II B.tech II semester

Lecture notes on

ADVANCED DATA STRUCTURES

Prepared by
Mr P Venkateswarlu
Assistant professor
Department of information technology

Data Structure

- A data structure is a specialized format for organizing, processing, retrieving and storing data.
- While there are several basic and advanced structure types, any data structure is designed to arrange data to suit a specific purpose so that it can be accessed and worked with in appropriate ways.

Data Structure

- In computer programming, a data structure may be selected or designed to store data for the purpose of working on it with various algorithms.
- Each data structure contains information about the data values, relationships between the data and functions that can be applied to the data.

Data Structure

- The data structure is basically a technique of organizing and storing of different types of data items in computer memory.
- It is considered as not only the storing of data elements but also the maintaining of the logical relationship existing between individual data elements.
- The Data structure can also be defined as a mathematical or logical model, which relates to a particular organization of different data elements.

Data Structure

- ***Data:***
  - Data is the basic entity of fact that is used in calculations or manipulation process.
  - The way of organizing of the data & performing the operations is called as data structure.

    Data structure=organized data+ operations
  - Operations
    - Insertion
    - Deletions
    - Searching
    - Traversing

Data Structure

- The organization must be convenient for users.
- Data structures are implemented in the real time in the following situations:
  - Car park
  - File storage
  - Machinery
  - Shortest path
  - Sorting
  - Networking
  - Evaluation of expressions

Data Structure

- Specification of data structure :
  - Data structures are considered as the main building blocks of a **computer program**.
    - **Organization of data**
    - **Accessing methods**
    - **Degree of associativity**
    - **Processing alternatives for information**

Algorithm + Data Structure = Program

Data Structure

- at the time of selection of data structure we should follow these two things so that our selection is efficient enough to solve our problem.
  - The data structure must be powerful enough to handle the different relationship existing between the data.
  - The structure of data also to be simple, so that we can efficiently process data when required.

Characteristics of data structures

- **Linear or non-linear:** This characteristic describes whether the data items are arranged in chronological sequence,

  such as with an array,

  or in an unordered sequence,

  such as with a graph.

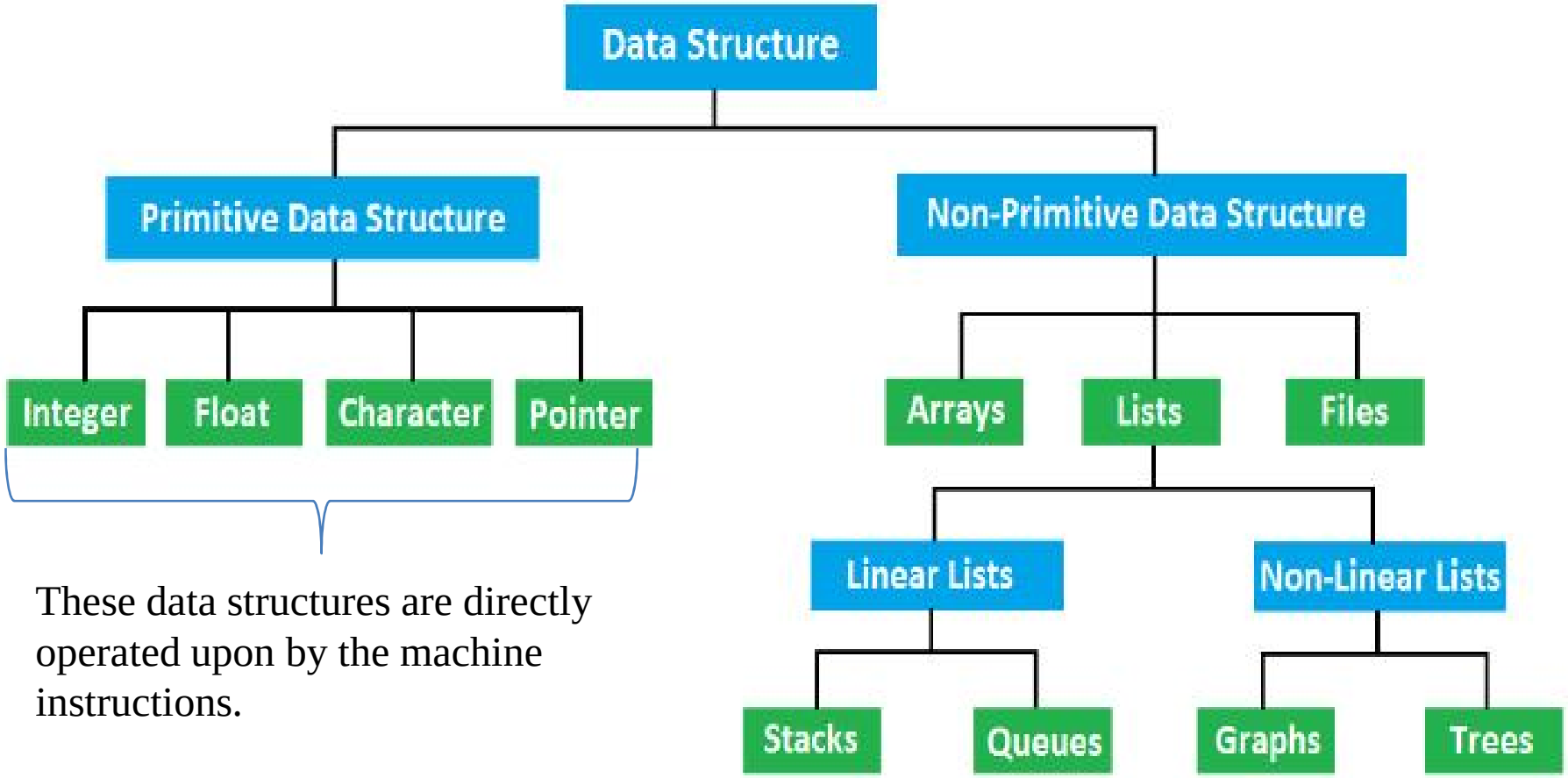
Characteristics of data structures

- **Homogeneous or non-homogeneous:** This characteristic describes whether all data items in a given repository are of the same type or of various types.

Characteristics of data structures

- **Static or dynamic:** This characteristic describes how the data structures are compiled. Static data structures have fixed sizes, structures and memory locations at compile time.
- Dynamic data structures have sizes, structures and memory locations that can shrink or expand depending on the use.

Types of data structures



These data structures are directly operated upon by the machine instructions.

Types of data structures

- ***Primitive data structure :***
  - The primitive data structures are known as basic data structures.
  - These data structures are directly operated upon by the machine instructions.
  - The primitive data structures have different representation on different computers.

Types of data structures

- ***Non-Primitive data structure :***
  - The non-primitive data structures are highly developed complex data structures.
  - Basically these are developed from the primitive data structure.
  - The non-primitive data structure is responsible for organizing the group of homogeneous and heterogeneous data elements.

Types of data structures

- Data structure types are determined by what types of operations are required or what kinds of algorithms are going to be applied.
- ***Arrays-***
  - An array stores a collection of items at adjoining memory locations.
  - Items that are the same type get stored together so that the position of each element can be calculated or retrieved easily.
  - Arrays can be fixed or flexible in length.

Types of data structures

- **Arrays-**

Subscript ⇨		0	1	2	3	4	5
Array name ⇨	x	3.1	3.8	4.4	5.1	2.4	3.6

One Dimenstional Array

Array Name ⇨	x	0	1	2	3	⇦ Column Subscripts
	0	2	4	5	6	
	1	6	0	9	8	
	2	-3	-6	1	-11	

⇧ Row Subscripts

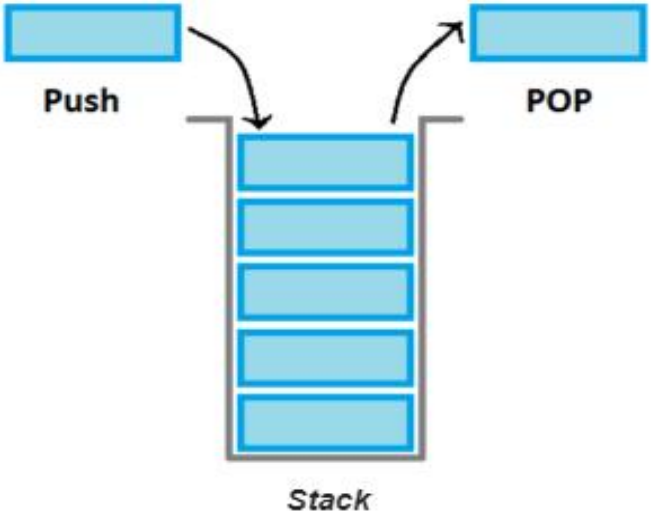
Two Dimensional Array

Types of data structures

- **Stacks-**

We can implement a stack using 2 ways:

- Static implementation (using arrays)
- Dynamic implementation (using pointers)

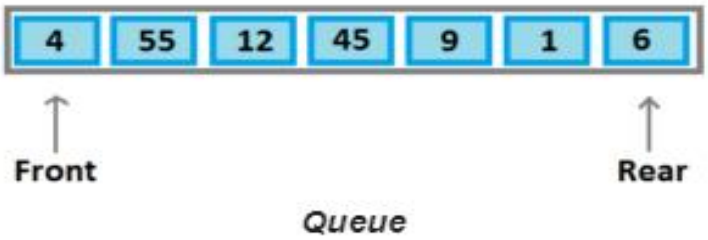


Types of data structures

- **Queues-**
  - A queue stores a collection of items similar to a stack; however the operation order can only be first in first out.

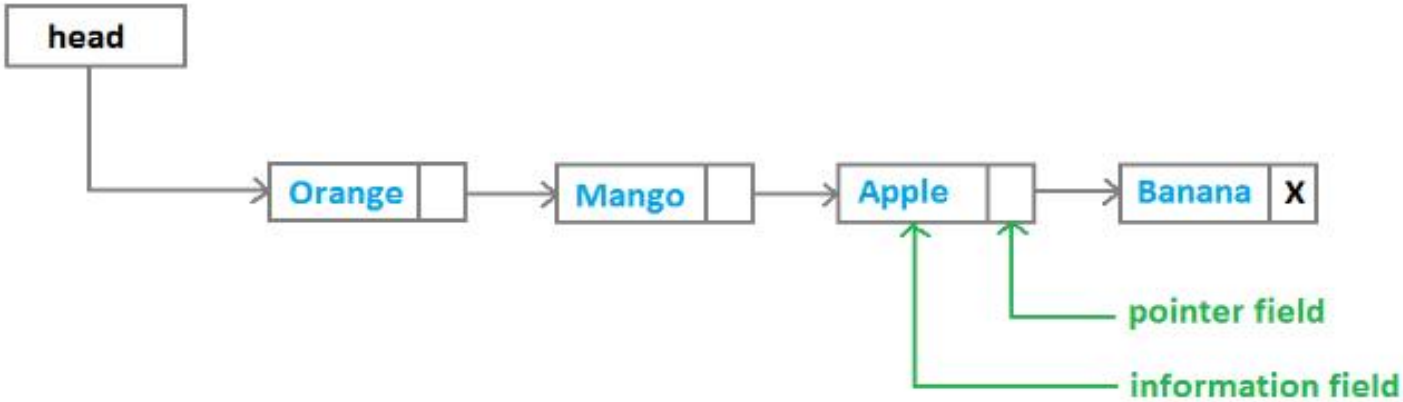
We can also implement queues using 2 ways :

- Using arrays
- Using pointers



Types of data structures

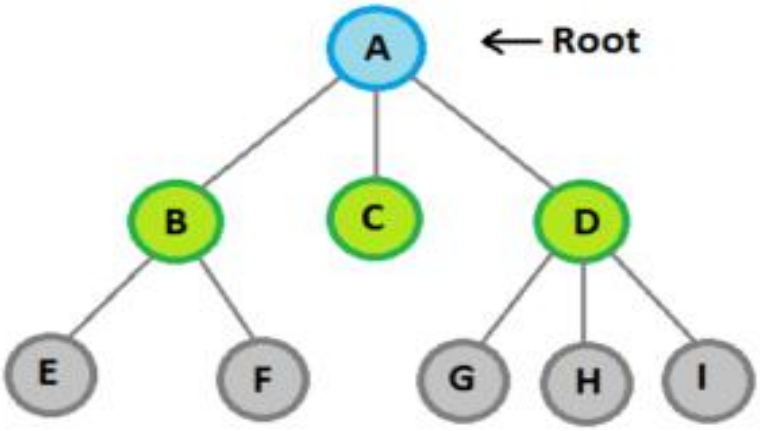
- **Linked lists-**
  - A linked list stores a collection of items in a linear order. Each element or node in a linked list contains a data item as well as a reference or link to the next item in the list.



Linear linked lists

Types of data structures

- **Trees-**
  - A tree stores a collection of items in an abstract hierarchical way.
  - Each node is linked to other nodes and can have multiple sub-values also known as children.



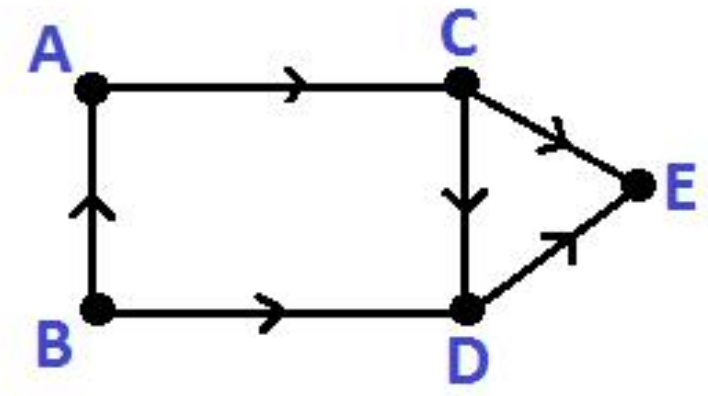
Trees

Types of data structures

- ***A Tree has the following characteristics :***
  - The top item in a hierarchy of a tree is referred as the root of the tree.
  - The remaining data elements are partitioned into a number of mutually exclusive subsets and they itself a tree and are known as the subtree.
  - Unlike natural trees trees in the data structure always grow in length towards the bottom.

Types of data structures

- **Graphs-**
  - A graph stores a collection of items in a non-linear fashion.
  - Graphs are made up of a finite set of nodes also known as vertices and lines that connect them also known as edges.
  - These are useful for representing real-life systems such as computer networks.



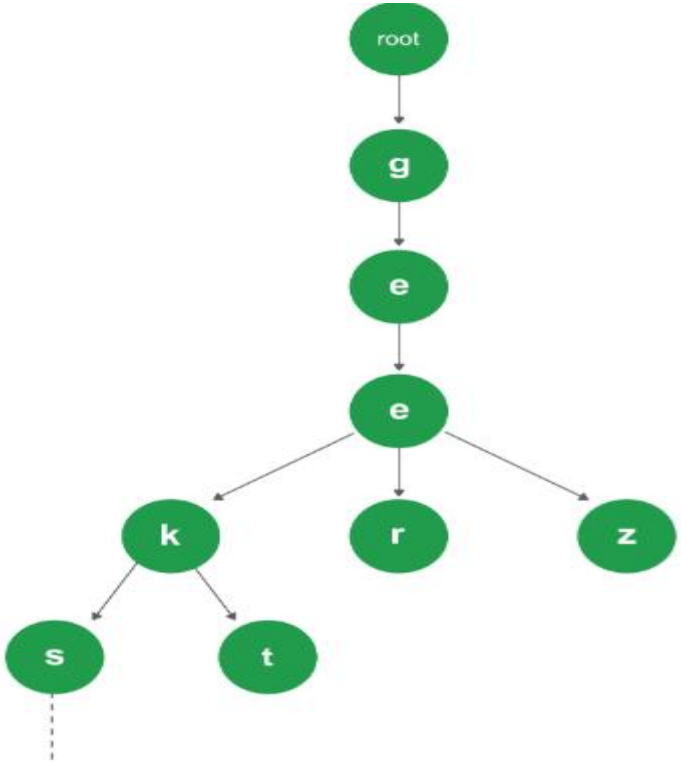
Graph

Types of data structures

- ***The different types of Graphs are :***
  - Directed Graph
  - Non-directed Graph
  - Connected Graph
  - Non-connected Graph
  - Simple Graph
  - Multi-Graph

Types of data structures

- **Tries-**
  - A trie or keyword tree, is a data structure that stores strings as data items that can be organized in a visual graph.



Types of data structures

- **Hash tables-**
  - A hash table or a hash map stores a collection of items in an associative array that plots keys to values.
  - A hash table uses a hash function to convert an index into an array of buckets that contain the desired data item.
  - Overcoming the drawbacks of linear data structures hashing is introduced.

Types of data structures

- ***Files :***
  - Files contain data or information, stored permanently in the secondary storage device such as Hard Disk and Floppy Disk.
  - It is useful when we have to store and process a large amount of data.
  - A file stored in a storage device is always identified using a file name like **HELLO.DAT** or **TEXTNAME.TXT** and so on.

Types of data structures

- ***Files :***
  - A file name normally contains a primary and a secondary name which is separated by a **dot(.)**.

Files

Fundamentals of data structures:

- **Fundamental Data Structures**
  - The following four data structures are used ubiquitously in the description of algorithms and serve as basic building blocks for realizing more complex data structures.
    - Sequences (also called as lists)
    - Dictionaries
    - Priority Queues
    - Graphs
  - Dictionaries and priority queues can be classified under a broader category called *dynamic sets*.
  - binary and general trees are very popular building blocks for implementing dictionaries and priority queues.

Fundamentals of data structures: Dictionaries

- A *dictionary* is a general-purpose data structure for storing a group of objects.
- A dictionary has a set of *keys* and each key has a single associated *value*.
- When presented with a key the dictionary will return the associated value.
- A dictionary is also called a *hash*, a *map*, a *hashmap* in different programming languages.

Fundamentals of data structures: Dictionaries

- For example the results of a classroom test could be represented as a dictionary with pupil's names as keys and their scores as the values
- results = { 'Detra' : 17,
  'Nova' : 84,
  'Charlie' : 22,
  'Henry' : 75,
  'Roxanne' : 92,
  'Elsa' : 29 }
- Instead of using the numerical index of the data we can use the dictionary names to return values
- >>> results['Nova']
  84
- >>> results['Elsa']
  29

Fundamentals of data structures: Dictionaries

- The keys in a dictionary must be simple types (such as integers or strings) while the values can be of any type.
- Different languages enforce different type restrictions on keys and values in a dictionary.
- Dictionaries are often implemented as hash tables.
- Keys in a dictionary must be unique an attempt to create a duplicate key will typically overwrite the existing value for that key.

Fundamentals of data structures: Dictionaries

- Dictionary is an abstract data structure that supports the following operations:
  - search(K key) (returns the value associated with the given key)
  - insert(K key, V value)
  - delete(K key)
- Each element stored in a dictionary is identified by a key of type K.
- Dictionary represents a mapping from keys to values.

Fundamentals of data structures: Dictionaries

- Dictionaries have numerous applications.
  - contact book
    - key: name of person; value:
  - telephone number table of program variable identiers
    - key: identier; value: address in memory
  - property-value collection
    - key: property name; value: associated value
  - natural language dictionary
    - key: word in language X; value: word in language Y
  - etc

Fundamentals of data structures: *operations on dictionaries*

- Dictionaries typically support several operations:
  - **retrieve a value** (depending on language, attempting to retrieve a missing key may give a default value or throw an exception)
  - **insert or update a value** (typically, if the key does not exist in the dictionary, the key-value pair is inserted; if the key already exists, its corresponding value is overwritten with the new one)
  - **remove a key-value** pair
  - test for existence of a key
- Note that items in a dictionary are unordered, so loops over dictionaries will return items in an arbitrary order.

