Chapter 5: Data Structure & Algorithm

5.1 Data Structure

The logical or mathematical model of a particular organization of data is called a data Structure.

<u>Linear Data Structure:</u>

A data structure is said to be linear if its elements form a sequence or a linear list.

Examples:

Array
Linked List
Stacks
Queues

<u>Non Linear Data Structure:</u>

A non-linear data structure is a data structure in which a data item is connected to several other data items. So that a given data item has the possibility to reach one-or-more data items.

Examples

Graphs and
Trees.

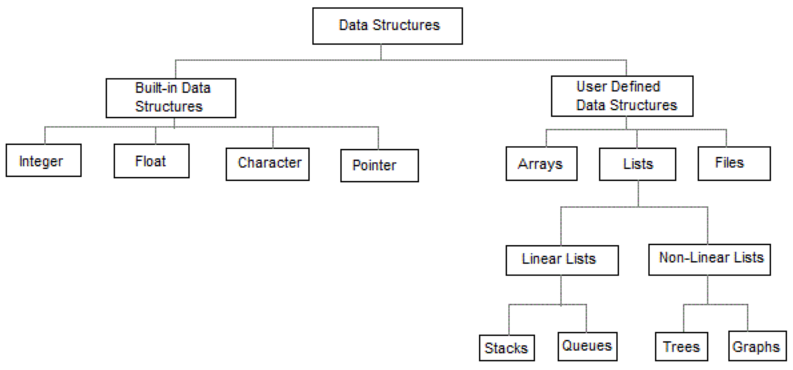
Traversal: Visit every part of the data structure
Search: Traversal through the data structure for a given element
Insertion: Adding new elements to the data structure
Deletion: Removing an element from the data structure.
Sorting: Rearranging the elements in some type of order (e.g Increasing or Decreasing)
Merging: Combining two similar data structures into
Arrays: Maximization, ordered lists, sparse matrices, representation of arrays.

Data Structure is a way of collecting and organizing data in such a way that we can perform operations on these data in an effective way. Data Structures is about rendering data elements in terms of some relationship, for better organization and storage. For example, we have data player's name "Virat" and age 26. Here "Virat" is of String data type and 26 is of integer data type.

<u>How is an Array different from Linked List?</u>

- The size of the arrays is fixed, Linked Lists are Dynamic in size.
- Inserting and deleting a new element in an array of elements is expensive, Whereas both insertion and deletion can easily be done in Linked Lists.
- Random access is not allowed in Linked Listed.
- Extra memory space for a pointer is required with each element of the Linked list.
- Arrays have better cache locality that can make a pretty big difference in performance.

5.1.1 Types of Data Structure



5.2 Algorithm

An algorithm is a finite set of instructions or logic, written in order, to accomplish a certain predefined task.

Algorithm is not the complete code or program, it is just the core logic (solution) of a problem, which can be expressed either as an informal high level description as pseudo-code or using a flowchart. An algorithm is said to be efficient and fast, if it takes less time to execute and consumes less memory space. The performance of an algorithm is measured on the basis of following properties: Time Complexity and Space Complexity.

1**.What is algorithm?**

Algorithm is a step by step procedure, which defines a set of instructions to be executed in certain order to get the desired output.

2.**Why we need to do algorithm analysis**?

A problem can be solved in more than one ways. So, many solution algorithms can be derived for a given problem. We analyze available algorithms to find and implement the best suitable algorithm.

3**.What are the criteria of algorithm analysis?**

An algorithm are generally analyzed on two factors − time and space. That is, how much **execution** time and how much **extra space** required by the algorithm.

4.**What is asymptotic analysis of an algorithm?**

Asymptotic analysis of an algorithm, refers to defining the mathematical boundation/framing of its run-time performance. Using asymptotic analysis, we can very well conclude the best case, average case and worst case scenario of an algorithm.

5. **What are asymptotic notations?**

Asymptotic analysis can provide three levels of mathematical binding of execution time of an algorithm −

- Best case is represented by Ω(n) notation.
- Worst case is represented by O(n) notation.
- Average case is represented by Θ(n) notation.

Complexity:

2) What is the Complexity of Algorithm?

The complexity of the algorithm is a way to classify how efficient an algorithm is compared to alternative ones. Its focus is on how execution time increases with the data set to be processed. The computational complexity of the algorithm is important in computing.

It is very suitable to classify algorithm based on the relative amount of time or relative amount of space they required and specify the growth of time/ space requirement as a function of input size.

Time complexity

Time complexity is a Running time of a program as a function of the size of the input.

Space complexity

Space complexity analyzes the algorithms, based on how much space an algorithm needs to complete its task. Space complexity analysis was critical in the early days of computing (when storage space on the computer was limited).

Nowadays, the problem of space rarely occurs because space on the computer is broadly enough.

We achieve the following types of analysis for complexity

5.2.1 Space Complexity & Time Complexity

It's the amount of memory space required by the algorithm, during the course of its execution. Space complexity must be taken seriously for multi-user systems and in situations where limited memory is available. An algorithm generally requires space for following components:

- ✓ Instruction Space: It's the space required to store the executable version of the program. This space is fixed, but varies depending upon the number of lines of code in the program.
- ✓ Data Space: It's the space required to store all the constants and variables value.
- ✓ Environment Space: It's the space required to store the environment information needed to resume the suspended function.

Time Complexity is a way to represent the amount of time needed by the program to run to completion. We will study this in details in our section.

5.2.2 Asymptotic Analysis

Asymptotic Analysis is the big idea that handles above issues in analyzing algorithms. In Asymptotic Analysis, we evaluate the performance of an algorithm in terms of input size (we don't measure the actual running time). We calculate, how does the time (or space) taken by an algorithm increases with the input size.

- ✓ **Worst Case Analysis (Usually Done):** In the worst case analysis, we calculate upper bound on running time of an algorithm.
- ✓ **Average Case Analysis (Sometimes done):** In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs.
- ✓ **Best Case Analysis (Bogus):** In the best case analysis, we calculate lower bound on running time of an algorithm.
- ✓
- ✓ **Worst-case** − the maximum number of steps taken on any instance of size **a**.
- ✓ **Best-case** − the minimum number of steps taken on any instance of size **a**.
- ✓ **Average case** − an average number of steps taken on any instance of size **a**.
- ✓ **Amortized** − A sequence of operations applied to the input of size **a** averaged over time.

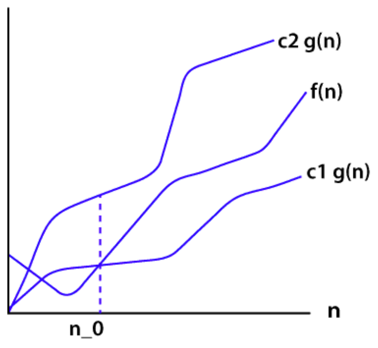
) What are the Asymptotic Notations?

Asymptotic analysis is used to measure the efficiency of an algorithm that doesn't depend on machine-specific constants and prevents the algorithm from comparing the time taking algorithm. Asymptotic notation is a mathematical tool that is used to represent the time complexity of algorithms for asymptotic analysis.

The three most used asymptotic notation is as follows.

θ Notation

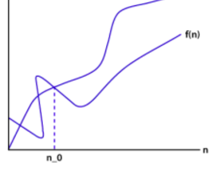
θ Notation defines the exact asymptotic behavior. To define a behavior, it bounds functions from above and below. A convenient way to get Theta notation of an expression is to drop low order terms and ignore leading constants.



f(n)= theta(g(n))

Big O Notation

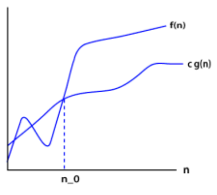
The Big O notation bounds a function from above, it defines an upper bound of an algorithm. Let's consider the case of insertion sort; it takes linear time in the best case and quadratic time in the worst case. The time complexity of insertion sort is $O(n^2)$. It is useful when we only have upper bound on time complexity of an algorithm.



f(n)=O(g(n))

Ω Notation

Just like Big O notation provides an asymptotic upper bound, the **Ω Notation** provides an asymptotic lower bound on a function. It is useful when we have lower bound on time complexity of an algorithm.



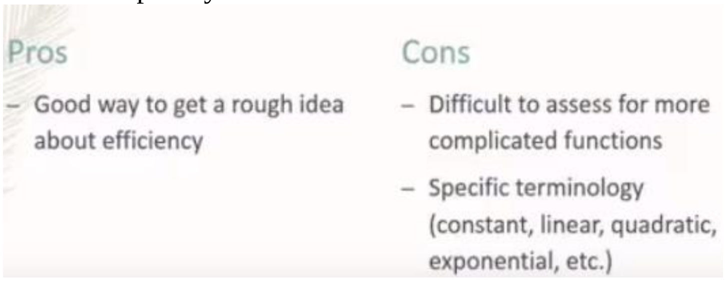
f(n)= Omega(g(n))

5.2.3 Asymptotic Notation

- ✓ Big Oh denotes "fewer than or the same as" <expression> iterations.
- ✓ Big Omega denotes "more than or the same as" <expression> iterations.
- ✓ Big Theta denotes "the same as" <expression> iterations.
- ✓ Little Oh denotes "fewer than" <expression> iterations.
- ✓ Little Omega denotes "more than" <expression> iterations.

Algorithm	Pseudo code
Formal definition with some specific characteristics that describes a process	Pseudocode is an informal and (often rudimentary) human readable description of an algorithm leaving many granular details of it.
Which could be executed by a Turing-complete computer machine to perform a specific task.	Writing a pseudocode has no restriction of styles and its only objective is to describe the high level steps of algorithm in a much realistic manner in natural language.
"algorithm" can be used to describe any high level task in computer science.	

Time Complexity:



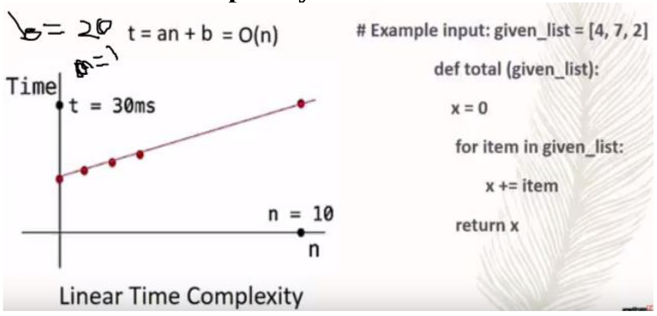
Pros

- Good way to get a rough idea about efficiency

Cons

- Difficult to assess for more complicated functions
- Specific terminology (constant, linear, quadratic, exponential, etc.)

Linear Time Complexity:

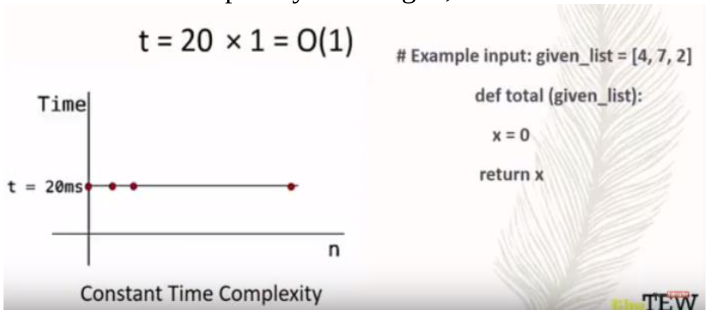


```
# Example input: given_list = [4, 7, 2]
def total (given_list):
    x = 0
    for item in given_list:
        x += item
    return x
```

When n=0 then t=20 and a=1;
Precise 0;
N=0 t=20
N=1 t=21
Each for loop take 1 ms
When run computer need 2s computer takes 2a much time
Lets called new time t2
T2=2an+40 =2O(n)
T2=O(n)
If computer slower or 10 time faster steel have linear complexity

Constant Complexity:
n=faster growing term here
a ,b are coefficient.

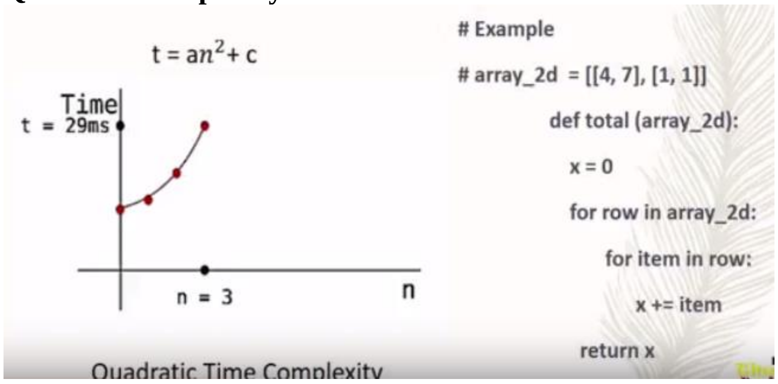
For bellow figure: t=c(constant)
Here 20 is coefficetn or t=an+b a=20,
Constant time complexity with Big O;



```
# Example input: given_list = [4, 7, 2]
def total (given_list):
    x = 0
    return x
```

Its always need 20 ms.
If its run at 5 times slow environment, then this time need 5*20 =100 ms.
Also we can write O(1)

Quadretic Complexity:



```
# Example
# array_2d  = [[4, 7], [1, 1]]
def total (array_2d):
    x = 0
    for row in array_2d:
        for item in row:
            x += item
    return x
```

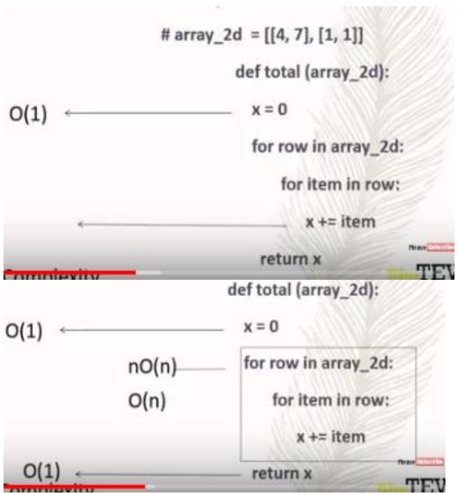
How we represent using Big Oh notation;
First remove coefficient a,

$$t = an + c = O(n^2)$$

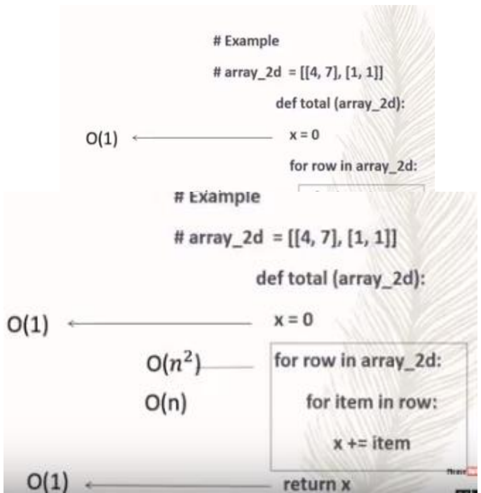
Time complexity & Big-O does not depend on environment. It just express faster.
2O(n)=O(n)
nO(n)=n(an+b)

$an^2 + bn$=O(n^2)
nO(x)=O(nx)
yO(x)=O(yx)

Again figure 3 quadretic there have n-row and n-column   n squre.



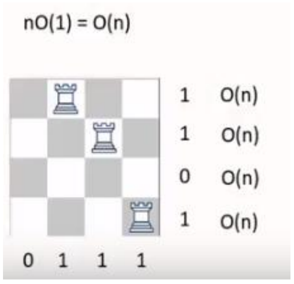
```
# array_2d  = [[4, 7], [1, 1]]
def total (array_2d):
    x = 0
    for row in array_2d:
        for item in row:
            x += item
    return x
```

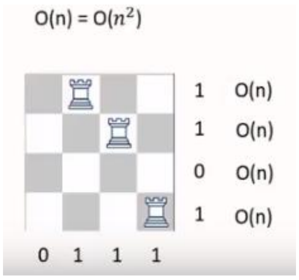


```
# Example
# array_2d  = [[4, 7], [1, 1]]
def total (array_2d):
    x = 0
    for row in array_2d:
        for item in row:
            x += item
    return x
```

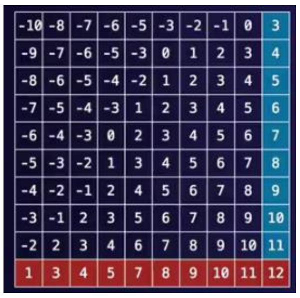
Chowdhury Al Akram
Convey: *01730980003*

Real life example:\





Worst case Scenario:
Counting the negative number from array:



-10	-8	-7	-6	-5	-3	-2	-1	0	3
-9	-7	-6	-5	-3	0	1	2	3	4
-8	-6	-5	-4	-2	1	2	3	4	5
-7	-5	-4	-3	1	2	3	4	5	6
-6	-4	-3	0	2	3	4	5	6	7
-5	-3	-2	1	3	4	5	6	7	8
-4	-2	-1	2	4	5	6	7	8	9
-3	-1	2	3	5	6	7	8	9	10
-2	2	3	4	6	7	8	9	10	11
1	3	4	5	7	8	9	10	11	12

When we calculate total number of negative number from top right corner to bottom left corner:

5.4 Searching Algorithms on Array

The process of finding a particular element in an array is called searching. There two popular searching techniques:

- Linear search, and
- Binary search.

The linear search compares each array element with the search key.
If the **search key** is a member of the array, typically the location of the search key is reported to indicate the presence of the search key in the array. Otherwise, a sentinel value is reported to indicate the absence of the search key in the array.

Swap without using third variable:

```
a= a + b;
b=a - b; // this will act like (a+b)-b, now b is equal to a.
a=a - b; // (a+b)-a, now, a is equal to b.
```

5.4.1 Linear Search

Each member of the array is visited until the search key is found.

A linear search searches an element or value from an array till the desired element or value is not found and it searches in a sequence order. It compares the element with all the other elements given in the list and if the element is matched it returns the value index else it return -1. Linear Search is applied on the unsorted or unordered list when there are fewer elements in a list.
To search the element 5 it will go step by step in a sequence order.

8	6	3	9	2	5

- **Example:**
  Write a program to search for the search key entered by the user in the following array:

(9, 4, 5, 1, 7, 78, 22, 15, 96, 45)

You can use the linear search in this example.

```
/* This program is an example of the Linear Search*/
        #include <stdio.h>
        #define SIZE 10
        int LinearSearch(int [], int);
        int main() {
        int a[SIZE]= {9, 4, 5, 1, 7, 78, 22, 15, 96, 45};
        int key, pos;
        printf("Enter the Search Key\n");
        scanf("%d", &key);
        pos = LinearSearch(a, key);
        if(pos == –1)
        printf("The search key is not in the array\n");
        else
        printf("The search key %d is at location %d\n", key, pos);
        return 0;
        }
        int LinearSearch (int b[ ], int skey) {
        int i;
        for (i=0; i < SIZE; i++)
        if(b[i] == skey)
        return i;
        return –1;
}
```

5.4.2 Binary Search

Binary Search is applied on the sorted array or list. In binary search, we first compare the value with the elements in the middle position of the array.

If the value is matched, then we return the value. If the value is less than the middle element, then it must lie in the lower half of the array and if it's greater than the element then it must lie in the upper half of the array. We repeat this procedure on the lower (or upper) half of the array. Binary Search is useful when there are large numbers of elements in an array.
To search an element 13 from the sorted array or list we have to follow below steps.

2	4	7	9	13	15

As we can see the above array is sorted in ascending order

- ✓ Binary search is applied on sorted lists only, so that we can make the search fast, by breaking the list every time
- ✓ Start with middle element
- ✓ If it is equal to the number we are searching then return
- ✓ If it is less than it then move to the right
- ✓ If it is more than it then move to the left
- ✓ And then repeat till you find the number

How Binary Search Works:

The binary search algorithm applied to our array DATA works as follows. During each stage of our algorithm, our search for ITEM is reduced to a segment of elements of DATA

DATA[BEG], DATA[BEG + 11, DATA[BEG + 2],..., DATA[END]

Note that the variables **BEG and END denote, respectively, the beginning and end locations of the segment under consideration**.

The algorithm compares **ITEM** with the middle clement **DATA[MID]** of the segment, **where MID** is obtained by

MID = INT((BEG + END)/2) (We use INT(A) for the integer value of A.)

If DATA[MID] = ITEM, then the search is successful and

we set **LOC : = MID.** Otherwise a new segment of DATA is obtained as follows:

(a) **If ITEM < DATA[MID]**, then ITEM can **appear only in the left half of the segment**:

DATA[BEG], DATA[BEG +11 .....DATA[MID —1]

So we reset END : = MID - 1 and begin searching again.

(b) **If ITEM> DATA[MID],** then -ITEM can appear only in the right half of the segment:

DATA[MID + fl DATA[MID + 2]..... DATA [END]

So we reset BEG := MID + 1 and begin searching again.

<u>EXAMPLE</u>

Let DATA be the following sorted 13-clement array:

DATA: 11, 22, 30, 33, 40, 44, , 60, 66, 77, 80, 88, 99

We apply the binary search t o DATA for 'different values of ITEM.

Suppose **ITEM = 40**. The search for ITEM in the array DATA is pictured in Fig. 4-6, where the values of **DATA[BEG]** and **DATA[END]** in each stage of the algorithm are indicated by circles and the value of DATA Specifically, BEG, END and MID will have the following successive values:

(I) Initially, **BEG= 1** and **END = 13**. Hence

MID = INT((1 + 13)/2] =7 and so DATA(MIDJ = 55

(2) **Since 40< 55, END has its value changed by END = MID - 1** =6

Hence **MID = INT [(1 + 6)/2 = 3 and so DATALMID] = 30**

(3) **Since 40>30, BEG has its value changed by BEG = MID + 1=4**

Hence MID INT[4 + 6)/2]=5 and so **DATA[MID]=40**

We have found ITEM in location LOC MID= 5.

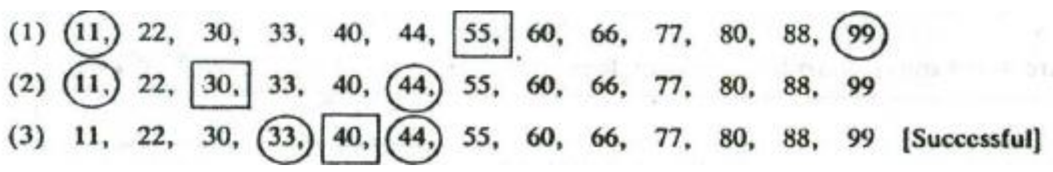


Fig. 4-6 Binary search for ITEM = 40.

Question: For column major order find out the address of the element score[15,3] from a 20X5 matrix array score with base value 100 and w=4. Exam-2015

```
#include <stdio.h>
#define SIZE 10
int BinarySearch(int [ ], int);
int main(){
int a[SIZE]= {3, 5, 9, 11, 15, 17, 22, 25, 37, 68};
int key, pos;
printf("Enter the Search Key\n");
scanf("%d",&key);
pos = BinarySearch(a, key);
if(pos == □1)
        printf("The search key is not in the array\n");
else
        printf("The search key %d is at location %d\n", key, pos);
return 0;
}

int BinarySearch (int A[], int skey){
int low=0, high=SIZE–1, middle;
while(low <= high){
middle = (low+high)/2;
if (skey == A[middle])
        return middle;
else if(skey <A[middle])
        high = middle – 1;
else
        low = middle + 1;
}
return –1;
  }
```

5.3 Sorting

Sorting is nothing but storage of data in sorted order; it can be in ascending or descending order. The term Sorting comes into picture with the term Searching. There are so many things in our real life that we need to search, like a particular record in database, roll numbers in merit list, a particular telephone number, any particular page in a book etc.

Input:

A sequence of n numbers $a_1, a_2, \ldots, a_n$

Output:

A permutation (reordering) $a_1', a_2', \ldots, a_n'$ of the input sequence such that $a_1' \leq a_2' \leq \cdots \leq a_n'$

Why Study Sorting Algorithms?

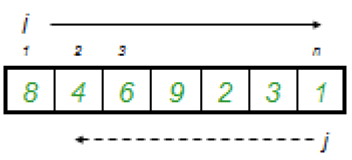
- There are a variety of situations that we can encounter
  - Do we have randomly ordered keys?
  - Are all keys distinct?
  - How large is the set of keys to be ordered?
  - Need guaranteed performance?
- Various algorithms are better suited to some of these situations

Some Definitions:

- Internal Sort
  - The data to be sorted is all stored in the computer's main memory.
- External Sort
  - Some of the data to be sorted might be stored in some external, slower, device.
- In Place Sort
  - The amount of extra space required to sort the data is constant with the input size.

