

UNIT 4 & 5

Programming Methodology and Data Structure

डेटा संरचना (Data Structure)

परिभाषा (Definition):

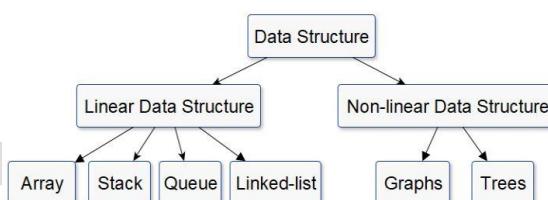
डेटा संरचना एक ऐसी प्रणाली है जो कंप्यूटर मेमोरी में डेटा को व्यवस्थित (organize), संग्रहित (store), और प्रबंधित (manage) करने का तरीका प्रदान करती है, ताकि डेटा पर प्रभावी ढंग से कार्य (processing) किया जा सके।

दूसरे शब्दों में, डेटा संरचना वह तरीका है जिससे हम डेटा को इस प्रकार व्यवस्थित करते हैं कि उस पर संचालन जैसे – खोज (search), जोड़ना (insert), हटाना (delete), और संशोधन (update) – कुशलता से किए जा सकें।

उपयोग (Uses):

- प्रदर्शन में सुधार (Efficiency): डेटा संरचनाएं एल्गोरिदम को तेज़ और स्मृति कुशल बनाती हैं।
- डेटा प्रबंधन (Data Management): बड़े और जटिल डेटा को व्यवस्थित ढंग से संग्रहित करती हैं।
- समस्या समाधान (Problem Solving): कंप्यूटर विज्ञान में विभिन्न समस्याओं को हल करने के लिए उपयुक्त डेटा संरचना का चयन किया जाता है।
- सॉफ्टवेयर विकास (Software Development): ऑपरेटिंग सिस्टम, डेटाबेस, कंपाइलर, और विभिन्न एप्लिकेशन में डेटा संरचनाएं आधारभूत भूमिका निभाती हैं।

प्रकार (Types of Data Structures)



डेटा संरचनाओं को मुख्यतः दो भागों में विभाजित किया जाता है:

1. (Linear Data Structure)

परिभाषा:

रेखीय डेटा संरचना वह होती है जिसमें सभी तत्वों को एक क्रम (sequence) में व्यवस्थित किया जाता है, अर्थात प्रत्येक तत्व का अगला और पिछला कोई न कोई तत्व होता है।

उदाहरण:

1. **Array (सरणी):** एक स्थिर आकार की डेटा संरचना जिसमें समान प्रकार के डेटा को क्रम में संग्रहित किया जाता है।
👉 उदाहरण: `int arr[5] = {10, 20, 30, 40, 50};`
2. **Linked List (लिंक्ड लिस्ट):** डेटा और उसके पते को साथ लेकर चलने वाली श्रृंखला।
3. **Stack (स्टैक):** LIFO (Last In First Out) सिद्धांत पर कार्य करता है।
4. **Queue (क्यू):** FIFO (First In First Out) सिद्धांत पर आधारित होता है।

2. (Non-Linear Data Structure)

परिभाषा:

अरेखीय डेटा संरचना वह होती है जिसमें तत्वों को रेखीय क्रम में नहीं रखा जाता, बल्कि पदानुक्रम (hierarchy) या नेटवर्क संरचना में व्यवस्थित किया जाता है।

उदाहरण:

1. **Tree (वृक्ष):** एक पदानुक्रमिक डेटा संरचना जिसमें एक मूल नोड (root node) होता है और उससे शाखाएं निकलती हैं।
👉 उदाहरण: Binary Tree, Binary Search Tree (BST), AVL Tree
2. **Graph (ग्राफ):** नोड्स और एजेस (nodes and edges) द्वारा बनाया गया नेटवर्क।
👉 उदाहरण: सोशल नेटवर्क, मार्ग नेटवर्क
3. **Heap (हीप):** एक विशेष प्रकार का ट्री जो पूर्ण बाइनरी ट्री होता है और विशेष क्रम का पालन करता है (जैसे Max Heap या Min Heap)।

एल्गोरिदम स्पेसिफिकेशन (Algorithm Specification)

1. परिचय (Introduction):

- एल्गोरिदम (**Algorithm**) किसी समस्या को हल करने के लिए चरणों का अनुक्रम (**Sequence of Steps**) है।
- एल्गोरिदम स्पेसिफिकेशन का मतलब है कि एल्गोरिदम को इस तरह से परिभाषित करना कि वह स्पष्ट (**Clear**), सटीक (**Precise**), और समझने में आसान हो।
- इसका उपयोग किसी प्रक्रिया को डिजाइन और प्रलेखित (**Document**) करने के लिए किया जाता है।

परफॉर्मेंस एनालिसिस (Performance Analysis):

- परफॉर्मेंस एनालिसिस का मतलब है कि किसी एल्गोरिदम की दक्षता (**Efficiency**) का विश्लेषण करना।
- एल्गोरिदम की दक्षता का मापन निम्नलिखित मानदंडों पर आधारित होता है:

(i) समय जटिलता (Time Complexity):

- यह बताता है कि एल्गोरिदम को निष्पादित करने में कितना समय लगेगा।
- इसे इनपुट के आकार (**Input Size**) के आधार पर मापा जाता है।
- नोटेशन:
 - **Big-O (O):** एल्गोरिदम का सबसे खराब समय।
 - **Theta (Θ):** औसत समय।
 - **Omega (Ω):** एल्गोरिदम का सबसे अच्छा समय।

उदाहरण:

- लिनियर सर्च (Linear Search): $O(n)$
- बाइनरी सर्च (Binary Search): $O(\log n)$

(ii) स्थान जटिलता (Space Complexity):