Fundamentals of data structures: *Implementations on dictionaries*

- simple implementations: sorted or unsorted sequences, direct addressing
- hash tables
- binary search trees (BST)
- AVL trees
- self-organising BST
- red-black trees
- (a,b)-trees (in particular: 2-3-trees)
- B-trees and other

Fundamentals of data structures: The Dictionary ADT

- The abstract data type that corresponds to the dictionary metaphor is known by several names.
- Other terms for keyed containers include the names map, table, search table, associative array, or hash.
- Whatever it is called, the idea is a data structure optimized for a very specific type of search.
- Elements are placed into the dictionary in key/value pairs.

Fundamentals of data structures: The Dictionary ADT

- To do a retrieval, the user supplies a key, and the container returns the associated value.
- Each key identifies one entry; that is, each key is unique.
- data is removed from a dictionary by specifying the key for the data value to be deleted

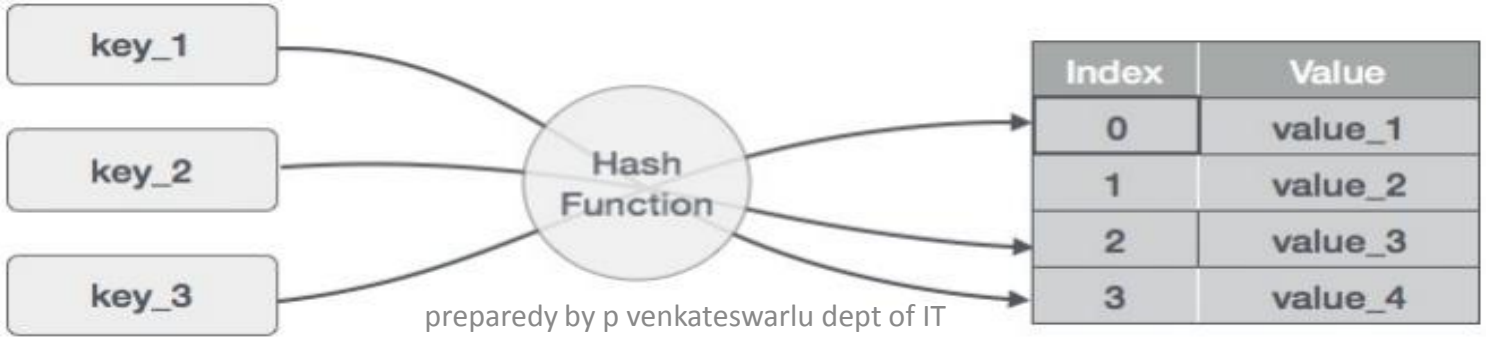
Operation	C++ Map<keytype, valuetype>	Java HashMap<keytype, valuetype>	C# hashtable
get	Map[key]	Get(key)	Hash[key]
put	Insert(key, value)	Put(key, value)	Add(key, value)
containsKey	Count(key)	containsKey(key)	
removeKey	Erase(key)	Remove(key)	

Fundamentals of data structures: Dictionary Implementation with Hash-Table

- Hash Table is a data structure which store data in associative manner.
- In hash table, data is stored in array format where each data values has its own unique index value.
- Access of data becomes very fast if we know the index of desired data.
- a data structure in which insertion and search operations are very fast irrespective of size of data.
- Hash Table uses array as a storage medium and uses hash technique to generate index where an element is to be inserted or to be located from.

Fundamentals of data structures: Dictionary Implementation with Hash-Table

- Hashing is a technique to convert a range of key values into a range of indexes of an array.
- We're going to use modulo operator to get a range of key values.
- Consider an example of hashtable of size 20, and following items are to be stored.
- Item are in key, value format.



Fundamentals of data structures: Dictionary Implementation with Hash-Table

	S.n.	Key	Hash	Array Index
1, 20	1	1	1 % 20 = 1	1
2, 70	2	2	2 % 20 = 2	2
42, 80	3	42	42 % 20 = 2	2
4, 25	4	4	4 % 20 = 4	4
12, 44	5	12	12 % 20 = 12	12
14, 32	6	14	14 % 20 = 14	14
17, 11	7	17	17 % 20 = 17	17
13, 78	8	13	13 % 20 = 13	13
37, 98	9	37	37 % 20 = 17	17

Fundamentals of data structures: Dictionary Implementation with Hash-Table

- **Linear Probing**
- the hashing technique used create already used index of the array.
- In such case, we can search the next empty location in the array by looking into the next cell until we found an empty cell.
- This technique is called linear probing

Fundamentals of data structures: Dictionary Implementation with Hash-Table

S.n.	Key	Hash	Array Index	After Linear Probing, Array Index
1	1	1 % 20 = 1	1	1
2	2	2 % 20 = 2	2	2
3	42	42 % 20 = 2	2	3
4	4	4 % 20 = 4	4	4
5	12	12 % 20 = 12	12	12
6	14	14 % 20 = 14	14	14
7	17	17 % 20 = 17	17	17
8	13	13 % 20 = 13	13	13
9	37	37 % 20 = 17	17	18

Fundamentals of data structures: Dictionary Implementation with Hash-Table

- Following are basic primary operations of a hashtable which are following.
  - *Search − search an element in a hashtable.*
  - *Insert − insert an element in a hashtable.*
  - *delete − delete an element from a hashtable*
- **DataItem** Define a data item having some data, and key based on which search is to be conducted in hashtable.

```
struct DataItem {
 int data;
int key;
};
```

Fundamentals of data structures: Dictionary Implementation with Hash-Table

- ***Hash Method*** Define a hashing method to compute the hash code of the key of the data item.

```
int hashCode(int key)
{
return key % SIZE;
}
```

Fundamentals of data structures: Dictionary Implementation with Hash-Table

- ***Insert Operation***
- Whenever an element is to be inserted.
- Compute the hash code of the key passed and locate the index using that hashcode as index in the array.
- Use linear probing for empty location if an element is found at computed hash code.

Fundamentals of data structures: Dictionary Implementation with Hash-Table

- ***Insert Operation***

```
void insert(int key,int data){
   struct DataItem *item = (struct DataItem*) malloc(sizeof(struct DataItem));
   item->data = data;
   item->key = key;

   //get the hash
   int hashIndex = hashCode(key);

   //move in array until an empty or deleted cell
   while(hashArray[hashIndex] != NULL && hashArray[hashIndex]->key != -1){
      //go to next cell
      ++hashIndex;

      //wrap around the table
      hashIndex %= SIZE;
   }

   hashArray[hashIndex] = item;
}
```

Fundamentals of data structures: Dictionary Implementation with Hash-Table

- **Delete Operation** Whenever an element is to be deleted.
- Compute the hash code of the key passed and locate the index using that hashcode as index in the array.
- Use linear probing to get element ahead if an element is not found at computed hash code.
- When found, store a dummy item there to keep performance of hashtable intact

Fundamentals of data structures: Dictionary Implementation with Hash-Table

```
struct DataItem* delete(struct DataItem* item){
   int key = item->key;

   //get the hash
   int hashIndex = hashCode(key);

   //move in array until an empty
   while(hashArray[hashIndex] !=NULL){

      if(hashArray[hashIndex]->key == key){
         struct DataItem* temp = hashArray[hashIndex];

         //assign a dummy item at deleted position
         hashArray[hashIndex] = dummyItem;
         return temp;
      }

      //go to next cell
      ++hashIndex;

      //wrap around the table
      hashIndex %= SIZE;
   }

   return NULL;
}
```

Fundamentals of data structures: Dictionary Implementation with Hash-Table

- ***Search Operation*** Whenever an element is to be searched.
- Compute the hash code of the key passed and locate the element using that hashcode as index in the array.
- Use linear probing to get element ahead if element not found at computed hash code.

Fundamentals of data structures: Dictionary Implementation with Hash-Table

- ***Search Operation***

```
struct DataItem *search(int key){
   //get the hash
   int hashIndex = hashCode(key);

   //move in array until an empty
   while(hashArray[hashIndex] != NULL){

      if(hashArray[hashIndex]->key == key)
         return hashArray[hashIndex];

      //go to next cell
      ++hashIndex;

      //wrap around the table
      hashIndex %= SIZE;
   }

   return NULL;
}
```

Fundamentals of data structures:

- ***SET:-*** A set is a collection of well defined elements. The members of a set are all different. A set is a group of "objects".
  - People in a class: { Alice, Bob, Chris }
  - Classes offered by a department: { CS 101, CS 202, … }
  - Colors of a rainbow: { red, orange, yellow, green, blue, purple }
  - States of matter { solid, liquid, gas, plasma }
  - States in the US: { Alabama, Alaska, Virginia, … }
  - Sets can contain non-related elements: { 3, a, red, Virginia }
- Although a set can contain anything, we will most often use sets of numbers
  - All positive numbers less than or equal to 5: {1, 2, 3, 4, 5}
  - A few selected real numbers: { 2.1, π, 0, -6.32, e }

Fundamentals of data structures:

- ***Properties of set :***
  - The set is defined by the capital letters
  - All the elements in the set are enclosed within { }
  - Every elements is separated by comma.
  - Eg: A={a,b,c,d}
- ***Representation of sets:***
- There are 3 types of representation sets
  - Tabular form/ Listing methods
  - Descriptive form / describe method
  - Set builder form/ recursive method

Fundamentals of data structures:

- ***Tabular Form:***
- Listing all the elements of a set and separated by commas and enclosed within curly brackets {}.

EX: $A=\{1,2,3,4,5\}$, $B=\{2,4,6,\ldots,50\}$, $C=\{1,3,5,7,9,\ldots\}$

- For example:

  (i) Let N denote the set of first five natural numbers.

  – Therefore, N = {1, 2, 3, 4, 5} → Roster Form

  (ii) The set of all vowels of the English alphabet.

  – Therefore, V = {a, e, i, o, u} → Roster Form

  (iii) The set of all odd numbers less than 9.

  – Therefore, X = {1, 3, 5, 7} → Roster Form

Fundamentals of data structures:

- ***Descriptive Form:***
- State in words the elements of a set. That is, the property of elements in the set defend as the set

EX:

A = Set ofthe first five natural numbers.

B = Set of positive even integers less or equal t

fifty

C = Set ofpositiveodd numbers.

(i) The set of odd numbers less than 7 is written as: {odd numbers less than 7}.

(ii) A set of football players with ages between 22 years to 30 years.

(iii) A set of numbers greater than 30 and smaller than 55.

Fundamentals of data structures:

- Set Builder Form:
- Writing in symbolic form the common characteristic shared by all the elements of the sets.

$$A=\{x:x\in\mathbb{N}\wedge x\leq 5\} \quad B=\{x:x\in E\wedge 0<x\leq 50\} \quad C=\{x:x\in O\wedge x>0\}$$

Complexity of Algorithms

- It is very convenient to classify algorithms based on the relative amount of time or relative amount of space they require and specify the growth of time /space requirements as a function of the input size.
- **Time Complexity**: Running time of the program as a function of the size of input.
- **Space Complexity**: Amount of computer memory required during the program execution, as a function of the input size.

Algorithm Analysis

- ***What is an algorithm?***
- Algorithm is a set of steps to complete a task. For example,
- Task: to make a cup of tea.
- Algorithm:
  - add water and milk to the kettle,
  - Boil it,
  - add tea leaves,
  - Add sugar,
  - and then serve it in cup

Algorithm Analysis

- What is Computer algorithm?
- a set of steps to accomplish or complete a task that is described precisely enough that a computer can run it.

Algorithm Analysis

- Characteristics of an algorithm:-
  - Must take an input.
  - Must give some output(yes/no, value etc.)
  - ***Definiteness–*** each instruction is clear and unambiguous.
  - ***Finiteness–*** algorithm terminates after a finite number of steps.
  - ***Effectiveness–*** every instruction must be basic i.e. simple instruction.

Algorithm Analysis

- An Algorithm is a sequence of steps to solve a problem.
- The Analysis of Algorithm is very important for designing algorithm to solve different types of problems in the branch of computer science and information technology.

Algorithm Analysis

- In the analysis of algorithms, it is common to estimate their complexity in the asymptotic sense.
- to estimate the complexity function for arbitrarily large input.

Algorithm Analysis

- Expectation from an algorithm
  - ***Correctness:-***
    - Correct: Algorithms must produce correct result. Produce an incorrect answer: Even if it fails to give correct results all the time still there is a control on how often it gives wrong result.
    - Approximation algorithm: Exact solution is not found, but near optimal solution can be found out.
  - ***Less resource usage:***
    - Algorithms should use less resources (time and space).

Algorithm Analysis

- The topic “Analysis of Algorithms” is concerned primarily with determining the memory (space) and time requirements (complexity) of an algorithm.
- The time complexity (or simply, complexity) of an algorithm is measured as a function of the problem size.

Algorithm Analysis

- Expectation from an algorithm
  - ***Resource usage:***
    - The time is considered to be the primary measure of efficiency.
    - We are also concerned with how much the respective algorithm involves the computer memory.
    - But mostly time is the resource that is dealt with.
    - And the actual running time depends on a variety of backgrounds: like the speed of the Computer, the language in which the algorithm is implemented, the compiler/interpreter, skill of the programmers etc.
    - mainly the resource usage can be divided into:
      1. Memory (space)
      2. Time

Algorithm Analysis

- Time taken by an algorithm?
  - ***performance measurement*** or Apostoriori Analysis:
    - Implementing the algorithm in a machine and then calculating the time taken by the system to execute the program successfully.
  - ***Performance Evaluation*** or Apriori Analysis.
    - Before implementing the algorithm in a system. This is done as follows

Algorithm Analysis

- Time taken by an algorithm?
  - How long the algorithm takes :-
    - will be represented as a function of the size of the input.
    - f(n) → how long it takes if ‘n’ is the size of input.
  - How fast the function that characterizes the running time grows with the input size.
    - “Rate of growth of running time”.
    - The algorithm with less rate of growth of running time is considered better.

Algorithm Analysis

- Some examples are given below.

  1. The complexity of an algorithm to sort n elements may be given as a function of n.

  2. The complexity of an algorithm to multiply an m×n matrix and an n×p matrix may be given as a function of m, n, and p.

  3. The complexity of an algorithm to determine whether x is a prime number may be given as a function of the number, n, of bits in x. Note that n = log2(x+ 1).

Algorithm Analysis

- We partition our discussion of algorithm analysis into the following sections.
  1. Operation counts.
  2. Step counts.
  3. Counting cache misses.
  4. Asymptotic complexity.
  5. Recurrence equations.
  6. Amortized complexity.
  7. Practical complexities.

Algorithm Analysis

- Operation counts:
  - One way to estimate the time complexity of a program or method is to select one or more operations, such as add, multiply, and compare, and to determine how many of each is done.
  - The success of this method depends on our ability to identify the operations that contribute most to the time complexity.

Algorithm Analysis

- Operation counts:
  - Finding the position of the largest element in a[0:n-1].

```
int max(int a[],int n)
{
if(n<1) return -1;
int positionof current max=0;
for (int i=1;i<n;i++)
if(a[positionofcurrent max]<a[i])
positionofcurrentmax=I;
return positionofcurrentmax;
}
```

Algorithm Analysis

- Operation counts:
  - an algorithm that returns the position of the largest element in the array a[0:n-1].
  - When $n > 0$, the time complexity of this algorithm can be estimated by determining the number of comparisons made between elements of the array a.
  - When $n \leq 1$, the for loop is not entered.
  - So no comparisons between elements of a are made.
  - When $n > 1$, each iteration of the for loop makes one comparison between two elements of a, and the total number of element comparisons is n-1.
  - The number of element comparisons is $\max\{n-1, 0\}$

Algorithm Analysis

- Operation counts:
- Sequential search.

```
int sequentialSearch(int [] a, int n, int x)
{
// search a[0:n-1] for x
int i;
for (i = 0; i < n && x != a[i]; i++);
if (i == n) return -1; // not found
else return i;
}
```

Algorithm Analysis

- Operation counts:
- Sequential search.
  - an algorithm that searches a[0:n-1] for the first occurrence of x.
  - The number of comparisons between x and the elements of a isn't uniquely determined by the problem size n.
  - For example, if n = 100 and x = a[0], then only 1 comparison is made.
  - However, if x isn't equal to any of the a[i]s, then 100 comparisons are made.
  - A search is successful when x is one of the a[i]s. All other searches are unsuccessful.
  - Whenever we have an unsuccessful search, the number of comparisons is n.
  - For successful searches the best comparison count is 1, and the worst is n.

Algorithm Analysis

- Step Counts:
  - In the step-count method, we attempt to account for the time spent in all parts of the algorithm.
  - A step is any computation unit that is independent of the problem size.
  - Thus 10 additions can be one step;
  - 100 multiplications can also be one step;
  - but n additions, where n is the problem size, cannot be one step.
  - The amount of computing represented by one step may be different from that represented by another

Algorithm Analysis

- Step Counts:
  - return a+b+b*c+(a+b-c)/(a+b)+4;
  - can be regarded as a single step if its execution time is independent of the problem size.
  - We may also count a statement such as
  - x = y;
  - as a single step

Algorithm Analysis

- Step Counts:
  - To determine the step count of an algorithm, we first determine the number of steps per execution (s/e) of each statement and the total number of times (i.e., frequency) each statement is executed.
  - Combining these two quantities gives us the total contribution of each statement to the total step count.
  - We then add the contributions of all statements to obtain the step count for the entire algorithm.

Algorithm Analysis

- Step Counts: Best-case step count

Statement	Step per execution	Frequency	Total steps
int sequentialSearch(int [] a, int n, int x)	0	0	0
{	0	0	0
int i;	1	1	1
for(i = 0; i < n && x != a[i]; i++);	1	1	1
if(i == n) return -1; // not found	1	1	1
else return i;	1	1	1
}	0	0	0
Total			4

Algorithm Analysis

- Step Counts: Worst-case step count

Statement	Step per execution	Frequency	Total steps
int sequentialSearch(int [] a, int n, int x)	0	0	0
{	0	0	0
int i;	1	1	1
for(i = 0; i < n && x != a[i]; i++);	1	n+1	n+1
if(i == n) return -1; // not found	1	1	1
else return i;	1	0	0
}	0	0	0
Total			n+3

Algorithm Analysis

- Asymptotic Notations are languages that allow us to analyze an algorithm's running time by identifying its behavior as the input size for the algorithm increases.
- This is also known as an algorithm's growth rate.
- The word **Asymptotic** means approaching a value or curve arbitrarily closely (i.e., as some sort of limit is taken).