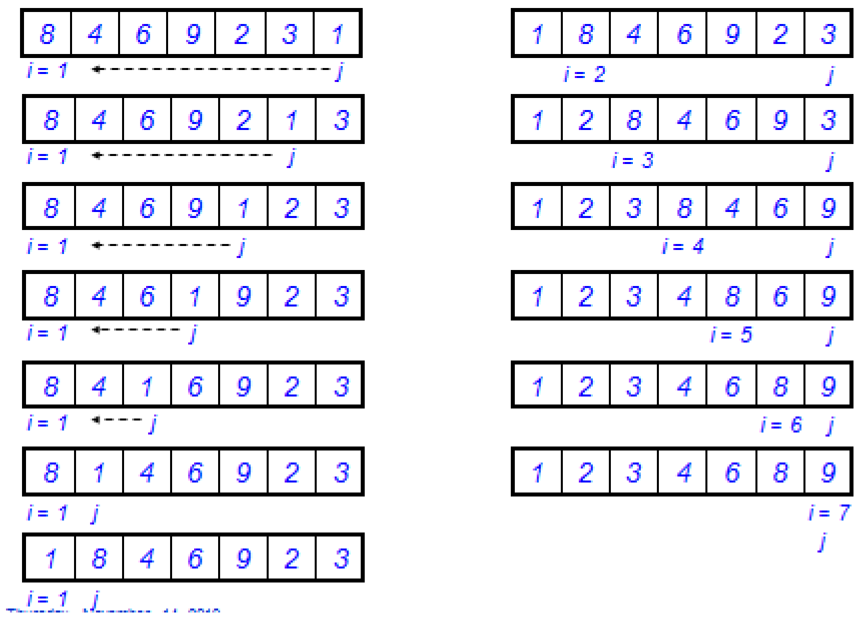
5.3.1 Bubble Sort

- **Idea:**
  - Repeatedly pass through the array
  - Swaps adjacent elements that are out of order



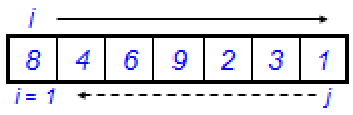
- Easier to implement, but slower than Insertion sort

Chowdhury Al Akram
Convey: *01730980003*



Alg.: BUBBLESORT(A)

```
for i ← 1 to length[A]
    do for j ← length[A] downto i + 1
        do if A[j] < A[j -1]
            then exchange A[j] ↔ A[j-1]
```



```
Alg.: BUBBLESORT(A)
    for i ← 1 to length[A]                          c1
        do for j ← length[A] downto i + 1           c2
Comparisons: ≈ n²/2   do if A[j] < A[j -1]          c3
Exchanges: ≈ n²/2     then exchange A[j] ↔ A[j-1]   c4
```

$$T(n) = c_1(n+1) + c_2 \sum_{i=1}^{n}(n-i+1) + c_3 \sum_{i=1}^{n}(n-i) + c_4 \sum_{i=1}^{n}(n-i)$$

$$= \Theta(n) + (c_2 + c_2 + c_4) \sum_{i=1}^{n}(n-i)$$

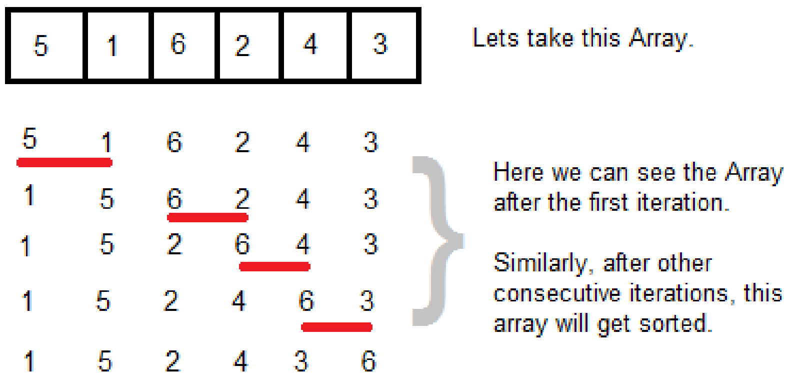
$$\text{where } \sum_{i=1}^{n}(n-i) = \sum_{i=1}^{n} n - \sum_{i=1}^{n} i = n^2 - \frac{n(n+1)}{2} = \frac{n^2}{2} - \frac{n}{2}$$

Thus, $T(n) = \Theta(n^2)$

Bubble sort is an elementary sorting algorithm, which works by repeatedly exchanging adjacent elements, if necessary. When no exchanges are required, the file is sorted.

Bubble Sort is an algorithm which is used to sort N elements that are given in a memory for e.g. an Array with N number of elements. Bubble Sort compares all the elements one by one and sort them based on their values. It is called Bubble sort, because with each iteration the smaller element in the list bubbles up towards the first place, just like a water bubble rises up to the water surface.

Sorting takes place by stepping through all the data items one-by-one in pairs and comparing adjacent data items and swapping each pair that is out of order.


5 1 6 2 4 3
Lets take this Array.
5 1 6 2 4 3
1 5 6 2 4 3
1 5 2 6 4 3
1 5 2 4 6 3
1 5 2 4 3 6
Here we can see the Array after the first iteration.
Similarly, after other consecutive iterations, this array will get sorted.

```
for i← 1 to length [A] do
for j ← length [A] down-to i +1 do
  if A[j] < A[j - 1] then
    Exchange A[j] ↔ A[j-1]
```

Implementation

```
Void bubbleSort(int numbers[], intarray_size) {
  inti, j, temp;
  for (i = (array_size - 1); i >= 0; i--)
  for (j = 1; j <= i; j++)
    if (numbers[j - 1] > numbers[j]) {
      temp = numbers[j-1];
      numbers[j - 1] = numbers[j];
      numbers[j] = temp;
    }
  }
```

Here we have two nested loops, and therefore calculating the run time is straight-forward

$$\sum_{k=1}^{n-1}(n-k)=n(n-1)-\frac{n(n-1)}{2}=\frac{n(n-1)}{2}=\Theta(n^2)$$

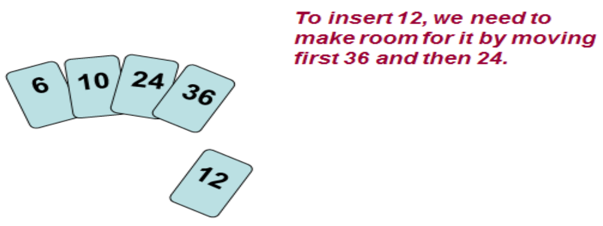
Clearly, the graph shows the n^2 nature of the bubble sort.
Consider the unsorted array to the right

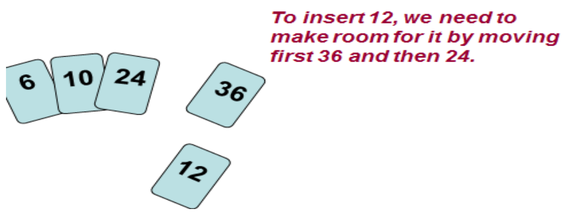
5.3.2 Insertion Sort

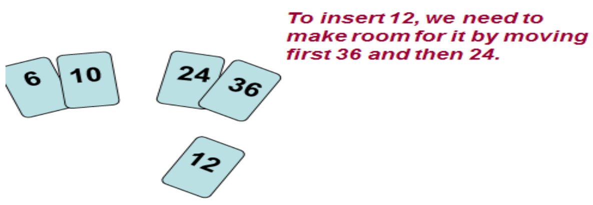
Idea: like sorting a hand of playing cards

- Start with an empty left hand and the cards facing down on the table.
- Remove one card at a time from the table, and insert it into the correct position in the left hand
  - compare it with each of the cards already in the hand, from right to left
- The cards held in the left hand are sorted
  - these cards were originally the top cards of the pile on the table

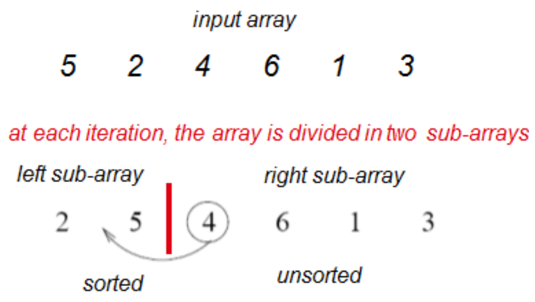
Insertion sort:

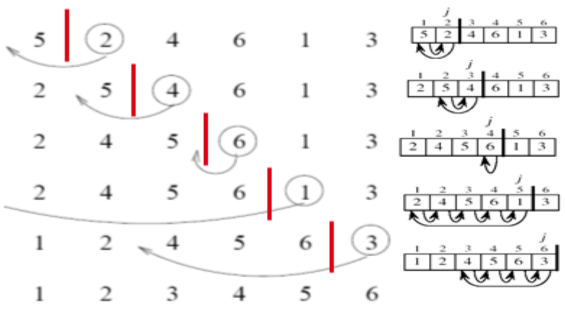

To insert 12, we need to make room for it by moving first 36 and then 24.
6
10
24
36
12


To insert 12, we need to make room for it by moving first 36 and then 24.
6
10
24
36
12


To insert 12, we need to make room for it by moving first 36 and then 24.
6
10
24
36
12

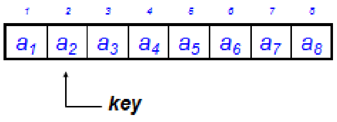
To insert 12, we need to make room for it by moving first 36 and then 24.
6
10
12
24
36





Algorithm:

```
Alg.: INSERTION-SORT(A)
  for j ← 2 to n
      do key ← A[ j ]
        ▷Insert A[ j ] into the sorted sequence A[1 . . j -1]
          i ← j - 1
          while i > 0 and A[i] > key
              do A[i + 1] ← A[i]
                 i ← i - 1
          A[i + 1] ← key
```



- Insertion sort – sorts the elements in place

Analysis of Insertion Sort:

INSERTION-SORT(A)	cost	times
for j ← 2 to n	c_1	n
do key ← A[j]	c_2	n-1
▷Insert A[j] into the sorted sequence A[1 . . j -1]	0	n-1
i ← j - 1	c_4	n-1
while i > 0 and A[i] > key	c_5	$\sum_{j=2}^{n} t_j$
do A[i + 1] ← A[i]	c_6	$\sum_{j=2}^{n} (t_j - 1)$
i ← i – 1	c_7	$\sum_{j=2}^{n} (t_j - 1)$
A[i + 1] ← key	c_8	n-1

t_j: # of times the while statement is executed at iteration j

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\sum_{j=2}^{n} t_j + c_6\sum_{j=2}^{n}(t_j - 1) + c_7\sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$$

Best Case Analysis:

- The array is already sorted
  - A[i] ≤ key upon the first time **the while** loop test is run (when *i* = *j* -1)
  - $t_j = 1$
- $T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) = (c_1 + c_2 + c_4 + c_5 + c_8)n + (c_2 + c_4 + c_5 + c_8)$
  - $= an + b = \Theta(n)$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\sum_{j=2}^{n} t_j + c_6\sum_{j=2}^{n}(t_j - 1) + c_7\sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$$

Worst Case Analysis:

- The array is in reverse sorted order — "**while** i > 0 and A[i] > key"
  - Always A[i] > key in while loop test
  - Have to compare key with all elements to the left of the j-th position $\Rightarrow$ compare with j-1 elements $\Rightarrow t_j = j$

using $\sum_{j=1}^{n} j = \frac{n(n+1)}{2} \to \sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1 \to \sum_{j=2}^{n}(j-1) = \frac{n(n-1)}{2}$ we have:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\frac{n(n-1)}{2} + c_7\frac{n(n-1)}{2} + c_8(n-1)$$

$= an^2 + bn + c$ a quadratic function of n

- $T(n) = \Theta(n^2)$ order of growth in n^2

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\sum_{j=2}^{n} t_j + c_6\sum_{j=2}^{n}(t_j - 1) + c_7\sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$$

Comparisons and Exchanges in Insertion Sort:

INSERTION-SORT*(A)*		cost	times
for j ← 2 **to** n		c_1	n
do key ← A[j]		c_2	$n-1$
Insert A[j] into the sorted sequence A[1 . . j -1]		0	$n-1$
i ← j - 1	$\approx n^2/2$ comparisons	c_4	$n-1$
while i > 0 and A[i] > key		c_5	$\sum_{j=2}^{n} t_j$
do A[i + 1] ← A[i]		c_6	$\sum_{j=2}^{n}(t_j - 1)$
i ← i – 1	$\approx n^2/2$ exchanges	c_7	$\sum_{j=2}^{n}(t_j - 1)$
A[i + 1] ← key		c_8	$n-1$

Insertion Sort – Summary:

- Advantages
  - Good running time for "almost sorted" arrays $\Theta(n)$
- Disadvantages
  - $\Theta(n^2)$ running time in worst and average case
  - $\approx n^2/2$ comparisons and $\approx n^2/2$ exchanges

It is a simple Sorting algorithm which sorts the array by shifting elements one by one. Following are some of the important characteristics of Insertion Sort.

- ✓ It has one of the simplest implementation
- ✓ It is efficient for smaller data sets, but very inefficient for larger lists.
- ✓ Insertion Sort is adaptive, that means it reduces its total number of steps if given a partially sorted list, hence it increases its efficiency.
- ✓ It is better than Selection Sort and Bubble Sort algorithms.

- ✓ Its space complexity is less, like Bubble Sorting; insertion sort also requires a single additional memory space.
- ✓ It is Stable, as it does not change the relative order of elements with equal keys

INSERTION-SORT*(A)*

```
for j ← 2 to n
        assign key ← A[ j ]
            Insert A[ j ] into the sorted sequence A[1 . . j -1]
           i ← j - 1
           while i > 0 and A[i] > key
                assign A[i + 1] ← A[i]
                    i ← i – 1
           A[i + 1] ← key
```

The numbers, which are needed to be sorted, are known as keys. Here is the algorithm of the insertion sort method.

Analysis

Run time of this algorithm is very much dependent on the given input. If the given numbers are sorted, this algorithm runs in O (n) time. If the given numbers are in reverse order, the algorithm runs in O (n^2) time.

Example

Unsorted list: **2 13 5 18 14**

1st iteration:
Key = a[2] = 13
a[1] = 2 < 13

Swap, no swap 2 13 5 18 14

2nd iteration:
Key = a[3] = 5
a[2] = 13 > 5

Swap 5 and 13 2 5 13 18 14

Next, a[1] = 2 < 13

Swap, no swap 2 5 13 18 14

3rd iteration:
Key = a[4] = 18
a[3] = 13 < 18,
a[2] = 5 < 18,
a[1] = 2 < 18

Swap, no swap 2 5 13 18 14

4th iteration:
Key = a[5] = 14
a[4] = 18 > 14

Swap 18 and 14 2 5 13 14 18

Next, a[3] = 13 < 14,
a[2] = 5 < 14,
a[1] = 2 < 14

So, no swap 2 5 13 14 18

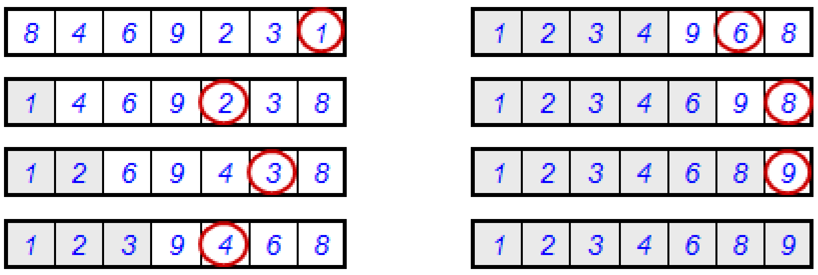
Finally,

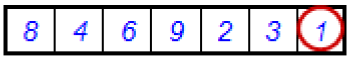
the sorted list is 2 5 13 14 18

5.3.3 Selection Sort

- Idea:
  - Find the smallest element in the array
  - Exchange it with the element in the first position
  - Find the second smallest element and exchange it with the element in the second position
  - Continue until the array is sorted

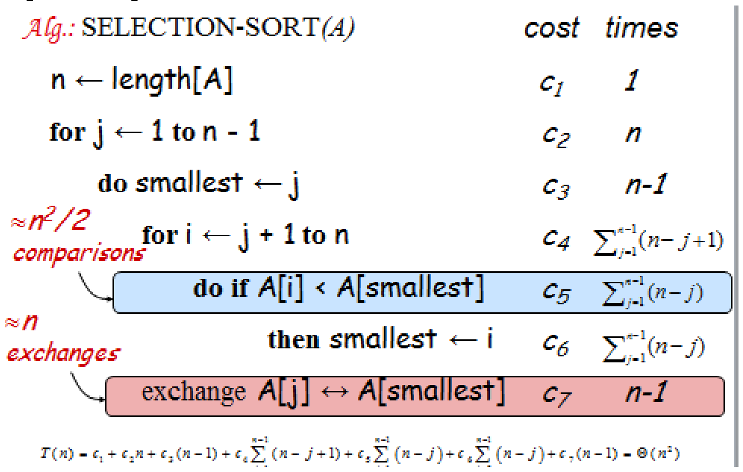
Example:





```
Alg.: SELECTION-SORT(A)
        n ← length[A]
        for j ← 1 to n – 1 do
                smallest ← j
                for i ← j + 1 to n do
                        if A[i] < A[smallest]
                                then smallest ← i
                exchange A[j] ↔ A[smallest]
```

Analysis of Selection Sort:



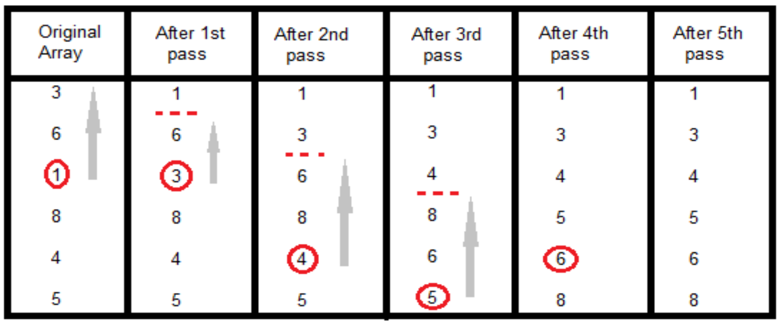
Alg.: SELECTION-SORT(A)	cost	times
n ← length[A]	c_1	1
for j ← 1 to n - 1	c_2	n
do smallest ← j	c_3	n-1
for i ← j + 1 to n	c_4	$\sum_{j=1}^{n-1}(n-j+1)$
do if A[i] < A[smallest]	c_5	$\sum_{j=1}^{n-1}(n-j)$
then smallest ← i	c_6	$\sum_{j=1}^{n-1}(n-j)$
exchange A[j] ↔ A[smallest]	c_7	n-1

$\approx n^2/2$ comparisons
$\approx n$ exchanges

$$T(n) = c_1 + c_2 n + c_3(n-1) + c_4\sum_{j=1}^{n-1}(n-j+1) + c_5\sum_{j=1}^{n-1}(n-j) + c_6\sum_{j=1}^{n-1}(n-j) + c_7(n-1) = \Theta(n^2)$$

Selection sorting is conceptually the simplest sorting algorithm. This algorithm first finds the smallest element in the array and exchanges it with the element in the first position, then finds the second smallest element and exchange it with the element in the second position, and continues in this way until the entire array is sorted.

In the first pass, the smallest element found is 1, so it is placed at the first position, then leaving first element, smallest element is searched from the rest of the elements, 3 is the smallest, so it is then placed at the second position. Then we leave 1 and 3, from the rest of the elements, we search for the smallest and put it at third position and keep doing this, until array is sorted.



Original Array	After 1st pass	After 2nd pass	After 3rd pass	After 4th pass	After 5th pass
3	1	1	1	1	1
6	6	3	3	3	3
1	3	6	4	4	4
8	8	8	8	5	5
4	4	4	6	6	6
5	5	5	5	8	8

```
SELECTION-SORT(A)
        n ← length[A]
        for j ← 1 to n - 1
                assign smallest ← j
                    for i ← j + 1 to n
                          if A[i] < A[smallest]
                                  then smallest ← i
                    exchange A[j] ↔ A[smallest]
```

Selection sort is quadratic in both the worst and the average case, and requires no extra memory.

For each ***i* from *1* to *n - 1*, there is one exchange and *n - i* comparisons, so there is a total of *n - 1* exchanges and**
(n − 1) + (n − 2) + ...+ 2 + 1 = n (n − 1)/2 comparisons.

These observations hold, no matter what the input data is.
In the worst case, t**his could be quadratic, but in the average case, this quantity is O (n log n).**

It implies that the running time of Selection sort is quite insensitive to the input.

Example

Unsorted list: 5 2 1 4 3

1st iteration:
Smallest = 5
2 < 5, smallest = 2
1 < 2, smallest = 1
4 > 1, smallest = 1
3 > 1, smallest = 1

Swap 5 and 1 1 2 5 4 3

2nd iteration:
Smallest = 2
2 < 5, smallest = 2
2 < 4, smallest = 2
2 < 3, smallest = 2

No Swap 1 2 5 4 3

3rd iteration:
Smallest = 5
4 < 5, smallest = 4
3 < 4, smallest = 3

Swap 5 and 3 1 2 3 4 5

4th iteration:
Smallest = 4

4 < 5, smallest = 4

No Swap 1 2 3 4 5

Finally,

the sorted list is 1 2 3 4 5

Merge Sort and Quick Sort:

Two well-known sorting algorithms adopt this divide-and-conquer strategy

Merge sort

Divide step is trivial – just split the list into two equal parts

Work is carried out in the conquer step by merging two sorted lists

Quick sort

Work is carried out in the divide step using a pivot element

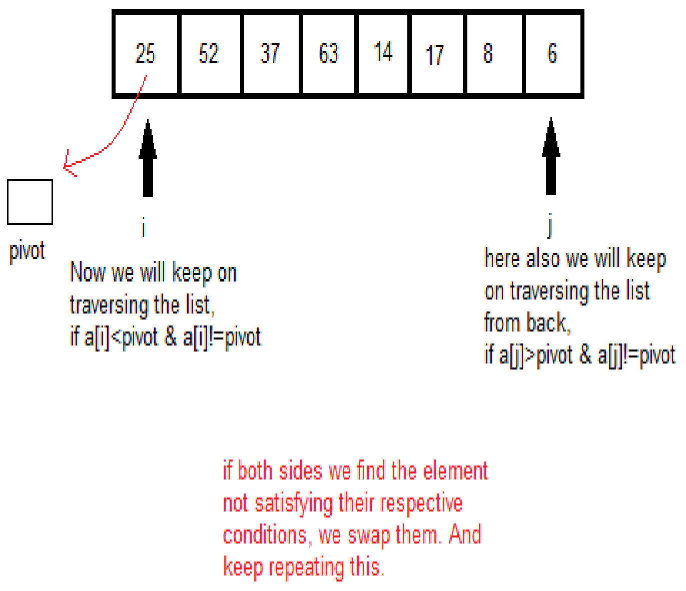
Conquer step is trivial

5.3.4 Quick Sort

It is used on the principle of divide-and-conquer. It is a good general purpose sort and it consumes relatively fewer resources during execution.

Quick Sort, as the name suggests, sorts any list very quickly. Quick sort is not stable search, but it is very fast and requires very less additional space. It is based on the rule of Divide and Conquer (also called partition-exchange sort). This algorithm divides the list into three main parts:

- ✓ Elements less than the Pivot element
- ✓ Pivot element
- ✓ Elements greater than the pivot element


25
52
37
63
14
17
8
6
pivot
i
Now we will keep on traversing the list, if a[i]<pivot & a[i]!=pivot
j
here also we will keep on traversing the list from back, if a[j]>pivot & a[j]!=pivot
if both sides we find the element not satisfying their respective conditions, we swap them. And keep repeating this.

In the list of elements, mentioned in below example, we have taken 25 as pivot. So after the first pass, the list will be changed like this.

6 8 17 14 25 63 37 52

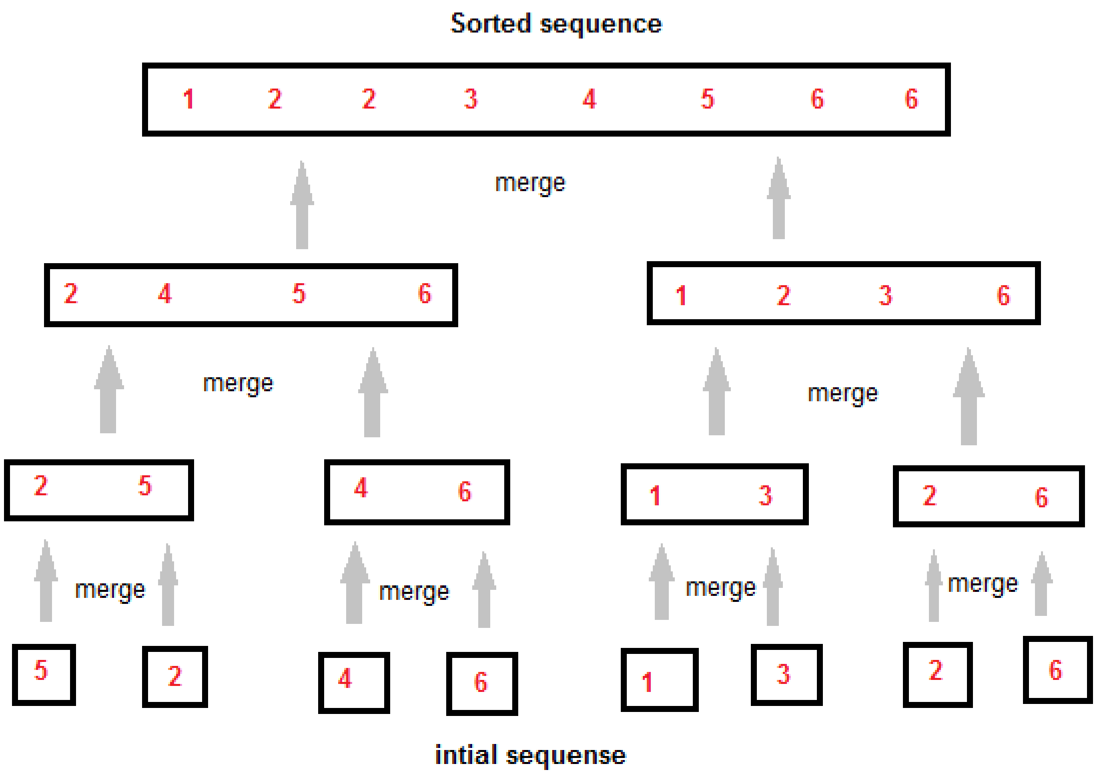
Hence after the first pass, pivot will be set at its position, with all the elements smaller to it on its left and all the elements larger than it on the right. Now 6 8 17 14 and 63 37 52 are considered as two separate lists, and same logic is applied on them, and we keep doing this until the complete list is sorted.

5.3.5 Merge Sort

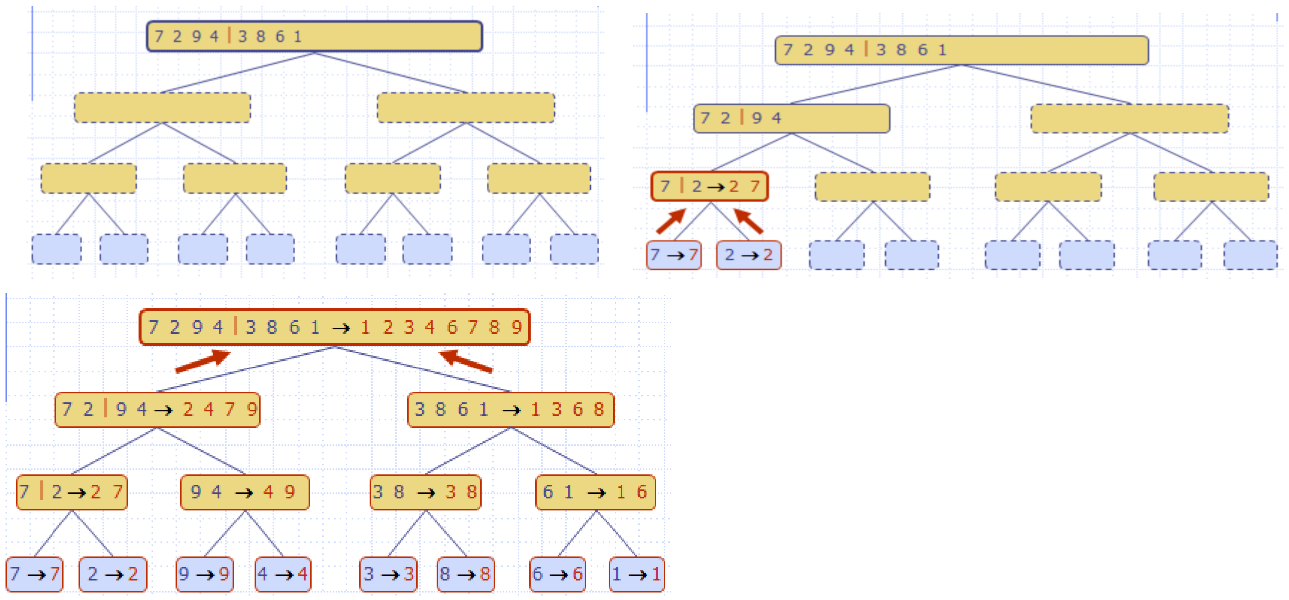
Merge Sort follows the rule of Divide and Conquer. But it doesn't divide the list into two halves. In merge sort the unsorted list is divided into N sublists, each having one element, because a list of one element is considered sorted.