- यह बताता है कि एल्गोरिदम को निष्पादित करने में कितनी मेमोरी लगेगी।

परफॉर्मेंस एनालिसिस (Performance Analysis):

- परफॉर्मेंस एनालिसिस का मतलब है कि किसी एल्गोरिदम की दक्षता (Efficiency) का विश्लेषण करना।
- एल्गोरिदम की दक्षता का मापन निम्नलिखित मानदंडों पर आधारित होता है:

(i) समय जटिलता (Time Complexity):

- यह बताता है कि एल्गोरिदम को निष्पादित करने में कितना समय लगेगा।
- इसे इनपुट के आकार (Input Size) के आधार पर मापा जाता है।

नोटेशन:

- Big-O (O): एल्गोरिदम का सबसे खराब समय।
- Theta (Θ): औसत समय।
- Omega (Ω): एल्गोरिदम का सबसे अच्छा समय।

उदाहरण:

- लिनियर सर्च (Linear Search): $O(n)$
- बाइनरी सर्च (Binary Search): $O(\log n)$

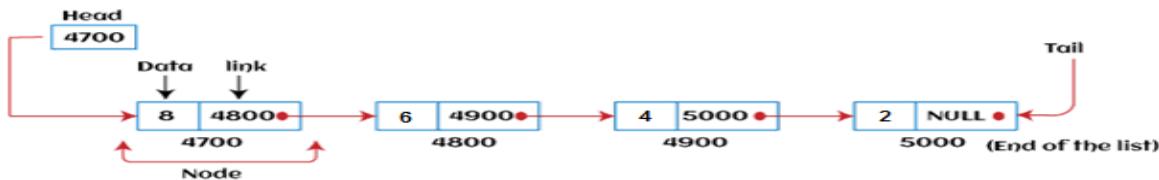
(ii) स्थान जटिलता (Space Complexity):

यह बताता है कि एल्गोरिदम को निष्पादित करने में कितनी मेमोरी लगेगी।

लिंक्ड लिस्ट (Linked List) क्या है?

लिंक्ड लिस्ट एक डेटा संरचना (Data Structure) है, जिसका उपयोग डेटा को क्रमबद्ध (sequential) रूप से संग्रहीत (store) करने के लिए किया जाता है। यह डेटा को स्टोर करने के लिए नोड्स (Nodes) का उपयोग करती है। प्रत्येक नोड में दो भाग होते हैं:

1. **डेटा (Data):** यह वह जानकारी है, जिसे हम स्टोर करना चाहते हैं।
2. **पॉइंटर (Pointer):** यह अगले नोड का पता (address) रखता है।



लिंक्ड लिस्ट क्यों उपयोगी है? (Why Linked List is Useful):

1. डायनामिक मेमोरी अलोकेशन (Dynamic Memory Allocation):

1. लिंक्ड लिस्ट में मेमोरी रन-टाइम के दौरान डायनामिक रूप से आवंटित (Allocate) होती है।
2. ऐसे की तरह पहले से मेमोरी का आकार तय करने की ज़रूरत नहीं होती।

2. नॉन-कंटिग्यूस मेमोरी (Non-Contiguous Memory):

1. लिंक्ड लिस्ट के सभी नोड्स नॉन-कंटिग्यूस (Non-Contiguous) मेमोरी में स्टोर होते हैं।
2. नोड्स को आपस में पॉइंटर्स (Pointers) की मदद से जोड़ा जाता है।

3. आकार की समस्या नहीं:

1. लिंक्ड लिस्ट का आकार प्रोग्राम की ज़रूरत के अनुसार बढ़ या घट सकता है।
2. यह उपलब्ध मेमोरी तक सीमित होता है।

4. इन्सर्शन और डिलीशन आसान:

1. लिंक्ड लिस्ट में किसी नोड को जोड़ने (Insertion) या हटाने (Deletion) के लिए सिर्फ पॉइंटर को अपडेट करना होता है।
2. डेटा शिफ्ट करने की ज़रूरत नहीं होती।

लिंक्ड लिस्ट के प्रकार

1. सिंगल लिंक्ड लिस्ट (Singly Linked List):

1. प्रत्येक नोड केवल अगले नोड की जानकारी रखता है।
2. दिशा: आगे (Forward)।

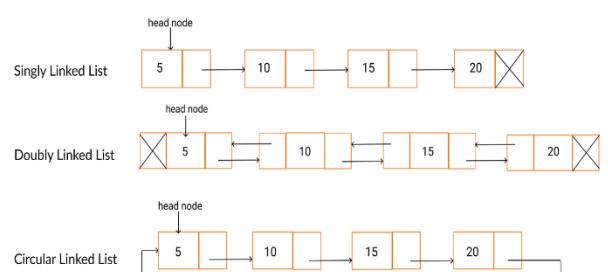
2. डबल लिंक्ड लिस्ट (Doubly Linked List):

1. प्रत्येक नोड में पिछले और अगले नोड, दोनों का पता होता है।
2. दिशा: आगे और पीछे (Forward और Backward)।

3. संकुलर लिंक्ड लिस्ट (Circular Linked List):

1. आखिरी नोड, पहले नोड से जुड़ा होता है।
2. यह एक चक्र (Circle) बनाता है।

Types of Linked List



Stack की परिभाषा (Definition of Stack in Detail):

Stack एक विशेष प्रकार की रेखीय डेटा संरचना (**Linear Data Structure**) है, जिसमें डेटा को क्रमबद्ध (**ordered**) तरीके से संग्रहित किया जाता है, लेकिन इसमें जोड़ने (insert) और हटाने (delete) की क्रिया केवल एक ही सिरे (एक ही सिरा – **Top**) से होती है। Stack को आमतौर पर "**LIFO**" (**Last In, First Out**) सिद्धांत पर आधारित माना जाता है, जिसका अर्थ है कि जो तत्व सबसे अंत में जोड़ा जाता है, वही सबसे पहले हटाया जाता है।