Asymptotic Notations

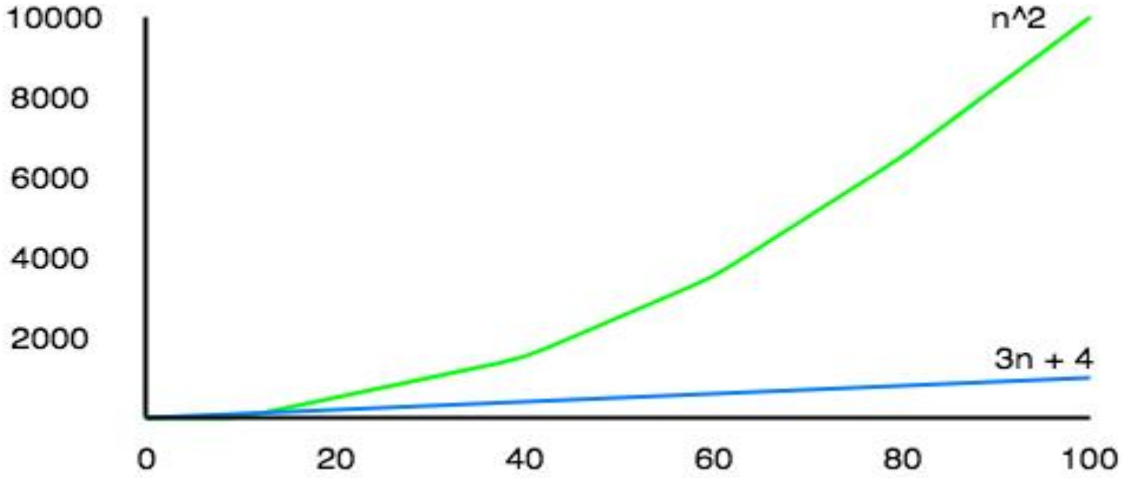
- **Asymptotic Notations** are the expressions that are used to represent the complexity of an algorithm.
- When it comes to analysing the complexity of any algorithm in terms of time and space, we can never provide an exact number to define the time required and the space required by the algorithm, instead we express it using some standard notations, also known as **Asymptotic Notations**.

Asymptotic Notations

- When we analyse any algorithm, we generally get a formula to represent the amount of time required for execution or the time required by the computer to run the lines of code of the algorithm, number of memory accesses, number of comparisons, temporary variables occupying memory space etc.

Asymptotic Notations

- If some algorithm has a time complexity of $T(n) = (n^2 + 3n + 4)$, which is a quadratic equation.
- For large values of n, the 3n + 4 part will become insignificant compared to the n^2 part.



For n = 1000, n^2 will be 1000000 while 3n + 4 will be 3004.

Asymptotic Notations

- When we compare the execution times of two algorithms the constant coefficients of higher order terms are also neglected.
- An algorithm that takes a time of $200n^2$ will be faster than some other algorithm that takes n^3 time, for any value of n larger than 200

Asymptotic Notations

- there are three **types of analysis** that we perform on a particular algorithm.
- **Best Case**: In which we analyse the performance of an algorithm for the input, for which the algorithm takes less time or space.
- **Worst Case**: In which we analyse the performance of an algorithm for the input, for which the algorithm takes long time or space.
- **Average Case**: In which we analyse the performance of an algorithm for the input, for which the algorithm takes time or space that lies between best and worst case.

Types of Data Structure Asymptotic Notation

1. **Big-O Notation (O)** – Big O notation specifically describes worst case scenario.
2. **Omega Notation (Ω)** – Omega(Ω) notation specifically describes best case scenario.
3. **Theta Notation (θ)** – This notation represents the average complexity of an algorithm.

Big-O Notation (O)

- Big O notation specifically describes worst case scenario.
- It represents the upper bound running time complexity of an algorithm.
- the longest amount of time an algorithm can possibly take to complete.. It provides us with an asymptotic upper bound for the growth rate of run-time of an algorithm.
- Lets take few examples to understand how we represent the time and space complexity using **Big O notation**.

Big-O Notation (O)

- **O(1)**
  - Big O notation O(1) represents the complexity of an algorithm that always execute in same time or space regardless of the input data.
  - **example**
    The following step will always execute in same time(or space) regardless of the size of input data.
    - Accessing array index(int num = arr[5])
    -

```
void printFirstElementOfArray(int arr[])
{
    printf("First element of array = %d",arr[0]);
}
```

This function runs in O(1) time (or "constant time") relative to its input. The input array could be 1 item or 1,000 items, but this function would still just require one step.

Big-O Notation (O)

- **O(n)**
  - Big O notation O(N) represents the complexity of an algorithm, whose performance will grow linearly (in direct proportion) to the size of the input data.
  - **O(n)example**

```
void printAllElementOfArray(int arr[], int size)
{
    for (int i = 0; i < size; i++)
    {
        printf("%d\n", arr[i]);
    }
}
```

  - This function runs in O(n) time (or "linear time"), where n is the number of items in the array. If the array has 10 items, we have to print 10 times. If it has 1000 items, we have to print 1000 times.

Big-O Notation (O)

- **O(n^2)**
  - Big O notation O(n^2) represents the complexity of an algorithm, whose performance is directly proportional to the square of the size of the input data.
  - **O(n^2) example**
    - Traversing a 2D array

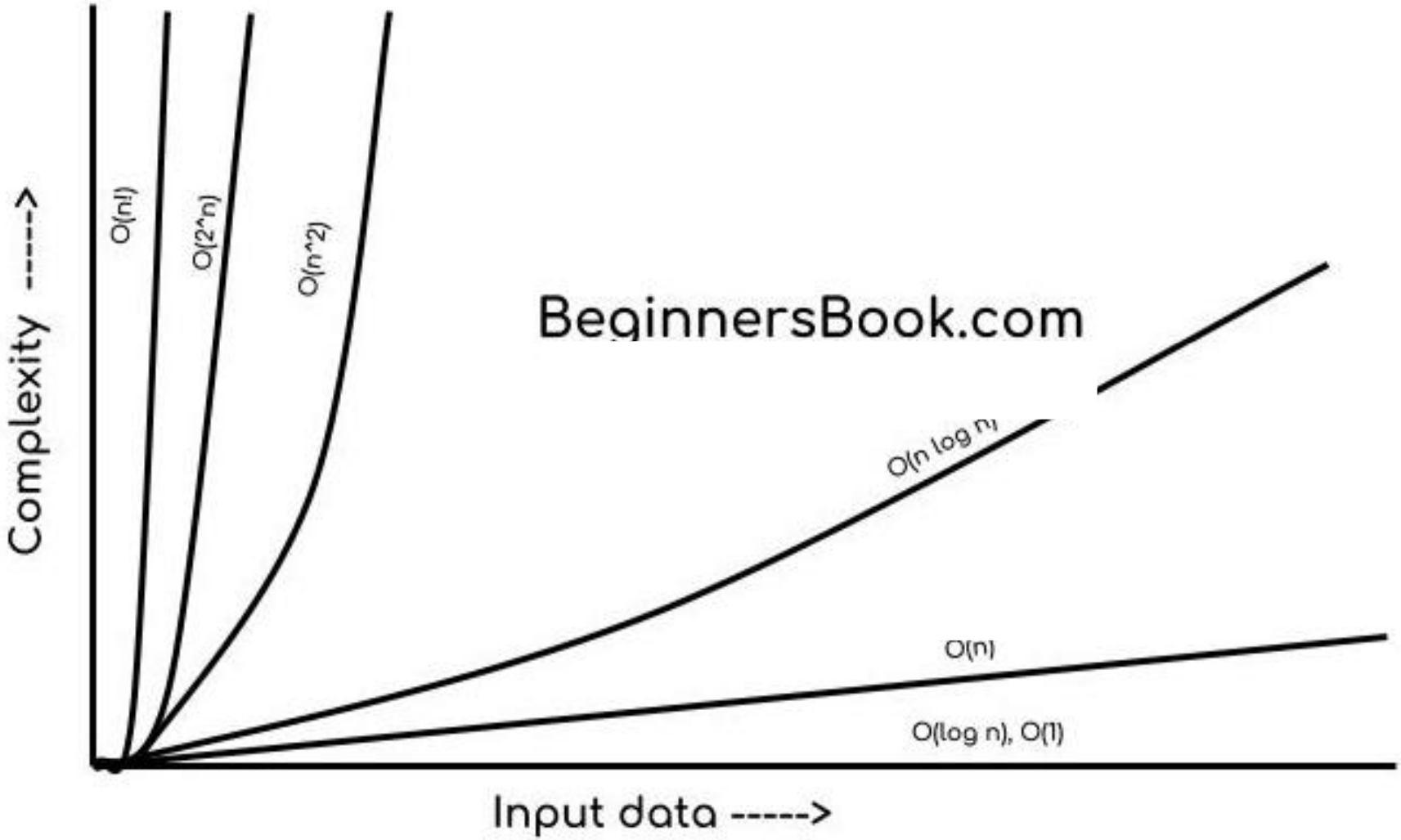
```
void printAllPossibleOrderedPairs(int arr[], int size)
{
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            printf("%d = %d\n", arr[i], arr[j]);
        }
     }
}
```

Big-O Notation (O)

- **O(n^2)**
  - Here we're nesting two loops. If our array has n items, our outer loop runs n times and our inner loop runs n times for each iteration of the outer loop, giving us n^2 total prints. Thus this function runs in $O(n^2)$ time (or "quadratic time"). If the array has 10 items, we have to print 100 times. If it has 1000 items, we have to print 1000000 times.

```
void printAllPossibleOrderedPairs(int arr[], int size)
{
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            printf("%d = %d\n", arr[i], arr[j]);
        }
    }
}
```

Big-O Notation (O)



Big-O Notation (O)

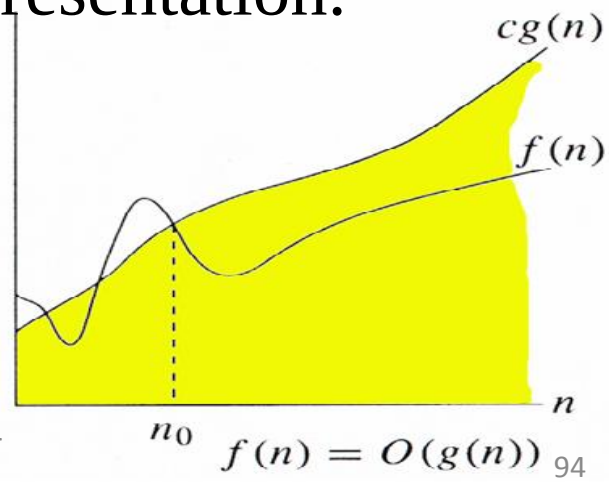
- It provides us with an ***asymptotic upper bound*** for the growth rate of runtime of an algorithm.
- Say f(n) is your algorithm runtime, and g(n) is an arbitrary time complexity you are trying to relate to your algorithm.
- A function ***f(n)*** can be represented is the order of ***g(n)*** that is ***O(g(n))*.**
- f(n) is O(g(n)), if for some real constants c (c > 0) and n_0, f(n) <= c g(n) for every input size n (n > n_0).

Big-O Notation (O)

- It tells us that a certain function will never exceed a specified time for any value of input n.
- Consider Linear Search algorithm, in which we traverse an array elements, one by one to search a given number.
- starting from the front of the array, we find the element or number we are searching for at the end, which will lead to a time complexity of n, where n represents the number of total elements.

Big-O Notation (O)

- But it can happen, that the element that we are searching for is the first element of the array, in which case the time complexity will be 1.
- when we use the big-O notation, we mean to say that the time complexity is O(n), which means that the time complexity will never exceed n, defining the upper bound, hence saying that it can be less than or equal to n, which is the correct representation.



Big-O Notation (O)

- For example

  f(n)=3n+2 g(n)=n

  f(n)=o(g(n)) means that f(n) is smaller than g(n)

  f(n)<=c*g(n)

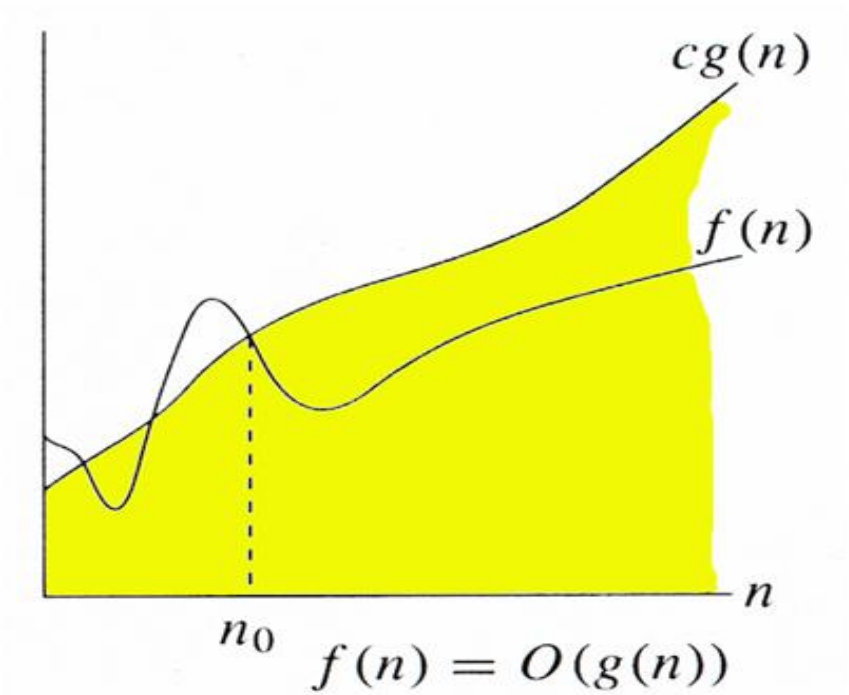
  3n+2<=c*n where c=4

  3n+2<=4*n

  n>=2

  3*2+2<=4*2

  8<=8



Omega Notation (Ω)

- Omega notation specifically describes best case scenario.
- It represents the lower bound running time complexity of an algorithm.
- So if we represent a complexity of an algorithm in Omega notation, it means that the **algorithm cannot be completed in less time than this.**
- It provides us with an ***asymptotic lower bound*** for the growth rate of runtime of an algorithm.

Omega Notation (Ω)

- This always indicates the minimum time required for any algorithm for all input values, therefore the best case of any algorithm.
- In simple words, when we represent a time complexity for any algorithm in the form of big-Ω, we mean that the algorithm will take atleast this much time to complete it's execution.

Omega Notation (Ω)

- The actual time complexity of the function which is determined by the time for an algorithm is increased
- Now you want to give a lower bound to that function i.e g(n) in such a way that c*g(n) is less then f(n) after some value of n i.e n_o.

  Which means that f(n)>=c*g(n)

  After some value of n i.e n>=n_o

  Where c is a constant if c>0 &n_o is an input i.e n_o>=1

Omega Notation (Ω)

- f(n)=3n+2 g(n)=n

  Can the function f(n) be bounded by g(n) which means f(n) has lower bound as g(n)

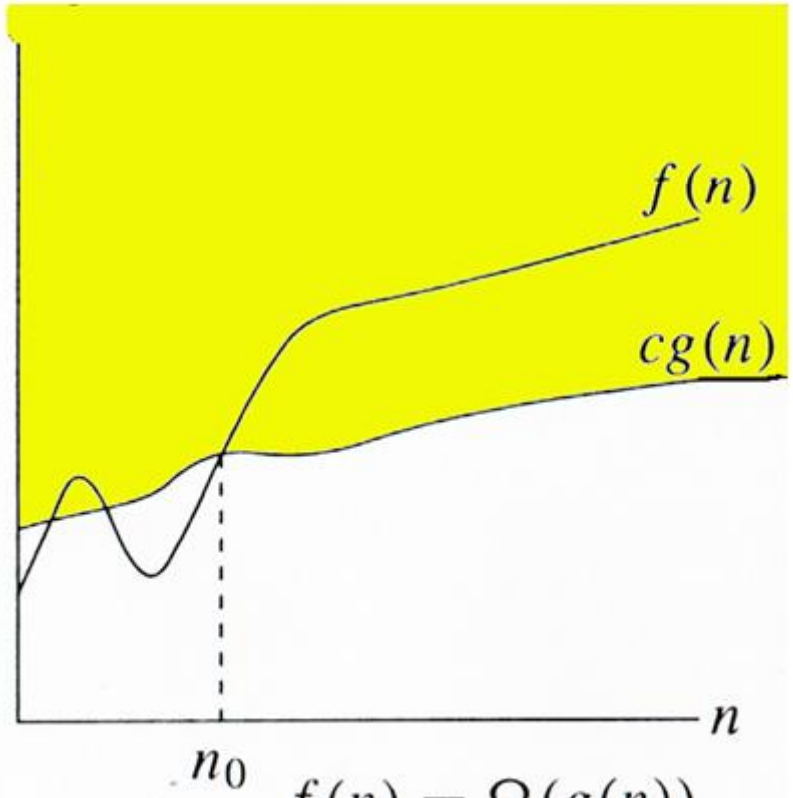
  f(n)=Ω g(n)

  f(n) >=c*g(n)

  3n+2 >=c*n where c=1,n_0>=1

  3*1+2 >=1* 1

  3n+2 >= Ω (n)



$f(n) = \Omega(g(n))$

Omega Notation (Ω)

- f(n)=3n+2 g(n)= n^2
- Can we check

  f(n)=Ω g(n)

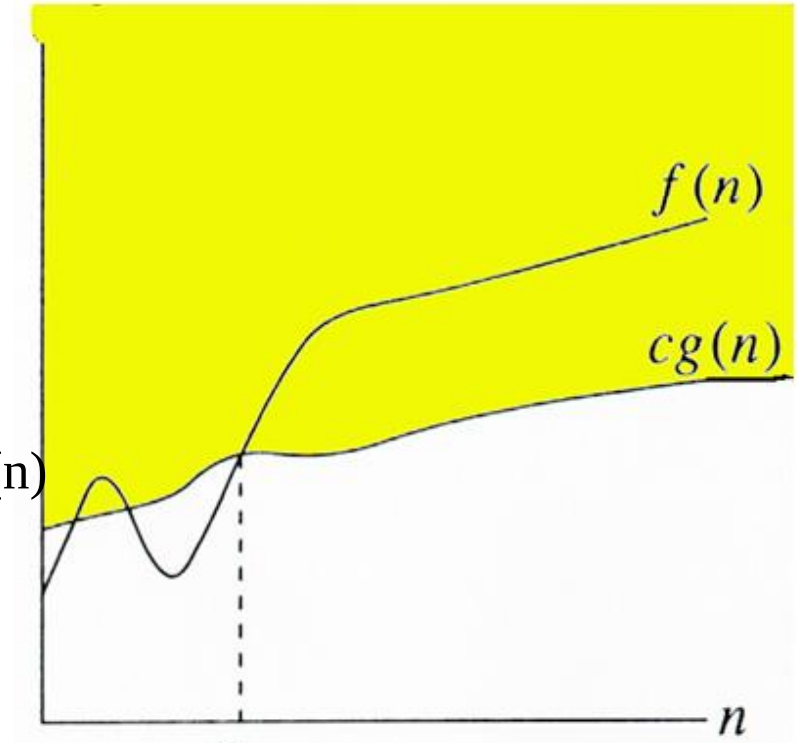
  f(n) >=c*g(n)

  3n+2 >=c* n^2

  3*4+2 >=1* 4^2

  3n+2 >= Ω (n)

  where c=1, n_0>=4
- Can the f(n) is lower bounded by g(n)?
- the f(n) can never be lower bounded by g(n)
- f(n)=Ω g(n) then any thing less then n can be lower bounded as Log n ,log log n…..



$f(n) = \Omega(g(n))$

Theta Notation (θ)

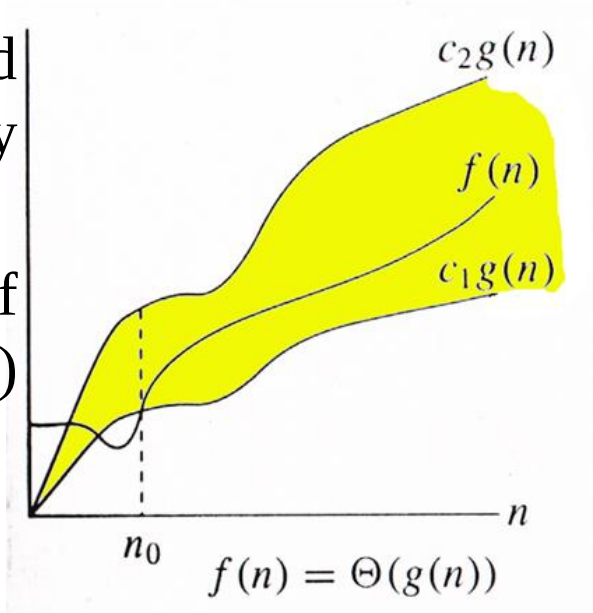
- This notation describes both upper bound and lower bound of an algorithm so we can say that it defines exact asymptotic behaviour.
- In the real case scenario the algorithm not always run on best and worst cases, the average running time lies between best and worst and can be represented by the theta notation.