Then, it repeatedly merges these sublists, to produce new sorted sublists, and at lasts one sorted list is produced.

Merge Sort is quite fast, and has a time complexity of O(n log n). It is also a stable sort, which means the "equal" elements are ordered in the same order in the sorted list.



Execution Example:



Merge Sort: Running Time:

The recurrence for the worst-case running time T(n) is

$$T(n) = \begin{cases} b & \text{if } n = 1 \\ 2T(n/2) + bn & \text{if } n > 1 \end{cases}$$

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

Solve this recurrence by

(1) iteratively expansion

(2) using the recursion tree

$$
\begin{aligned}
T(n) &= 2T(n/2) + bn \\
&= 2(2T(n/2^2)) + b(n/2)) + bn \\
&= 2^2T(n/2^2) + 2bn \\
&= 2^3T(n/2^3) + 3bn \\
&= 2^4T(n/2^4) + 4bn \\
&= \ldots \\
&= 2^iT(n/2^i) + ibn
\end{aligned}
$$

◆Note that base, $T(n) = b$, case occurs when $2^i = n$.
That is, $i = \log n$.

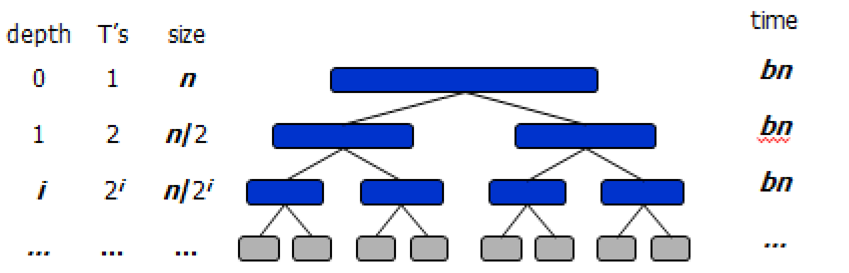
◆So, $T(n) = bn + bn\log n$

◆Thus, $T(n)$ is $O(n \log n)$.

Merge Sort: Running Time (Recursion Tree)

Draw the recursion tree for the recurrence relation and look for a pattern:

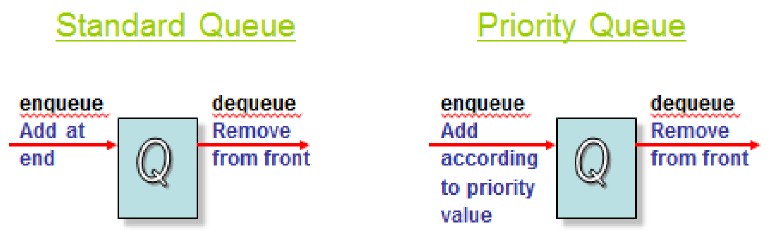
$$T(n) = \begin{cases} b & \text{if } n = 1 \\ 2T(n/2) + bn & \text{if } n \geq 2 \end{cases}$$



Total time = $bn + bn \log n$
(last level plus all previous levels)

Heaps & Heap Sort Priority Queues:

Priority Queues:



Applications of Priority Queues:

- ❖ Line-up of Incoming Planes at Airport
  - ○ Possible Criteria for Priority?
- ❖ Operating Systems Priority Queues?
- ❖ Several criteria could be mapped to a priority status

Max-Priority Queue Operations

Insert(S, x) – Inserts element x into set S, according to its priority

Maximum(*S*) – Returns, but does not remove, the element of *S* with the largest key

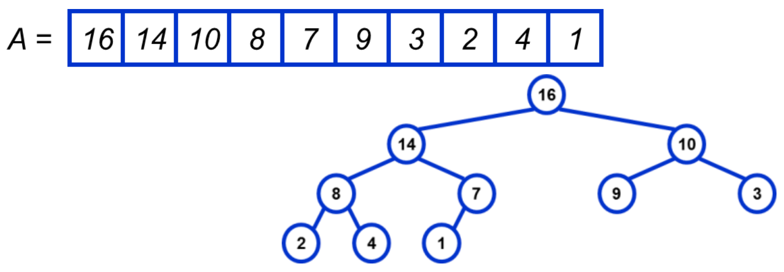
Extract-Max(*S*) – Removes and returns the element of *S* with the largest key

Increase-Key(*S, x, k*) – Increases the value of element *x*'s keyto the new value *k*

Possible Implementations?

Binary Heaps:

- ❖ The (binary) heap data structure is an array object that can be viewed as a complete binary tree
  - Each node of the tree corresponds to an element of the array that stores the value in the node.
  - The tree is completely filled on all levels except possibly the lowest, where it is filled from the left up to a point.



- To represent a complete binary tree as an array:
  - The root node is A[1]
  - Node *i* is A[*i*]
  - The parent of node *i* is A[*i*/2]
  - The left child of node *i* is A[2*i*]
  - The right child of node *i* is A[2*i* + 1]

```
Parent(i)
  return floor(i/2)
```

```
Right(i)
  return 2i+1
```

```
Left(i)
   return 2i
```

Types of Binary Heaps:

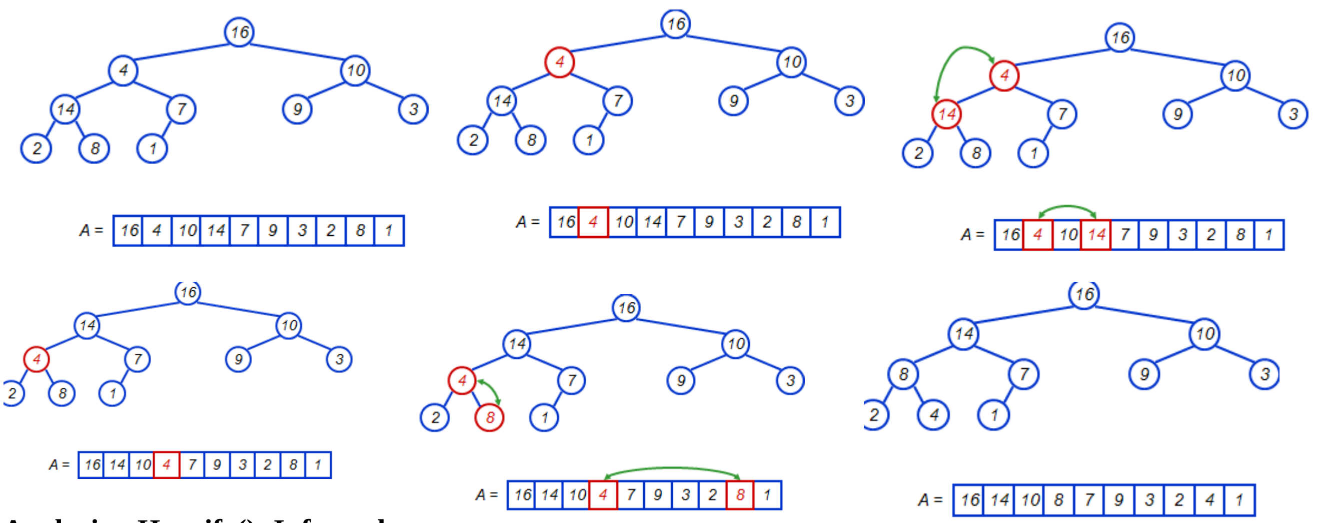
- ❖ Min-Heaps:
  - The element in the root is less than or equal to all elements in both of its sub-trees
  - Both of its sub-trees are Min-Heaps
- ❖ Max-Heaps:
  - The element in the root is greater than or equal to all elements in both its sub-trees
  - Both of its sub-trees are Max-Heaps

- ❖ Max-Heaps satisfy the *heap property*:
  - ▪ $A[Parent(i)] \geq A[i]$ for all nodes $i > 1$
  - In other words, the value of a node is at most the value of its parent
  - The largest element in a max-heap is stored at the root
- ❖ Min-Heaps satisfy the *heap property*:
  - ▪ $A[Parent(i)] \leq A[i]$ for all nodes $i > 1$
  - In other words, the value of a node is at least the value of its parent
  - The smallest element in a min-heap is stored at the root

Max-Heap Operations: Max-Heapify():

- ❖ Max-Heapify(): maintain the max-heap property
  - Given: a node *i* in the heap with children *l* and *r*

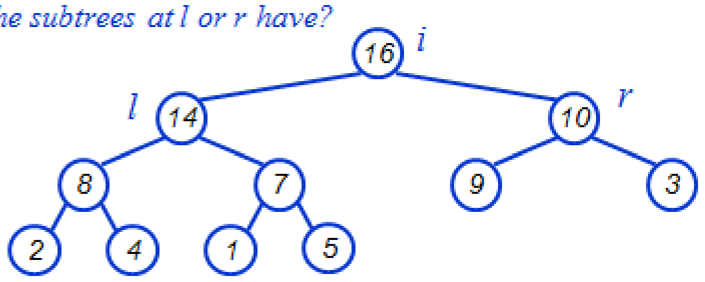
- Given: two subtrees rooted at *l* and *r*, assumed to be heaps
- Problem: The subtree rooted at *i* may violate the heap property (*How?*)
- Action: let the value of the parent node “float down” so subtree at *i* satisfies the heap property
    - *What do you suppose will be the basic operation between i, l, and r*


A = 16 4 10 14 7 9 3 2 8 1
A = 16 4 10 14 7 9 3 2 8 1
A = 16 4 10 14 7 9 3 2 8 1
A = 16 14 10 4 7 9 3 2 8 1
A = 16 14 10 4 7 9 3 2 8 1
A = 16 14 10 8 7 9 3 2 4 1

Analyzing Heapify(): Informal:

- ❖ Aside from the recursive call, what is the running time of **Heapify()**?
- ❖ How many times can **Heapify()** recursively call itself?
- ❖ What is the worst-case running time of **Heapify()** on a heap of size n?

- Fixing up relationships among the elements $A[i]$, $A[l]$, and $A[r]$ takes $\Theta(1)$ time
- *If the heap at i has n elements, at most how many elements can the subtrees at l or r have?*



- Answer: $2n/3$ (worst case: bottom row half full)
- So time taken by `Heapify()` is given by

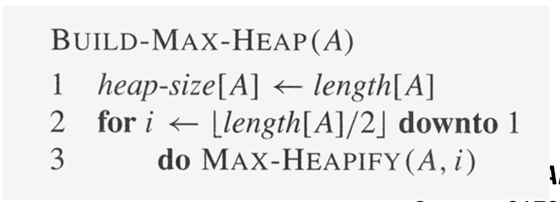
  $T(n) \le T(2n/3) + \Theta(1)$

- ❖ So we have
  - $T(n) \le T(2n/3) + \Theta(1)$
- ❖ Solving the recurrence, we have
  - $T(n) = O(\lg n)$
- ❖ Thus, **Heapify()** takes $O(h)$ time for a node at height h.

Heap Operations: BuildHeap():

- ❖ We can build a heap in a bottom-up manner by running **Heapify()** on successive subarrays
  - Fact: for array of length n, all elements in the range $A[\lfloor n/2 \rfloor + 1 \ldots n]$ are heaps (*Why?*)
  - So
    - Walk backwards through the array from $n/2$ to 1, calling **Heapify()** on each node.
    - Order of processing guarantees that the children of node i are heaps when i is processed

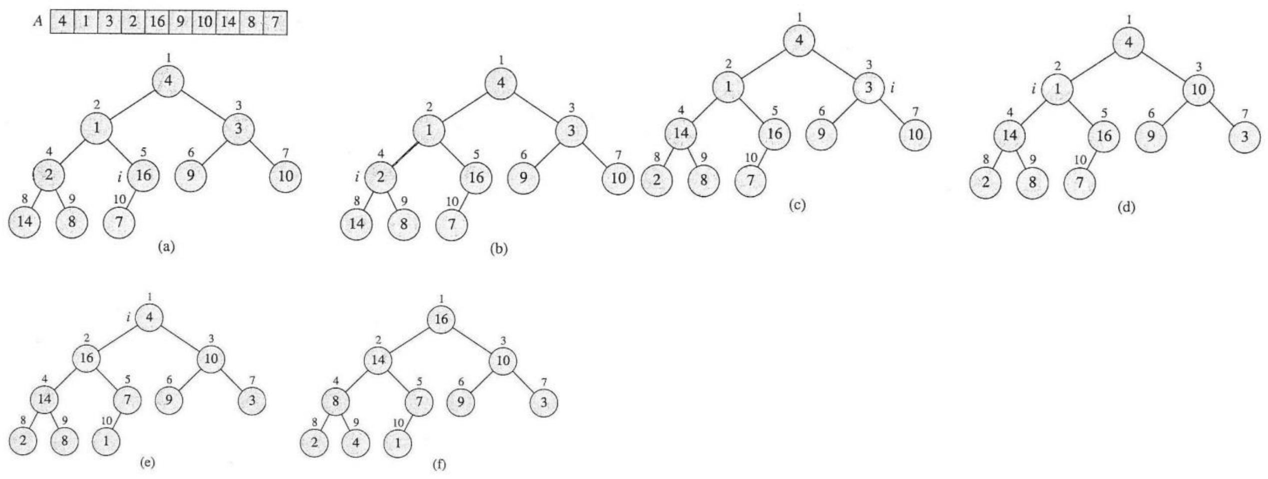
Converts an unorganized array A into a max- heap:



```
BUILD-MAX-HEAP(A)
1  heap-size[A] ← length[A]
2  for i ← ⌊length[A]/2⌋ downto 1
3        do MAX-HEAPIFY(A, i)
```

BuildHeap() Example:

- ❖ Work through example
  A = {4, 1, 3, 2, 16, 9, 10, 14, 8, 7}



Analyzing BuildHeap():

- ❖ Each call to **Heapify()** takes $O(\lg n)$ time
- ❖ There are $O(n)$ such calls (specifically, $\lfloor n/2 \rfloor$)
- ❖ Thus the running time is $O(n \lg n)$
  - o *Is this a correct asymptotic upper bound?*
  - o *Is this an asymptotically tight bound?*
- ❖ A tighter bound is $O(n)$
  - o *How can this be? Is there a flaw in the above reasoning?*

Analyzing BuildHeap(): Tight:

- ❖ To **Heapify()** a subtree takes $O(h)$ time, where h is the height of the subtree
  - o $h = O(\lg m)$, m = # nodes in the subtree
  - o The height of most subtrees is small
- ❖ Fact: an n-element heap has at most $\lceil n/2^{h+1} \rceil$ nodes of height h
- ❖ Prove that **BuildHeap()** takes $O(n)$ time

HeapSort

- ❖ Given **BuildHeap()**, a sorting algorithm can easily be constructed:
  - o Maximum element is at A[1]
  - o Discard by swapping with element at A[n]
    - ▪ Decrement heap_size[A]
    - ▪ A[n] now contains correct value
  - o Restore heap property at A[1] by calling **Heapify()**
  - o Repeat, always swapping A[1] for A[heap_size(A)]

```
Heapsort(A)
{
        BuildHeap(A);
        for (i = length(A) downto 2)
        {
                Swap(A[1], A[i]);
                heap_size(A) = heap_size(A) – 1;
                Heapify(A, 1);
        }
}
```

Analyzing Heapsort:

- ❖ The call to **BuildHeap()** takes $O(n)$ time
- ❖ Each of the $n - 1$ calls to **Heapify()** takes $O(\lg n)$ time

- Thus the total time taken by **HeapSort()**
  $= O(n) + (n - 1)\, O(\lg n)$
  $= O(n) + O(n \lg n)$
  $= O(n \lg n)$

Priority Queues:

- Heapsort is a nice algorithm, but in practice Quicksort (coming up) usually wins
- The heap data structure is incredibly useful for implementing *priority queues*
  - A data structure for maintaining a set *S* of elements, each with an associated value or *key*
  - Supports the operations **Insert()**, **Maximum()**, and **ExtractMax()**

Priority Queue Operations:

Insert(*S, x*) – Inserts element *x* into set *S*, according to its priority
Maximum(*S*) – Returns, but does not remove, the element of *S* with the largest key
Extract-Max(*S*) – Removes and returns the element of *S* with the largest key
Increase-Key(*S, x, k*) – Increases the value of element *x*'s key to the new value *k*

How could we implement these operations using a heap?

Priority Queue Operations:

```
HEAP-MAXIMUM(A)
1  return A[1]
```

```
HEAP-EXTRACT-MAX(A)
1  if heap-size[A] < 1
2      then error "heap underflow"
3  max ← A[1]
4  A[1] ← A[heap-size[A]]
5  heap-size[A] ← heap-size[A] − 1
6  MAX-HEAPIFY(A, 1)
7  return max
```

```
HEAP-INCREASE-KEY(A, i, key)
1  if key < A[i]
2      then error "new key is smaller than current key"
3  A[i] ← key
4  while i > 1 and A[PARENT(i)] < A[i]
5       do exchange A[i] ↔ A[PARENT(i)]
6          i ← PARENT(i)
```

```
MAX-HEAP-INSERT(A, key)
1  heap-size[A] ← heap-size[A] + 1
2  A[heap-size[A]] ← −∞
3  HEAP-INCREASE-KEY(A, heap-size[A], key)
```

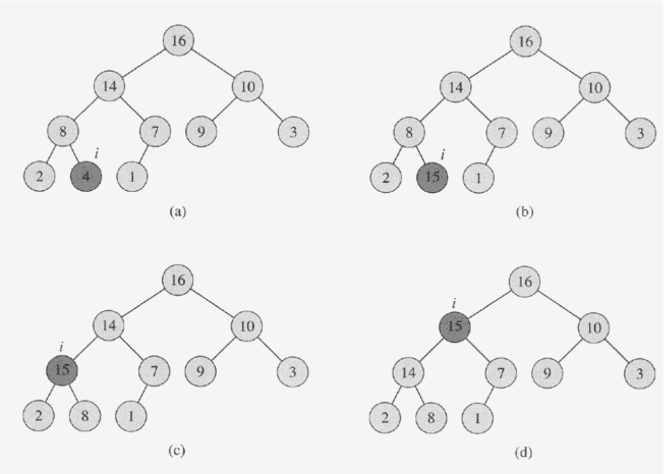


Figure 6.5 The operation of HEAP-INCREASE-KEY. **(a)** The max-heap of Figure 6.4(a) with a node whose index is *i* heavily shaded. **(b)** This node has its key increased to 15. **(c)** After one iteration of the **while** loop of lines 4–6, the node and its parent have exchanged keys, and the index *i* moves up to the parent. **(d)** The max-heap after one more iteration of the **while** loop. At this point, $A[\text{PARENT}(i)] \ge A[i]$. The max-heap property now holds and the procedure terminates.

5.3.6 Heap Sort

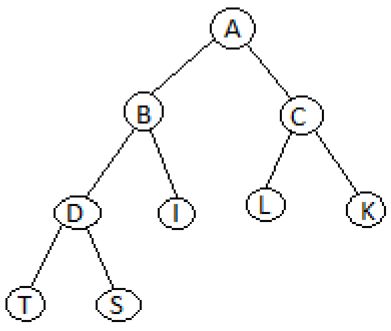
Heap Sort is one of the best sorting methods being in-place and with no quadratic worst-case scenarios. Heap sort algorithm is divided into two basic parts:

- ✓ Creating a Heap of the unsorted list
- ✓ Then a sorted array is created by repeatedly removing the largest/smallest element from the heap, and inserting it into the array. The heap is reconstructed after each removal.

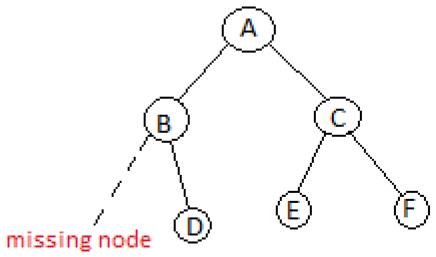
What is a Heap?

Heap is a special tree-based data structure that satisfies the following special heap properties:

1. **Shape Property:** Heap data structure is always a Complete Binary Tree, which means all levels of the tree are fully filled.

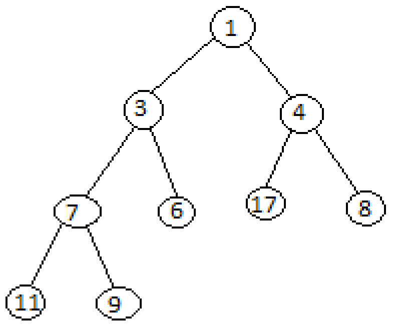


Complete Binary Tree



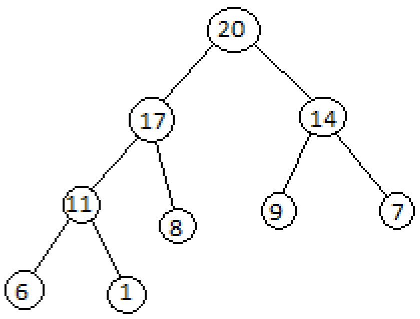
In-Complete Binary Tree

2. **Heap Property:** All nodes are either [greater than or equal to] or [less than or equal to] each of its children. If the parent nodes are greater than their children, heap is called a Max-Heap, and if the parent nodes are smaller than their child nodes, heap is called Min-Heap.



Min-Heap

In min-heap, first element is the smallest. So when we want to sort a list in ascending order, we create a Min-heap from that list, and picks the first element, as it is the smallest, then we repeat the process with remaining elements.



Max-Heap

In max-heap, the first element is the largest, hence it is used when we need to sort a list in descending order.

Initially on receiving an unsorted list, the first step in heap sort is to create a Heap data structure (Max-Heap or Min-Heap). Once heap is built, the first element of the Heap is either largest or smallest (depending upon Max-Heap or Min-Heap), so we put the first element of the heap in our array. Then we again make heap using the remaining elements, to again pick the first element of the heap and put it into the array. We keep on doing the same repeatedly until we have the complete sorted list in our array.

5.3.7 Time & Space Complexity of Sorting Algorithm

Name	Worst Case	Best Case	Average Case	Space
Bubble Sort	O(n^2)	O(n)	O(n^2)	O(1)
Insertion Sort	O(n^2)	O(n)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Quick Sort	O(n^2)	O(n log n)	O(n log n)	O(n log n)
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)
Heap Sort	O(n log n)	O(n log n)	O(n log n)	O(n)

- ✓ Insertion sort is better than bubble sort and selection sort
- ✓ Insertion sort is better for small data set and some sorted data set
- ✓ Merge Sort is better for linked list
- ✓ Heap Sort is very fast and widely used

36) Which sorting algorithm is considered the fastest?

- ✓ There are many types of sorting algorithms: quick sort, bubble sort, balloon sort, radix sort, merge sort, etc. Not one can be considered the fastest because each algorithm is designed for a particular data structure and data set. It would depend on the data set that you would want to sort.

5.5 Stack

Stack is an ordered list of similar data type its a LIFO structure. (Last in First out).

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

5.5.1 Basic features of Stack

- ✓ Stack is an ordered list of similar data type.
- ✓ Stack is a LIFO structure (Last in First out).
- ✓ push() function is used to insert new elements into the Stack and pop() is used to delete an element from the stack. Both insertion and deletion are allowed at only one end of Stack called Top.
- ✓ Stack is said to be in Overflow state when it is completely full and is said to be in Underflow state if it is completely empty.

5.5.2 Applications of Stack

- ✓ The simplest application of a stack is to reverse a word. You push a given word to stack - letter by letter - and then pop letters from the stack.

- ✓ There are other uses also like: Parsing, Expression Conversion (Infix to Postfix, Postfix to Prefix etc.) and many more.

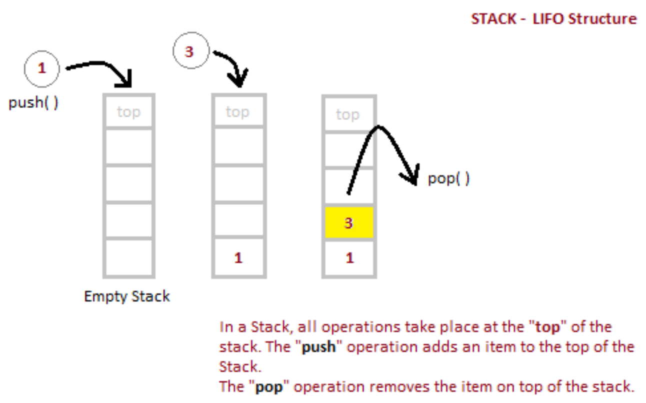
5.5.3 Implementation of Stack

Stack can be easily implemented using an Array or a Linked List. Arrays are quick, but are limited in size and Linked List requires overhead to allocate, link, unlink, and deallocate, but is not limited in size.

5.5.4 Analysis of Stacks

Below mentioned are the time complexities for various operations that can be performed on the Stack data structure.

- ✓ Push Operation: O(1)
- ✓ Pop Operation: O(1)
- ✓ Top Operation: O(1)
- ✓ Search Operation: O(n)


STACK - LIFO Structure
1
push( )
3
top
top
top
pop( )
3
1
1
Empty Stack
In a Stack, all operations take place at the "top" of the stack. The "push" operation adds an item to the top of the Stack.
The "pop" operation removes the item on top of the stack.

What is Stack and where it can be used?