◆ उदाहरण द्वारा समझें:

Stack की तुलना हम प्लेटों के ढेर (**Stack of plates**) से कर सकते हैं। जब हम एक-एक करके प्लेटें ऊपर-ऊपर रखते हैं, तो जो प्लेट सबसे अंत में रखी जाती है, उसे सबसे पहले उठाया जा सकता है। नीचे की प्लेटें तब तक नहीं हटाई जा सकतीं जब तक ऊपर की प्लेटें न हटाई जाएं। यही नियम Stack में भी लागू होता है।

◆ विशेषताएँ:

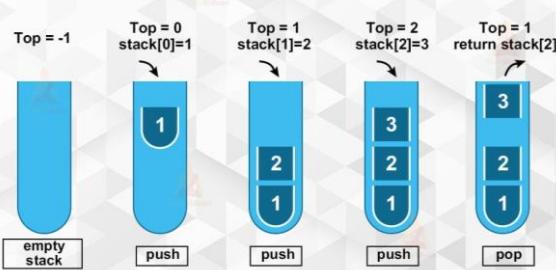
- Stack में सिर्फ एक ही पॉइंट (**Top**) होता है जहाँ से सभी कार्य किए जाते हैं।
- इसमें नए तत्व को जोड़ने के लिए **push()** और हटाने के लिए **pop()** क्रिया का उपयोग किया जाता है।
- यह संरचना स्मृति (memory) प्रबंधन, प्रोग्रामिंग एल्गोरिदम, और कई कंप्यूटर प्रक्रियाओं में अत्यंत महत्वपूर्ण भूमिका निभाती है।

उपयोग (Uses of Stack):

1. फंक्शन कॉल स्टैक (Function calls) – रिकर्सन में
2. ब्राउज़र हिस्ट्री (Back/Forward navigation)
3. **Undo/Redo** ऑपरेशन
4. **Expression evaluation** (जैसे postfix, prefix)
5. **Syntax Parsing** (Compiler में)
6. **Memory management** (Runtime Stack)
7. **(Operations on Stack):**

क्रिया	विवरण (Description)
1. push()	स्टैक के शीर्ष पर एक नया तत्व जोड़ता है।
2. pop()	स्टैक के शीर्ष से तत्व हटाता है। यदि स्टैक खाली है तो Underflow स्थिति उत्पन्न होती है।
3. peek() / top()	स्टैक के शीर्ष पर वर्तमान तत्व को दिखाता है, लेकिन उसे हटाता नहीं है।
4. isEmpty()	जांचता है कि स्टैक खाली है या नहीं।
5. isFull()	यह जांचता है कि स्टैक पूरा भर चुका है (सिर्फ array आधारित स्टैक में लागू)।

Stack in Data structure



Stack के मुख्य Operations with Explanation and Algorithm

☒ 1. **push (x) Operation**

◆ **Explanation:**

Stack में **push ()** ऑपरेशन का उपयोग किसी नए डेटा एलिमेंट को जोड़ने के लिए किया जाता है। यह एलिमेंट हमेशा **top** (शीर्ष) पर जोड़ा जाता है। यदि Stack पहले से ही full है, तो इसे और एलिमेंट नहीं जोड़ा जा सकता — इसे **Stack Overflow** कहते हैं।

◆ **Algorithm for push (x)**

vbnnet

CopyEdit

```
Step 1: If TOP == SIZE - 1 then
        Display "Stack Overflow"
        Exit
Step 2: TOP ← TOP + 1
Step 3: STACK[TOP] ← x
Step 4: Exit
```

2. **pop() Operation

◆ **Explanation:**

Stack में **pop()** ऑपरेशन का उपयोग **top** पर मौजूद एलिमेंट को हटाने (**delete**) के लिए किया जाता है। यदि Stack खाली है, तो कोई एलिमेंट हटाया नहीं जा सकता — इसे **Stack Underflow** कहा जाता है।

◆ **Algorithm for pop()**

```
vbnet
CopyEdit
Step 1: If TOP == -1 then
        Display "Stack Underflow"
        Exit
Step 2: x ← STACK[TOP]
Step 3: TOP ← TOP - 1
Step 4: Return x
Step 5: Exit
```

3. **peek() or top()** Operation

◆ **Explanation:**

peek() या **top()** ऑपरेशन का उपयोग Stack के **शीर्ष (top)** पर मौजूद एलिमेंट को देखने के लिए किया जाता है, बिना उसे हटाए। यह उपयोगकर्ता को Stack की वर्तमान स्थिति जानने में मदद करता है।

◆ **Algorithm for peek()**

```
vbnet
CopyEdit
Step 1: If TOP == -1 then
        Display "Stack is Empty"
        Exit
Step 2: Return STACK[TOP]
Step 3: Exit
```

4. **isEmpty()** Operation

◆ **Explanation:**

isEmpty() ऑपरेशन यह जांचने के लिए किया जाता है कि Stack में कोई एलिमेंट मौजूद है या नहीं। यदि TOP -1 पर है, तो Stack पूरी तरह खाली होता है।

◆ **Algorithm for isEmpty()**

```
vbnet
CopyEdit
Step 1: If TOP == -1 then
        Return TRUE
    Else
        Return FALSE
Step 2: Exit
```