Theta Notation (θ)

- Theta commonly written as **Θ** is an Asymptotic Notation to denote the ***asymptotically tight bound*** on the growth rate of runtime of an algorithm.

Theta Notation (θ)

- If we have a function f(n) then we should find the upper and lower bound by a function just by the value of some constant.
- If f(n) is bounded by c1*g(n) and c2*g(n) then we can say that f(n) is **θ (g(n)).**
- So the constants c1 &c2 could be different and moreover after a value we could taken any value
- Which means that after the value of no both of them are c1*g(n) less then f(n) and c2*g(n) greater then f(n)



Theta Notation (θ)

- F(n)= g(n) if f(n) is bounded by g(n) both in the lower and upper
- C1*g(n)<=f(n)<=c2g(n) where c1,c2>0 n>=no no>=1

f(n)=o(g(n)) means that f(n) is smaller than g(n) i.e upper bound

f(n)<=c*g(n)

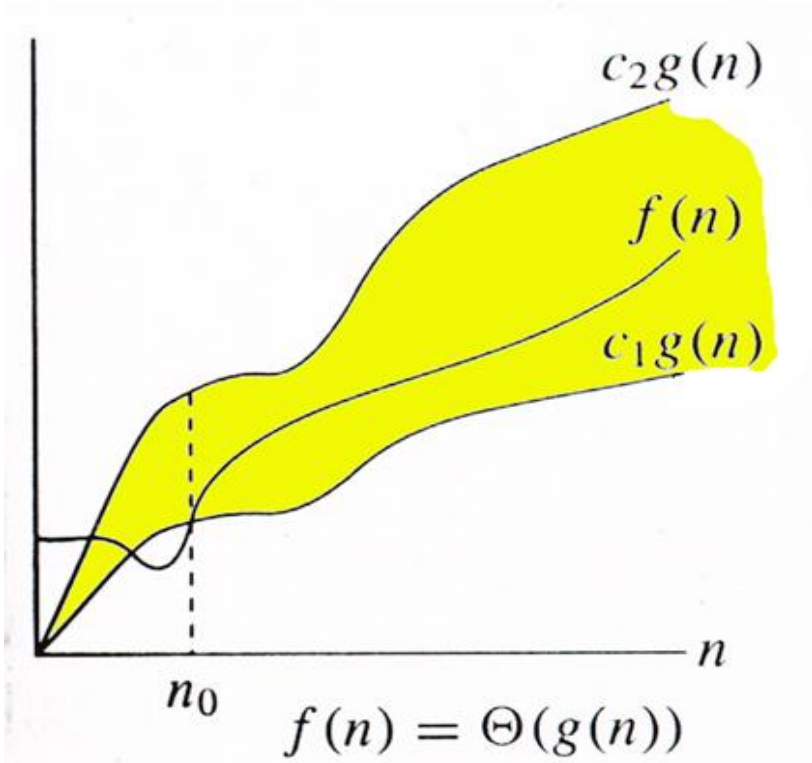
3n+2<=c*n

3n+2<=4*n

n>=2

3*2+2<=4*2

8<=8



Theta Notation (θ)

- F(n)= g(n) if f(n) is bounded by g(n) both in the lower and upper
- C1*g(n)<=f(n)<=c2g(n) where c1,c2>0 n>=no no>=1

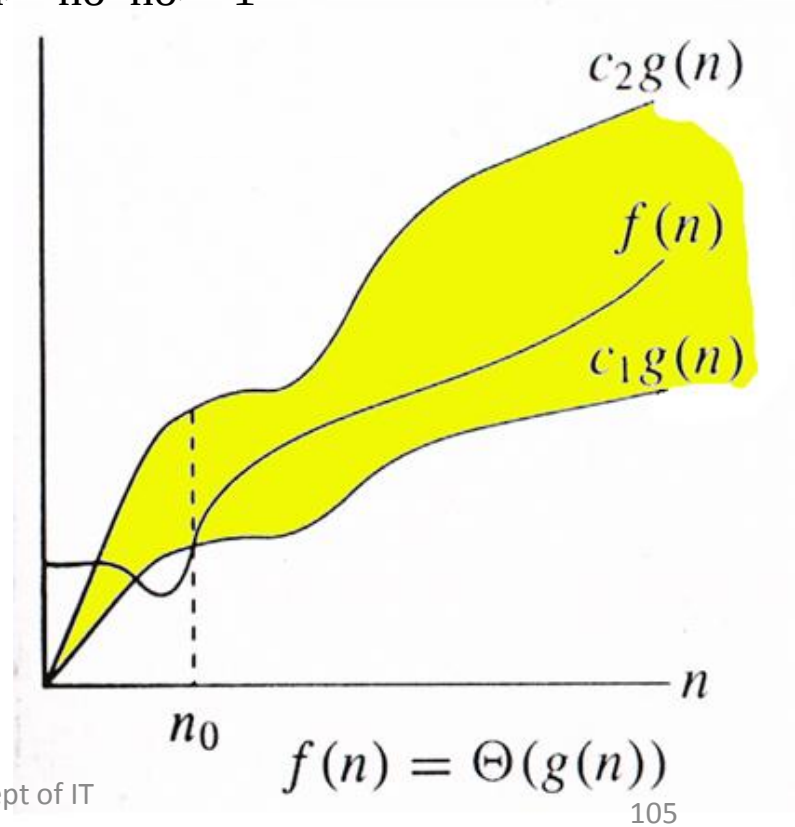
f(n)=Ω g(n)

f(n) >=c*g(n)

3n+2 >=c*n where c=1,n_o>=1

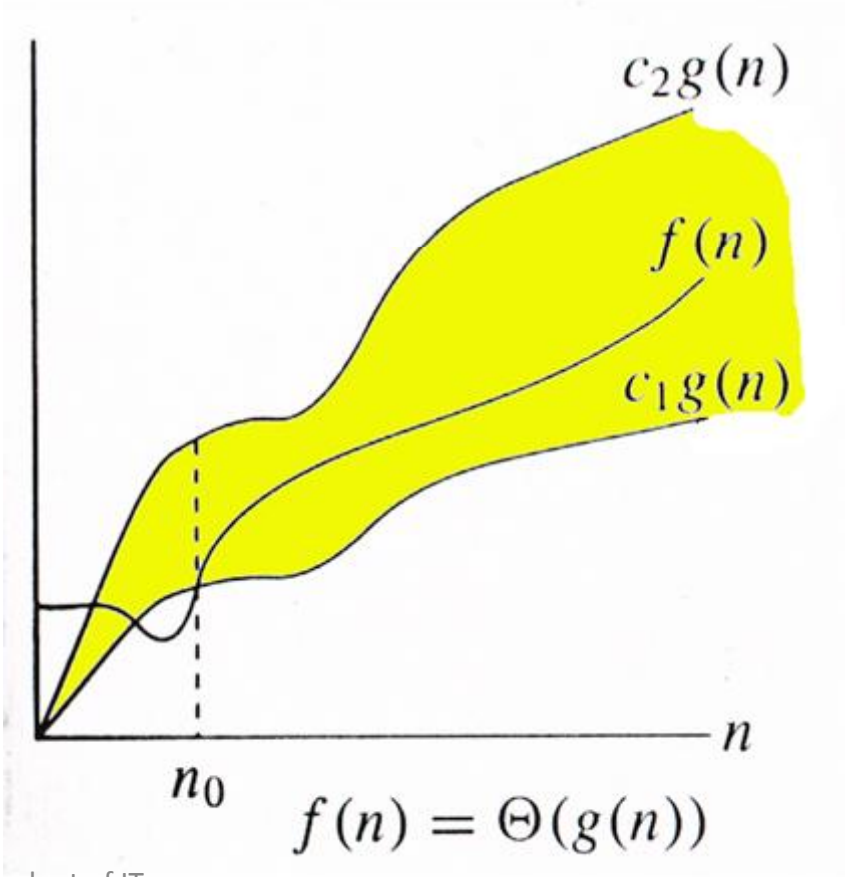
3*1+2 >=1* 1

3n+2 >= Ω (n)



Theta Notation (θ)

- https://www.youtube.com/watch?v=aGjL7YXI31Q



Amortized Analysis

- In computer science, **amortized analysis** is a method for analyzing a given algorithm's complexity, or how much of a resource, especially time or memory, it takes to execute.
- **Amortized analysis** is a method of analyzing the costs associated with a data structure that averages the worst operations out over time.
- a data structure has one particularly costly operation, but it doesn't get performed very often.

Amortized Analysis

- In the Hash-table the most of the time the searching time complexity is O(1) but sometimes it executes O(n) operations.
- When we want to search or insert an element in a hash table for most of the cases it is constant time taking the task but when a collision occurs it needs O(n) times operations for collision resolution.

Amortized Analysis

- Cake-making is pretty complex but it's essentially two main steps:
  - Mix batter (fast).
  - Bake in an oven (slow, and you can only fit one cake in at a time).
- Mixing the batter takes relatively little time when compared with baking. Afterwards, you reflect on the cake-making process.
- When deciding if it is slow, medium, or fast, you choose medium because you average the two operations—slow and fast—to get medium.

Amortized Analysis

- There are three main types of amortized analysis:
  - aggregate analysis
  - the accounting method and
  - the potential method.

What is Hashing

- Hashing is an algorithm (via a hash function) that maps large data sets of variable length, called keys, to smaller data sets of a fixed length
- A hash table (or hash map) is a data structure that uses a hash function to efficiently map keys to values, for efficient search and retrieval
- Map large integers to smaller integers
- Map non-integer keys to integers

What is Hashing

- Widely used in many kinds of computer software, particularly for associative arrays, database indexing, caches, and sets

Operations	Sorted Array	Balanced BST	Hashing
Insertion	O(n)	O(log n)	O(1) avg
Deletion	O(n)	O(log n)	O(1) avg
Retrieval	O(log n)	O(log n)	O(1) avg

Hash Functions

- simple/fast to compute,
- Avoid collisions
- have keys distributed evenly among cells
- Each uses a hash table for average complexity to insert , erase, and find in O(1)
- hash function is a one-to-one mapping between keys and hash values. So no collision occurs

characteristics of a good hash function

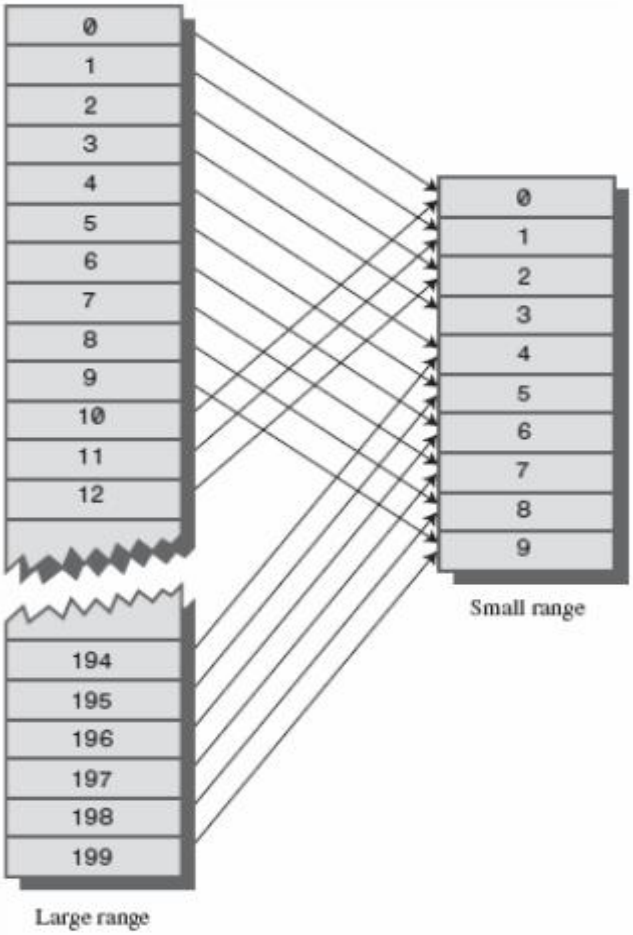
- The characteristics of a good hash function are as follows.
  - It avoids collisions.
  - It tends to spread keys evenly in the array.
  - It is easy to compute (i.e., computational time of a hash function should be O(1)).

Collision Resolution

- **Collision:** when two keys map to the same location in the hash table.
- Collisions occur when two keys, k1 and k2, are not equal, but h(k1) = h(k2).
- Two ways to resolve collisions:
  - Separate Chaining (open hashing)
  - Open Addressing (linear probing, quadratic probing, double hashing) (closed hashing)

Several approaches for dealing with collisions are

- Example: K = {0, 1, ..., 199}, M = 10, for each key k in K, f(k) = k % M



Pigeon Hole Principle

- The **pigeonhole principle** states that if n items are put into m containers, with n>m, then at least one container must contain more than one item.

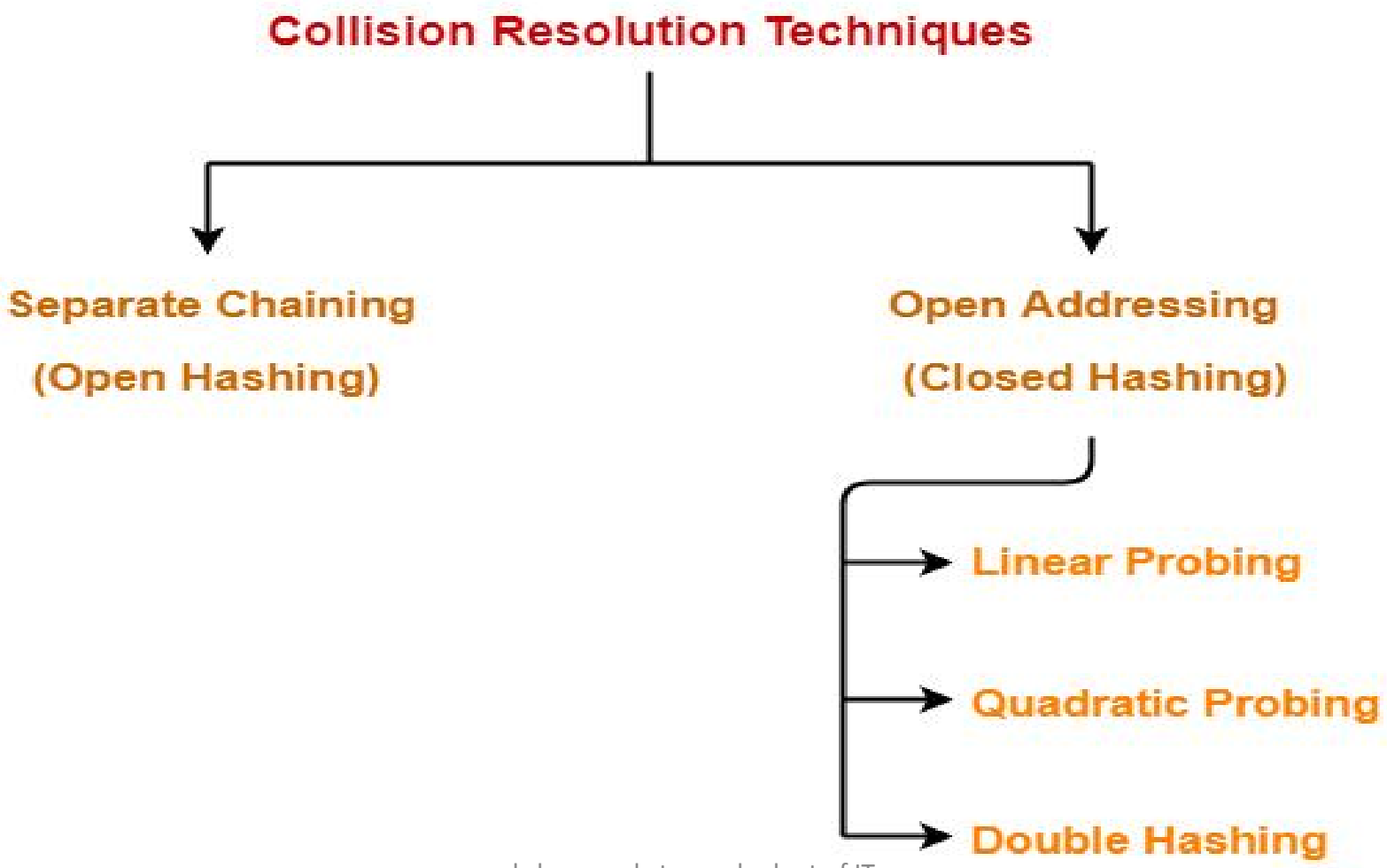
Pigeon Hole Principle

- Pigeons in holes. Here there are $n = 10$ pigeons in $m = 9$ holes. Since 10 is greater than 9, the pigeonhole principle says that at least one hole has more than one pigeon

Pigeon Hole Principle

- Recall for hash tables we let…
  - n = # of entries (i.e. keys)
  - m = size of the hash table
- If n > m, is every entry in the table used?
  - No. Some may be blank?
- Is it possible we haven't had a collision?
  - No. Some entries have hashed to the same location
- Pigeon Hole Principle says given n items to be slotted into m holes and n > m there is at least one hole with more than 1 item
- So if n > m, we know we've had a collision
- We can only avoid a collision when n < m

Collision Resolution



Collusion Resolution Methods

- Three methods in open addressing are linear probing, quadratic probing, and double hashing.
- These methods are of the division hashing method because the hash function is f(k) = k % M.
- Some other hashing methods are middle-square hashing method, multiplication hashing method, and Fibonacci hashing method, and so on.

Linear Probing Method

- The hash table in this case is implemented using an array containing M nodes, each node of the hash table has a field k used to contain the key of the node.
- M can be any positive integer but M is often chosen to be a prime number.
- When the hash table is initialized, all fields k are assigned to -1.

Linear Probing Method

- When a node with the key k needs to be added into the hash table, the hash function

$$f(k) = k \% M$$

- will specify the address $i = f(k)$ (i.e., an index of an array) within the range $[0, M - 1]$.

Linear Probing Method

- If there is no conflict, then this node is added into the hash table at the address i.
- If a conflict takes place, then the hash function rehashes first time f 1 to consider the next address (i.e., i + 1).
- If conflict occurs again, then the hash function rehashes second time f 2 to examine the next address (i.e., i + 2).
- This process repeats until the available address found then this node will be added at this address.

Linear Probing Method

- The rehash function at the time t (i.e., the collision number t = 1, 2, ...) is presented as follows

$$f_t(k) = (f(k) + t) \% M = (i + t) \% M$$

- When searching a node, the hash function f(k) will identify the address i (i.e., i = f(k)) falling between 0 and M - 1.

Linear Probing Method

- Let us consider a simple hash function as “key mod 7” and sequence of keys as 50, 700, 76, 85, 92, 73, 101.

Step-01:

Draw the hash table

For the given hash function, the possible range of hash values is [0, 6].