Stack is a linear data structure which the order LIFO(Last In First Out) or FILO(First In Last Out) for accessing elements. Basic operations of stack are : **Push, Pop , Peek**

Applications of Stack:

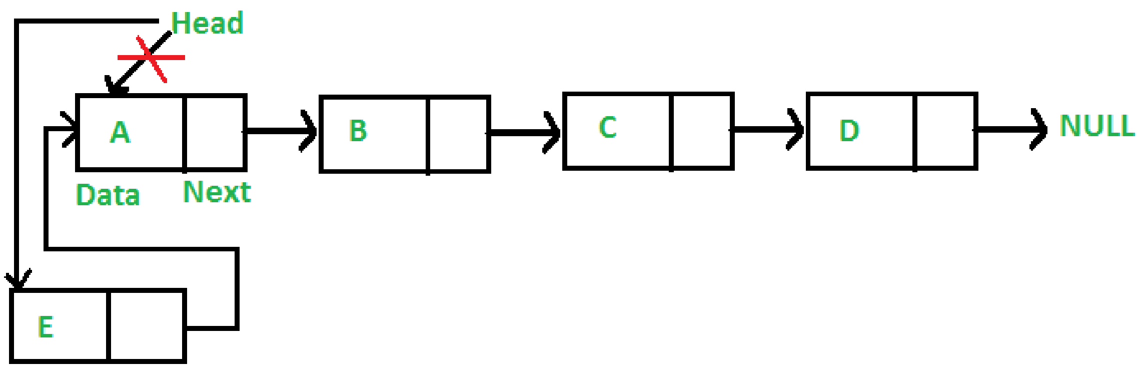
1. Infix to Postfix Conversion using Stack
2. Evaluation of Postfix Expression
3. Reverse a String using Stack
4. Implement two stacks in an array
5. Check for balanced parentheses in an expression

How stack implementation using Linklist:

A node can be added in three ways:

Add a node at the front: (A 4 steps process)

The new node is always added before the head of the given Linked List. And newly added node becomes the new head of the Linked List. For example if the given Linked List is 10->15->20->25 and we add an item 5 at the front, then the Linked List becomes 5->10->15->20->25. Let us call the function that adds at the front of the list is push(). The push() must receive a pointer to the head pointer, because push must change the head pointer to point to the new nod

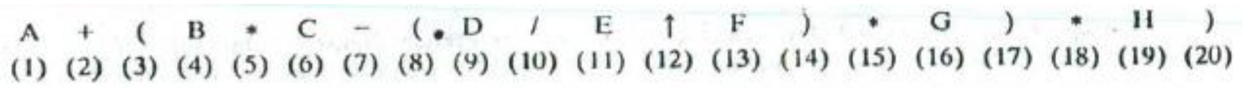


Time complexity of push() is O(1) as it does constant amount of work

Question: Simulate the postfix expression evaluation algorithm using 12,6,7,6,2,+,*,12,4,/,- by showing stack contents as each element is scanned. Exam-2016,2014

Question: Simulate the infix to postfix transformation algorithm for- A+(B*C-(D/E↑F)*G)*H by showing the stack's contents as each element is scanned. Exam-2013

We simulate Algoiithm 6.4 to transform 0 into its equivalent postlix cxprcsion P.
First we push "(" onto STACK, and then we add ")" to the end of Q to obtain:



A	+	(	B	*	C	-	(	D	/	E	↑	F	)	*	G	)	*	H	)
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)	(16)	(17)	(18)	(19)	(20)

The elem ents of 0 have now been labeled from left to right for easy reference. Figure 6-8 shows the status of STACK and of the string I' as each element of 0 is scanned. Observe it:

(I) Each operand is simply added to P and does not change STACK.
(2) The subtraction operator (-) in row 7 sends * front to P before it (-) is pushed onto STACK.
(3) The right parenthesis in row 14 scnds t and then / from STACK to P. and then removes the left parenthesis Ironi he top of STACK.
(4) The right parenthesis in row 20 sends * and then + from STACK to P. and (lien removes the left parenthesis from the top of STACK.

After Step 20 is executed, the STACK is empty and P: ABCI)EF 1/G - 11* +
which is the required postfix equivalent of 0.

Question: Convert the following infix expression to its equivalent prefix and postfix expression.
(i) A*B/C+D↑ (E-F*G)/H
(ii) 1+2*3/4↑ 5*6-7*8 Exam-2016
Question: Convert the following infix expression to its equivalent prefix and postfix expression.
Question: (i) A*B+(C*D/E)*F+(G↑ H)
(ii) 1*2/3+(4-5↑6)+7-8 Exam-2014
Question: Convert the following infix expression to its equivalent prefix and postfix expression
(i) A+B*C/D-E(F/G+H↑K)
(ii) (1+2)↑3/4*5+7-8↑ 9 Exam-2015
Question: Convert the following infix expression into their equivalent prefix and postfix expressions: Exam-2013
(i) A+B*C/(D-E)+F/G*H
(ii)10*B+5/D-(12+F)/50

5.6 Queue

Queue is also an abstract data type or a linear data structure, in which
The first element is inserted from one end called REAR (also called tail), and the deletion of existing element takes place from the other end called as FRONT (also called head).

This makes queue as FIFO data structure, which means that element inserted first will also be removed first.

Dequeue, Front, Rear:
The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added. Both Queues and Stacks can be implemented using Arrays and Linked Lists.

5.6.1 Basic features of Queue

- ✓ Like Stack, Queue is also an ordered list of elements of similar data types.
- ✓ Queue is a FIFO (First in First Out) structure.

- ✓ Once a new element is inserted into the Queue, all the elements inserted before the new element in the queue must be removed, to remove the new element
- ✓ peek( ) function is often used to return the value of first element without dequeuing it

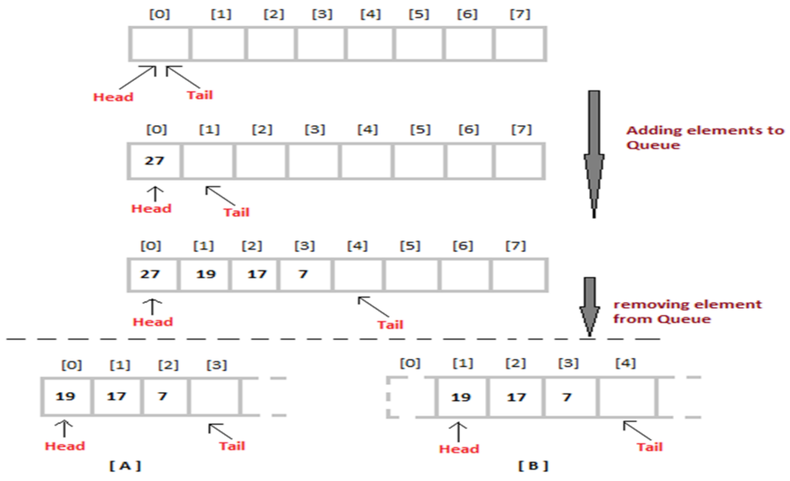
5.6.2 Applications of Queue

Queue, as the name suggests is used whenever we need to have any group of objects in an order in which the first one coming in, also gets out first while the others wait for their turn, like in the following scenarios:

- ✓ Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
- ✓ In real life, Call Center phone systems will use Queues, to hold people calling them in an order, until a service representative is free.
- ✓ Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive, First come first served.

5.6.3 Implementation of Queue

Queue can be implemented using an Array, Stack or Linked List. The easiest way of implementing a queue is by using an Array. Initially the head (FRONT) and the tail (REAR) of the queue points at the first index of the array (starting the index of array from 0).



When we remove element from Queue, we can follow two possible approaches (mentioned [A] and [B] in above diagram). In [A] approach, we remove the element at head position, and then one by one move all the other elements on position forward. In approach [B] we remove the element from head position and then move head to the next position.

5.6.4 Analysis of Queue

- ✓ Enqueue: O(1)
- ✓ Dequeue: O(1)
- ✓ Size: O(1)

<u>**Priority Queue:**</u>

A priority queue is an abstract data type which is like a regular queue or stack data structure, but where additionally each element has a "priority" associated with it. In a priority queue, an element with high priority is served before an element with low priority.

<u>**Important of Priority Queue:**</u>

- 🕮 Priority queues provide extra flexibility over sorting, which is required because jobs often enter the system at arbitrary intervals. It is much more cost-effective to insert a new job into a priority queue than to re-sort everything.
- 🕮 The need to perform certain jobs may vanish before they are executed, meaning that they must be removed from the queue.
- 🕮 A prototype of a priority queue is a timesharing system: programs of high priority are processed first,
- 🕮 The main property of the one-way list representation of a priority queue is that is the clement in the queue that should be processed first always appears at the beginning of the one-way list.
- 🕮 It is a very simple matter to delete and process an clement from our priority queuc. The outline of the algorithm follow

How to implement a stack using queue?

A stack can be implemented using two queues. Let stack to be implemented be 's' and queues used to implement be 'q1' and 'q2'. Stack 's' can be implemented in two ways:

Method 1 (By making push operation costly)
Method 2 (By making pop operation costly) See Implement Stack using Queues

How to implement a queue using stack?

A queue can be implemented using two stacks. Let queue to be implemented be q and stacks used to implement q be stack1 and stack2. q can be implemented in two ways:

Method 1 (By making enQueue operation costly)
Method 2 (By making deQueue operation costly) See Implement Queue using Stacks

Which Data Structure Should be used for implementiong LRU cache?

We use two data structures to implement an LRU Cache.

Queue which is implemented using a doubly linked list. The maximum size of the queue will be equal to the total number of frames available (cache size).The most recently used pages will be near rear end and least recently pages will be near front end.

A Hash with page number as key and address of the corresponding queue node as value. See How to implement LRU caching scheme? What data structures should be used?

How to check if a given Binary Tree is BST or not?
If inorder traversal of a binary tree is sorted, then the binary tree is BST. The idea is to simply do inorder traversal and while traversing keep track of previous key value. If current key value is greater, then continue, else return false. See A program to check if a binary tree is BST or not for more details.

What is a Queue, how it is different from stack and how is it implemented?
Queue is a linear structure which follows the order is **F**irst **I**n **F**irst **O**ut (FIFO) to access elements. Mainly the following are basic operations on queue: **Enqueue, Dequeue**, **Front, Rear**
The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added. Both Queues and Stacks can be implemented using Arrays and Linked Lists.

Array:
34) What are ARRAYs?

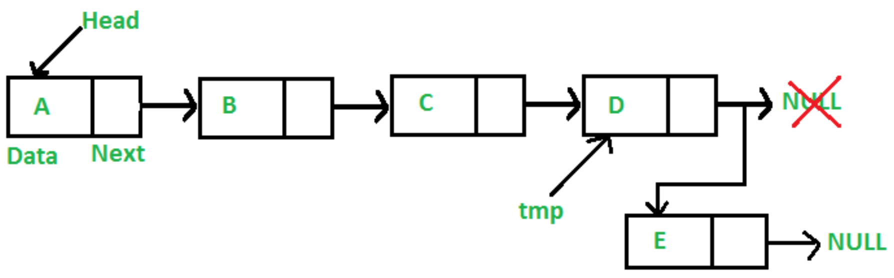
When dealing with arrays, data is stored and retrieved using an index that refers to the element number in the data sequence. This means that data can be accessed in any order. In programming, an array is declared as a variable having a number of indexed elements.
Differentiate STACK from ARRAY.
Stack follows a LIFO pattern. It means that data access follows a sequence wherein the last data to be stored when the first one to be extracted. Arrays, on the other hand, does not follow a particular order and instead can be accessed by referring to the indexed element within the array.

LinkListh:

How implement Queue using Linklist:

Add a node at the end: (6 steps process)

The new node is always added after the last node of the given Linked List. For example if the given Linked List is 5->10->15->20->25 and we add an item 30 at the end, then the Linked List becomes 5->10->15->20->25->30.
Since a Linked List is typically represented by the head of it, we have to traverse the list till end and then change the next of last node to new node.



Delete a node from front: (A 4 steps process)

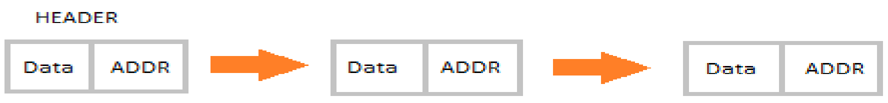
The new node is always added before the head of the given Linked List. And newly added node becomes the new head of the Linked List. For example if the given Linked List is 10->15->20->25 and we add an item 5 at the front, then the Linked List becomes 5->10->15->20->25. Let us call the function that adds at the front of the list is push(). The push() must receive a pointer to the head pointer, because push must change the head pointer to point to the new node

5.7 Linked List

Linked list is defined as a data structure that is formed by linearly collecting data elements but each element is defined as a separate object (node). Each node consists of two items- one is data and the other is a reference to next object.

Linked List is a linear data structure that can store a collection of item and it is very common data structure which consists of group of nodes in a sequence which is divided in two parts.

Each node consists of its own data and the address of the next node and forms a chain. Linked Lists are used to create trees and graphs.



5.7.1 Advantages of Linked List

- ✓ They are a dynamic in nature which allocates the memory when required.
- ✓ Insertion and deletion operations can be easily implemented.
- ✓ Stacks and queues can be easily executed.
- ✓ Linked List reduces the access time.

5.7.2 Disadvantages of Linked Lists

- ✓ The memory is wasted as pointers require extra memory for storage.
- ✓ No element can be accessed randomly; it has to access each node sequentially.
- ✓ Reverse Traversing is difficult in linked list.

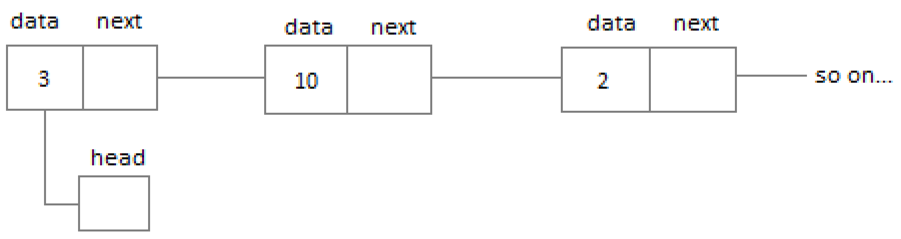
5.7.3 Applications of Linked Lists

- ✓ Linked lists are used to implement stacks, queues, graphs, etc.
- ✓ Linked lists let you insert elements at the beginning and end of the list.

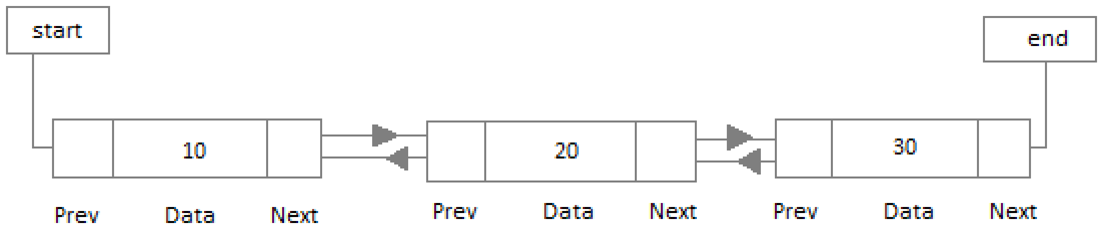
- ✓ In Linked Lists we don't need to know the size in advance.

5.7.4 Types of Linked Lists

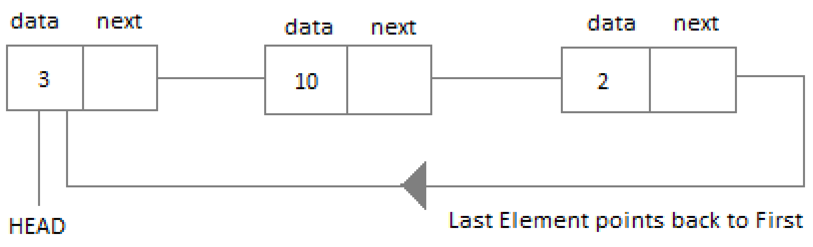
1. **Singly Linked List:** Singly linked lists contain nodes which have a data part as well as an address part i.e. next, which points to the next node in sequence of nodes. The operations we can perform on singly linked lists are insertion, deletion and traversal.



2. **Doubly Linked List:** In a doubly linked list, each node contains two links the first link points to the previous node and the next link points to the next node in the sequence.



3. **Circular Linked List:** In the circular linked list the last node of the list contains the address of the first node and forms a circular chain.



5.7.5 Application of Circular Linked List

- ✓ The real life application where the circular linked list is used is our Personal Computers, where multiple applications are running. All the running applications are kept in a circular linked list and the OS gives a fixed time slot to all for running.
- ✓ The Operating System keeps on iterating over the linked list until all the applications are completed.
- ✓ Another example can be Multiplayer games. All the Players are kept in a Circular Linked List and the pointer keeps on moving forward as a player's chance ends.

Inserting Into a Linked list:

Let LIST be a linked list with successive nodes A and B, as pictured in Fig. 5-14(a). Suppose a node N is to be inserted into the list between nodes A and B.

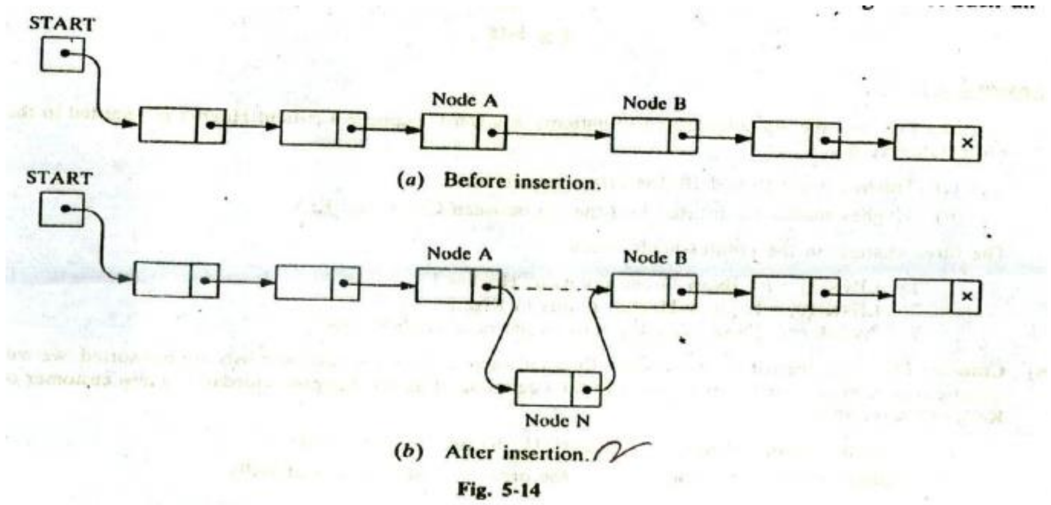


Fig. 5-14

The schematic diagram of such an insertion appears in Fig. 5-14(b). That is, node **A now points to the new node N, and node N points to node B**, to which A previously pointed

- ❖ What type of memory allocation is referred for Linked lists?
    - o Dynamic memory allocation is referred for Linked lists.
- ❖ Mention what is traversal in linked lists?
    - o Term Traversal is used to refer the operation of processing each element in the list.
- ❖ Describe what is Node in link list? And name the types of Linked Lists?
    - o Together (data + link) is referred as the Node.

Types of Linked Lists are,

Singly Linked List
Doubly Linked List
Multiply Linked List
Circular Linked List

Mention what is the difference between Linear Array and Linked List?

Linear Array	**Linked List**
• Deletion and Insertions are difficult.	• Deletion and Insertions can be done easily.
• For insertion and deletion, it needs movements	• For insertion and deletion, it does not require movement of nodes
• In it space is wasted	• In it space is not wasted
• It is expensive	• It is not expensive
• It cannot be reduced or extended according to requirements	• It can be reduced or extended according to requirements
• To avail each element same amount of time is required.	• To avail each element different amount of time is required.
• In consecutive memory locations elements are stored.	• Elements may or may not be stored in consecutive memory locations
• We can reach there directly if we have to go to a particular element	• To reach a particular node, you need to go through all those nodes that come before that node.

1. **Question 1. How To Find Middle Element Of A Singly Linked List In One Pass?**
   Answer :
   You should clarify what does mean by one pass in this question. If Interviewer says that you cannot loop twice and you just have to use

   one loop, then you can use the **two pointer approach to solving this problem. In the two pointer approach, you have two pointers, fast and slow**. In each step, the **fast pointer moves two nodes**, while slow pointer just steps one node. So, **when fast pointer will point to the last node i.e. where next node is null, the slow pointer will be pointing to the middle node of the linked list.**

   tortoise and hare algorithm algorithm
2. **Question 2. How To Check If Linked List Contains Loop In Java? How To Find The Starting Node Of The Loop?**
   Answer :
   This is another interesting linked list problem which can be solved using the two pointer approach discussed in the first question. This is also known as **tortoise and hare algorithm**.

   Basically, you have two pointers fast and slow and they move with different speed i.e. fast moves 2 notes in each iteration and slow moves one node.

   If linked list contains cycle then at some point in time, both fast and slow pointer will meet and point to the same node, if this didn't happen and one of the pointer reaches the end of linked list means linked list doesn't contain any loop.
1. **Question 4. How To Reverse A Singly Linked List Without Recursion In Java?**
   Answer :
   The previously linked list interview question becomes even more challenging when the interviewer asked you to solve the problem without recursion.

   you need to keep reversing links on the node until you reach the end, which will then become new head.
2. **Question 5. How Would You Remove A Node From A Doubly Linked List?**
   In order to remove a node from the doubly linked list**, you need to go through that node and then change the links so that it points to the next node. Removing nodes from head and tail is easy in linked list but removing a node from the middle of the linked list requires you to travel to the node hence take O(n) time**.
3. **Question 6. Write A Program To Convert A Binary Tree Into A Doubly Linked List?**
   Answer :
   This problem is opposite of question 25 where you need to write a program to convert a double linked list to the balanced binary tree. The left and right pointers in nodes of a binary tree will be used as previous and next pointers respectively in converted doubly linked list. The order of nodes in the doubly linked list must be same as Inorder of the given Binary Tree. The first node of Inorder traversal (left most node in the binary tree) must be the head node of the doubly linked list.
4. **Question 7. How To Remove Duplicate Nodes In An Unsorted Linked List?**
   Answer :
   This problem is similar earlier problems related to String and arrays i.e. removing duplicate elements in an array (see) or removing duplicate characters from given String .

   You need to write a program to remove all duplicate nodes from an unsorted linked list in Java. For example if the linked list is 22->21->22->31->41->23->21 then your program should convert the list to 22->21->31->41->23. This question is also given in the famous Cracking the Coding Interview book so you can look at their solution as well.
5. **Question 8. How To Find The Length Of A Singly Linked List In Java?**
   Answer :
   you **can iterate over linked list and keep a count of nodes until you reach the end** of the linked list where next node will be null. The value of the counter is the length of linked list.
6. **Question 10. Write A Program To Print A Linked List In Reverse Order? E.g. Print Linked List From Tail To Head?**
   Answer :
   You can print nodes of linked list in reverse order by using Stack data structure in two steps:

Step 1: Traverse the linked list from the head and put the value of each node into Stack until you reach the last node. This will take O(n) time.
Step 2: Pop the elements out from the stack and print. This will take O(1) time.
Input: 1->2->3
Output: 3 2 1

7. **Question 11. How To Find The Kith Node From The End In A Singly Linked List?**
   Answer :
   - This is one of the tricky but frequently asked linked list questions. Some of you may be wondering how do you find kth node from end, singly linked list can only traverse in one direction and that is forward then how do you count nodes from the end.
   - Well, you don't have to, you can still move forward and count nodes from the end? Actually, that's the trick. You can use two pointers to find the Nth node from the end in a singly linked list. They are known as fast and slow points.
   - You start slow pointer when the fast pointer reaches to the Kth node from start e.g. if you have to find 3rdnode from the end then you start slow pointer when the fast pointer reaches to the 3rd node. This way, when your fast pointer reaches to the end, your slow pointer will be on the 3rd node from the end.
8. **Question 12. How To Delete Alternate Nodes Of A Linked List?**
   Answer :
   You are given a Singly Linked List. Starting from the second node delete all alternate nodes of it. For example, if the given linked list is 1->4->8->10->15 then your function should convert it to 1->8->15.
9. **Question 13. What Is The Difference Between An Array And Linked List In Java?**
   Answer :
   Array stores elements at the **adjacent memory location, while linked list stores them at scattered**, which means searching is easy in an array and difficult in linked list but adding and removing an element from start and end is easy in linked list. See here for more differences between array and linked list.
10. **Question 14. Difference Between Singly And Doubly Linked List In Java?**
    Answer :
    The key difference between a single and double linked list data structure in java is that **singly linked list only contains pointer to next node, which means you can only traverse in one direction i.e. forward, but doubly linked list contains two points, both previous and next nodes, hence you can traverse to both forward and backward direction.**
11. **Question 15. How To Implement A Linked List Using Generics In Java?**
    Answer :
    It's not easy to implement a linked using generics in Java, especially if have not written any parametric or generic class, but it's a good exercise to get familiar with both linked list data structure as well generics in Java.
12. **Question 16. How To Insert A Node At The Beginning Of The List?**
    This involves creating a new node (with the new data, say int 10), making its link point to the current first node pointed to by head (data value 2) and lasting changing head to point to this new node. Simple, right.
13. **Question 17. How To Insert A Node At The End Of The List?**
    Answer :
    This case is a little bit more evolved. If you have a tail pointer, it is as easy as inserting at the beginning of the list. If you do not have a tail pointer, you will have to create the new node, traverse the list till you reach the end (i.e. the next pointer is NULL) and then make that last node's next pointer point to the new node.
14. **Question 18. How Do You Traverse A Linked List In Java?**
    Answer :
    There are multiple ways to traverse a linked list in Java e.g. you can use traditional for, while, or do-while loop and go through the linked list until you reach the end of the linked list. Alternatively, you can use enhanced for loop of Java 1.5 or Iterator to traverse through a linked list in Java. From JDK 8 onwards, you can also use java.util.stream.Stream for traversing a linked list.
15. **Question 19. How Do You Find The Sum Of Two Linked List Using Stack In Java?**
    Answer :
    This is a relatively difficult linked questions when you compare this to reversing a linked list or adding/removing elements from the linked list. In order to calculate the sum of linked list, you calculate the sum of values held at nodes in the same position, for example, you add values at first node on both the

linked list to find the first node of resultant linked list. If the length of both linked list is not same then you only add elements from shorter linked list and just copy values for remaining nodes from the long list.

16. **Question 20. How Do You Convert A Sorted Doubly Linked List To A Balanced Binary Search Tree In Java?**
    Answer :
    You need to write a program to convert a given doubly Linked, which is sorted in ascending order to construct a Balanced Binary Search Tree which has same the values as the given doubly linked list. The challenge is usually increased by putting a restriction to construct the BST in-place i.e. no new node should be allocated for tree conversion)

    Input: A Doubly linked list 10 20 30 40 50 60 70
    Output: A balanced binary search tree BST
```
        40

      /

    20      60

  /         /

 10   30  40   70
```
17. **Question 21. How Do You Calculate The Sum Of Two Linked List Using Recursion In Java?**
    Answer :
    You cannot use the java.util.LinkdList class but you have to write your own linked list implementation in Java to solve this problem.
18. **Question 22. How To Implement Lru Cache In Java Using Linked List?**
    Answer :
    An LRU cache is the cache where you remove least recently used an element when the cache is full or about to fill. It's relatively easy in Java if you are allowed to use one of the Collection class e.g. you can use a LinkedHashMap to implement LRU cache in Java, but you should also prepare how you can use a doubly linked list to create an LRU cache.
19. **Question 23. How Do You Reverse Every Alternate K Nodes Of A Linked List In Java?**
    Answer :
    You have been given a singly linked list and you need to write a function to reverse every alternate k nodes (where k is an input to the function) in an efficient way. You also need to calculate the time and space complexity of your algorithm.