5. **isFull()** Operation

◆ Explanation:

isFull() ऑपरेशन यह जांचता है कि Stack में और कोई नया एलिमेंट जोड़ा जा सकता है या नहीं। यह केवल Array-आधारित Stack के लिए उपयोगी है। यदि TOP = SIZE - 1 है, तो Stack पूर्ण (Full) माना जाता है।

◆ Algorithm for isFull()

```
vbnet
CopyEdit
Step 1: If TOP == SIZE - 1 then
        Return TRUE
    Else
        Return FALSE
```

Step 2: Exit

Queue (क्यू) – विस्तार से समझाएँ

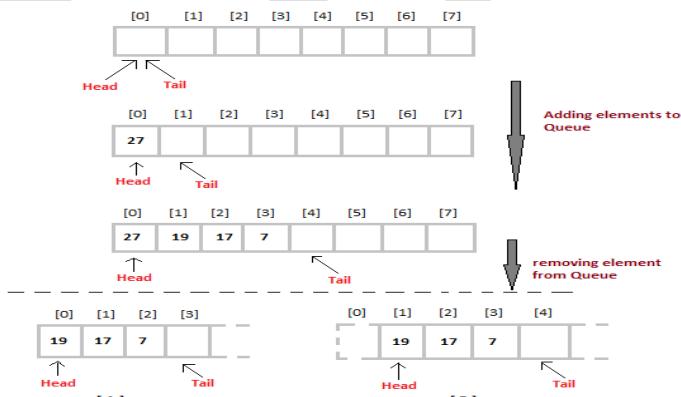
◆ परिभाषा (Definition):

Queue एक रेखीय डेटा संरचना (Linear Data Structure) है जो FIFO (First In, First Out) सिद्धांत पर कार्य करती है।

अर्थात् जो डेटा सबसे पहले Queue में डाला गया है, वही सबसे पहले निकाला जाएगा। यह ठीक ऐसे ही होता है जैसे लोग लाइन में खड़े रहते हैं – जो पहले आता है, सेवा पहले उसे मिलती है।

◆ विशेषताएँ (Characteristics):

- दो मुख्य सिरों (ends) होते हैं:
 - Front:** जहाँ से डेटा निकाला (deletion) जाता है
 - Rear:** जहाँ पर डेटा जोड़ा (insertion) जाता है
- डेटा को क्रमबद्ध तरीके से संग्रहीत किया जाता है।
- Queue में **overflow** और **underflow** जैसी स्थितियाँ आ सकती हैं।



❖ मुख्य क्रियाएँ (Operations on Queue)

1. enqueue (x) – Insertion

Explanation:

Queue के rear पर नया डेटा जोड़ने की प्रक्रिया को enqueue कहते हैं। यदि queue full हो जाती है, तो **Overflow** स्थिति उत्पन्न होती है।

Algorithm:

```
Algorithm ENQUEUE (x)
```

```
1. if rear == SIZE - 1 then
        print "Queue Overflow"
        Exit
```

```
2. if front == -1 then
    front ← 0
3. rear ← rear + 1
4. queue[rear] ← x
5. Exit
```

☒ 2. **dequeue () – Deletion**

Explanation:

Queue के **front** से डेटा हटाने की प्रक्रिया को **dequeue** कहा जाता है। यदि queue खाली है, तो **Underflow** स्थिति होती है।

Algorithm:

Algorithm DEQUEUE()

```
1. if front == -1 or front > rear then
    print "Queue Underflow"
    Exit
2. x ← queue[front]
3. front ← front + 1
4. return x
5. Exit
```

☒ 3. **peek () – Front Element को देखना**

Explanation:

peek() या **front()** का उपयोग queue में सबसे पहले डाले गए (front) एलिमेंट को देखने के लिए किया जाता है, बिना उसे हटाए।

Algorithm:

Algorithm PEEK()

```
1. if front == -1 or front > rear then
    print "Queue is Empty"
    Exit
2. return queue[front]
3. Exit
```

☒ 4. **isEmpty () – खाली जांचना**

Explanation:

यह ऑपरेशन यह जांचता है कि queue में कोई डेटा मौजूद है या नहीं।

Algorithm:

Algorithm isEmpty()

```
1. if front == -1 or front > rear then
    return TRUE
2. else
    return FALSE
3. Exit
```

☒ 5. **isFull () – पूरा भरा है या नहीं**

Explanation:

यह जांचता है कि queue पूरी भर चुकी है या नहीं।

Algorithm:

Algorithm isFull()

```

1. if rear == SIZE - 1 then
    return TRUE
2. else
    return FALSE
3. Exit

```

नोट्स:

शब्द

अर्थ

front	Queue का वह सिरा जहाँ से डेटा हटाया जाता है
rear	Queue का वह सिरा जहाँ नया डेटा जोड़ा जाता है
SIZE	Queue की अधिकतम सीमा
queue []	Queue को स्टोर करने वाला array

Queue के प्रकार (Types of Queue)

Queue डेटा संरचना के विभिन्न प्रकार होते हैं जो उपयोग के आधार पर अलग-अलग कार्य करते हैं। प्रमुखतः **5** प्रकार के Queue होते हैं:

1. Simple Queue (साधारण क्यू)

◆ परिभाषा:

Simple Queue वह सामान्य queue होता है जिसमें डेटा केवल एक सिरे (**rear**) से जोड़ा जाता है और दूसरे सिरे (**front**) से हटाया जाता है। यह पूरी तरह से **FIFO (First In, First Out)** सिद्धांत पर आधारित होता है।

◆ विशेषताएँ:

- Insertion → Rear से
- Deletion → Front से

◆ उदाहरण:

बैंक की कतार, टिकट खिड़की पर लाइन

2. Circular Queue (वृत्ताकार क्यू)

◆ परिभाषा:

Circular Queue एक ऐसा queue होता है जिसमें लास्ट पोजिशन के बाद फिर से पहला स्थान इस्तेमाल किया जा सकता है। यह एक **circle** के रूप में कार्य करता है।

◆ विशेषताएँ:

- Memory की बर्बादी नहीं होती
- Front और Rear circular रूप से चलते हैं ($(\text{rear} + 1) \% \text{ SIZE}$)
- यह Overflow की स्थिति को कम करता है

◆ उदाहरण:

CPU Scheduling, Buffer Management

3. Double-Ended Queue (Deque – डबल एंडेड क्यू)

◆ परिभाषा:

Deque एक ऐसा queue होता है जिसमें डेटा को दोनों सिरों (**front** और **rear**) से जोड़ा और हटाया जा सकता है। यह flexible होता है।

- ◆ प्रकार:
 - **Input Restricted Deque:** Insertion केवल एक सिरे से होती है, लेकिन deletion दोनों ओर से हो सकती है।
 - **Output Restricted Deque:** Deletion केवल एक सिरे से होती है, लेकिन insertion दोनों ओर से हो सकती है।
- ◆ उदाहरण:

Palindrome checking, Job Scheduling

4. Priority Queue (प्राथमिकता क्यू)

- ◆ परिभाषा:

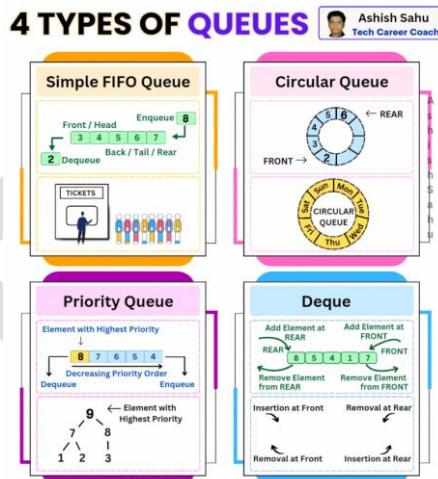
Priority Queue एक ऐसा queue होता है जिसमें प्रत्येक एलिमेंट के साथ एक प्राथमिकता (**priority**) जुड़ी होती है, और हटाने का क्रम **priority** के आधार पर होता है, न कि FIFO के अनुसार।

- ◆ विशेषताएँ:

- उच्च प्राथमिकता वाला एलिमेंट पहले हटाया जाता है
- यदि दो एलिमेंट की प्राथमिकता समान है, तो FIFO लागू होता है

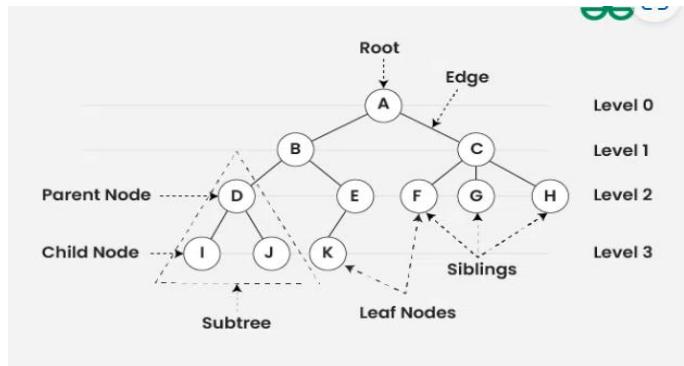
- ◆ उदाहरण:

Operating System में Task Scheduling, Emergency Room System



Tree Definition

ट्री एक पदानुक्रमित और (non-linear data structure) है, जिसमें नोड्स (nodes) किनारों (edges) द्वारा जुड़े होते हैं। यह एक मूल नोड (root node) से शुरू होता है और नीचे की ओर बाल नोड्स (child nodes) तक फैलता है। ऐरे और लिंक्ड लिस्ट जैसी रैखिक डेटा संरचनाओं के विपरीत, ट्री डेटा को पदानुक्रमिक रूप से व्यवस्थित करता है, जिससे कुशल खोज (searching) और क्रमबद्ध (sorting) संचालन संभव हो पाते हैं। कंप्यूटर विज्ञान में, ट्री का उपयोग फ़ाइल सिस्टम, डेटाबेस और पदानुक्रमित वर्गीकरण जैसी संरचनाओं को व्यवस्थित करने के लिए किया जाता है।



- ट्री में पहला node **root node** कहलाता है।
- एक node को दूसरे node से जोड़ने वाली कड़ी **edge** कहलाती है।
- एक **parent node** अपने child nodes से जुड़ा होता है। parent node को **internal node** भी कहा जाता है।
- एक node के zero, one, या many child nodes हो सकते हैं।
- प्रत्येक node का केवल **one parent node** होता है।
- जिन nodes के कोई child nodes नहीं होते, उन्हें **leaves** या **leaf nodes** कहा जाता है।
- **tree height**, root node से लेकर leaf node तक की maximum edges की संख्या होती है।
- किसी node की **height**, उस node से leaf node तक की maximum edges की संख्या होती है।
- **tree size**, ट्री में मौजूद कुल nodes की संख्या होती है।

types of tree

Binary Trees:

इस ट्री में प्रत्येक node के अधिकतम दो child nodes हो सकते हैं - एक **left child node** और एक **right child node**। यह संरचना अधिक जटिल ट्री प्रकारों जैसे कि **Binary Search Trees (BSTs)** और **AVL Trees** की नींव है।

Binary Search Trees (BSTs):

यह Binary Tree का एक प्रकार है, जहां प्रत्येक node के लिए, उसका left child node छोटे मान (value) वाला होता है और right child node बड़े मान वाला होता है।

AVL Trees:

यह Binary Search Tree का एक प्रकार है जो स्वयं को balance करता है, जिससे प्रत्येक node के लिए, उसके left और right subtrees की ऊँचाई में अंतर अधिकतम एक होता है। इस संतुलन (balance) को nodes के insert या delete होने पर rotations के माध्यम से बनाए रखा जाता है।

Binary Tree एक प्रकार की tree data structure है जिसमें प्रत्येक node के अधिकतम दो child nodes हो सकते हैं - एक **left child node** और एक **right child node**।

इस प्रतिबंध (restriction), कि प्रत्येक node के अधिकतम दो child nodes हो सकते हैं, के कई लाभ हैं:

- **Traversing, searching, insertion** और **deletion** जैसे एल्गोरिदम को समझना, लागू करना और उन्हें तेज़ी से चलाना आसान हो जाता है।
- डेटा को **Binary Search Tree (BST)** में व्यवस्थित रखने से searching अत्यधिक कुशल (efficient) हो जाती है।
- सीमित संख्या में child nodes होने के कारण trees को balance करना आसान हो जाता है, जैसे कि **AVL Binary Tree** का उपयोग करके।
- Binary Trees को **arrays** के रूप में भी represent किया जा सकता है, जिससे यह memory के उपयोग के दृष्टिकोण से अधिक कुशल हो जाता है।

Graph (डेटा स्ट्रक्चर)

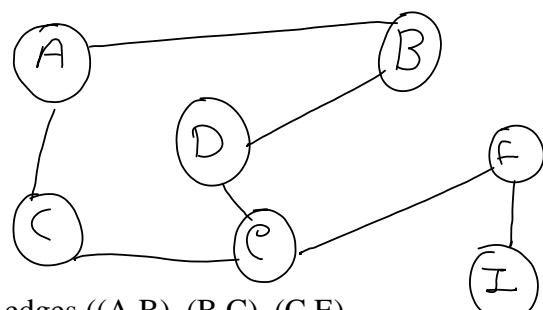
परिचय:

Graph को एक ऐसे समूह के रूप में परिभाषित किया जाता है जिसमें **vertices** (nodes) और **edges** होते हैं जो इन vertices को आपस में जोड़ते हैं।

Graph को एक cyclic tree की तरह देखा जा सकता है, जिसमें vertices आपस में किसी भी प्रकार का जटिल संबंध बनाए रख सकते हैं — यह एक सामान्य **parent-child** structure की तरह नहीं होता।

Definition: Graph G को एक ordered set के रूप में परिभाषित किया जा सकता है:
G(V, E) जहाँ:

- **V(G)** → vertices का set है
- **E(G)** → edges का set है जो इन vertices को आपस में जोड़ता है



उदाहरण के लिए, एक Graph G(V, E) जिसमें 5 vertices (A, B, C, D, E) और 6 edges ((A,B), (B,C), (C,E), (E,D), (D,B), (D,A)) हैं, नीचे के चित्र में दिखाया गया है।

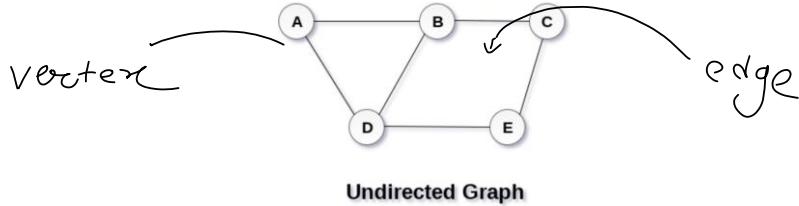
Types of Graph:

1. Undirected Graph (अदिश ग्राफ़):

इस प्रकार के Graph में edges के साथ कोई direction जुड़ी नहीं होती।

यदि vertex A और B के बीच कोई edge है, तो traversal दोनों दिशाओं में संभव है — यानी A से B और B से A।

ऊपर दिए गए उदाहरण में जो Graph दिखाया गया है, वह एक **undirected graph** है क्योंकि उसकी edges के साथ कोई direction नहीं है।



Undirected Graph

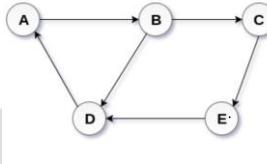
2. Directed Graph (दिशात्मक ग्राफ़):

Directed Graph में edges एक ordered pair के रूप में होती हैं।

इसमें edge एक specific direction को दिखाती है — जैसे vertex A से vertex B की ओर।

A को **initial node** कहा जाता है

- B को **terminal node** कहा जाता है
इसका मतलब यह है कि traversal केवल A से B की दिशा में ही हो सकता है, B से A नहीं।



Directed Graph

Hashing (हैशिंग)

Hashing कंप्यूटर साइंस में एक प्रसिद्ध तकनीक है, जिसमें बड़े डेटा सेट्स को एक **fixed-length value** में मैप किया जाता है। यह एक प्रक्रिया है जिसमें किसी भी **variable size** के डेटा को एक **fixed size** के डेटा में बदला जाता है। डेटा को तेजी से खोजने की क्षमता के कारण hashing डेटा संरचना (Data Structure) का एक महत्वपूर्ण भाग है।

What is Hashing?