So, draw an empty hash table consisting of 7 buckets as

Linear Probing Method

- Let us consider a simple hash function as “key mod 7” and sequence of keys as 50, 700, 76, 85, 92, 73, 101.

Step-02: Insert the given keys in the hash table one by one.
The first key to be inserted in the hash table = 50.
Bucket of the hash table to which key 50 maps = 50 mod 7 = 1.
So, key 50 will be inserted in bucket-1 of the hash table as

0	
1	50
2	
3	
4	
5	
6	

Linear Probing Method

- Let us consider a simple hash function as “key mod 7” and sequence of keys as 50, 700, 76, 85, 92, 73, 101.

Step-03:

The next key to be inserted in the hash table = 700.
Bucket of the hash table to which key 700 maps = 700 mod 7 = 0.
So, key 700 will be inserted in bucket-0 of the hash table as-

0	700
1	50
2	
3	
4	
5	
6	

Linear Probing Method

- Let us consider a simple hash function as “key mod 7” and sequence of keys as 50, 700, 76, 85, 92, 73, 101.

Step-04:

The next key to be inserted in the hash table = 76.
Bucket of the hash table to which key 76 maps = 76 mod 7 = 6.
So, key 76 will be inserted in bucket-6 of the hash table as-

0	700
1	50
2	
3	
4	
5	
6	76

Linear Probing Method

Step-05: The next key to be inserted in the hash table = 85.
Bucket of the hash table to which key 85 maps = 85 mod 7 = 1.
Since bucket-1 is already occupied, so collision occurs.
To handle the collision, linear probing technique keeps probing linearly until an empty bucket is found.
The first empty bucket is bucket-2.
So, key 85 will be inserted in bucket-2 of the hash table as-

0	700
1	50
2	85
3	
4	
5	
6	76

$$f_t(k) = (f(k) + t) \% M = (i + t) \% M$$

Linear Probing Method

Step-06: The next key to be inserted in the hash table = 92.
Bucket of the hash table to which key 92 maps = 92 mod 7 = 1.
Since bucket-1 is already occupied, so collision occurs.
To handle the collision, linear probing technique keeps probing linearly until an empty bucket is found.
The first empty bucket is bucket-3.
So, key 92 will be inserted in bucket-3 of the hash table as

0	700
1	50
2	85
3	92
4	
5	
6	76

$$f_t(k) = (f(k) + t) \% M = (i + t) \% M$$

Linear Probing Method

Step-07: The next key to be inserted in the hash table = 73.
Bucket of the hash table to which key 73 maps = 73 mod 7 = 3.
Since bucket-3 is already occupied, so collision occurs.
To handle the collision, linear probing technique keeps probing linearly until an empty bucket is found.
The first empty bucket is bucket-4.
So, key 73 will be inserted in bucket-4 of the hash table as-

0	700
1	50
2	85
3	92
4	73
5	
6	76

$$f_t(k) = (f(k) + t) \% M = (i + t) \% M$$

Linear Probing Method

Step-08: The next key to be inserted in the hash table = 101.
Bucket of the hash table to which key 101 maps = 101 mod 7 = 3.
Since bucket-3 is already occupied, so collision occurs.
To handle the collision, linear probing technique keeps probing linearly until an empty bucket is found.
The first empty bucket is bucket-5.
So, key 101 will be inserted in bucket-5 of the hash table as

0	700
1	50
2	85
3	92
4	73
5	101
6	76

$$f_t(k) = (f(k) + t) \% M = (i + t) \% M$$

Linear Probing Method

- Example: insert keys 32, 53, 22, 92, 17, 34, 24, 37, and 56 into a hash table of size M = 10
  1. insert keys 32 into a hash table of size M = 10

Linear Probing Method

insert keys 32 into a hash table of size M = 10 i.e M-1=9

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Hash Functions Distribute keys to locations in hash table

Hash function is then applied to the integer value 32 such that it maps to a value between 0 to M-1 where M is the table size then modulo hashing is used

$$f(k) = k \% M$$ Here k=32 M=10

$$f(k) = 32 \% 10 = 2$$

will specify the address i = f(k) (i.e., an index of an array) within the range [0, M - 1].
Index position i =2 then insert 32 in 3 position

Linear Probing Method

insert keys 32 into a hash table of size M = 10 i.e M-1=9

0	
1	
2	32
3	
4	
5	
6	
7	
8	
9	

Hash Functions Distribute keys to locations in hash table

Hash function is then applied to the integer value 32 such that it maps to a value between 0 to M-1 where M is the table size then modulo hashing is used

$$f(k) = k \% M$$ Here k=32 M=10

$$f(k) = 32 \% 10 = 2$$

will specify the address i = f(k) (i.e., an index of an array) within the range [0, M - 1].
Index position i =2 then insert 32 in 3 position

Linear Probing Method

insert keys 53 into a hash table of size M = 10 i.e M-1=9

0	
1	
2	32
3	53
4	
5	
6	
7	
8	
9	

Hash Functions Distribute keys to locations in hash table

Hash function is then applied to the integer value 53 such that it maps to a value between 0 to M-1 where M is the table size then modulo hashing is used

$$f(k) = k \% M$$

Here k=53 M=10

$$f(k) = 53 \% 10 = 3$$

will specify the address i = f(k) (i.e., an index of an array) within the range [0, M - 1].
Index position i =3 then insert 32 in 4 position

Linear Probing Method

insert keys 22 into a hash table of size M = 10 i.e M-1=9

0	
1	
2	32/22
3	53
4	
5	
6	
7	
8	
9	

Hash Functions Distribute keys to locations in hash table

Hash function is then applied to the integer value 22 such that it maps to a value between 0 to M-1 where M is the table size then modulo hashing is used

$$f(k) = k \% M$$ Here k=22 M=10

$$f(k) = 22 \% 10 = 2$$

will specify the address i = f(k) (i.e., an index of an array) within the range [0, M - 1].
Index position i = then insert 32 in 2 position

If a conflict takes place, then the hash function rehashes first time f 1 to consider the next address

$$f_t(k) = (f(k) + t) \% M = (i + t) \% M$$

Linear Probing Method

insert keys 22 into a hash table of size M = 10 i.e M-1=9

Then must be probe (move) for one time for finding empty slot

0	
1	
2	32/22
3	53
4	
5	
6	
7	
8	
9	

$f(k) = k \% M$ Here k=22 M=10

$f(k) = 22 \% 10 = 2$

will specify the address i = f(k) (i.e., an index of an array) within the range [0, M - 1].
Index position i = then insert 32 in 2 position

If a conflict takes place, then the hash function rehashes first time f 1 to consider the next address

$$f_t(k) = (f(k) + t) \% M = (i + t) \% M$$

Quadratic probing

- Quadratic probing operates by taking the original hash index and adding successive values of an arbitrary quadratic polynomial until an open slot is found.
- An example sequence using quadratic probing is:

$$H + 1^2, H + 2^2, H + 3^2, H + 4^2, \ldots, H + k^2$$

Quadratic probing

- it better avoids the clustering problem that can occur with linear probing.
- Let h(k) be a hash function that maps an element k to an integer in [0,m-1], where m is the size of the table.
- Let the ith probe position for a value k be given by the function

$$h(k,i) = (h(k) + c_1 i + c_2 i^2) \pmod{m}$$

Quadratic probing

- When a node with the key k needs to be added into the hash table, the hash function

$$f(k) = k \% M$$

- will specify the address i within the range [0, M - 1] (i.e., i = f(k))

Quadratic probing

- If there is no conflict, then this node is added into the hash table at the address i.
- If a conflict takes place, then the hash function rehashes first time f 1 to consider the address f(k) + 1^2
- If conflict occurs again, then the hash function rehashes second time f 2 to examine the address f(k) +2^2
- This process repeats until the available address found then this node will be added at this address.

Quadratic probing

- The rehash function at the time t (i.e., the collision number t = 1, 2, ...) is presented as follows.

$$f_t(k) = (f(k) + t^2) \% M = (i + t^2) \% M$$

- When searching a node, the hash function f(k) will identify the address i (i.e., i = f(k)) falling between 0 and M - 1

Quadratic probing

- Example: insert the keys :76,40,48,5,20

Step-01:

Draw the hash table

For the given hash function, the possible range of hash values is [0, 6].

So, draw an empty hash table consisting of 7 buckets as

0	
1	
2	
3	
4	
5	
6	

Quadratic probing

- Example: insert the keys :76,40,48,5,20

Step-01:

Insert the given keys in the hash table one by one.
The first key to be inserted in the hash table = 76.
Bucket of the hash table to which key 76 maps = 76 mod 7 = 6.
So, key 76 will be inserted in bucket-7 of the hash table as

0	
1	
2	
3	
4	
5	
6	**76**

$$f(k) = k \% M$$

76%7=6

Quadratic probing

- Example: insert the keys :76,40,48,5,20

Step-02:

The next key to be inserted in the hash table =40
Bucket of the hash table to which key 40 maps = 40 mod 7 = 5.
So, key 40 will be inserted in bucket-6 of the hash table as

0	
1	
2	
3	
4	
5	**40**
6	76

$$f(k) = k \% M$$

40%7=5

Quadratic probing

- Example: insert the keys :76,40,48,5,20

Step-03: The next key to be inserted in the hash table =48
Bucket of the hash table to which key 48 maps = 48 mod 7 = 6.
Since bucket-6 is already occupied, so collision occurs.
To handle the collision, quadratic probing technique keeps probing until an empty bucket is found.

0	
1	
2	
3	
4	
5	40
6	76

48+ %7=6

Quadratic probing

- Example: insert the keys :76,40,48,5,20

Step-04: The next key to be inserted in the hash table =48
Bucket of the hash table to which key 48 maps = 48 mod 7 = 6.
Since bucket-6 is already occupied, so collision occurs.
To handle the collision, quadratic probing technique keeps probing until an empty bucket is found.
The first empty bucket is bucket-0.
So, key 48 will be inserted in bucket-0 of the hash table as-

0	**48**
1	
2	
3	
4	
5	40
6	76

$$f(k) = (i + t^2) \% M$$

$48 + 1^2 \% 7 = 49 \% 7 = 0$

Quadratic probing

Step-05: The next key to be inserted in the hash table = 5.
Bucket of the hash table to which key 5 maps = 5 mod 7 =5 .
Since bucket-5 is already occupied, so collision occurs.
To handle the collision, quadratic probing technique keeps probing until an empty bucket is found
The first empty bucket is bucket-2.
So, key 5 will be inserted in bucket-2 of the hash table as-

0	48
1	
2	**5**
3	
4	
5	40
6	76

$f(k) = (i + t^2) \% M$

$5 + 1^2 \% 7 = 6 \% 7 = 6$

$5 + 2^2 \% 7 = 9 \% 7 = 2$

$5 \% 7 = 5$

Quadratic probing

Step-05: The next key to be inserted in the hash table = 20.
Bucket of the hash table to which key 20 maps = 20 mod 7 =6 .
Since bucket-6 is already occupied, so collision occurs.
To handle the collision, quadratic probing technique keeps probing until an empty bucket is found
The first empty bucket is bucket-3.
So, key 20 will be inserted in bucket-3 of the hash table as-

0	48
1	
2	5
3	**20**
4	
5	40
6	76

$$f(k) = (i + t^2) \% M$$

$20+ 1^2 \%7=21\%7=3$

$20+ 2^2 \%7=24\%7=3$

$20 \%7=6$

Quadratic probing

insert the keys 10, 15, 16, 20, 30, 25, 26, and 36 into a hash table of size M = 10

Chaining

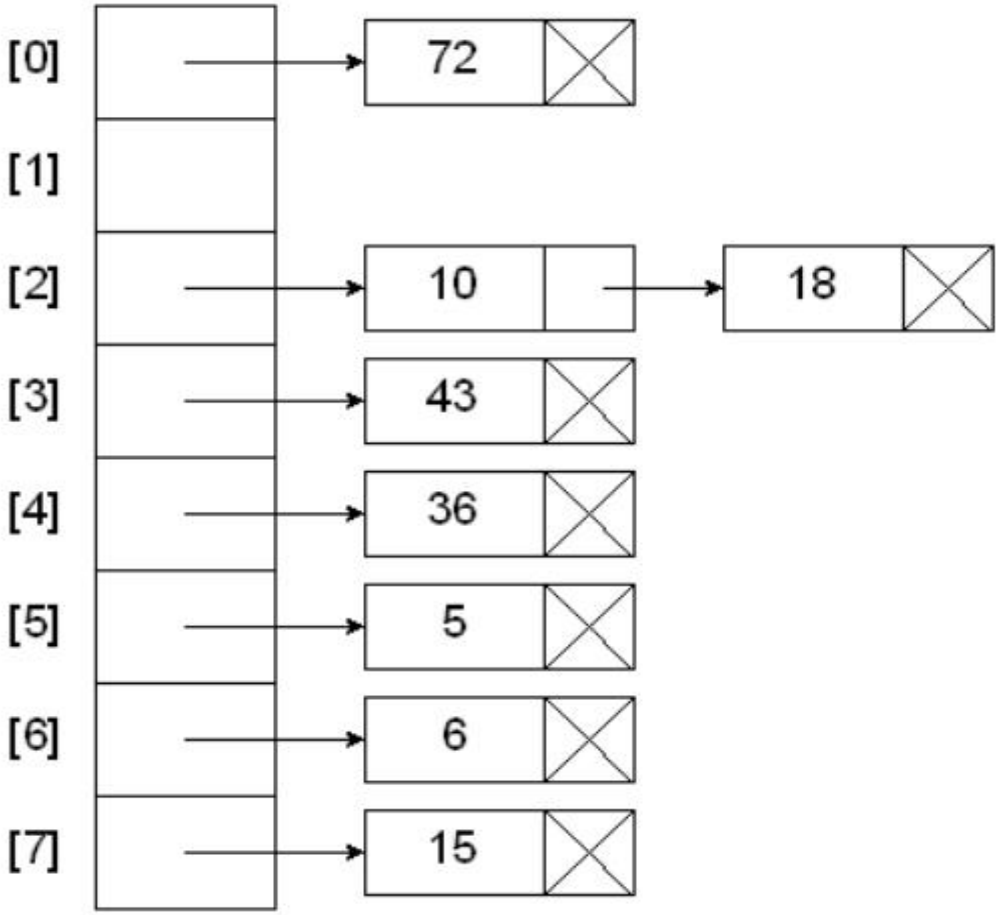
- A chain is simply a linked list of all the elements with the same hash key.
- A linked list is created at each index in the hash table.
- Hash Function: h (K) = K mod m
- Example: Assume a table has 8 slots (m=8). Using the chaining, insert the following elements into the hash table. 36, 18, 72, 43, 6, 10, 5, and 15 are inserted in the order.

Chaining

Hash key = key % table size

36 % 8 = 4
18 % 8 = 2
72 % 8 = 0
43 % 8 = 3
6 % 8 = 6
10 % 8 = 2
5 % 8 = 5
15 % 8 = 7

Chaining



Chaining

- A data item's key is hashed to the index in simple hashing, and the item is inserted into the linked list at that index.
- Other items that hash to the same index are simply added to the linked list.
- Cost is proportional to length of list.
- Average length = N / M, where N is size of array and M is the number of linked lists.

Double hashing

- Double hashing requires that the size of the hash table is a prime number.
- Double hashing uses the idea of applying a second hash function to the key when a collision occurs.
- The primary hash function determines the home address. If the home address is occupied, apply a second hash function to get a number c (c relatively prime to N). This c is added to the home address to produce an overflow addresses: if occupied, proceed by adding c to the overflow address, until an empty slot is found.
- Primary hash function is similar to direct hashing. Hash1(key) = Key % table size. A popular secondary hash function is: $Hash_2(key) = R - (key \% R)$ where R is a prime number that is smaller than the size of the table.

Double hashing

- Example: Assume a table has 10 slots. Primary hash function is H1(key) = key mod 10, and secondary hash function is H2(key)=7-(key mod 7). With Double hashing, insert the following elements in the given order.

89, 18, 49, 58 and 69 are inserted into a hash table

Double Hashing

$$f(i) = i * g(k)$$

where g is a second hash function

- Probe sequence:

  0th probe = h(k) mod TableSize

  1th probe = (h(k) + g(k)) mod TableSize

  2th probe = (h(k) + 2*g(k)) mod TableSize

  3th probe = (h(k) + 3*g(k)) mod TableSize

  . . .

  i^{th} probe = (h(k) + i*g(k)) mod TableSize

Double Hashing Example

$h(k) = k \bmod 7$ and $g(k) = 5 - (k \bmod 5)$

	76	93	40	47	10	55
0						
1				47	47	47
2		93	93	93	93	93
3					10	10
4						55
5			40	40	40	40
6	76	76	76	76	76	76
Probes	1	1	1	2	1	2

Extensible hashing

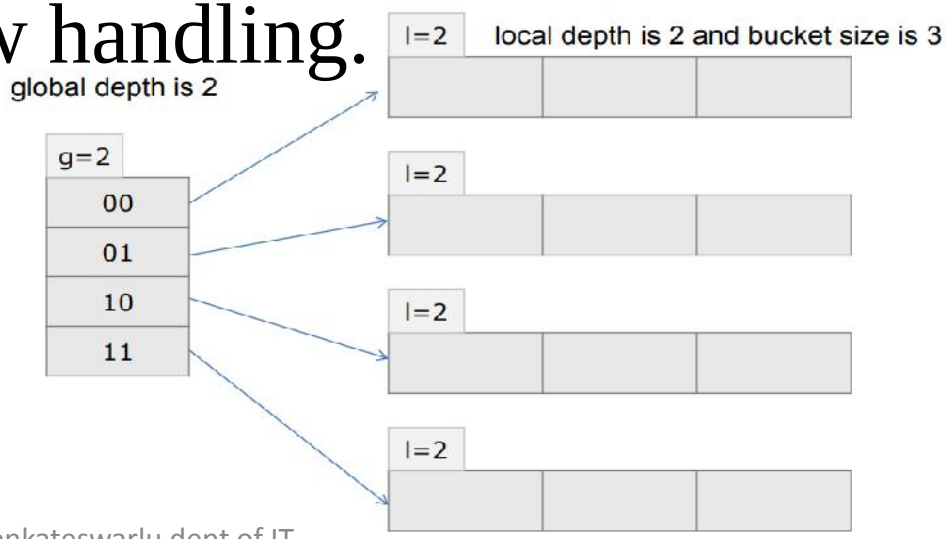
- It is a technique which handles a large amount of data.
- The data to be placed in the hash table is by extracting certain number of bits
- Extensible hashing grow and shink similar to B-tress
- In extensible hashing referring the size of directory the elements are to be placed in buckets.

Extensible hashing

- Extendible hashing uses a directory to access its buckets.
- This directory is usually small enough to be kept in main memory and has the form of an array with 2d entries, each entry storing a bucket address (pointer to a bucket).
- The variable d is called the global depth of the directory

Extensible hashing

- Multiple directory entries may point to the same bucket.
- Every bucket has a local depth leqd.
- The difference between local depth and global depth affects overflow handling.