    Example:
    Inputs: 1->2->3->4->5->6->7->8->9->NULL and k = 3
    Output: 3->2->1->4->5->6->9->8->7->NULL.
20. **Question 24. How Do Add Two Numbers Represented Using Linked List In Java?**
    Answer :
    You have given two numbers represented by two linked lists, write a function that returns the sum of these two lists. The sum list is linked list representation of addition of two input numbers. There are two restrictions to solve this problem i.e. you cannot modify the lists and you are not allowed to use explicit extra space. You can use recursion to solve this problem.

    Input:
    First List: 1->2->3 // represents number 123
    Second List: 9->9->9 // represents number 999
    Output:
    Resultant list: 1->1->2->2 // represents number 1122
    That's all about some of the frequently asked linked list based coding questions from Programming Interviews. As I said, linked list is one of the essential data structure and you should have a good command over it, especially if you are preparing for Google or Amazon job interview. Once you solve these problems, you can try solving questions given in Algorithm Design Manual by Steven S. Skiena. They are tougher but can really improve your data structure and algorithm skills.
21. **Question 25. What Is A Linked List?**
    Answer :
    Linked list is an ordered set of data elements, each containing a link to its successor (and typically its predecessor).

22. **Question 26. How Many Pointers Are Required To Implement A Simple Linked List?**
    Answer :
    You can find generally 3 pointers engaged:
    - A head pointer, pointing to the start of the record.
    - A tail pointer, pointing on the last node of the list. The key property in the last node is that its subsequent pointer points to nothing at all (NULL).
    - A pointer in every node, pointing to the next node element.
23. **Question 27. How Many Types Of Linked Lists Are There?**
    Answer :
    Singly linked list, doubly linked list, multiply linked list, Circular Linked list.
24. **Question 28. How To Represent A Linked List Node?**
    Answer :
    The simplest representation of a linked list node is wrapping the data and the link using a typedef structure and giving the structure as a Node pointer that points to the next node. An example representation in C is

```
/*ll stands for linked list*/
typedef struct ll
{
   int data;
   struct ll *next;
} Node;
```

25. **Question 29. Describe The Steps To Insert Data At The Starting Of A Singly Linked List?**
    Answer :
    Inserting data at the beginning of the linked list involves creation of a new node, inserting the new node by assigning the head pointer to the new node next pointer and updating the head pointer to the point the new node. Consider inserting a temp node to the first of list

```
Node *head;
void InsertNodeAtFront(int data)
{
   /* 1. create the new node*/
   Node *temp = new Node;
   temp->data = data;
   /* 2. insert it at the first position*/
   temp->next = head;
   /* 3. update the head to point to this new node*/
   head = temp;
}
```

26. **Question 30. How To Insert A Node At The End Of Linked List?**
    Answer :
    This case is a little bit more complicated. It depends on your implementation. If you have a tail pointer, it's simple. In case you do not have a tail pointer, you will have to traverse the list till you reach the end (i.e. the next pointer is NULL), then create a new node and make that last node's next pointer point to the new node.

```
void InsertNodeAtEnd(int data)
{
  /* 1. create the new node*/
   Node *temp = new Node;
  temp->data = data;
   temp->next = NULL;
   /* check if the list is empty*/
   if (head == NULL)
```

```
    {
        head = temp;
        return;
    }
    else
    {
        /* 2. traverse the list till the end */
        Node *traveller = head;
        while (traveler->next != NULL)
            traveler = traveler->next;
       /* 3. update the last node to point to this new node */
        traveler->next = temp;
    }
}
```

27. **Question 31. How To Insert A Node In Random Location In The List?**

Answer :

As above, you'd initial produce the new node. Currently if the position is one or the list is empty, you'd insert it initially position. Otherwise, you'd traverse the list until either you reach the specified position or the list ends. Then you'd insert this new node. Inserting within the middle is that the difficult case as you has got to make sure you do the pointer assignment within the correct order. First, you'd set the new nodes next pointer to the node before that the new node is being inserted. Then you'd set the node to the position to purpose to the new node. Review the code below to get an idea.

```
void InsertNode(int data, int position)
{
    /* 1. create the new node */
    Node *temp = new Node;
    temp->data = data;
    temp->next = NULL;
    /* check if the position to insert is first or the list is empty */
    if ((position == 1) || (head == NULL))
    {
        // set the new node to point to head
        // as the list may not be empty
        temp->next = head;
        // point head to the first node now
        head = temp;
        return;
    }
    else
    {
        /* 2. traverse to the desired position */
    Node *t = head;
int currPos = 2;
        while ((currPos < position) && (t->next != NULL))
        {
            t = t->next;
            currPos++;
        }
        /* 3. now we are at the desired location */
        /* 4 first set the pointer for the new node */
        temp->next = t->next;
        /* 5 now set the previous node pointer */
        t->next = temp;
    }
}
```

28. **Question 32. How To Delete A Node From Linked List?**
    Answer :
    - Set the head to point to the node that head is pointing to.
    - Traverse to the desired position or till the list ends; whichever comes first
    - You have to point the previous node to the next node.
29. **Question 33. How To Reverse A Singly Linked List?**
    Answer :
    - First, set a pointer (*current) to point to the first node i.e. current=head.
    - Move ahead until current!=null (till the end)
    - set another pointer (*next) to point to the next node i.e. next=current->next
    - store reference of *next in a temporary variable (*result) i.e. current->next=result
    - swap the result value with current i.e. result=current
    - And now swap the current value with next. i.e. current=next
    - return result and repeat from step 2
    - A linked list can also be reversed using recursion which eliminates the use of a temporary variable.
30. **Question 34. Compare Linked Lists And Dynamic Arrays?**
    Answer :
    - A **dynamic array is a data structure that allocates all elements contiguously in memory, and keeps a count of the present number of elements.** If the area reserved for the dynamic array is exceeded, it's reallocated and traced, a costly operation.
    - **Linked lists have many benefits over dynamic arrays. Insertion or deletion of an element at a specific point of a list, is a constant-time operation**, whereas insertion in a dynamic array at random locations would require moving half the elements on the average, and all the elements in the worst case.
    - Whereas one can delete an element from an array in constant time by somehow marking its slot as vacant, this causes fragmentation that impedes the performance of iteration.
31. **Question 35. What Is A Circular Linked List?**
    Answer :
    In the last node of a singly linear list, the link field often contains a null reference. A less common convention is to make the **last node to point to the first node of the list**; in this case the list is said to be 'circular' or 'circularly linked'.
32. **Question 36. What Is The Difference Between Singly And Doubly Linked Lists?**
    Answer :
    A doubly linked list whose nodes contain three fields: an integer value and two links to other nodes one to point to the previous node and other to point to the next node. Whereas a singly linked list contains points only to the next node.
33. **Question 37. What Are The Applications That Use Linked Lists?**
    Answer :
    Both stacks and queues are often implemented using linked lists, other applications are skip list, binary tree, unrolled linked list, hash table, heap, self-organizing list.
34. **Question 38. How To Remove Loops In A Linked List (or) What Are Fast And Slow Pointers Used For?**
    Answer :

The best solution runs in O(N) time and uses O(1) space. This method uses two pointers (one slow pointer and one fast pointer). The slow pointer traverses one node at a time, while the fast pointer traverses twice as fast as the first one. If the linked list has loop in it, eventually the fast and slow pointer will be at the same node. On the other hand, if the list has no loop, obviously the fast pointer will reach the end of list before the slow pointer does. Hence we detect a loop.

35. **Question 39. What Will You Prefer To Use A Singly Or A Doubly Linked Lists For Traversing Through A List Of Elements?**
Answer :
Double-linked lists require more space per node than singly liked lists, and their elementary operations such as insertion, deletion are more expensive; but they are often easier to manipulate because they allow fast and easy sequential access to the list in both directions. On the other hand, doubly linked lists cannot be used as persistent data structures. So, for traversing through a list of node, doubly linked list would be a better choice.

7) Mention what are the applications of Linked Lists?
Applications of Linked Lists are,

- Linked lists are used to implement queues, stacks, graphs, etc.
- In Linked Lists you don’t need to know the size in advance.
- Linked lists let you insert elements at the beginning and end of the list.

8) What does the dummy header in linked list contain?
In linked list, the dummy header contains the first record of the actual data
9) Mention the steps to insert data at the starting of a singly linked list?
Steps to insert data at the starting of a singly linked list include,

- Create a new node
- Insert new node by allocating the head pointer to the new node next pointer
- Updating the head pointer to the point the new node.

```
Node *head;
void InsertNodeAtFront(int data)
{
/* 1. create the new node*/
Node *temp = new Node;
temp->data = data;
/* 2. insert it at the first position*/
temp->next = head;
/* 3. update the head to point to this new node*/
head = temp;
}
```

10) Mention what is the difference between singly and doubly linked lists?
A doubly linked list nodes contain three fields:

- An integer value and
- Two links to other nodes
- one to point to the previous node and
- other to point to the next node.

Whereas a singly linked list contains points only to the next node.
11) Mention what are the applications that use Linked lists?
Both queues and stacks are often implemented using linked lists. Other applications are list, binary tree, skip, unrolled linked list, hash table, etc.
12) Explain how to add an item to the beginning of the list?
To add an item to the beginning of the list, you have to do the following:

- Create a new item and set its value

- Link the new item to point to the head of the list
- Set the head of the list to be our new item

If you are using a function to do this operation, you need to alter the head variable. To do this, you must pass a pointer to the pointer variable (a double pointer). so you will be able to modify the pointer itself.

13) Mention what is the biggest advantage of linked lists?
The biggest benefit of linked lists is that you do not specify a fixed size for your list. The more elements you add to the chain, the bigger the chain gets.

14) Mention how to delete first node from singly linked list?
To delete first node from singly linked list

- Store Current Start in Another Temporary Pointer
- Move Start Pointer One position Ahead
- Delete temp i.e Previous Starting Node as we have Updated Version of Start Pointer

15) Mention how to display Singly Linked List from First to Last?
To display Singly Linked List from First to Last,

- Create a linked list using create().
- You cannot change the address stored inside global variable "start" therefore you have to declare one temporary variable -"temp" of type node
- To traverse from start to end, you should allot address of Starting node in Pointer variable i.e temp.

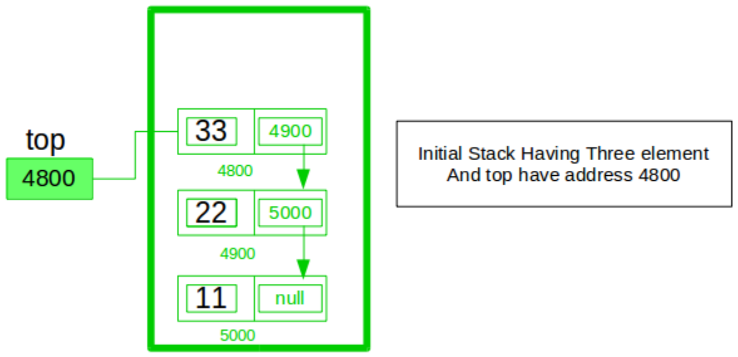
16) Mention how to insert a new node in linked list where free node will be available?
To insert a new node in linked list the free node will be available in Avail list.

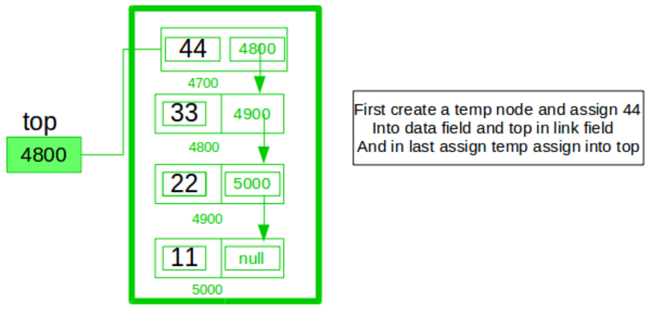
17) Mention for which header list, you will found the last node contains the null pointer?
For grounded header list you will found the last node contains the null pointer.

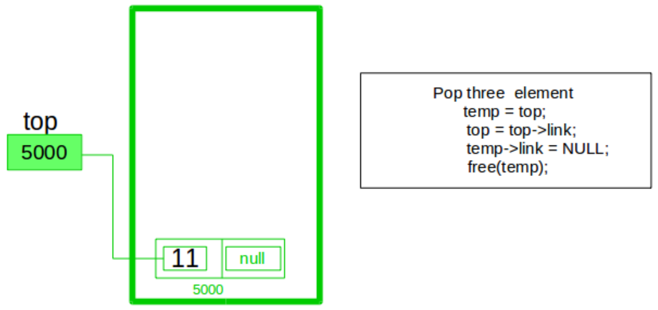
15. How to remove loops in a linked list (or) what are fast and slow pointers used for?
The best solution runs in O(N) time and uses O(1) space. This method uses two pointers (one slow pointer and one fast pointer). The slow pointer traverses one node at a time, while the fast pointer traverses twice as fast as the first one. If the linked list has loop in it, eventually the fast and slow pointer will be at the same node. On the other hand, if the list has no loop, obviously the fast pointer will reach the end of list before the slow pointer does. Hence we detect a loop.

16. What will you prefer to use a singly or a doubly linked lists for traversing through a list of elements?
Double-linked lists require more space per node than singly liked lists, and their elementary operations such as insertion, deletion are more expensive; but they are often easier to manipulate because they allow fast and easy sequential access to the list in both directions. On the other hand, doubly linked lists cannot be used as persistent data structures. So, for traversing through a list of node, doubly linked list would be a better choice.

Implement a stack using singly linked list:
Implement a stack using single linked list concept. all the single linked list operations perform based on Stack operations LIFO(last in first out) and with the help of that knowledge we are going to implement a stack using single linked list. using single linked lists so how to implement here it is linked list means what we are storing the information in the form of nodes and we need to follow the stack rules and we need to implement using single linked list nodes so what are the rules we need to follow in the implementation of a stack a simple rule that is last in first out and all the operations we should perform so with the help of a top variable only with the help of top variables are how to insert the elements let's see




top
4800
44
4800
4700
33
4900
4800
22
5000
4900
11
null
5000
First create a temp node and assign 44
Into data field and top in link field
And in last assign temp assign into top



A stack can be easily implemented through the linked list. In stack Implementation, a stack contains a top pointer. Which is "head" of the stack where pushing and popping items happens at the head of the list. first node have null in link field and second node link have first node address in link field and so on and last node address in "top" pointer.
The main advantage of using linked list over an arrays is that it is possible to implements a stack that can shrink or grow as much as needed. In using array will put a restriction to the maximum capacity of the array which can lead to stack overflow. Here each new node will be dynamically allocate. so overflow is not possible.
Stack Operations:

1. **push()** **:** Insert the element into linked list nothing but which is the top node of Stack.
2. **pop()** **:** Return top element from the Stack and move the top pointer to the second node of linked list or Stack.
3. **peek()**: Return the top element.
4. **display():** Print all element of Stack.

Stack Operations using Linked List

To implement a stack using a linked list, we need to set the following things before implementing actual operations.

- **Step 1 -** Include all the **header files** which are used in the program. And declare all the **user defined functions**.
- **Step 2 -** Define a '**Node**' structure with two members **data** and **next**.
- **Step 3 -** Define a **Node** pointer '**top**' and set it to **NULL**.
- **Step 4 -** Implement the **main** method by displaying Menu with list of operations and make suitable function calls in the **main** method.

push(value) - Inserting an element into the Stack

We can use the following steps to insert a new node into the stack...

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether stack is **Empty** (**top** == **NULL**)
- **Step 3 -** If it is **Empty**, then set **newNode** → **next** = **NULL**.
- **Step 4 -** If it is **Not Empty**, then set **newNode** → **next** = **top**.
- **Step 5 -** Finally, set **top** = **newNode**.

pop() - Deleting an Element from a Stack

We can use the following steps to delete a node from the stack...

- **Step 1 -** Check whether **stack** is **Empty** (**top == NULL**).
- **Step 2 -** If it is **Empty**, then display **"Stack is Empty!!! Deletion is not possible!!!"** and terminate the function
- **Step 3 -** If it is **Not Empty**, then define a **Node** pointer '**temp**' and set it to '**top**'.
- **Step 4 -** Then set '**top** = **top** → **next**'.

- **Step 5 -** Finally, delete '**temp**'. (**free(temp)**).

display() - Displaying stack of elements

We can use the following steps to display the elements (nodes) of a stack...

- **Step 1 -** Check whether stack is **Empty** (**top** == **NULL**).
- **Step 2 -** If it is **Empty**, then display **'Stack is Empty!!!'** and terminate the function.
- **Step 3 -** If it is **Not Empty**, then define a Node pointer **'temp'** and initialize with **top**.
- **Step 4 -** Display '**temp** → **data** --->' and move it to the next node. Repeat the same until **temp** reaches to the first node in the stack. (**temp** → **next** != **NULL**).
- **Step 5 -** Finally! Display '**temp** → **data** ---> **NULL**'.

Implementation of Stack using Linked List | C Programming

```
#include<stdio.h>
#include<conio.h>
struct Node
{
  int data;
  struct Node *next;
}*top = NULL;

void push(int);
void pop();
void display();
void main()
{
  int choice, value;
  clrscr();
  printf("\n:: Stack using Linked List ::\n");
  while(1){
    printf("\n****** MENU ******\n");
    printf("1. Push\n2. Pop\n3. Display\n4. Exit\n");
    printf("Enter your choice: ");
    scanf("%d",&choice);
    switch(choice){
            case 1: printf("Enter the value to be insert: ");
                    scanf("%d", &value);
                    push(value);
                    break;
            case 2: pop(); break;
            case 3: display(); break;
            case 4: exit(0);
            default: printf("\nWrong selection!!! Please try again!!!\n");
    }
  }
}
void push(int value)
{
  struct Node *newNode;
  newNode = (struct Node*)malloc(sizeof(struct Node));
  newNode->data = value;
  if(top == NULL)
    newNode->next = NULL;
  else
    newNode->next = top;
  top = newNode;
  printf("\nInsertion is Success!!!\n");
}
void pop()
{
  if(top == NULL)
```

```
      printf("\nStack is Empty!!!\n");
   else{
      struct Node *temp = top;
      printf("\nDeleted element: %d", temp->data);
      top = temp->next;
      free(temp);
   }
}
void display()
{
   if(top == NULL)
      printf("\nStack is Empty!!!\n");
   else{
      struct Node *temp = top;
      while(temp->next != NULL){
              printf("%d--->",temp->data);
              temp = temp -> next;
      }
      printf("%d--->NULL",temp->data);
   }
}
```

Queue Implementation using Link list:

Queue is a linear data structure which follows FIFO i.e. First-In-First-Out method.

The two ends of a queue are called Front and Rear, where Insertion always takes place at the Rear and the elements are accessed or removed from the Front.

While implementing queues using arrays:
We cannot increase the size of array, if we have more elements to insert than the capacity of array.
If we create a very large array, we will be wasting a lot of memory space.
Therefore if we implement Queue using Linked list we can solve these problems, as in Linked list Nodes are created dynamically as an when required.

So using a linked list we can create a Queue of variable size and depending on our need we can increase or decrease the size of the queue.

Thus instead of using the head pointer with Linked List, we will use two pointers, front and rear to keep track of both ends of the list, so that we can perfrom all the operations in O(1).

```
# Source Code - C++
#include <iostream>
using namespace std;

// Structure of Node.
struct Node
{
int data;

Node *link;
};

Node *front = NULL;
Node *rear = NULL;

//Function to check if queue is empty or not
```

```
bool isempty()
{
 if(front == NULL && rear == NULL)
 return true;
 else
 return false;
}

//function to enter elements in queue
void enqueue ( int value )
{
 Node *ptr = new Node();
 ptr->data= value;
 ptr->link = NULL;

 //If inserting the first element/node
 if( front == NULL )
 {
  front = ptr;
  rear = ptr;
 }
 else
 {
  rear ->link = ptr;
  rear = ptr;
 }
}

//function to delete/remove element from queue
void dequeue ( )
{
 if( isempty() )
 cout<<"Queue is empty\n";
 else
 //only one element/node in queue.
 if( front == rear)
 {
  free(front);
  front = rear = NULL;
 }
 else
 {
  Node *ptr = front;
  front = front->link;
  free(ptr);
 }
}

//function to show the element at front
void showfront( )
{
 if( isempty())
 cout<<"Queue is empty\n";
 else
 cout<<"element at front is:"<<front->data;
}

//function to display queue
```

```
void displayQueue()
{
 if (isempty())
  cout<<"Queue is empty\n";
 else
 {
  Node *ptr = front;
  while( ptr !=NULL)
  {
   cout<<ptr->data<<" ";
   ptr= ptr->link;
  }
 }
}

//Main Function
int main()
{
 int choice, flag=1, value;
 while( flag == 1)
 {
  cout<<"\n1.enqueue 2.dequeue 3.showfront 4.displayQueue 5.exit\n";
  cin>>choice;
  switch (choice)
  {
  case 1: cout<<"Enter Value:\n";
          cin>>value;
          enqueue(value);
          break;
  case 2: dequeue();
          break;
  case 3: showfront();
          break;
  case 4: displayQueue();
          break;
  case 5: flag = 0;
          break;
  }
 }

 return 0;
}
```

5.8 Trees

Unlike Arrays, Linked Lists, Stack and Queues, which are linear data structures, trees are hierarchical data structures.

5.8.1 Applications of trees

- ✓ Manipulate hierarchical data.
- ✓ Make information easy to search (see tree traversal).
- ✓ Manipulate sorted lists of data.
- ✓ As a workflow for compositing digital images for visual effects.
- ✓ Router algorithms
- ✓ Form of a multi-stage decision-making (see business chess).

5.8.2 Binary Tree

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

Properties of Binary Tree

1) The maximum number of nodes at level 'l' of a binary tree is 2l-1
2) Maximum number of nodes in a binary tree of height 'h' is 2h – 1.
3) In a Binary Tree with N nodes, minimum possible height or minimum number of levels is [Log2^(N+1)]
4) A Binary Tree with L leaves has at least [Log2^L] + 1 levels
5) In Binary tree, number of leaf nodes is always one more than nodes with two children

5.8.3 Types

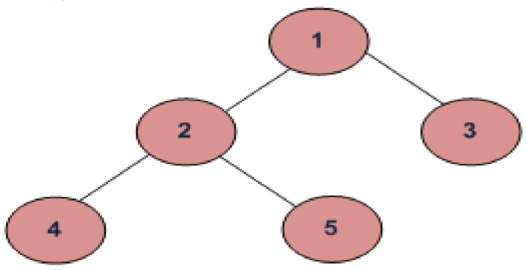
- ✓ **Full Binary Tree:** A Binary Tree is full if every node has 0 or 2 children. Following are examples of full binary tree.
- ✓ **Complete Binary Tree:** A Binary Tree is complete Binary Tree if all levels are completely filled except possibly the last level and the last level has all keys as left 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.
- ✓ **Balanced Binary Tree:** A binary tree is balanced if height of the tree is O(Log n) where n is number of nodes. For Example, AVL tree maintain O(Log n) height by making sure that the difference between heights of left and right subtrees is 1. Red-Black trees maintain O(Log n) height by making sure that the number of Black nodes on every root to leaf paths are same and there are no adjacent red nodes. Balanced Binary Search trees are performance wise good as they provide O(log n) time for search, insert and delete.
- ✓ **A degenerate (or pathological) tree:** A Tree where every internal node has one child. Such trees are performance-wise same as linked list.

How to check if a given Binary Tree is BST or not?

If inorder traversal of a binary tree is sorted, then the binary tree is BST. The idea is to simply do inorder traversal and while traversing keep track of previous key value. If current key value is greater, then continue, else return false. See A program to check if a binary tree is BST or not for more details.

5.8.4 Tree Traversals

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



Depth First Traversals:

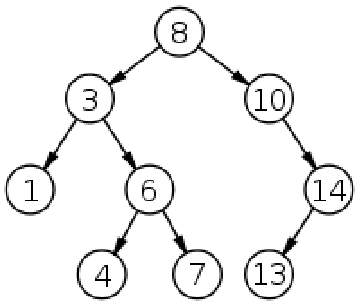
- ✓ Inorder (Left, Root, Right): 4 2 5 1 3
- ✓ Preorder (Root, Left, Right): 1 2 4 5 3
- ✓ Postorder (Left, Right, Root): 4 5 2 3 1

Breadth First or Level Order Traversal: 1 2 3 4 5

5.8.5 Binary Search Tree

Binary Search Tree is a node-based binary tree data structure which has the following properties:

- ✓ The left subtree of a node contains only nodes with keys less than the node's key.
- ✓ The right subtree of a node contains only nodes with keys greater than the node's key.
- ✓ The left and right subtree each must also be a binary search tree.
- ✓ There must be no duplicate nodes.



The above properties of Binary Search Tree provide an ordering among keys so that the operations like search, minimum and maximum can be done fast. If there is no ordering, then we may have to compare every key to search a given key.

Question: For the expression: *+a-bc*-de/fgh draw the tree and perform inorder and postorder traversal. Exam-2015
Answer:

Question: What is binary search tree? Why binary search tree is important ? Exam-2016.,2013

Question: What is binary search tree? Exam-2015
Answer:

Binary Search Tree:

The binary tree T is called a binary search tree (or binary sorted tree) if each node N of has the following property:

- 🕮 The value at N is greater than every value in the left subtree of N and
- 🕮 N is less than every value in the right subtree of N. (It is not difficult to see that this property guarantees that the inorder traversal of T will yield a sorted listing of the elements of T.)

Example of Binary Search Tree:

Consider the binary tree T in Fig. 7-21. T is a binary search tree; that is, every node N in T exceeds every number in its left subtree and is less than every number in its right subtree.

Suppose the 23 were replaced by 35. Then T would still be a binary search tree. On the other hand, suppose the 23 were replaced by 40. Then T would not be a binary search tree, since the 38 would not be greater than the 40 in its left subtree.