Hashing algorithm का उपयोग किसी इनपुट (जैसे: string या integer) को एक **fixed-size output** (जिसे **hash code** या **hash value** कहते हैं) में बदलने के लिए किया जाता है। यह hash value फिर किसी **array** या **hash table** में डेटा को स्टोर या retrieve करने के लिए **index** के रूप में उपयोग की जाती है।

एक अच्छी hash function **deterministic** होती है, यानी किसी दिए गए इनपुट के लिए यह हमेशा एक ही hash value देगी।

Hashing का उपयोग आमतौर पर किसी डेटा के लिए एक **unique identifier** बनाने के लिए किया जाता है, ताकि उस डेटा को बड़े dataset में तेजी से ढूँढा जा सके।

उदाहरण: वेब ब्राउज़र password को सुरक्षित रूप से स्टोर करने के लिए hashing का उपयोग करते हैं। जब यूजर पासवर्ड दर्ज करता है, तो ब्राउज़र उसे hash value में बदल देता है और पहले से स्टोर की गई hash value से मिलान करता है। (Browser :— Chrome, edge, opera)

◆ What is a Hash Key?

Hashing में, **hash key** (जिसे **hash value** या **hash code** भी कहते हैं) एक **fixed-size numerical** या **alphanumeric value** होती है जो input डेटा से उत्पन्न होती है।

Hash key को प्राप्त करने के लिए एक **mathematical hash function** को input डेटा पर लागू किया जाता है। यह key डेटा का **digital fingerprint** होती है — यानी डेटा में थोड़ा भी बदलाव करने पर hash key पूरी तरह बदल जाती है।

Hash Key के उपयोग:

- डेटा की सत्यता (data integrity) की जांच
- तेजी से डेटा स्टोर और एक्सेस करने के लिए
- Hash Table में quick lookup के लिए

◆ How Hashing Works?

Hashing प्रक्रिया को तीन मुख्य चरणों में समझा जा सकता है:

$$13 \% 6 = 1$$

$$25 \% 6 = 1$$

$$20 \% 6 = 2$$

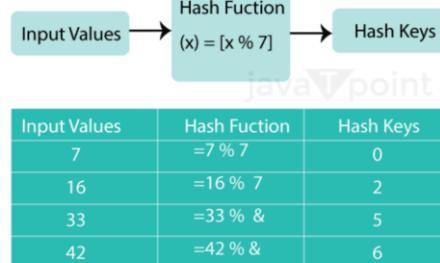
1. **Input:** डेटा को hashing algorithm में इनपुट के रूप में दिया जाता है।
2. **Hash Function:** यह algorithm इनपुट डेटा पर mathematical function लागू करती है और एक fixed-size hash value देती है।
3. **Output:** प्राप्त hash value का उपयोग डेटा को स्टोर या खोजने के लिए किया जाता है, जैसे कि array या hash table में।

Hashing Algorithms:

कुछ प्रमुख hashing algorithms निम्नलिखित हैं:

6	13	25	20	16	
0	1	2	3	4	5

- **MD5** → 128-bit hash value उत्पन्न करता है।
- **SHA-1** → 160-bit hash value उत्पन्न करता है।
- **SHA-256** → अधिक सुरक्षित algorithm जो 256-bit hash value देता है।



Hashing Data Structure

$$6 \% 6 = ?$$

0	7
1	16
2	
3	
4	
5	33
6	42

$$16 \% 6 = 4$$

$$33 \% 7 = 5$$

$$16 \% 7 = 2$$

$$42 \% 7 = 6$$

◆ Hashing in Data Structure

☒ Hash Function:

Hash function एक mathematical function होती है जो किसी input (key) को लेकर एक **fixed-size hash value** उत्पन्न करती है।

यह function **deterministic** होनी चाहिए, ताकि एक ही input के लिए हमेशा एक जैसा output मिले।

अच्छी hash function की विशेषताएँ:

- अलग-अलग inputs के लिए अलग hash values

- समान input के लिए समान output
- समान रूप से hash values का वितरण (uniform distribution)

Types of Hash Functions:

◆ 1. Division Method:

इसमें key को table size से divide किया जाता है और remainder को hash value के रूप में लिया जाता है।

उदाहरण: यदि key = 23 और table size = 10 हो, तो hash value = $23 \% 10 = 3$

◆ 2. Multiplication Method:

इसमें key को एक constant से multiply किया जाता है और उसके fractional part से hash value निकाली जाती है।

उदाहरण: यदि key = 23 और constant = 0.618 हो, तो

hash value = $\text{floor}(10 * (0.618 * 23 \% 1)) = \text{floor}(2.236) = 2$

◆ 3. Universal Hashing:

इस विधि में hash function को hash functions के एक family में से random तरीके से चुना जाता है।

यह biased input या संभावित हमलों से सुरक्षा प्रदान करता है और एक प्रकार की randomness लाता है।

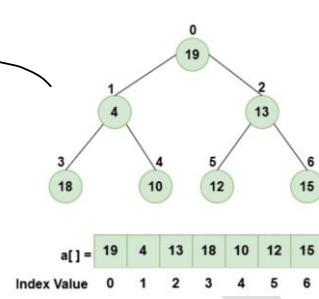
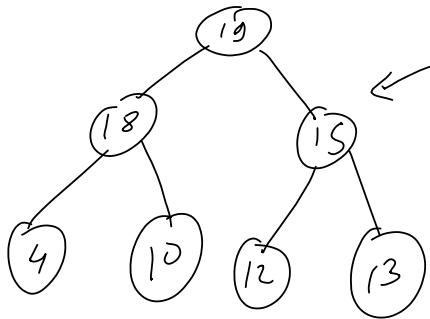
◆ परिभाषा (Definition):

Heap एक विशेष प्रकार का **Binary Tree** होता है, जो कि हमेशा एक **Complete Binary Tree** होता है। इसका मतलब यह है कि सभी लेवल पूरी तरह भरे होते हैं, केवल अंतिम लेवल को छोड़कर, जो कि बाएं से दाएं भरा जाता है। *Max heap*

Heap में एक विशेष **Order Property** होती है:

Clue : —

- **Max-Heap:** हर node का मान उसके children के मान से अधिक या बराबर होता है।

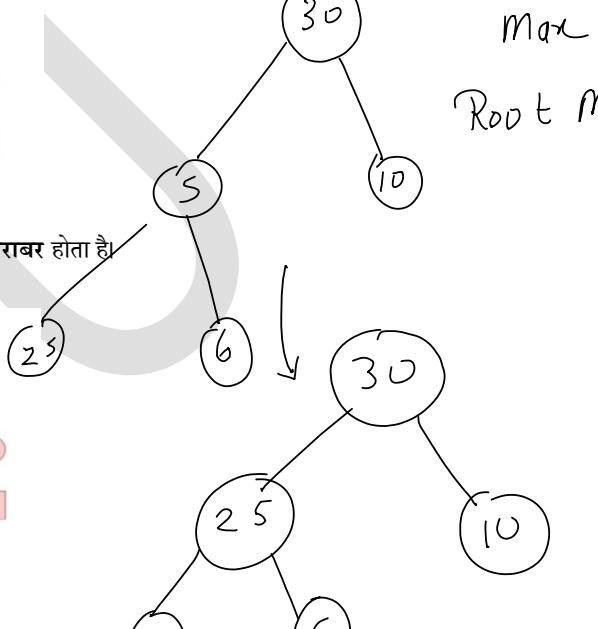
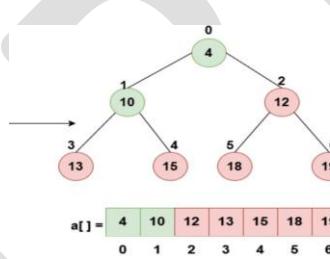


30 , 5 10 , 2 5 , 6

Max

Root Max

- **Min-Heap:** हर node का मान उसके children के मान से कम या बराबर होता है।



◆ **Insertion** (प्रवेश):

Heap में नया एलिमेंट जोड़ने की प्रक्रिया:

1. नए एलिमेंट को heap के अंत में जोड़ा जाता है (Complete Binary Tree बनाए रखने के लिए)।
2. इसके बाद **Up-heap** (या Bubble-up) प्रक्रिया होती है, जिसमें एलिमेंट को ऊपर ले जाया जाता है जब तक कि heap की property सही न हो जाए।

◆ **Deletion** (हटाना):

Heap में आमतौर पर **root node** (सबसे ऊपर का एलिमेंट) हटाया जाता है:

1. Root node को हटाया जाता है।
2. अंतिम एलिमेंट को root की जगह पर रखा जाता है।
3. फिर उसे नीचे की ओर **Down-heap** (या Heapify) किया जाता है ताकि heap property किर से संतुलित हो जाए।

AVL Tree एक **Self-Balancing Binary Search Tree** (स्वयं-संतुलन बनाए रखने वाला बाइनरी सर्च ट्री) होता है, जिसमें किसी भी नोड के बाएं और दाएं सबट्री (उपवृक्ष) की ऊँचाई (height) के बीच का अंतर **1** से अधिक नहीं होता है। इसका उद्देश्य Binary

Search Tree (BST) के संरचनात्मक दोष को सुधारना है, जहाँ लगातार डेटा इनसर्शन से Tree असंतुलित (Unbalanced) हो सकता है और search time बढ़ सकता है।

इस ट्री को 1962 में दो रूसी गणितज्ञों, **Adelson-Velsky** और **Landis** ने परिभाषित किया था, और उन्हीं के नाम पर इसे **AVL Tree** कहा जाता है।

संतुलन बनाए रखने के लिए **AVL Tree** में यह जांच की जाती है:

- हर नोड के लिए उसका **Balance Factor** निकाला जाता है, जो बाएं और दाएं subtree की ऊँचाई के अंतर से निर्धारित होता है।
- यदि यह अंतर **-1, 0, या 1** है, तो Tree संतुलित माना जाता है।
- यदि यह अंतर इससे अधिक हो जाए, तो Tree को बैलेंस करने के लिए रोटेशन (**Rotations**) का उपयोग किया जाता है।

◆ रोटेशन (**Rotations**) की सहायता से **AVL Tree** में संरचनात्मक परिवर्तन करके उसे फिर से संतुलित किया जाता है:

- Left Rotation (LL)
- Right Rotation (RR)
- Left-Right Rotation (LR)
- Right-Left Rotation (RL)

Binary Search Tree

परिभाषा (Definition):

Binary Search Tree (BST) एक ऐसा **Binary Tree** होता है जिसमें प्रत्येक नोड के बाएं (left) सब-ट्री में सभी नोड्स का मान उस नोड से छोटा होता है, और दाएं (right) सब-ट्री में सभी नोड्स का मान उस नोड से बड़ा होता है।

यह संरचना डेटा को इस प्रकार संगठित करती है कि **search, insertion** और **deletion** जैसे ऑपरेशन्स कुशलता से किए जा सकें।

◆ मुख्य गुण (**Properties**):

- हर नोड के अधिकतम दो बच्चे होते हैं।
- Left नोड < Root नोड < Right नोड
- Inorder traversal करने पर नोड्स sorted क्रम में आते हैं।

BST के उपयोग (**Applications**):

- Searching
- Sorting
- Indexing
- Data Lookup

AVL Tree and BST में अंतर:

विशेषता	Binary Search Tree (BST)	AVL Tree
संरचना	Ordered Binary Tree	Self-Balancing BST
बैलेंसिंग	नहीं होती	हर नोड पर बैलेंसिंग होती है
Search Time	Worst Case: O(n)	Always: O(log n)
Speed	तेज तभी जब बैलेंस हो	हमेशा संतुलित और तेज
उपयोग	साधारण कार्यों के लिए	जब performance महत्वपूर्ण हो