Extensible hashing

- Suppose that g=2 and bucket size = 3.
- Suppose that we have records with these keys and hash function h(key) = key mod 64:

1111	3333	1235	2378	1212	1456	2134	2345	1111	8231	2222	9999

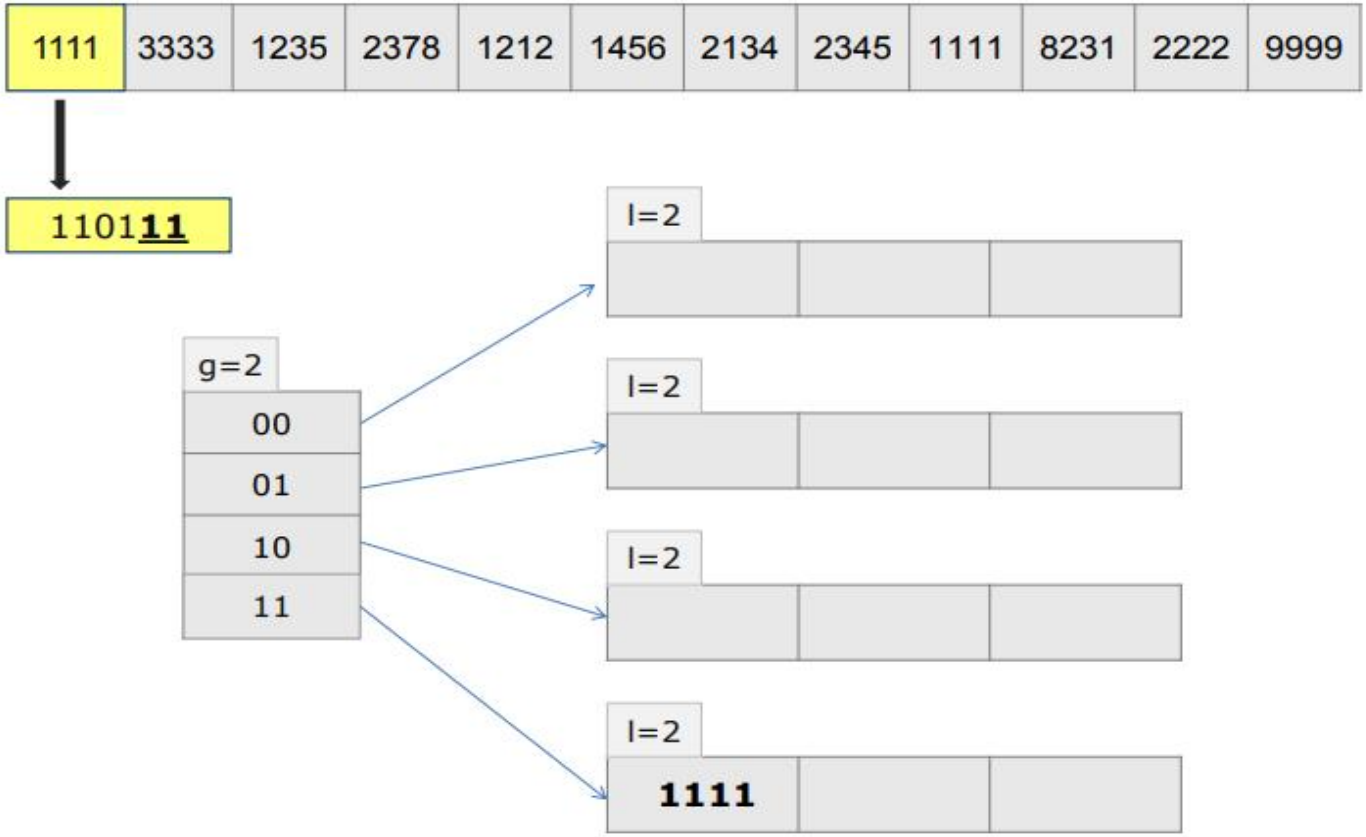
Extensible hashing

- Suppose that we have records with these keys and hash function h(key) = key mod 64:

key	h(key) = key mod 64	bit pattern
1111	23	010111
3333	5	000101
1235	19	010011
2378	10	001010
1212	60	111100
1456	48	110000
2134	22	010110
2345	41	101001
1111	23	010111
8231	39	100111
2222	46	101110
9999	15	001111