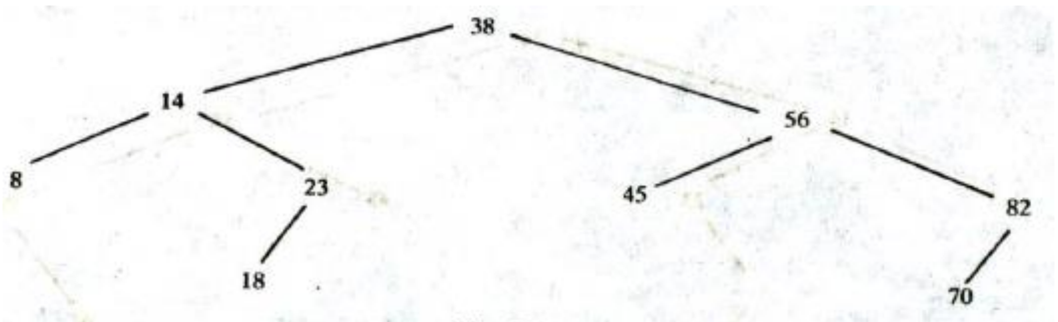


Fig. 7-21

5.9 Heap

A Heap is a Binary Tree with following properties:

- ✓ It's a complete tree (All levels are completely filled except possibly the last level and the last level has all keys as left as possible). This property of Binary Heap makes them suitable to be stored in an array.

- ✓ A Binary Heap is either Min Heap or Max Heap. In a Min Binary Heap, the key at root must be minimum among all keys present in Binary Heap. The same property must be recursively true for all nodes in Binary Tree. Max Binary Heap is similar to Min Heap.

5.9.1 Applications of Heaps:

- ✓ Heap Sort: Heap Sort uses Binary Heap to sort an array in O(nLog(n)) time
- ✓ **Priority Queue:** Priority queues can be efficiently implemented using Binary Heap because it supports insert(), delete() and extractmax(), decreaseKey() operations in O(logn) time. Binomoial Heap and Fibonacci Heap are variations of Binary Heap. These variations perform union also efficiently
- ✓ **Graph Algorithms:** The priority queues are especially used in Graph Algorithms like Dijkstra's Shortest Path and Prim's Minimum Spanning Tree
- ✓ **Many problems can be efficiently solved using Heaps. See following for example**.
  - o K'th Largest Element in an array.
  - o Sort an almost sorted array
  - o Merge K Sorted Arrays.

5.10 Hashing, Hash Data Structure and Hash Table

Hashing is the process of mapping large amount of data item to a smaller table with the help of a hashing function.

The essence of hashing is to facilitate the next level searching method when compared with the linear or binary search. The advantage of this searching method is its efficiency to hand vast amount of data items in a given collection (i.e. collection size). Due to this hashing process, the result is a Hash data structure that can store or retrieve data items in an average time disregard to the collection size.

5.10.1 Hash Function:

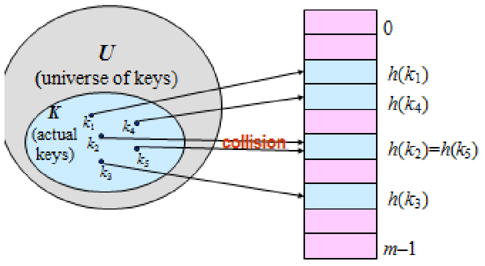
A function that converts a given big phone number to a small practical integer value. The mapped integer value is used as an index in hash table. In simple terms, a hash function maps a big number or string to a small integer that can be used as index in hash table.
A good hash function should have following properties

- ✓ Efficiently computable.
- ✓ Should uniformly distribute the keys (Each table position equally likely for each key)

For example for phone numbers a bad hash function is to take first three digits. A better function is considering last three digits. Please note that this may not be the best hash function. There may be better ways.

5.10.2 Hash Table:

An array that stores pointers to records corresponding to a given phone number. An entry in hash table is NIL if no existing phone number has hash function value equal to the index for the entry.

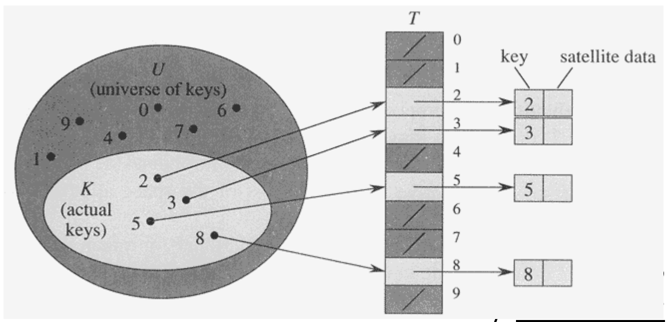


Dictionary:

- ❖ **Dictionary:**
  - ○ Dynamic-set data structure for storing items indexed using *keys*.
  - ○ Supports operations Insert, Search, and Delete.
  - ○ Applications:
    - ▪ Symbol table of a compiler.
    - ▪ Memory-management tables in operating systems.
    - ▪ Large-scale distributed systems.
- ❖ **Hash Tables:**
  - ○ Effective way of implementing dictionaries.
  - ○ Generalization of ordinary arrays.

Direct-Address Tables:

- ❖ Direct-address Tables are ordinary arrays
- ❖ Facilitate direct addressing
  - ○ Element whose key is k is obtained by indexing into the k^{th} position of the array
- ❖ Applicable when we can afford to allocate an array with one position for every possible key
  - ○ i.e. when the universe of keys U is small
- ❖ Dictionary operations can be implemented to take $O(1)$ time



- Suppose:
    - The range of keys is 0..*m*-1
    - Keys are distinct
- The idea:
    - Set up an array T[0..m-1] in which
        - T[*i*] = *x*   if $x \in T$ and key[*x*] = *i*
        - T[*i*] = NULL   otherwise
    - This is called a *direct-address table*
        - Operations take O(1) time **!**
        - ***So what's the problem?***

- Suppose:
    - The range of keys is 0..*m*-1
    - Keys are distinct
- The idea:
    - Set up an array T[0..m-1] in which
        - T[*i*] = *x*   if $x \in T$ and key[*x*] = *i*
        - T[*i*] = NULL   otherwise
    - This is called a *direct-address table*
        - Operations take O(1) time !
        - *So what's the problem?*

- Direct addressing works well when the range *m* of keys is relatively small
- But what if the keys are 32-bit integers?
    - Problem 1: direct-address table will have 2^{32} entries, more than 4 billion
    - Problem 2: even if memory is not an issue, the time to initialize the elements to NULL may be
- Solution: map keys to smaller range 0..*m*-1
- This mapping is called a *hash function*

- Motivation: symbol tables
    - A compiler uses a *symbol table* to relate symbols to associated data
        - Symbols: variable names, procedure names, etc.
        - Associated data: memory location, call graph, etc.
    - For a symbol table (also called a *dictionary*), we care about search, insertion, and deletion
    - We want these to be fast, but don't care about sorted order
- The structure we will use is a *hash table*
    - Supports all the above in O(1) expected time !

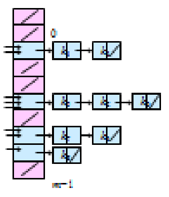
- **Notation:**
    - *U* : Universe of all possible keys.
    - *K* : Set of keys actually stored in the dictionary.
    - $|K| = n$.
- When U is very large,
    - Arrays are not practical.
    - $|K| << |U|$.
- Use a table of size proportional to $|K|$ – The hash tables.
    - However, we lose the direct-addressing ability.
    - Define functions that map keys to slots of the hash table.
- Hash function *h*: Mapping from *U* to the slots of a hash table *T*[0..*m*–1].
    - $h : U \rightarrow \{0,1,\ldots, m–1\}$
- With arrays, key *k* maps to slot *A*[*k*].
- With hash tables, key *k* maps or "hashes" to slot *T*[*h*[*k*]].
- *h*[*k*] is the *hash value* of key *k*.

Issues with Hashing:

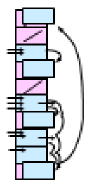
- ❖ Multiple keys can hash to the same slot – collisions are possible.
    - Design hash functions such that collisions are minimized.
    - But avoiding collisions is impossible.
        - Design collision-resolution techniques.
- ❖ Search will cost $\Theta(n)$ time in the worst case.
    - However, all operations can be made to have an expected complexity of $\Theta(1)$.

Methods of Resolution:

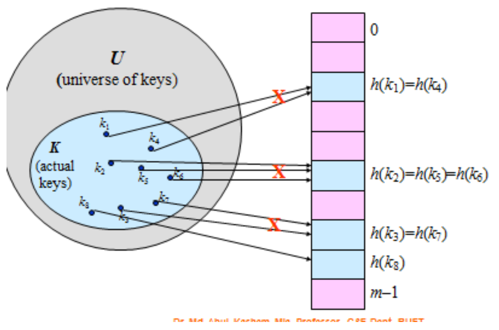
- Chaining:
    - Store all elements that hash to the same slot in a linked list.
    - Store a pointer to the head of the linked list in the hash table slot.



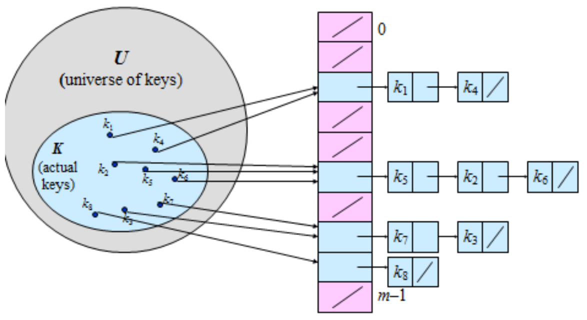
- Open Addressing:
    - All elements are stored in hash table itself.
    - When collisions occur, use a systematic (consistent) procedure to store elements in free slots of the table.



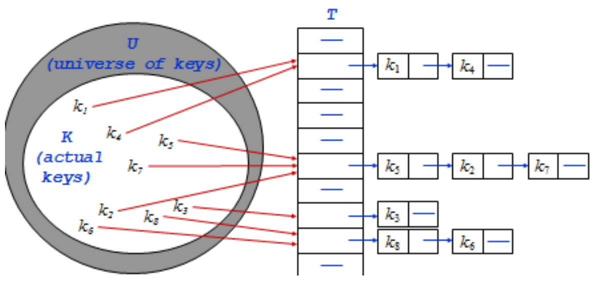
Collision Resolution by changing:



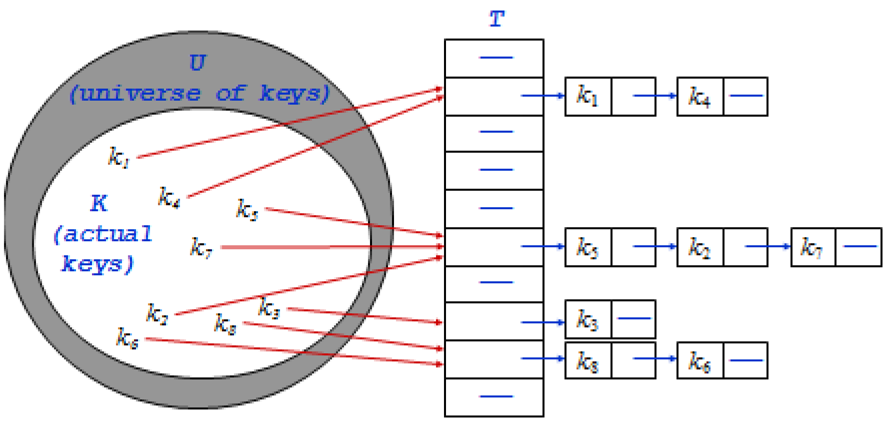
- Chaining puts elements that hash to the same slot in a linked list:



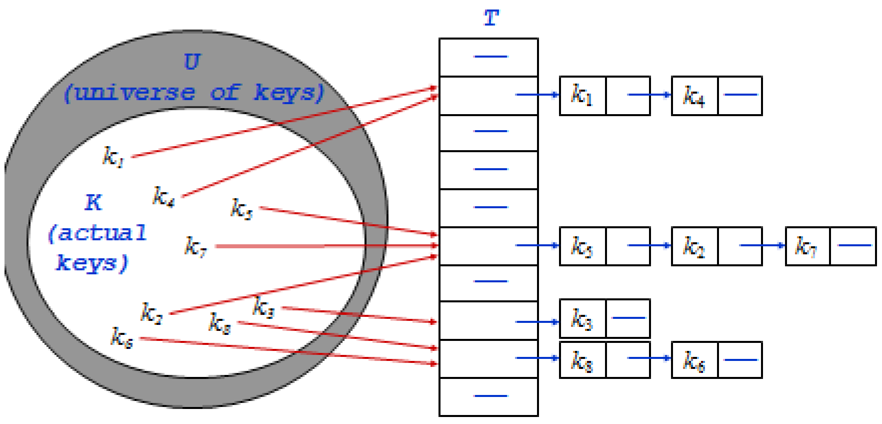
Chaining: ***How do we insert an element?***



How do we delete an element?



How do we search for a element with a given key?



Analysis of Chaining:

- Assume *simple uniform hashing*: each key in table is equally likely to be hashed to any slot
- Given n keys and m slots in the table: the *load factor* $\alpha = n/m$ = average # keys per slot
- *What will be the average cost of an unsuccessful search for a key?*
- A: $O(1+\alpha)$
- *What will be the average cost of a successful search?*   A: $O(1 + \alpha/2) = O(1 + \alpha)$

Draw the 11-item hash table that results from using the hash function $h(k) = (2k + 5) \bmod 11$, to hash the keys 12, 44, 13, 88, 23, 94, 11, 39, 20, 16, and 5, assuming collisions are handled by chaining.

Open Addressing:

- Basic idea:
  - To insert: if slot is full, try another slot, …, until an open slot is found (*probing*)
  - To search, follow same sequence of probes as would be used when inserting the element
    - If reach element with correct key, return it
    - If reach a NULL pointer, element is not in table
- Good for fixed sets (adding but no deletion)
  - Example: spell checking
- Table needn't be much bigger than n

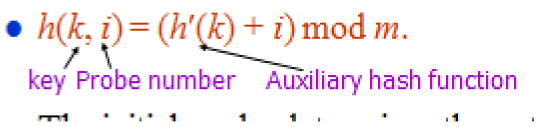
Probe Sequence:

- Sequence of slots examined during a key search constitutes a *probe sequence*.
- Probe sequence must be a permutation of the slot numbers.
  - We examine every slot in the table, if we have to.
  - We don't examine any slot more than once.
- The hash function is extended to:
  - $h : U \times \underbrace{\{0, 1, \dots, m-1\}}_{\text{probe number}} \rightarrow \underbrace{\{0, 1, \dots, m-1\}}_{\text{slot number}}$
- $\langle h(k,0), h(k,1), \dots, h(k,m-1)\rangle$ should be a permutation of $\langle 0, 1, \dots, m-1 \rangle$.

Computing Probe Sequences:

- ❖ The ideal situation is ***uniform hashing***:
  - Generalization of simple uniform hashing.
  - Each key is equally likely to have any of the $m!$ permutations of $\langle 0, 1, \ldots, m-1\rangle$ as its probe sequence.
- ❖ It is hard to implement true uniform hashing.
  - Approximate with techniques that at least guarantee that the probe sequence is a permutation of $\langle 0, 1, \ldots, m-1\rangle$.
- ❖ Some techniques:
  - Use ***auxiliary hash functions***.
    - Linear Probing.
    - Quadratic Probing.
    - Double Hashing.
  - Can't produce all $m!$ probe sequences.

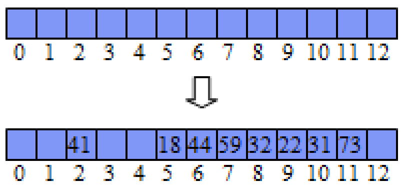
Linear Probing:



- ❖ The initial probe determines the entire probe sequence.
  - $T[h'(k)], T[h'(k)+1], \ldots, T[m-1], T[0], T[1], \ldots, T[h'(k)-1]$
  - Hence, only m distinct probe sequences are possible.
- ❖ Suffers from ***primary clustering***:
  - Long runs of occupied sequences build up.
  - Long runs tend to get longer, since an empty slot preceded by i full slots gets filled next with probability $(i+1)/m$.
  - Hence, average search and insertion times increase.

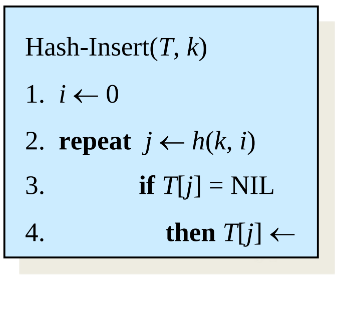
Ex: Linear Probing:

- ❖ Example:
  - $\boldsymbol{h'(k)} = \boldsymbol{k} \bmod 13$
  - $h(k, i) = (\boldsymbol{h'(k)} + i) \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



Operation Insert:

Act as though we were searching, and insert at the first NIL slot found.



```
Hash-Insert(T, k)
1.  i ← 0
2.  repeat  j ← h(k, i)
3.            if T[j] = NIL
4.               then T[j] ←
```

Pseudo-code for Search:

```
Hash-Search (T, k)
1.  i ← 0
2.  repeat  j ← h(k, i)
3.          if T[j] = k
4.             then return j
5.          i ← i + 1
6.  until T[j] = NIL or i = m
7.  return NIL
```

Deletion:

- ❖ Cannot just turn the slot containing the key we want to delete to contain NIL. Why?
- ❖ Use a special value DEL instead of NIL when marking a slot as empty during deletion.
    - ○ ***Search*** should treat DEL as though the slot holds a key that does not match the one being searched for.
    - ○ ***Insert*** should treat DEL as though the slot were empty, so that it can be reused.
- ❖ **Disadvantage:** Search time is no longer dependent on α.
    - ○ Hence, chaining is more common when keys have to be deleted.

Quadratic Probing:

- $h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m \quad c_1 \neq c_2$

key Probe number Auxiliary hash function

- ❖ The initial probe position is $T[h'(k)]$, later probe positions are offset by amounts that depend on a quadratic function of the probe number i.
- ❖ Must constrain c_1, c_2, and m to ensure that we get a full permutation of $\langle 0, 1, \ldots, m-1\rangle$.
- ❖ Can suffer from ***secondary clustering***:
    - ○ If two keys have the same initial probe position, then their probe sequences are the same.

Double Hashing:

- $h(k, i) = (h_1(k) + i\, h_2(k)) \bmod m$

key Probe number Auxiliary hash functions

- ❖ Two auxiliary hash functions.
    - ○ h_1 gives the initial probe. h_2 gives the remaining probes.
- ❖ Must have $h_2(k)$ relatively prime to m, so that the probe sequence is a full permutation of $\langle 0, 1, \ldots, m - 1\rangle$.
    - ○ Choose m to be a power of 2 and have $h_2(k)$ always return an odd number. Or,
    - ○ Let m be prime, and have $1 < h_2(k) < m$.
- ❖ $\Theta(m^2)$ different probe sequences.
    - ○ One for each possible combination of $h_1(k)$ and $h_2(k)$.
    - ○ Close to the ideal uniform hashing.

5.10.3 Collision Handling:

Since a hash function gets us a small number for a big key, there is possibility that two keys result in same value. The situation where a newly inserted key maps to an already occupied slot in hash table is called collision and must be handled using some collision handling technique. Following are the ways to handle collisions:

- ✓ **Chaining:** The idea is to make each cell of hash table point to a linked list of records that have same hash function value. Chaining is simple, but requires additional memory outside the table.
- ✓ **Open Addressing:** In open addressing, all elements are stored in the hash table itself. Each table entry contains either a record or NIL. When searching for an element, we one by one examine table slots until the desired element is found or it is clear that the element is not in the table.

5.10.4 Hash Table Example:

Here, we construct a hash table for storing and retrieving data related to the citizens of a county and the social-security number of citizens are used as the indices of the array implementation (i.e. key). Let's assume that the table size is 12, therefore the hash function would be Value modulus of 12.

Hence, the Hash Function would equate to: (sum of numeric values of the characters in the data item) %12

Let us consider the following social-security numbers and produce a hash code:

- ✓ 120388113D => 1+2+0+3+8+8+1+1+3+13=40
  Hence, (40)%12 => Hashcode=4
- ✓ 310181312E => 3+1+0+1+8+1+3+1+2+14=34
  Hence, (34)%12 => Hashcode=10
- ✓ 041176438A => 0+4+1+1+7+6+4+3+8+10=44
  Hence, (44)%12 => Hashcode=8

Therefore, the Hashtable content would be as follows:

0	
1	
2	
3	
4	SSN:120388113D
5	
6	
7	
8	SSN:041176438A
9	
10	SSN:310181312E
11	

5.11 Graph

a Graph is a non-linear data structure made up of nodes/vertices and edges.
Graph is a data structure that consists of following two components:

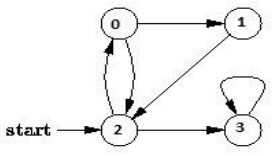
- ✓ A finite set of vertices also called as nodes.
- ✓ A finite set of ordered pair of the form (u, v) called as edge. The pair is ordered because (u, v) is not same as (v, u) in case of directed graph(di-graph). The pair of form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost.

5.11.1 Breadth First Traversal or BFS for a Graph

Breadth First Traversal (or Search) for a graph is similar to Breadth First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again.

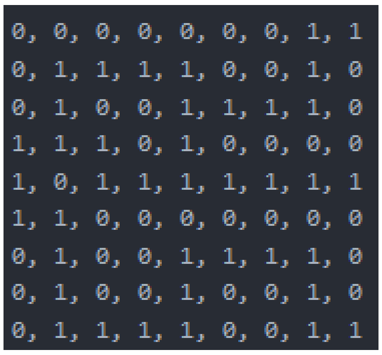
To avoid processing a node more than once, we use a Boolean visited array. For simplicity, it is assumed that all vertices are reachable from the starting vertex.

For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process. A Breadth First Traversal of the following graph is 2, 0, 3, 1.



There are many kinds of graphs, undirected graphs, directed graphs, vertex labeled graphs, cyclic graphs, edge-labeled graphs, weighted graphs etc.
Nonetheless, they are most often used to represent networks be it a city network, city streets, a terrain for AI to pass or social network connections

Find the shortest path in a Maze. The path blocks are represented as 1 and the wall blocks are represented as 0. You can move in four directions (up, down, left, right)



```
0, 0, 0, 0, 0, 0, 0, 1, 1
0, 1, 1, 1, 1, 0, 0, 1, 0
0, 1, 0, 0, 1, 1, 1, 1, 0
1, 1, 1, 0, 1, 0, 0, 0, 0
1, 0, 1, 1, 1, 1, 1, 1, 1
1, 1, 0, 0, 0, 0, 0, 0, 0
0, 1, 0, 0, 1, 1, 1, 1, 0
0, 1, 0, 0, 1, 0, 0, 1, 0
0, 1, 1, 1, 1, 0, 0, 1, 1
```

Explanation

Breadth First Search (BFS for short) is a graph-traversal algorithm, often used for finding the shortest path from the starting node to the goal node.
BFS visits "layer-by-layer". This means that in a Graph like shown below, it first visits all the children of the starting node. These children are treated as the "second layer".

After visiting all the children of the starting node, the algorithm then visits all the children of the already visited children, practically moving to the third layer.

It's important to note that BFS will reach the end of the graph and then calculate the shortest path possible. It guarantees to find the shortest possible path, though it pays the price of not being as memory friendly as some other algorithms.

To accomplish this, BFS uses a `Queue` and this is an important feature of the algorithm. The terminal condition of the algorithm is that the `Queue` is empty, so it's going to go until it visits every single node.

Let's take a look at the pseudo-code behind the implementation:

```
Queue queue
node set visited true
queue.enqueue(node)

while queue not empty
    current = queue.dequeue()
    for v in current children
        if v is not visited
            v set visited true
            queue.enqueue(v)
```

We start off by adding the starting node to the queue. Since the queue is not empty, we set the current node to be the root node, and remove it from the queue.

We then check all of the current node's children and visit them if we haven't visited them before. After that, we add the children to the queue.

Each of these children will become the next `current` node. And since the queue data structure follows the **FIFO** (First in, first out) structure, we visit the rest of the children from the second layer, before continuing to visit the children from the proceeding layers.

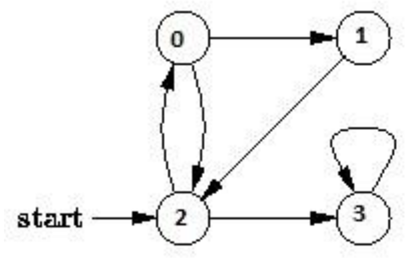
5.11.2 Applications of Breadth First Traversal

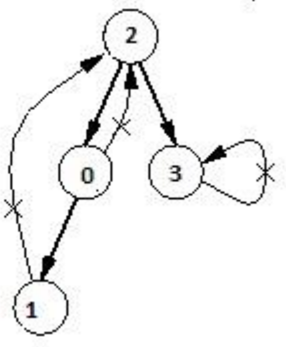
- ✓ Shortest Path and Minimum Spanning Tree for weighted graph
- ✓ Path finder
- ✓ GPS navigation
- ✓ Robotic
- ✓ Peer to Peer Networks
- ✓ Crawlers in Search Engines
- ✓ Social Networking Websites
- ✓ GPS Navigation systems
- ✓ Broadcasting in Network
- ✓ In Garbage Collection
- ✓ Cycle detection in undirected graph
- ✓ Ford–Fulkerson algorithm
- ✓ To test if a graph is Bipartite
- ✓ Path Finding
- ✓ Finding all nodes within one connected component

5.11.3 Depth First Traversal or DFS for a Graph

Depth First Traversal (or Search) for a graph is similar to Depth First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a Boolean visited array.

For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process. A Depth First Traversal of the following graph is 2, 0, 1, 3.





Depth First Search (DFS) is another graph traversal algorithm, similar to Breadth First Search.

DFS searches as far as possible along a branch and then backtracks to search as far as possible in the next branch. This means that in the proceeding Graph, it starts off with the first neighbor, and continues down the line as far as possible.

Once it reaches the final node in that branch (1), it backtracks to the first node where it was faced with a possibility to change course (5) and visits that whole branch, in our case only (2).

Then it backtracks again to the node (5) and since it's already visited nodes (1) and (2), it backtracks to (3) and re-routes to the next branch (8).

The algorithm continues traversing in this fashion until all nodes have been visited and the stack is empty.

DFS doesn't guarantee to find the shortest possible path and can settle for a local optimum, unlike BFS. Thanks to this, it's more memory friendly, though not by a lot.

To accomplish this, DFS uses a `Stack`, which is the main difference between these two algorithms. A `Stack` follows a **LIFO** (Last-in, first-out) structure, which is the reason it goes as far as possible, before taking the other layers into consideration.

You can implement DFS with the help of recursion or iteration. The recursive approach is shorter and simpler, though both approaches are pretty much the same performance-wise.

Let's take a look at the recursive pseudo-code behind the implementation:

```
dfs(node)
node set visited true

for n in node neighbors
    if n is not visited
    dfs(n)
```

We start off by calling the algorithm on the given node and set that it's visited.

We then check all of the node's neighbors, and for each one, run the same algorithm if it isn't visited. This means that before advancing to the next neighbor, it visits all of the children of the first neighbor.

Then goes back to the top, and visits the second neighbor - which advances the search for all of its children, and so on.

You might've noticed the lack of `Stack` in this example and I'd like to address it properly. Both approaches use a `Stack` structure in the background, though in the recursive approach, we don't explicitly define it like so.

Let's take a look at the iterative pseudo-code behind the implementation:

```
dfs(node)
    Stack stack
    node set visited true
    stack.push(node)

    while stack is not empty
        current = stack.pop()
        for n in node neighbors
            if n is not visited
                n set visited true
                stack.push(n)
```

Here, we add the starting node to the `Stack`. Since the stack is not empty, we set the `current` node to be the starting node and take it out of the stack.

We consider all of the neighbors, one by one. The first neighbor gets added to the stack, which makes it the new `current` node in the next iteration. This way, all of the children of this node will be checked before the node next to this one, from the original neighbor list.

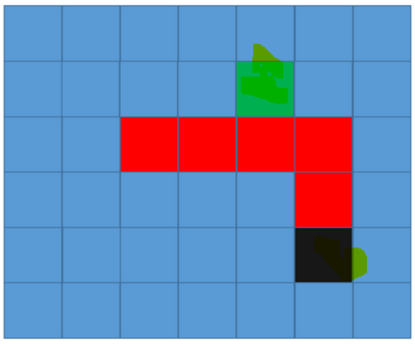
5.11.4 Applications of Depth First Search

- ✓ For an unweighted graph, DFS traversal of the graph produces the minimum spanning tree and all pair shortest path tree.
- ✓ Detecting cycle in a graph
- ✓ Path Finding
- ✓ Topological Sorting
- ✓ To test if a graph is bipartite
- ✓ Finding Strongly Connected Components of a graph
- ✓ Solving puzzles with only one solution, such as mazes

Breadth first search:

A* Search

Given the following graph, where the black node is the starting node, the red nodes are the walls and the green node is the goal node, find the shortest path to the green node:



Explanation

A* Search is quite different than the two we've previously talked about. It's also a graph traversal algorithm, however, it's used to handle real-life pathfinding scenarios.

A* search relies on a knowledge and heuristic cost function for the given node as a way to decide which node it should visit next. In most cases, this cost function is denoted as `f(x)`.

The cost function is calculated as a sum of two other functions:

- `g(x)` - The function that represents the cost of moving from the starting node to a given node.
- `h(x)` - The estimated cost from the given node to the goal node.

We obviously don't know the exact movement cost from the given note to the goal node, which is the reason why `h(x)` is also often referred to as the "heuristic".

The algorithm always picks the node with the **lowest `f(x)` value**. This is logical, considering how the function is calculated.

The fewer steps we take from the starting point combined with how close we get to the goal makes the value of `f(x)` lower if we're going with the shortest path to the goal. Walking away from the goal, and making more steps than needed to get there increases the `f(x)` function.

A Search Applications*

"Where can we use A* Search?

This algorithm is generally used for most of the shortest path problems. It's often used for real-life searching scenarios as well as video games. Most NPCs and AI players rely on A* to intelligently search for a path, fast and efficient.

Shortest Path   Algorithm:

Dijkstra Algorithm:

Dijkstra Algorithm is a notorious graph traversal algorithm for finding the shortest path from a given node/vertex to another.

There are several implementations of this algorithm and some even use different data structures and have different applications. We'll cover the classic one - finding the shortest path between two nodes.

Dijkstra works in a slightly different fashion to the previous algorithms. Instead of starting at the root node, it starts at the goal node and backtracks its way to the starting node.

The algorithm does this by building a set of notes with the minimum distance from the source, dictated by the "weight" of the edges.

It's built with a few variables:

- **dist** - The distance from the goal node (most often referred to as the source node) to every other node in the graph. The `dist` to the source node is 0 since that's where we start. Additionally, the distance to all other nodes is set to infinity. As the algorithm traverses the graph, the distance to all other nodes gets recalculated according to the current node.
- **Q** - Similar to BFS, we need to define a Queue data type and fill it with all of the nodes from the graph. The terminal condition for the algorithm is if the Queue is empty.
- **Set** - We'll also need a set to contain all of the visited nodes, in order to not visit them again. In the end, the set will contain all nodes from the graph.

Dijkstra's algorithm solves the single-source shortest-paths problem on a directed weighted graph G = (V, E), where all the edges are non-negative (i.e., w(u, v) ≥ 0 for each edge (u, v) Є E).

In the following algorithm, we will use one function Extract-Min(), which extracts the node with the smallest key.

Below are the detailed steps used in Dijkstra's algorithm to find the shortest path from a single source vertex to all other vertices in the given graph.

Algorithm

1) Create a set sptSet (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.

2) Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.

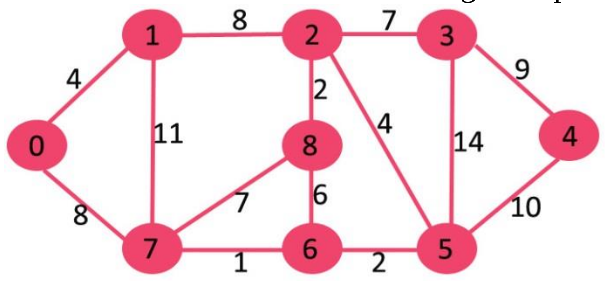
3) While sptSet doesn't include all vertices

….a) Pick a vertex u which is not there in sptSet and has minimum distance value.

….b) Include u to sptSet.

….c) Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

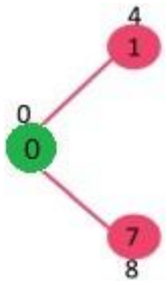
Let us understand with the following example:



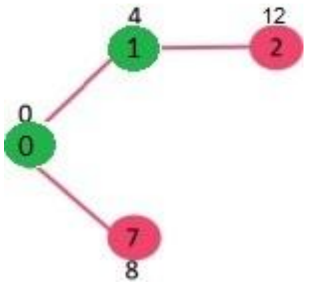
The set *sptSet* is initially empty and distances assigned to vertices are

A={0, INF, INF, INF, INF, INF, INF, INF}

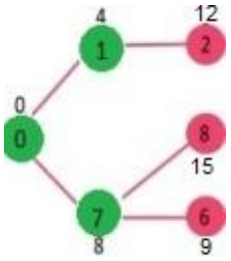
where INF indicates infinite. Now pick the vertex with minimum distance value. The vertex 0 is picked, include it in *sptSet*. So *sptSet* becomes {0}. After including 0 to *sptSet*, update distance values of its adjacent vertices. Adjacent vertices of 0 are 1 and 7. The distance values of 1 and 7 are updated as 4 and 8. Following subgraph shows vertices and their distance values, only the vertices with finite distance values are shown. The vertices included in SPT are shown in green colour.



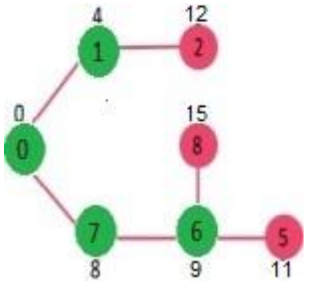
Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). The vertex 1 is picked and added to sptSet. So sptSet now becomes {0, 1}. Update the distance values of adjacent vertices of 1. The distance value of vertex 2 becomes 12.



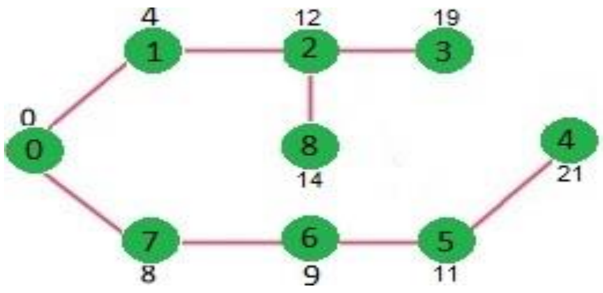
Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 7 is picked. So sptSet now becomes {0, 1, 7}. Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively).



Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 6 is picked. So sptSet now becomes {0, 1, 7, 6}. Update the distance values of adjacent vertices of 6. The distance value of vertex 5 and 8 are updated.



We repeat the above steps until *sptSet* does include all vertices of given graph. Finally, we get the following Shortest Path Tree (SPT).



Analysis

The complexity of this algorithm is fully dependent on the implementation of Extract-Min function. If extract min function is implemented using linear search, the complexity of this algorithm is O(V2 + E).

In this algorithm, if we use min-heap on which Extract-Min() function works to return the node from Q with the smallest key, the complexity of this algorithm can be reduced further**.**

Example

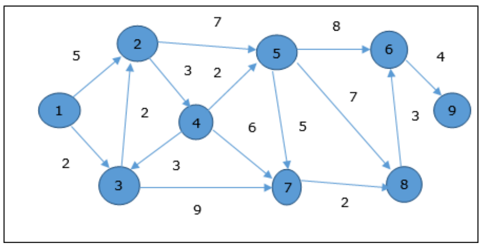
Let us consider vertex ***1*** and ***9*** as the start and destination vertex respectively. Initially, all the vertices except the start vertex are marked by ∞ and the start vertex is marked by ***0***.

Vertex	Initial	Step1 V_1	Step2 V_3	Step3 V_2	Step4 V_4	Step5 V_5	Step6 V_7	Step7 V_8	Step8 V_6
1	0	0	0	0	0	0	0	0	0
2	∞	5	4	4	4	4	4	4	4
3	∞	2	2	2	2	2	2	2	2
4	∞	∞	∞	7	7	7	7	7	7
5	∞	∞	∞	11	9	9	9	9	9
6	∞	∞	∞	∞	∞	17	17	16	16
7	∞	∞	11	11	11	11	11	11	11
8	∞	∞	∞	∞	∞	16	13	13	13
9	∞	∞	∞	∞	∞	∞	∞	∞	20

Hence, the minimum distance of vertex ***9*** from vertex ***1*** is ***20***. And the path is

1→ 3→ 7→ 8→ 6→ 9

This path is determined based on predecessor information.



Notes:

1) The code calculates shortest distance, but doesn't calculate the path information. We can create a parent array, update the parent array when distance is updated (like prim's implementation) and use it show the shortest path from source to different vertices.

2) The code is for undirected graph, same dijkstra function can be used for directed graphs also.

3) The code finds shortest distances from source to all vertices. If we are interested only in shortest distance from the source to a single target, we can break the for the loop when the picked minimum distance vertex is equal to target (Step 3.a of the algorithm).

4) Time Complexity of the implementation is O(V^2). If the input graph is represented using adjacency list, it can be reduced to O(E log V) with the help of binary heap. Please see

Dijkstra's Algorithm for Adjacency List Representation for more details.

5) Dijkstra's algorithm doesn't work for graphs with negative weight edges. For graphs with negative weight edges, Bellman–Ford algorithm can be used, we will soon be discussing it as a separate post.

Application:

This algorithm is generally used for many of the shortest path problems, and it's most definitely a notable algorithm.

Google Maps uses Dijkstra to find the shortest path in navigation. IP Routing also relies heavily exactly on Dijkstra and it has applications in computer networking.

Comparing BFS, DFS, A* and Dijkstra:

"Which one should I use? Which one is the best?"
The answer to this common question is: **all of them**.
Each of these algorithms, although seemingly similar, offer advantages over the other, based on the situation you're implementing them in and your specific case.
Here are some cases where you could be able to choose which one you would use:

Cause	BFS	DFS	A*	Dijkstra
Finding currently alive members in a family tree.	BFS would waste too much time considering levels that shouldn't be considered	In such a case, it would be best to use DFS since mostly the furthest levels are alive	It wouldn't make sense to use Dijkstra or A* for this purpose.	It wouldn't make sense to use Dijkstra or A* for this purpose.
Finding the deceased members in a family tree.	BFS would traverse this graph layer by layer and add all relevant members to the lis	In such a case, using DFS would be less practical than BFS as most of the deceased members would be located near the top.		
Finding the shortest path from point A to point B in a graph/map	Using BFS and DFS here can be very impractical.	Using BFS and DFS here can be very impractical.	In such a case, although a bit controversial, using A* search over Dijkstra is common practice.	BFS assumes that the weight between all nodes is the same, whereas the actual weight/cost could vary. This is where Dijkstra comes into play!
			A* focuses on	Dijkstra takes the cost of

			reaching the goal, and only makes steps that seem promising and reasonable, according to its functions.	moving on an edge into account. If the cost of going straight is bigger than going around it, it will go around. This may translate to congested roads on a map for an example.
			whereas A* only takes the reasonable ones. Because of this, A* is faster than Dijkstra and is often considered "the intelligent searching algorithm".	Dijkstra takes all other nodes into consideration,

Some also refer to A* as the "informed" Dijkstra algorithm. If you take away the heuristic of A* - which means that it chooses the next step only according to the cost so far, you practically get Dijkstra, in reverse though. This makes A* run not greedily towards the goal, but rather in all directions considering every single node in the graph, which again is the same as Dijkstra.
If you're looking for an optimized search and results looking exclusively at the goal - A* is the choice for you. If you're looking for an algorithm to compute the shortest path between the starting point and every single node in the tree, which includes the goal node - Dijkstra is the choice for you.

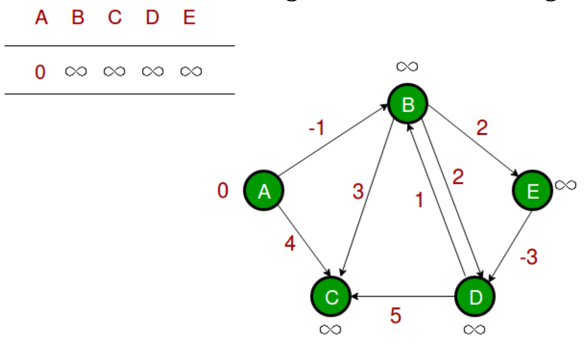
Bellman Ford:
Given a graph and a source vertex src in graph, find shortest paths from src to all vertices in the given graph. The graph may contain negative weight edges.
We have discussed Dijkstra's algorithm for this problem. Dijkstra's algorithm is a Greedy algorithm and time complexity is O(VLogV) (with the use of Fibonacci heap). Dijkstra doesn't work for Graphs with negative weight edges, Bellman-Ford works for such graphs. Bellman-Ford is also simpler than Dijkstra and suites well for distributed systems. But time complexity of Bellman-Ford is O(VE), which is more than Dijkstra.
Recommended: Please solve it on "PRACTICE " first, before moving on to the solution.

Algorithm
Following are the detailed steps.

Input: Graph and a source vertex src
Output: Shortest distance to all vertices from src. If there is a negative weight cycle, then shortest distances are not calculated, negative weight cycle is reported.

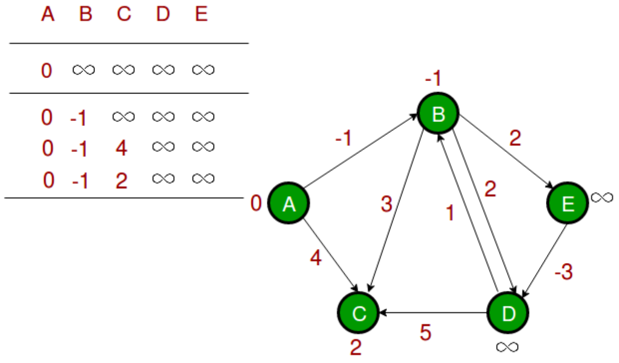
1) This step initializes distances from source to all vertices as infinite and distance to source itself as 0. Create an array dist[] of size |V| with all values as infinite except dist[src] where src is source vertex.
2) This step calculates shortest distances. Do following |V|-1 times where |V| is the number of vertices in given graph.
…..a) Do following for each edge u-v
………………If dist[v] > dist[u] + weight of edge uv, then update dist[v]
………………….dist[v] = dist[u] + weight of edge uv
3) This step reports if there is a negative weight cycle in graph. Do following for each edge u-v
……If dist[v] > dist[u] + weight of edge uv, then "Graph contains negative weight cycle"
The idea of step 3 is, step 2 guarantees shortest distances if graph doesn't contain negative weight cycle. If we iterate through all edges one more time and get a shorter path for any vertex, then there is a negative weight cycle
How does this work? Like other Dynamic Programming Problems, the algorithm calculate shortest paths in bottom-up manner. It first calculates the shortest distances which have at-most one edge in the path. Then, it calculates shortest paths with at-most 2 edges, and so on. After the i-th iteration of outer loop, the shortest paths with at most i edges are calculated. There can be maximum |V| – 1 edges in any simple path, that is why the

outer loop runs |v| – 1 times. The idea is, assuming that there is no negative weight cycle, if we have calculated shortest paths with at most i edges, then an iteration over all edges guarantees to give shortest path with at-most (i+1) edges (Proof is simple, you can refer this or MIT Video Lecture)
Example
Let us understand the algorithm with following example graph. The images are taken from this source.



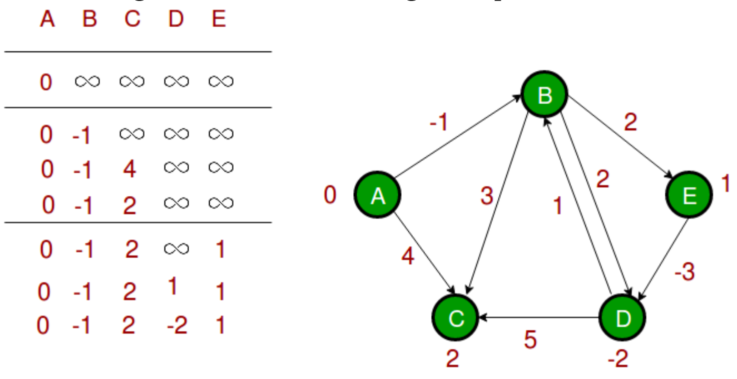
A	B	C	D	E
0	∞	∞	∞	∞

Let all edges are processed in following order: (B, E), (D, B), (B, D), (A, B), (A, C), (D, C), (B, C), (E, D). We get following distances when all edges are processed first time. The first row in shows initial distances. The second row shows distances when edges (B, E), (D, B), (B, D) and (A, B) are processed. The third row shows distances when (A, C) is processed. The fourth row shows when (D, C), (B, C) and (E, D) are processed.



A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞

The first iteration guarantees to give all shortest paths which are at most 1 edge long. We get following distances when all edges are processed second time (The last row shows final values).
et following distances when all edges are processed second time (The last row shows final values).



A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞
0	-1	2	∞	1
0	-1	2	1	1
0	-1	2	-2	1

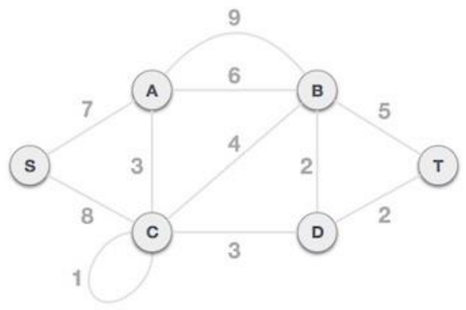
The second iteration guarantees to give all shortest paths which are at most 2 edges long. The algorithm processes all edges 2 more times. The distances are minimized after the second iteration, so third and fourth iterations don't update the distances.
Notes
1) Negative weights are found in various applications of graphs. For example, instead of paying cost for a path, we may get some advantage if we follow the path.
2) Bellman-Ford works better (better than Dijksra's) for distributed systems. Unlike Dijksra's where we need to find minimum value of all vertices, in Bellman-Ford, edges are considered one by one.
Exercise
1) The standard Bellman-Ford algorithm reports shortest path only if there is no negative weight cycles. Modify it so that it reports minimum distances even if there is a negative weight cycle.

2) Can we use Dijksra’s algorithm for shortest paths for graphs with negative weights – one idea can be, calculate the minimum weight value, add a positive value (equal to absolute value of minimum weight value) to all weights and run the Dijksra’s algorithm for the modified graph. Will this algorithm work?
Let the given source vertex be 0. Initialize all distances as infinite, except the distance to source itself. Total number of vertices in the graph is 5, so all edges must be processed 4 times.

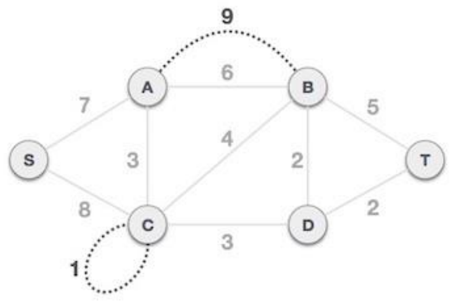
Kruskal Algorithm:

Question 117. What Is A Minimum Spanning Tree?
A minimum spanning tree of an undirected graph G is a tree formed from graph edges that connects all the vertices of G at the lowest total cost.

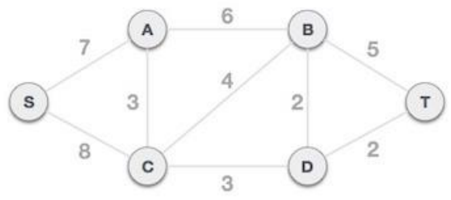
Kruskal's algorithm to find the minimum cost spanning tree uses the greedy approach. This algorithm treats the graph as a forest and every node it has as an individual tree. A tree connects to another only and only if, it has the least cost among all available options and does not violate MST properties.
To understand Kruskal's algorithm let us consider the following example −



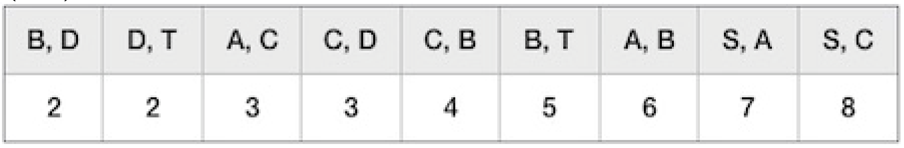
Step 1 - Remove all loops and Parallel Edges
Remove all loops and parallel edges from the given graph.



In case of parallel edges, keep the one which has the least cost associated and remove all others.

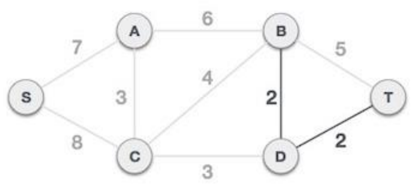


Step 2 - Arrange all edges in their increasing order of weight
The next step is to create a set of edges and weight, and arrange them in an ascending order of weightage (cost).

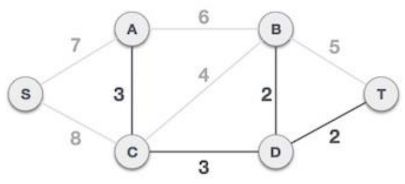


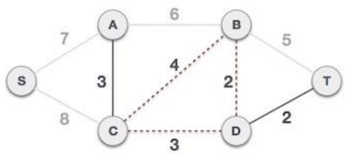
B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8

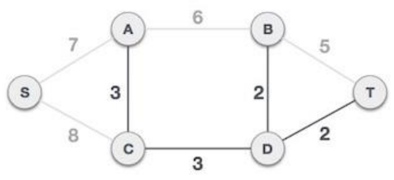
Step 3 - Add the edge which has the least weightage
Now we start adding edges to the graph beginning from the one which has the least weight. Throughout, we shall keep checking that the spanning properties remain intact. In case, by adding one edge, the spanning tree property does not hold then we shall consider not to include the edge in the graph.

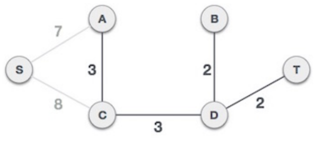


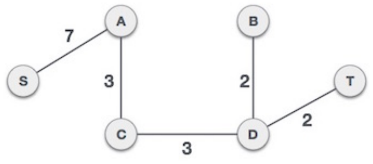
The least cost is 2 and edges involved are B,D and D,T. We add them. Adding them does not violate spanning tree properties, so we continue to our next edge selection.
Next cost is 3, and associated edges are A,C and C,D. We add them again −


Next cost in the table is 4, and we observe that adding it will create a circuit in the graph. −


We ignore it. In the process we shall ignore/avoid all edges that create a circuit.


We observe that edges with cost 5 and 6 also create circuits. We ignore them and move on.


Now we are left with only one node to be added. Between the two least cost edges available 7 and 8, we shall add the edge with cost 7.


By adding edge S,A we have included all the nodes of the graph and we now have minimum cost spanning tree.

What is Minimum Spanning Tree?

Given a connected and undirected graph, a spanning tree of that graph is a subgraph that is a tree and connects all the vertices together. A single graph can have many different spanning trees. A minimum spanning tree (MST) or minimum weight spanning tree 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. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.
How many edges does a minimum spanning tree has?
A minimum spanning tree has (V – 1) edges where V is the number of vertices in the given graph.
What are the applications of Minimum Spanning Tree?
See this for applications of MST.

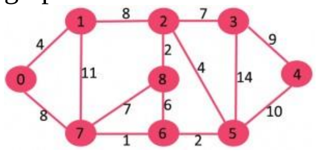
Below are the steps for finding MST using Kruskal's algorithm

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

The step#2 uses Union-Find algorithm to detect cycle. So we recommend to read following post as a prerequisite.

Union-Find Algorithm | Set 1 (Detect Cycle in a Graph)
Union-Find Algorithm | Set 2 (Union By Rank and Path Compression)

The algorithm is a Greedy Algorithm. The Greedy Choice is to pick the smallest weight edge that does not cause a cycle in the MST constructed so far. Let us understand it with an example: Consider the below input graph.



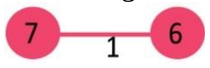
The graph contains 9 vertices and 14 edges. So, the minimum spanning tree formed will be having (9 – 1) = 8 edges.

After sorting:

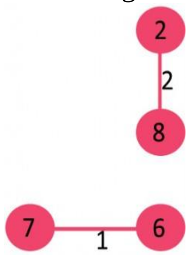
Weight	Src	Dest
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5

Now pick all edges one by one from sorted list of edges

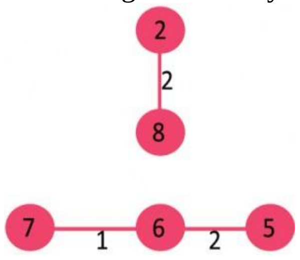
1. Pick edge 7-6: No cycle is formed, include it.



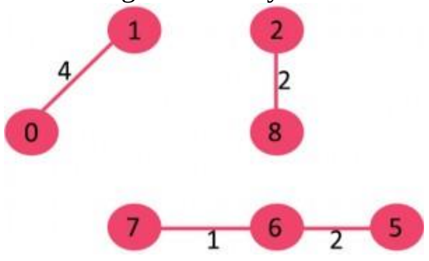
2. Pick edge 8-2: No cycle is formed, include it.