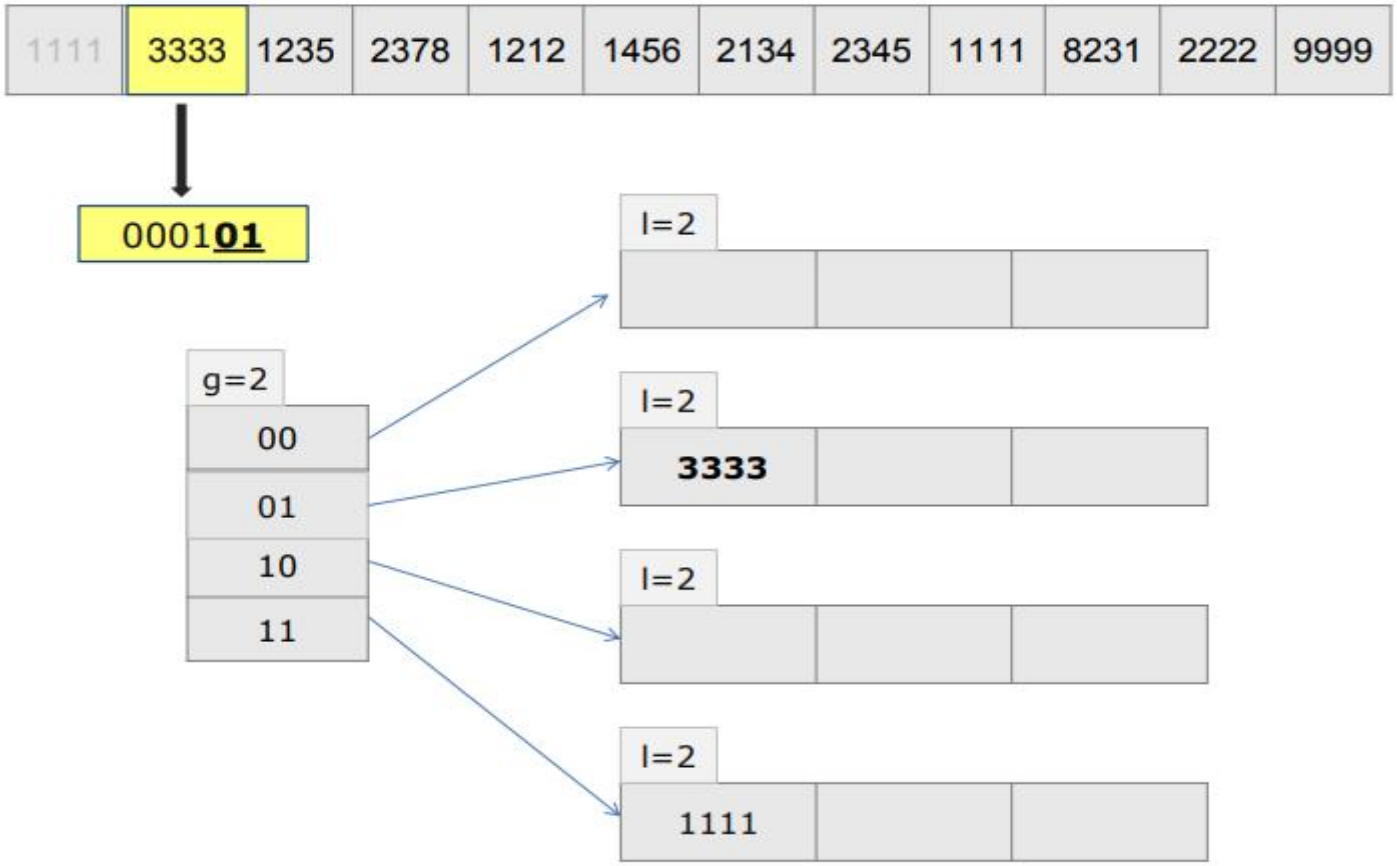
Extensible hashing

- Insert 1111 i.e 110111



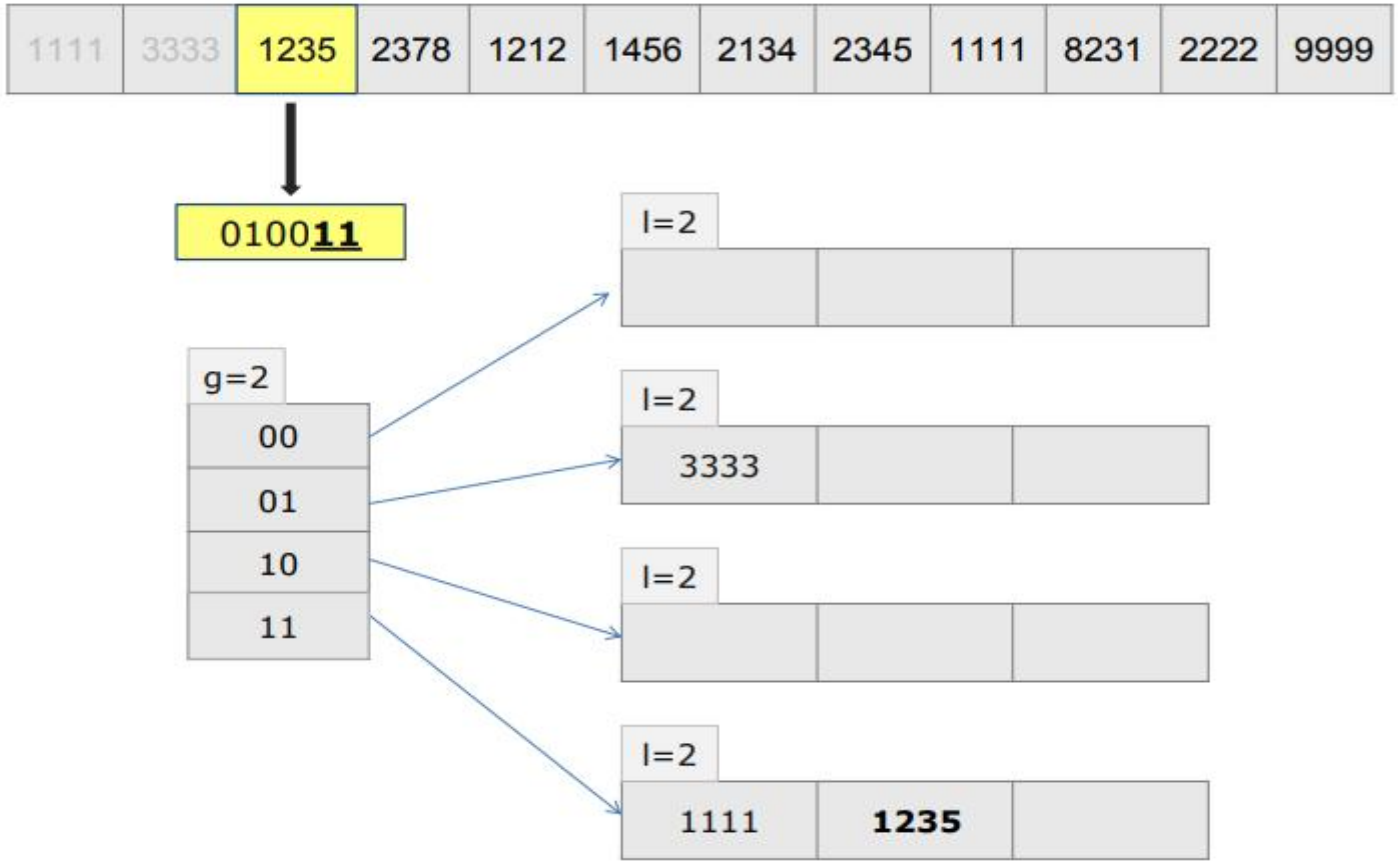
Extensible hashing

- Insert 3333 i.e 000101



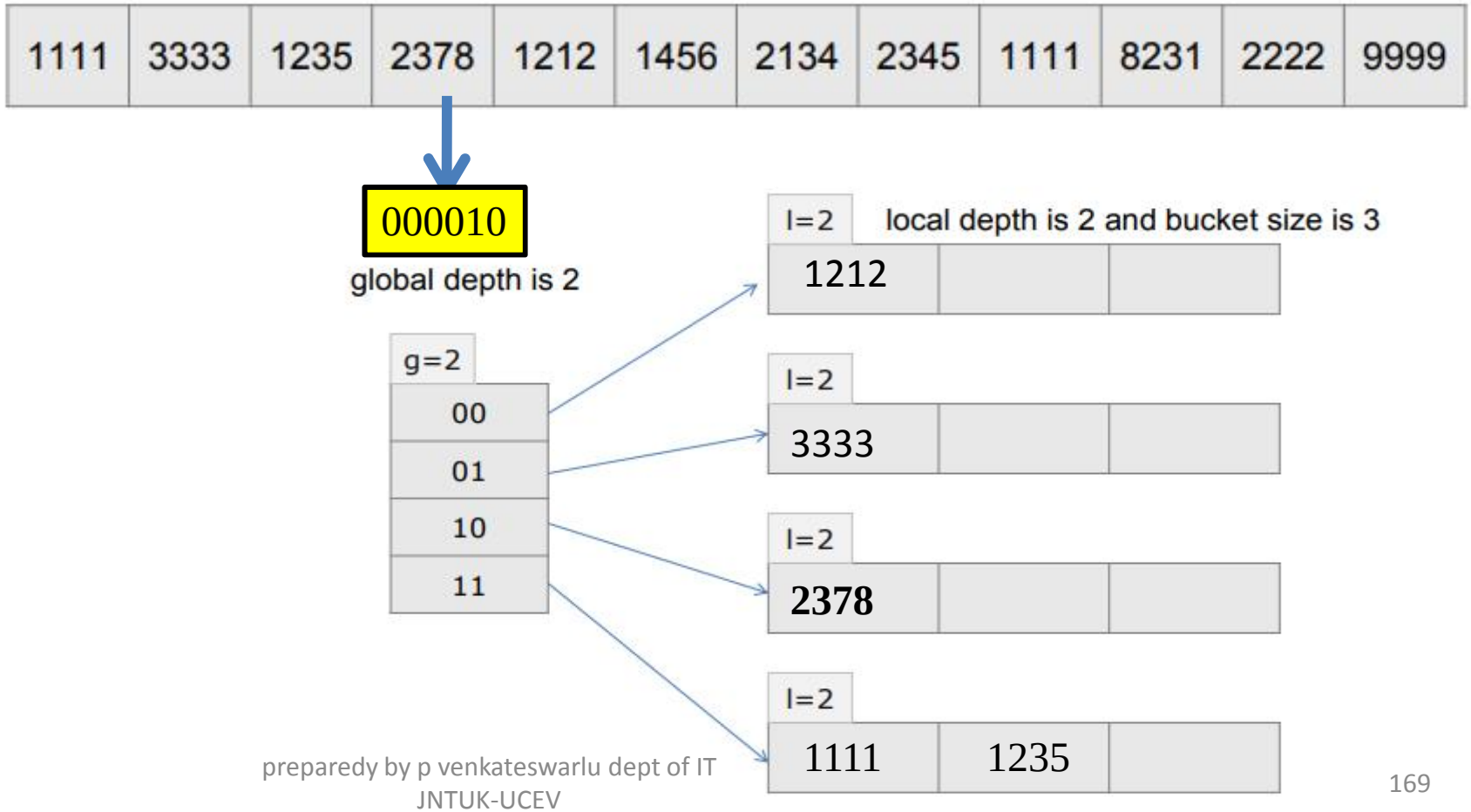
Extensible hashing

- Insert 1235 i.e 010011



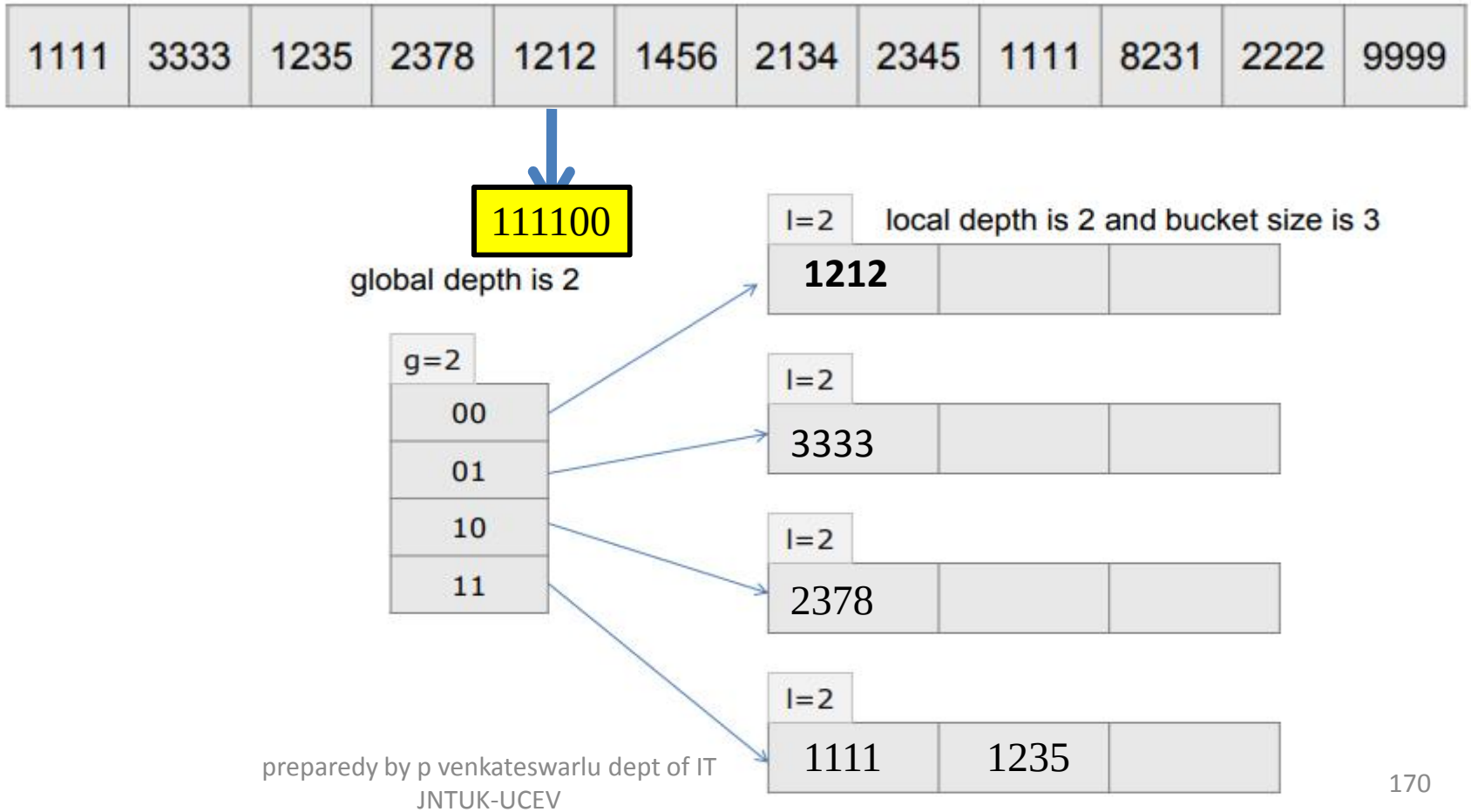
Extensible hashing

- Insert 2378 i.e 000010



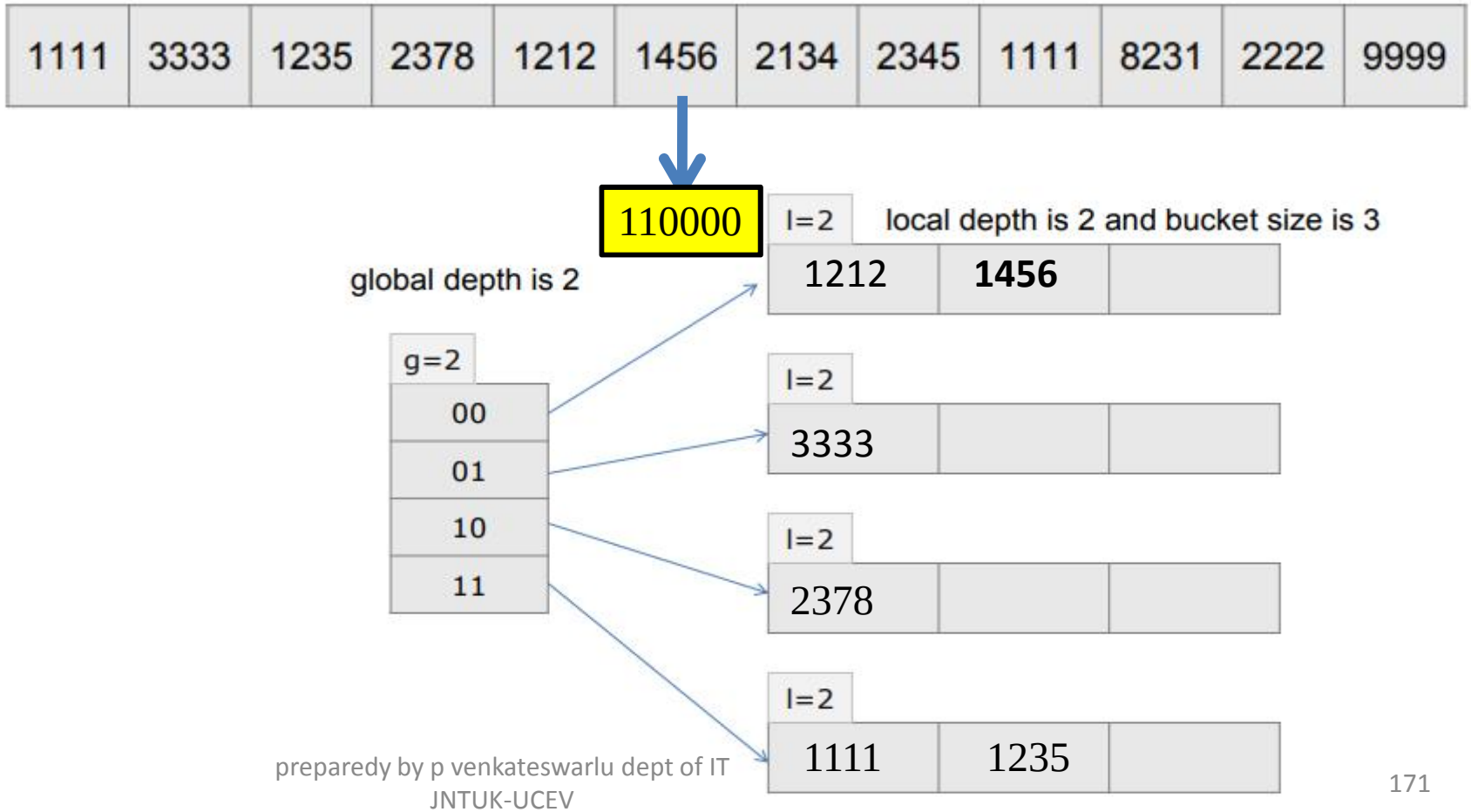
Extensible hashing

- Insert 1212 i.e 111100



Extensible hashing

- Insert 1456 i.e 110000

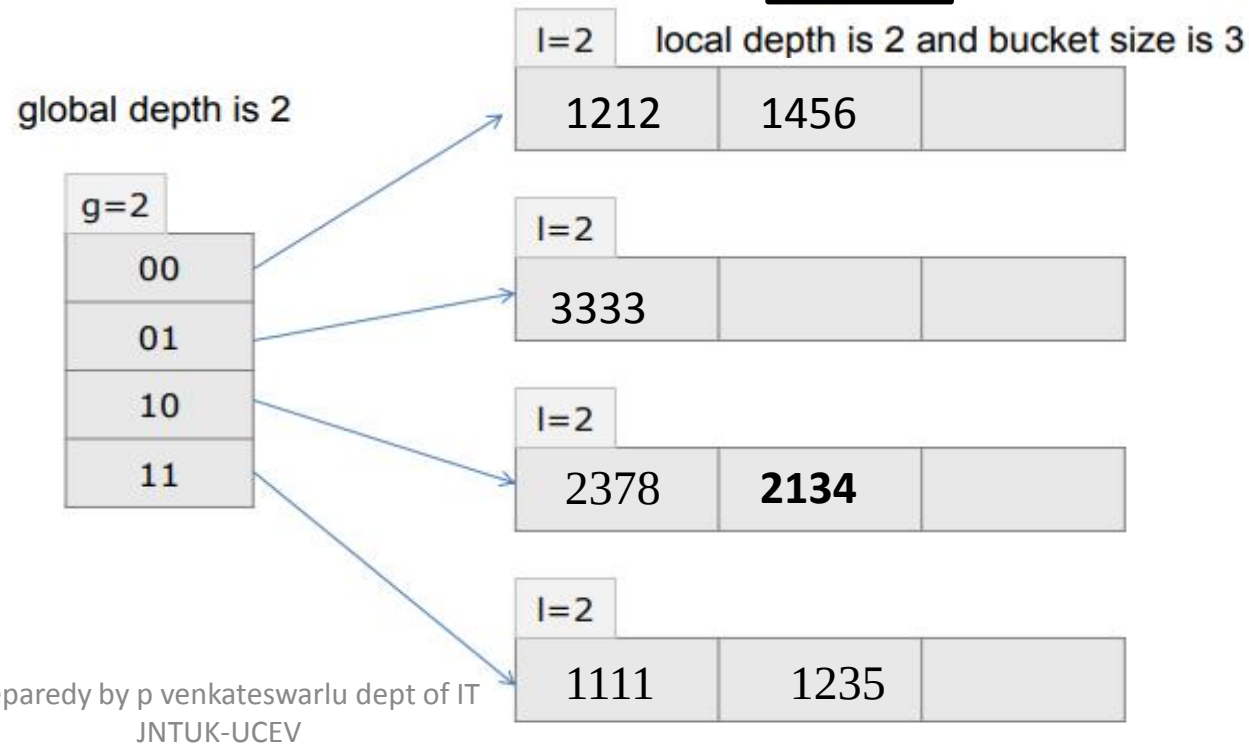


Extensible hashing

- Insert 2134 i.e 010110

1111	3333	1235	2378	1212	1456	2134	2345	1111	8231	2222	9999

010110

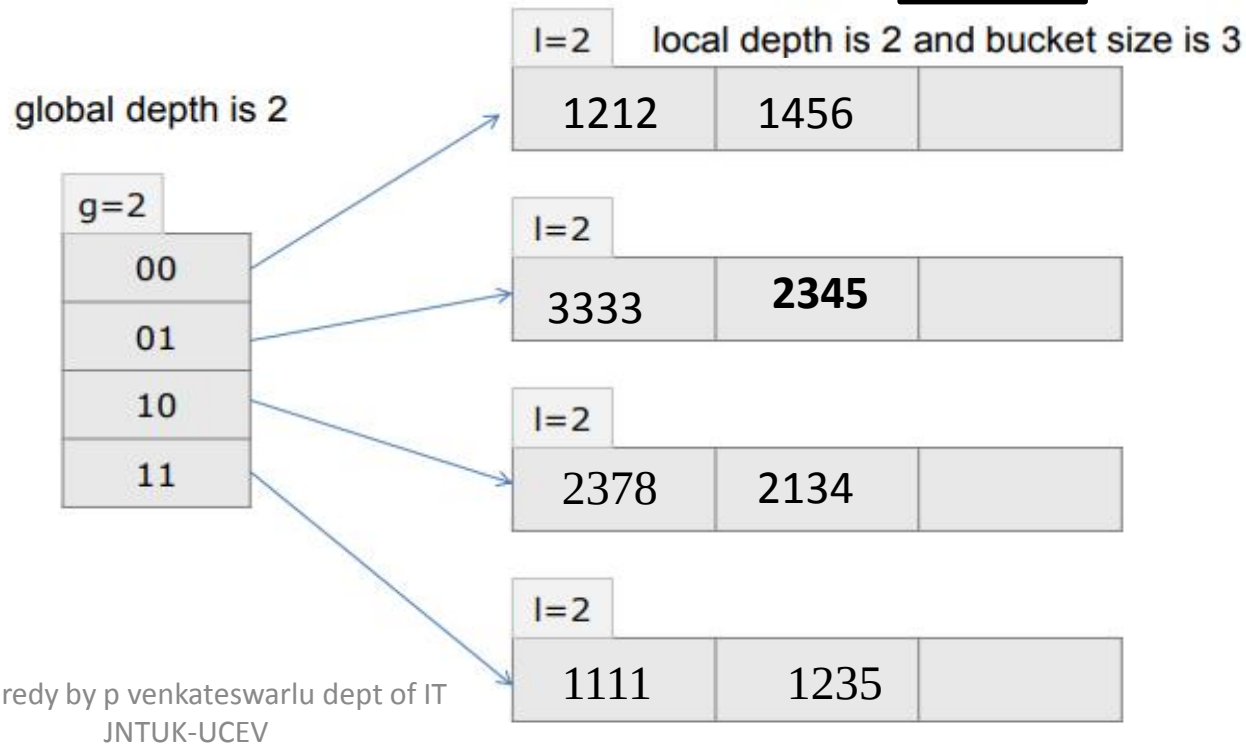


Extensible hashing

- Insert 2345 i.e 101001

1111	3333	1235	2378	1212	1456	2134	2345	1111	8231	2222	9999

101001

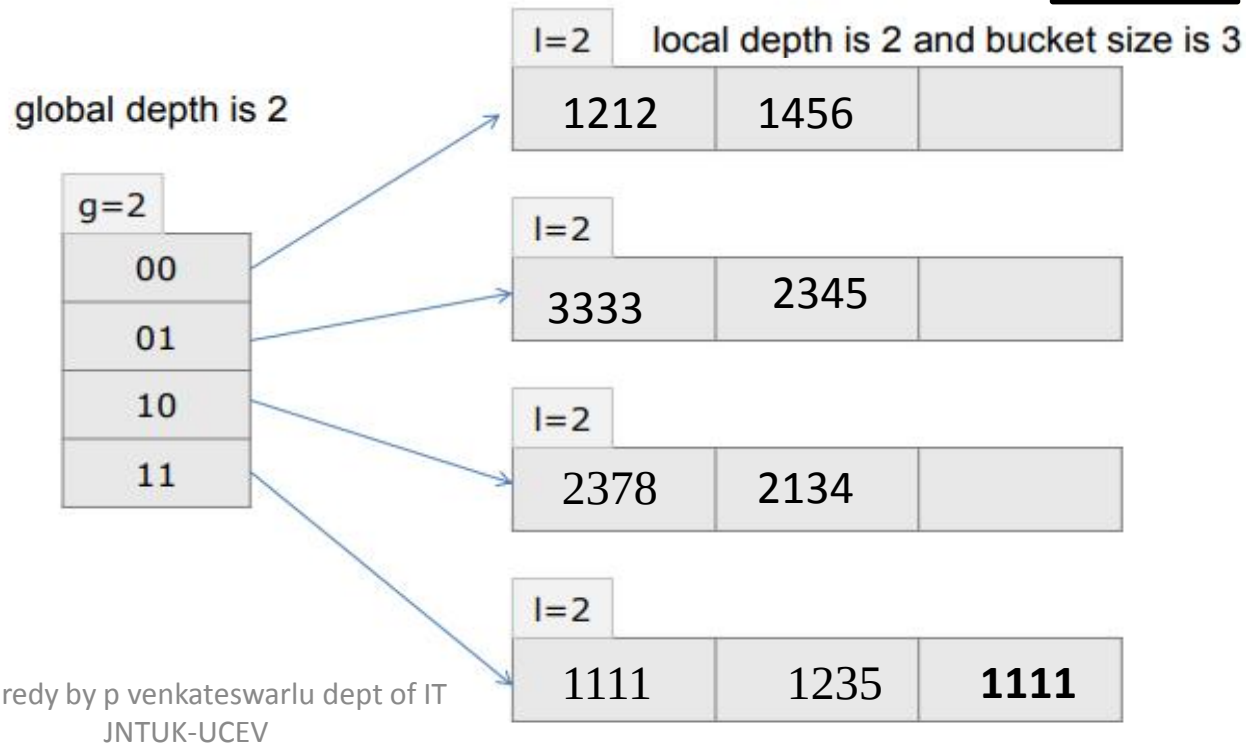


Extensible hashing

- Insert 1111 i.e 110111

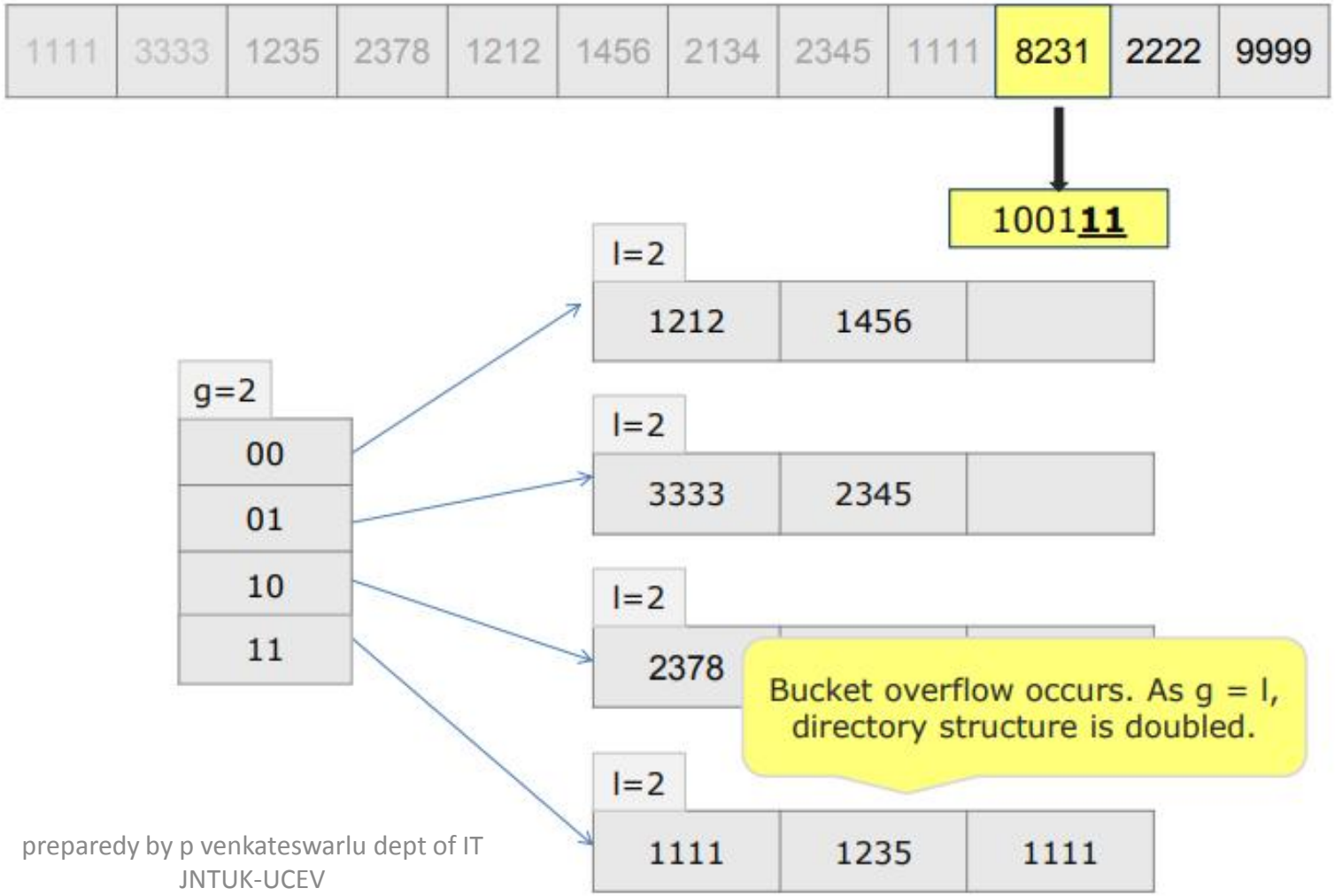
1111	3333	1235	2378	1212	1456	2134	2345	1111	8231	2222	9999

110111



Extensible hashing

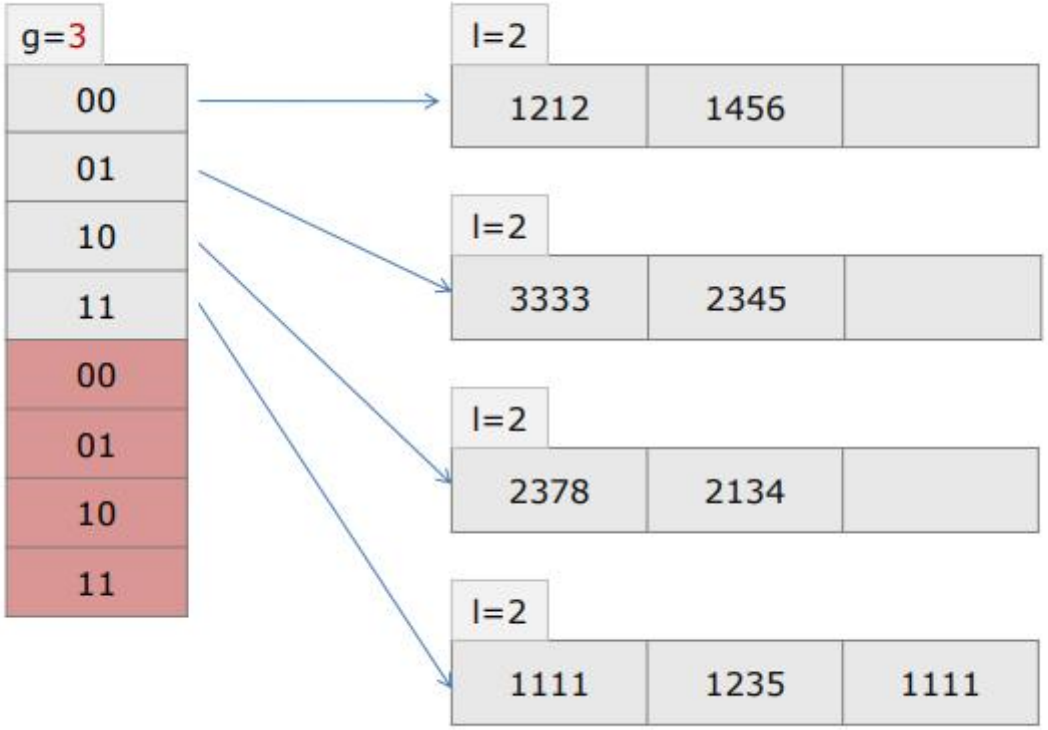
- Insert 8231 i.e 100111



Extensible hashing

- Insert 8231 i.e 100111

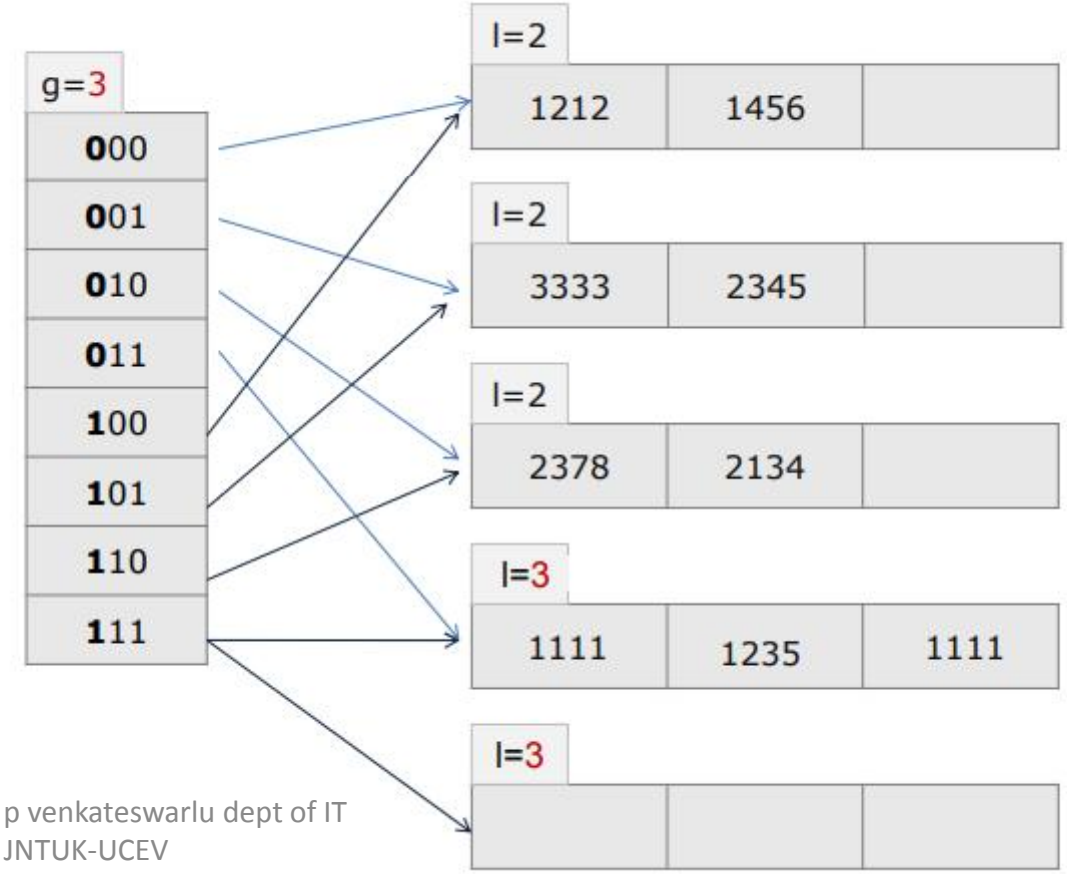
1111	3333	1235	2378	1212	1456	2134	2345	1111	8231	2222	9999