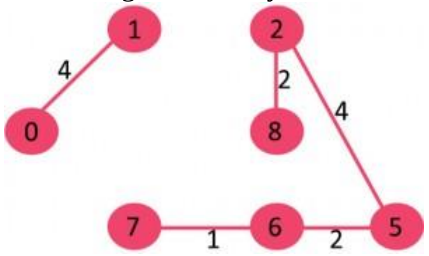
3. Pick edge 6-5: No cycle is formed, include it.



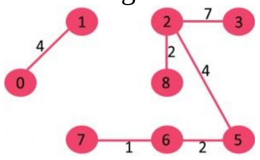
4. Pick edge 0-1: No cycle is formed, include it.



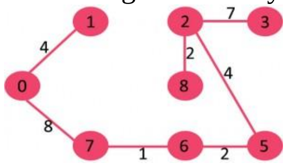
5. Pick edge 2-5: No cycle is formed, include it.



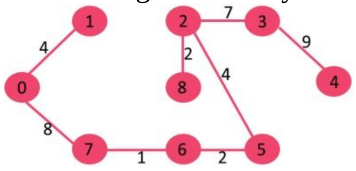
6. Pick edge 8-6: Since including this edge results in cycle, discard it.
7. Pick edge 2-3: No cycle is formed, include it.



8. Pick edge 7-8: Since including this edge results in cycle, discard it.
9. Pick edge 0-7: No cycle is formed, include it.



10. Pick edge 1-2: Since including this edge results in cycle, discard it.
11. Pick edge 3-4: No cycle is formed, include it.



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

A* algorithm:

Coplexity of A* algorithm:

Hilclimbing algorithm:

Simulated annealing algorithm:

Graph Sorting path algorithm:

5.12 Greedy Algorithm

Greedy is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. Greedy algorithms are used for optimization problems.

An optimization problem can be solved using Greedy if the problem has the following property: At every step, we can make a choice that looks best at the moment, and we get the optimal solution of the complete problem.
Example: Kruskal’s Minimum Spanning Tree (MST), Prim’s Minimum Spanning Tree, Dijkstra’s Shortest Path, Huffman Coding

5.13 Dynamic Programming

Dynamic Programming is an algorithmic paradigm that solves a given complex problem by breaking it into subproblems and stores the results of subproblems to avoid computing the same results again.

Following are the two main properties of a problem that suggest that the given problem can be solved using Dynamic programming:

- ✓ Overlapping Subproblems: Dynamic Programming is mainly used when solutions of same subproblems are needed again and again. In dynamic programming, computed solutions to subproblems are stored in a table so that zthese don’t have to recompute.
- ✓ Optimal Substructure: A given problems has Optimal Substructure Property if optimal solution of the given problem can be obtained by using optimal solutions of its subproblems.

Example: Fibonacci Series, Tower of Hanoi, Project Scheduling, Shortest Path by Dijkstra

5.14 Backtracking

Backtracking is an algorithmic paradigm that tries different solutions until finds a solution that “works”. Problems which are typically solved using backtracking technique have following property in common.

These problems can only be solved by trying every possible configuration and each configuration is tried only once. A Naive solution for these problems is to try all configurations and output a configuration that follows given problem constraints. Backtracking works in incremental way and is an optimization over the Naive solution where all possible configurations are generated and tried.
Example: The Knight’s tour problem, Rat in a Maze, N-Queen Problem, Sudoku, Word Break Problem

5.15 Divide and Conquer

Like Greedy and Dynamic Programming, Divide and Conquer is an algorithmic paradigm. A typical Divide and Conquer algorithm solves a problem using following three steps.

- ✓ Divide: Break the given problem into subproblems of same type.
- ✓ Conquer: Recursively solve these subproblems
- ✓ Combine: Appropriately combine the answers

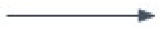
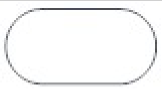
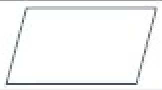
Example: Binary Search, Quicksort, Merge Sort, Closest Pair of Points

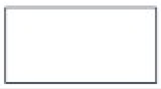
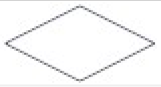
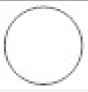
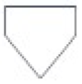
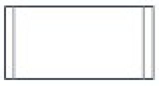
5.16 Flowchart

Flowchart is a diagrammatic representation of an algorithm. Flowchart is very helpful in writing program and explaining program to others.

5.16.1 Symbols Used In Flowchart

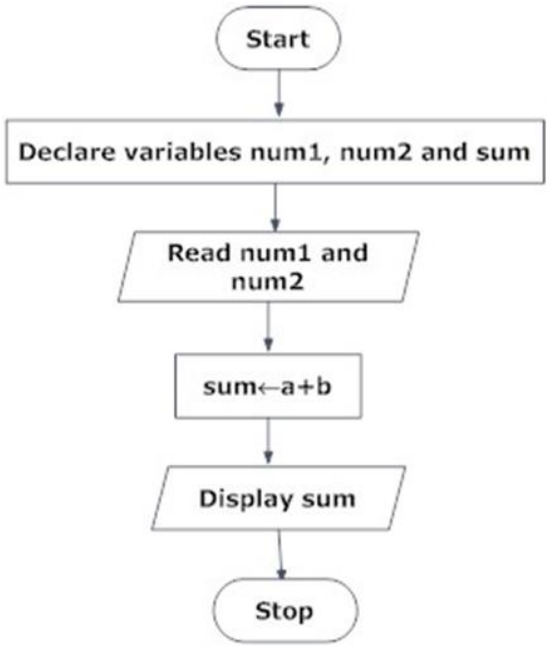
Different symbols are used for different states in flowchart, for example: Input/Output and decision making has different symbols. The table below describes all the symbols that are used in making flowchart

Symbol	**Purpose**	**Description**
	Flow line	Used to indicate the flow of logic by connecting symbols.
	Terminal(Stop/Start)	Used to represent start and end of flowchart.
	Input/output	Used for input and output operation.

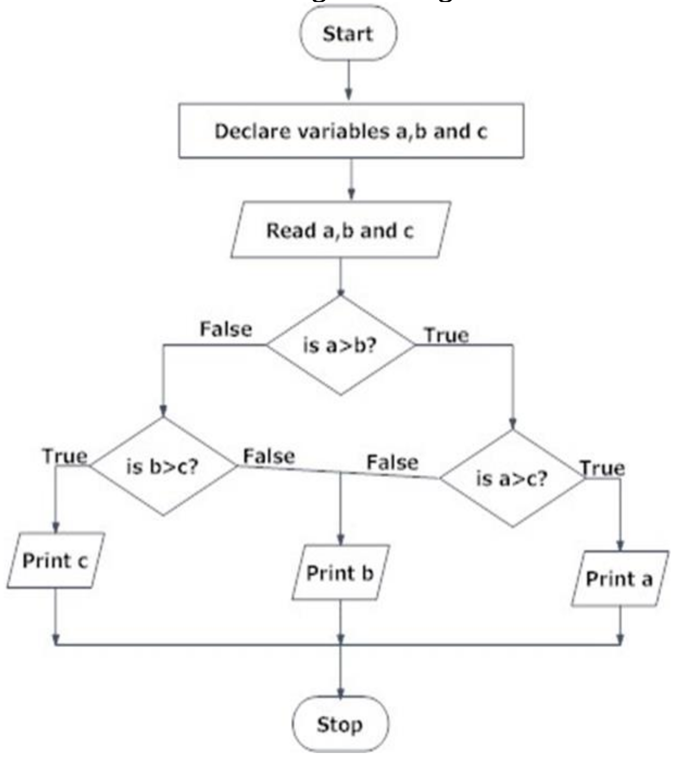
Symbol	Purpose	Description
	Processing	Used for arithmetic operations and data-manipulations.
	Decision	Used to represent the operation in which there are two alternatives, true and false.
	On-page Connector	Used to join different flow line
	Off-page Connector	Used to connect flowchart portion on different page.
	Predefined Process/Function	Used to represent a group of statements performing one processing task.

5.16.2 Example of Flow Chart

Flowchart to add two numbers



Flowchart to find the largest among 3 numbers



5.17 Pseudocode

Pseudocode is a compact and informal high-level description of a program using the conventions of a programming language, but intended more for humans. Pseudocode omits programming level details (like declaration of variables, looping syntax ...) and so it makes things very easy to understand for human being and implement it in any programming language easily.

5.17.1 Pseudo code of Fibonacci Series

```
1. Start
2. Declare variables i, a, b, show
3. Initialize the variables, a=0, b=1, and show =0
4. Enter the number of terms of Fibonacci series to be printed
5. Print First two terms of series
6. Use loop for the following steps
       a. show=a+b
```

    b. a=b
    c. b=show
    d. increase value of i each time by 1
    e. print the value of show
7. End

5.17.2 Infix to Postfix Conversion Algorithm

Let Q be any infix expression and we have to convert it to postfix expression P. For this the following procedure will be followed.

1) Push left parenthesis onto STACK and add right parenthesis at the end of Q.
2) Scan Q from left to right and repeat step 3 to 6 for each element of Q until the STACK is empty.
3) If an operand is encountered add it to P.
4) If a left parenthesis is encountered push it onto the STACK.
5) If an operator is encountered, then
    a. Repeatedly pop from STACK and add to P each operator which has same precedence as or higher precedence than the operator encountered.
    b. Push the encountered operator onto the STACK.
6) If a right parenthesis is encountered, then
    a. Repeatedly pop from the STACK and add to P each operator until a left parenthesis is encountered.
    b. Remove the left parenthesis; do not add it to P.
7) Exit

5.17.3 Write an algorithm to find the factorial of a number

- ✓ Step 1: Start
- ✓ Step 2: Declare variables n, factorial and i
- ✓ Step 3: Initialize variables
    - factorial←1
    - i←1
- ✓ Step 4: Read value of n
- ✓ Step 5: Repeat the steps until i=n
    - 5.1: factorial←factorial*i
    - 5.2: i←i+1
- ✓ Step 6: Display factorial
- ✓ Step 7: Stop

5.17.4 Write an algorithm to add two numbers

- ✓ Step 1: Start
- ✓ Step 2: Declare variables num1, num2 and sum.
- ✓ Step 3: Read values num1 and num2.
- ✓ Step 4: Add num1 and num2 and assign the result to sum.
    - sum←num1+num2
- ✓ Step 5: Display sum
- ✓ Step 6: Stop

5.17.5 Write an algorithm to find the largest among three different numbers

- ✓ Step 1: Start
- ✓ Step 2: Declare variables a,b and c.
- ✓ Step 3: Read variables a,b and c.
- ✓ Step 4:
    - If a>b
        - If a>c
            - Display a is the largest number.
        - Else
            - Display c is the largest number.
    - Else
        - If b>c
            - Display b is the largest number.
        - Else

```
        • Display c is the greatest number.
✓ Step 5: Stop
```

5.17.6 Write an algorithm to find the Fibonacci series till term≤1000

```
✓ Step 1: Start
✓ Step 2: Declare variables first_term,second_term and temp.
✓ Step 3: Initialize variables first_term←0 second_term←1
✓ Step 4: Display first_term and second_term
✓ Step 5: Repeat the steps until second_term≤1000
    o 5.1: temp←second_term
    o 5.2: second_term←second_term+first term
    o 5.3: first_term←temp
    o 5.4: Display second_term
✓ Step 6: Stop
```

5.17.7 Write an algorithm to check whether a number entered by user is prime or not

```
✓ Step 1: Start
✓ Step 2: Declare variables n,i,flag.
✓ Step 3: Initialize variables
    o flag←1
    o i←2
✓ Step 4: Read n from user.
✓ Step 5: Repeat the steps until i<(n/2)
    o 5.1 If remainder of n÷i equals 0
        ▪ flag←0
        ▪ Go to step 6
    o 5.2 i←i+1
✓ Step 6:
    o If flag=0
        ▪ Display n is not prime
    o else
        ▪ Display n is prime
✓ Step 7: Stop
```

5.18 Test Yourself

1. Write Infix to postfix algorithm
2. Draw flow chart to find largest number among three numbers.
3. Write Pseudo code of Fibonacci series.
4. Write an algorithm to find the shortest path from a source node S to destination node D on a given input graph G (V, E, and W). Here V is the set of vertices. E is the set of edges and W is the set of weights associated with edges.
5. Write the collision handling methodology of Hash Data Structure.

19) Differentiate NULL and VOID
Null is a value, whereas Void is a data type identifier. A variable that is given a Null value indicates an empty value. The void is used to identify pointers as having no initial size.
20) What is the primary advantage of a linked list?
A linked list is an ideal data structure because it can be modified easily. This means that editing a linked list works regardless of how many elements are in the list.
21) What is the difference between a PUSH and a POP?
Pushing and popping applies to the way data is stored and retrieved in a stack. A push denotes data being added to it, meaning data is being "pushed" into the stack. On the other hand, a pop denotes data retrieval, and in particular, refers to the topmost data being accessed.
22) What is a linear search?
A linear search refers to the way a target key is being searched in a sequential data structure. In this method, each element in the list is checked and compared against the target key. The process is repeated until found or if the end of the file has been reached.

23) How does variable declaration affect memory allocation?
The amount of memory to be allocated or reserved would depend on the data type of the variable being declared. For example, if a variable is declared to be of integer type, then 32 bits of memory storage will be reserved for that variable.
24) What is the advantage of the heap over a stack?
The heap is more flexible than the stack. That's because memory space for the heap can be dynamically allocated and de-allocated as needed. However, the memory of the heap can at times be slower when compared to that stack.
26) What is Data abstraction?
Data abstraction is a powerful tool for breaking down complex data problems into manageable chunks. This is applied by initially specifying the data objects involved and the operations to be performed on these data objects without being overly concerned with how the data objects will be represented and stored in memory.
27) How do you insert a new item in a binary search tree?
Assuming that the data to be inserted is a unique value (that is, not an existing entry in the tree), check first if the tree is empty. If it's empty, just insert the new item in the root node. If it's not empty, refer to the new item's key. If it's smaller than the root's key, insert it into the root's left subtree, otherwise, insert it into the root's right subtree.
28) How does a selection sort work for an array?
The selection sort is a fairly intuitive sorting algorithm, though not necessarily efficient. In this process, the smallest element is first located and switched with the element at subscript zero, thereby placing the smallest element in the first position.
The smallest element remaining in the subarray is then located next to subscripts 1 through n-1 and switched with the element at subscript 1, thereby placing the second smallest element in the second position. The steps are repeated in the same manner till the last element.
29) How do signed and unsigned numbers affect memory?
In the case of signed numbers, the first bit is used to indicate whether positive or negative, which leaves you with one bit short. With unsigned numbers, you have all bits available for that number. The effect is best seen in the number range (an unsigned 8-bit number has a range 0-255, while the 8-bit signed number has a range -128 to +127.
30) What is the minimum number of nodes that a binary tree can have?
A binary tree can have a minimum of zero nodes, which occurs when the nodes have NULL values. Furthermore, a binary tree can also have 1 or 2 nodes.
31) What are dynamic data structures?
Dynamic data structures are structures that expand and contract as a program runs. It provides a flexible means of manipulating data because it can adjust according to the size of the data.
32) In what data structures are pointers applied?
Pointers that are used in linked list have various applications in the data structure. Data structures that make use of this concept include the Stack, Queue, Linked List and Binary Tree.
33) Do all declaration statements result in a fixed reservation in memory?
Most declarations do, with the exemption of pointers. Pointer declaration does not allocate memory for data, but for the address of the pointer variable. Actual memory allocation for the data comes during run-time.

38) Give a basic algorithm for searching a binary search tree.

1. if the tree is empty, then the target is not in the tree, end search
2. if the tree is not empty, the target is in the tree
3. check if the target is in the root item
4. if a target is not in the root item, check if a target is smaller than the root's value
5. if a target is smaller than the root's value, search the left subtree
6. else, search the right subtree

39) What is a dequeue?
A dequeue is a double-ended queue. This is a structure wherein elements can be inserted or removed from either end.
40) What is a bubble sort and how do you perform it?
A bubble sort is one sorting technique that can be applied to data structures such as an array. It works by comparing adjacent elements and exchanges their values if they are out of order. This method lets the smaller values "bubble" to the top of the list, while the larger value sinks to the bottom.
41) What are the parts of a linked list?

A linked list typically has two parts: the head and the tail. Between the head and tail lie the actual nodes. All these nodes are linked sequentially.

42) How does selection sort work?

Selection sort works by picking the smallest number from the list and placing it at the front. This process is repeated for the second position towards the end of the list. It is the simplest sort algorithm.

43) What is a graph?

A graph is one type of data structure that contains a set of ordered pairs. These ordered pairs are also referred to as edges or arcs and are used to connect nodes where data can be stored and retrieved.

44) Differentiate linear from a nonlinear data structure.

The linear data structure is a structure wherein data elements are adjacent to each other. Examples of linear data structure include arrays, linked lists, stacks, and queues. On the other hand, a non-linear data structure is a structure wherein each data element can connect to more than two adjacent data elements. Examples of nonlinear data structure include trees and graphs.

45) What is an AVL tree?

An AVL tree is a type of binary search tree that is always in a state of partially balanced. The balance is measured as a difference between the heights of the subtrees from the root. This self-balancing tree was known to be the first data structure to be designed as such.

46) What are doubly linked lists?

Doubly linked lists are a special type of linked list wherein traversal across the data elements can be done in both directions. This is made possible by having two links in every node, one that links to the next node and another one that connects to the previous node.

47) What is Huffman’s algorithm?

Huffman’s algorithm is used for creating extended binary trees that have minimum weighted path lengths from the given weights. It makes use of a table that contains the frequency of occurrence for each data element.

48) What is Fibonacci search?

Fibonacci search is a search algorithm that applies to a sorted array. It makes use of a divide-and-conquer approach that can significantly reduce the time needed in order to reach the target element.

49) Briefly explain recursive algorithm.

Recursive algorithm targets a problem by dividing it into smaller, manageable sub-problems. The output of one recursion after processing one sub-problem becomes the input to the next recursive process.

50) How do you search for a target key in a linked list?

To find the target key in a linked list, you have to apply sequential search. Each node is traversed and compared with the target key, and if it is different, then it follows the link to the next node. This traversal continues until either the target key is found or if the last node is reached.

Question 8. What Actions Are Performed When A Function Returns?

Answer :

i) Return address is retrieved.
ii) Function’s data area is freed.
iii) Branch is taken to the return address.

1. **Question 6. What Is Sequential Search?**
   Answer :
   In sequential search each item in the array is compared with the item being searched until a match occurs. It is applicable to a table organized either as an array or as a linked list.
2. **Question 7. What Actions Are Performed When A Function Is Called?**
   Answer :
   When a function is called
   i) arguments are passed.
   ii) local variables are allocated and initialized.
   ii) transferring control to the function.

Question 69. What Is The Difference Between Null And Void Pointer?

Answer :

NULL can be value for pointer type variables.
VOID is a type identifier which has not size.
NULL and void are not same. Example: void* ptr = NULL;

1. **Question 81. List Some Of The Static Data Structures In C?**
   Answer :
   Some of the static data structures in C are arrays, pointers, structures etc.
2. **Question 82. Define Dynamic Data Structures?**

Answer :
A data structure formed when the number of data items are not known in advance is known as dynamic data structure or variable size data structure.

3. **Question 83. List Some Of The Dynamic Data Structures In C?**
Answer :
Some of the dynamic data structures in C are linked lists, stacks, queues, trees etc.

1. **uestion 95. State The Different Ways Of Representing Expressions?**
Answer :
The different ways of representing expressions are
   - Infix Notation.
   - Prefix Notation.
   - Postfix Notation.
2. **Question 96. State The Advantages Of Using Infix Notations?**
Answer :
   - It is the mathematical way of representing the expression.
   - It is easier to see visually which operation is done from first to last.
3. **Question 97. State The Advantages Of Using Postfix Notations?**
Answer :
   - Need not worry about the rules of precedence.
   - Need not worry about the rules for right to left associativity.
   - Need not need parenthesis to override the above rules.

Question 116. Name Two Algorithms Two Find Minimum Spanning Tree?
Answer :
- Kruskal's algorithm.
- Prim's algorithm.

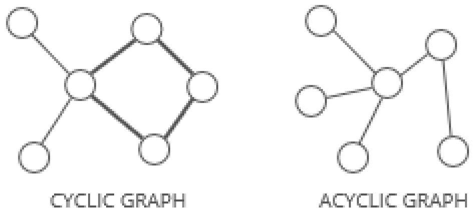
Question 128. What Is A Simple Graph?
Answer :
A simple graph is a graph, which has not more than one edge between a pair of nodes than such a graph is called a simple graph.

Directed or undirected

In **directed** graphs, edges point from the node at one end to the node at the other end. In **undirected** graphs, the edges simply connect the nodes at each end.

Cyclic or acyclic

A graph is **cyclic** if it has a cycle—an unbroken series of nodes with no repeating nodes or edges that connects back to itself. Graphs without cycles are **acyclic**.

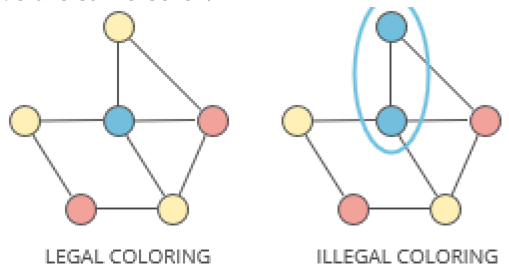


Weighted or unweighted

If a graph is **weighted**, each edge has a "weight." The weight could, for example, represent the distance between two locations, or the cost or time it takes to travel between the locations

Legal coloring

A **graph coloring** is when you assign colors to each node in a graph. A **legal coloring** means no adjacent nodes have the same color:



Adjacency matrix

A matrix of 0s and 1s indicating whether node x connects to node y (0 means no, 1 means yes).

Adjacency list

A list where the index represents the node and the value at that index is a list of the node's neighbors:

Edge list

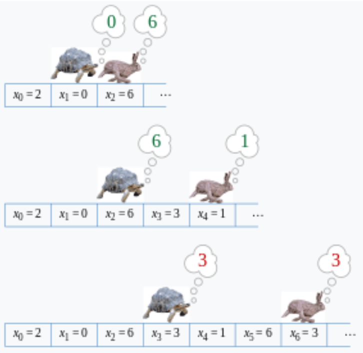
A list of all the edges in the grap

Advanced graph algorithms

If you have lots of time before your interview, these advanced graph algorithms pop up occasionally:

- **Dijkstra's Algorithm:** Finds the shortest path from one node to all other nodes in a *weighted* graph.
- **Topological Sort:** Arranges the nodes in a *directed*, *acyclic* graph in a special order based on incoming edges.
- **Minimum Spanning Tree:** Finds the cheapest set of edges needed to reach all nodes in a *weighted* graph.

Floyd's Tortoise and Hare[edit]


0
6
$x_0=2$ $x_1=0$ $x_2=6$ ...
6
1
$x_0=2$ $x_1=0$ $x_2=6$ $x_3=3$ $x_4=1$ ...
3
3
$x_0=2$ $x_1=0$ $x_2=6$ $x_3=3$ $x_4=1$ $x_5=6$ $x_6=3$ ...

Floyd's "tortoise and hare" cycle detection algorithm, applied to the sequence 2, 0, 6, 3, 1, 6, 3, 1, ...

Floyd's cycle-finding algorithm is a pointer algorithm that uses only two pointers, which move through the sequence at different speeds. It is also called the "tortoise and the hare algorithm", alluding to Aesop's fable of The Tortoise and the Hare.

The algorithm is named after Robert W. Floyd, who was credited with its invention by Donald Knuth.[3][4] However, the algorithm does not appear in Floyd's published work, and this may be a misattribution: Floyd describes algorithms for listing all simple cycles in a directed graph in a 1967 paper,[5] but this paper does not describe the cycle-finding problem in functional graphs that is the subject of this article. In fact, Knuth's statement (in 1969), attributing it to Floyd, without citation, is the first known appearance in print, and it thus may be a folk theorem, not attributable to a single individual.[6]

The key insight in the algorithm is as follows. If there is a cycle, then, for any integers $i \geq \mu$ and $k \geq 0$, $x_i = x_{i + k\lambda}$, where λ is the length of the loop to be found and μ is the index of the first element of the cycle. Based on this, it can then be shown that $i = k\lambda \geq \mu$ for some k if and only if $x_i = x_{2i}$. Thus, the algorithm only needs to check for repeated values of this special form, one twice as far from the start of the sequence as the other, to find a period v of a repetition that is a multiple of λ. Once v is found, the algorithm retraces the sequence from its start to find the first repeated value x_μ in the sequence, using the fact that λ divides v and therefore that $x_\mu = x_{\mu + v}$. Finally, once the value of μ is known it is trivial to find the length λ of the shortest repeating cycle, by searching for the first position $\mu + \lambda$ for which $x_{\mu + \lambda} = x_\mu$.

The algorithm thus maintains two pointers into the given sequence, one (the tortoise) at x_i, and the other (the hare) at x_{2i}. At each step of the algorithm, it increases i by one, moving the tortoise one step forward and the hare two steps forward in the sequence, and then compares the sequence values at these two pointers. The smallest value of $i > 0$ for which the tortoise and hare point to equal values is the desired value v.

The following Python code shows how this idea may be implemented as an algorithm.

```
def floyd(f, x0):
```

```
# Main phase of algorithm: finding a repetition x_i = x_2i.
# The hare moves twice as quickly as the tortoise and
# the distance between them increases by 1 at each step.
# Eventually they will both be inside the cycle and then,
# at some point, the distance between them will be
# divisible by the period λ.
tortoise = f(x0) # f(x0) is the element/node next to x0.
hare = f(f(x0))
while tortoise != hare:
    tortoise = f(tortoise)
    hare = f(f(hare))

# At this point the tortoise position, v, which is also equal
# to the distance between hare and tortoise, is divisible by
# the period λ. So hare moving in circle one step at a time,
# and tortoise (reset to x0) moving towards the circle, will
# intersect at the beginning of the circle. Because the
# distance between them is constant at 2v, a multiple of λ,
# they will agree as soon as the tortoise reaches index μ.

# Find the position μ of first repetition.
mu = 0
tortoise = x0
while tortoise != hare:
    tortoise = f(tortoise)
    hare = f(hare)   # Hare and tortoise move at same speed
    mu += 1

# Find the length of the shortest cycle starting from x_μ
# The hare moves one step at a time while tortoise is still.
# lam is incremented until λ is found.
lam = 1
hare = f(tortoise)
while tortoise != hare:
    hare = f(hare)
    lam += 1

     return lam, mu
```