Extensible hashing

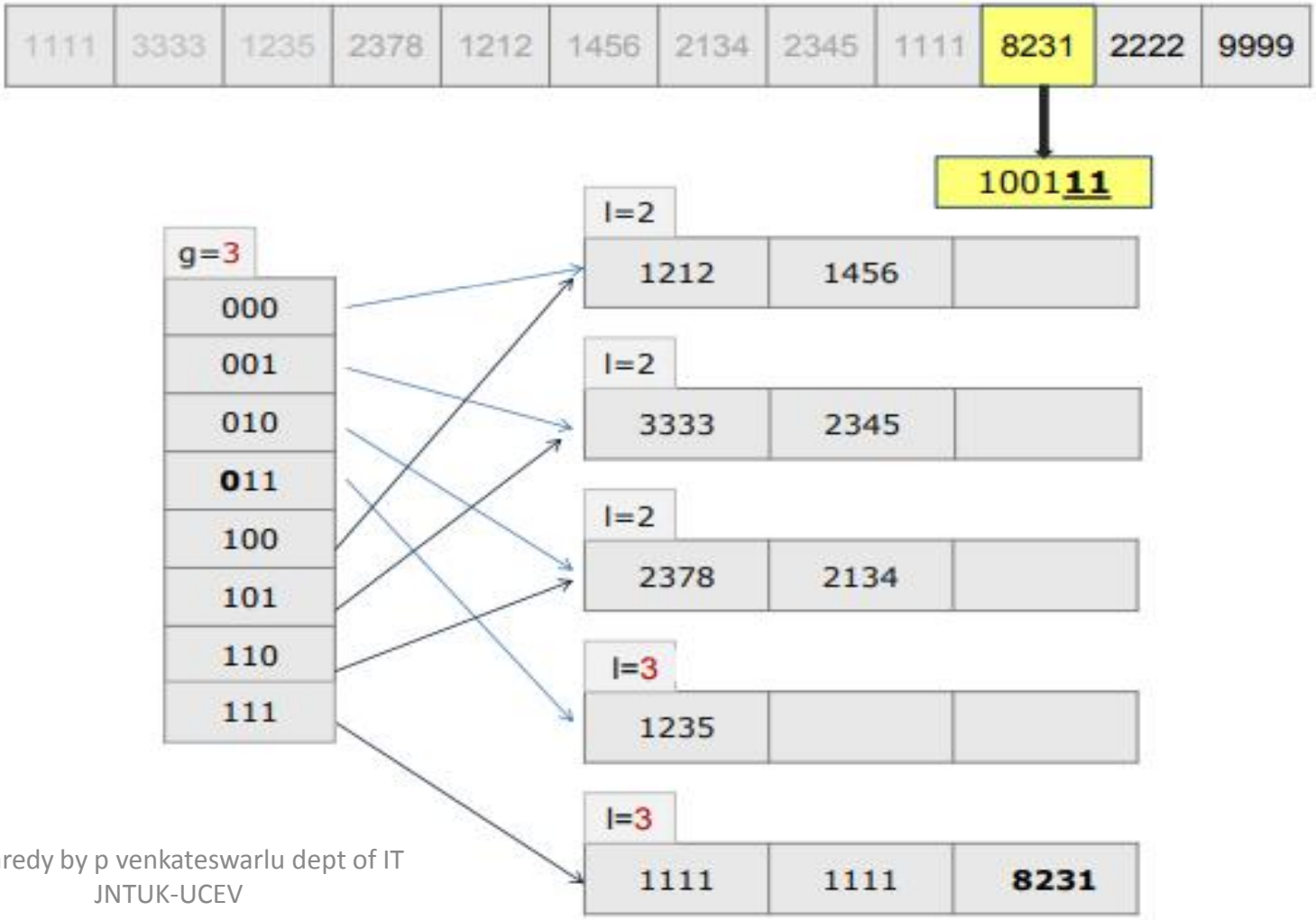
- Insert 8231 i.e 100111

1111	3333	1235	2378	1212	1456	2134	2345	1111	8231	2222	9999



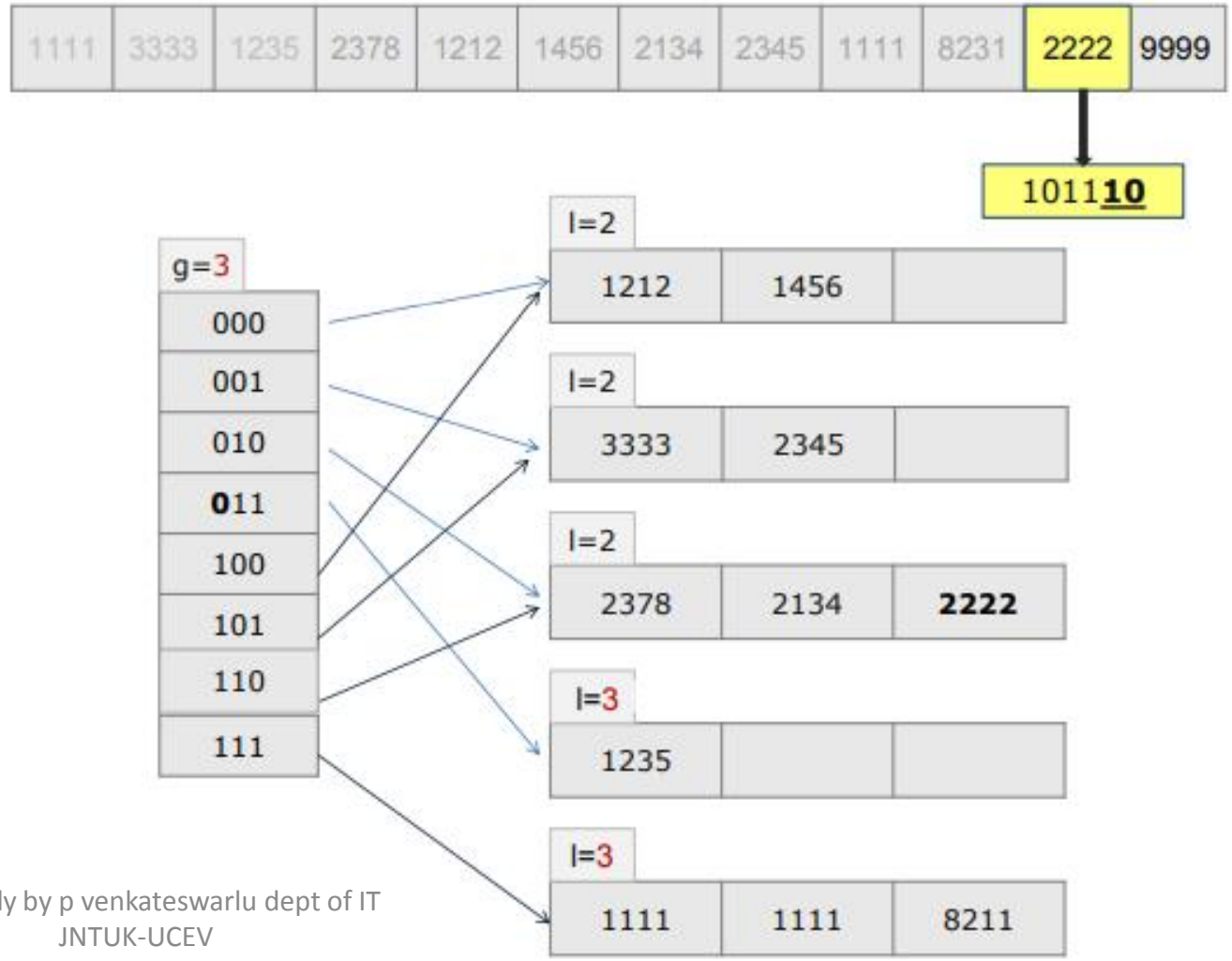
Extensible hashing

- Insert 8231 i.e 100111



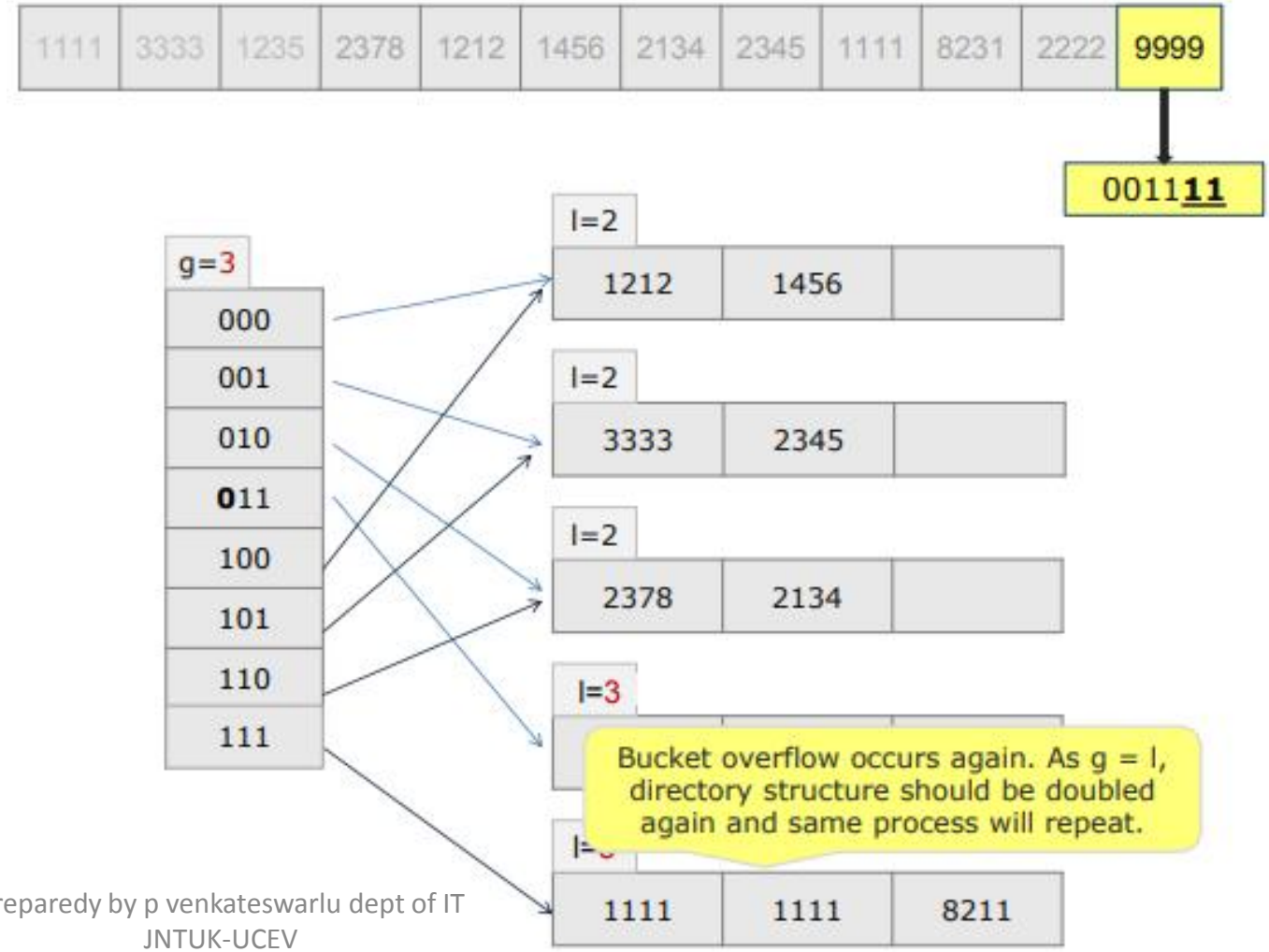
Extensible hashing

- Insert 2222 i.e 101110



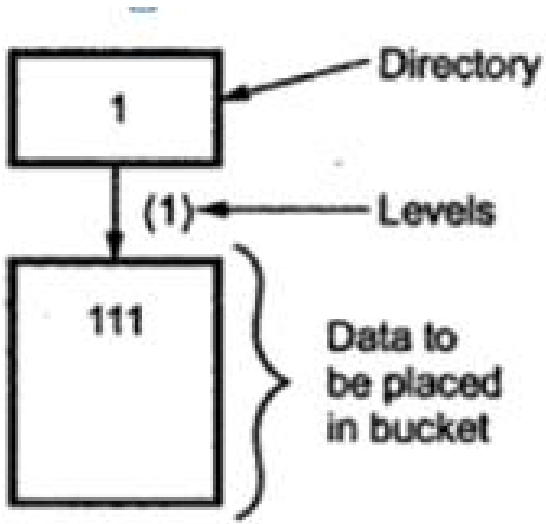
Extensible hashing

- Insert 9999 i.e 001111



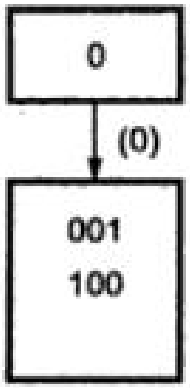
Extensible hashing

- The bucket can hold the data of its global depth.
- If data in bucket is more than global depth then split the bucket and double the directory



Extensible hashing

- Consider we have to insert 1, 4, 5, 7, 8, 10 assume each page can hold 2 data entries (2 is the depth)
- Step 1: insert 1, 4



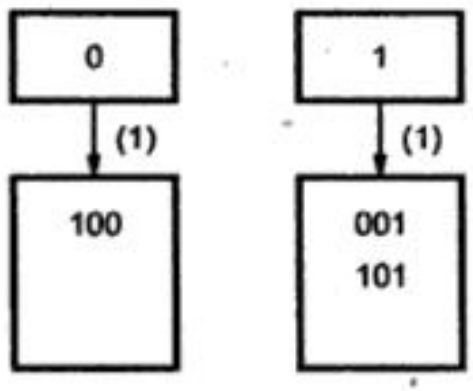
1 = 001

4 = 100

We will examine last bit of data and insert the data in bucket.

Extensible hashing

- Consider we have to insert 1, 4, 5, 7, 8, 10 assume each page can hold 2 data entries (2 is the depth)
- Step 2: insert 5 the bucket is full hence double the directory.



1 = 001

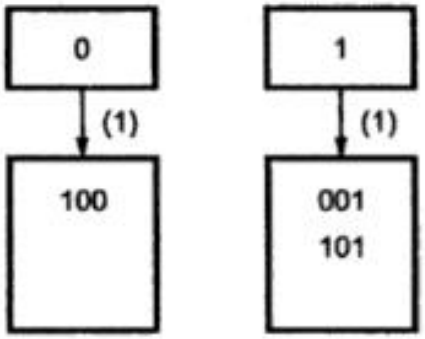
4 = 100

5 = 101

Based on last bit the data is inserted.

Extensible hashing

- Consider we have to insert 1, 4, 5, 7, 8, 10 assume each page can hold 2 data entries (2 is the depth)
- Step 3: insert 7 but as the depth is full we can not insert 7 here then double the directory and split the bucket.

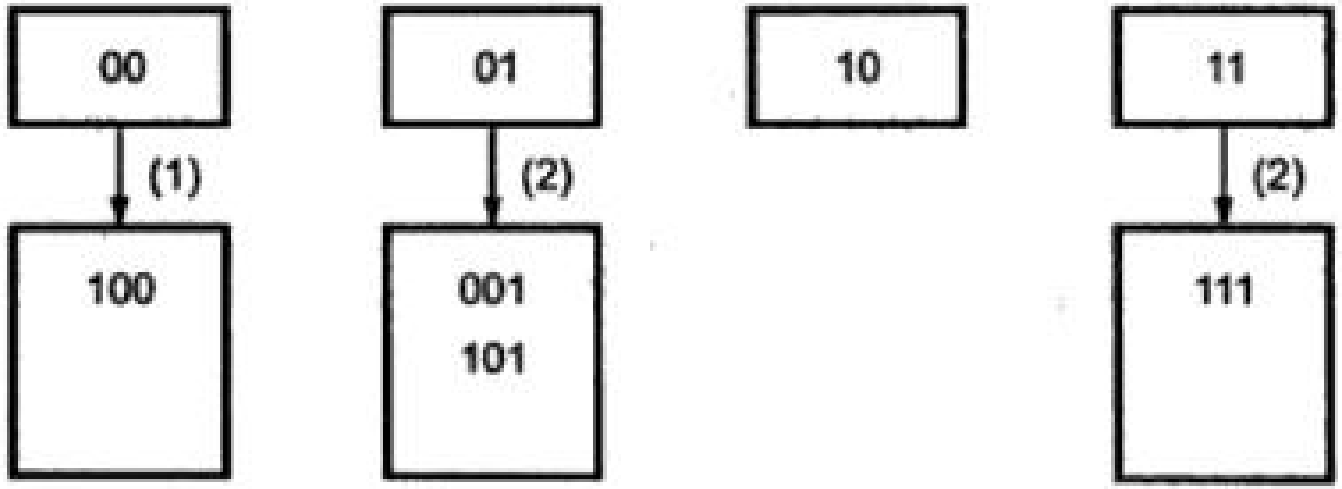


7 = 111

Insert 7 : But as depth is full we can not insert 7 here. Then double the directory and split the bucket.

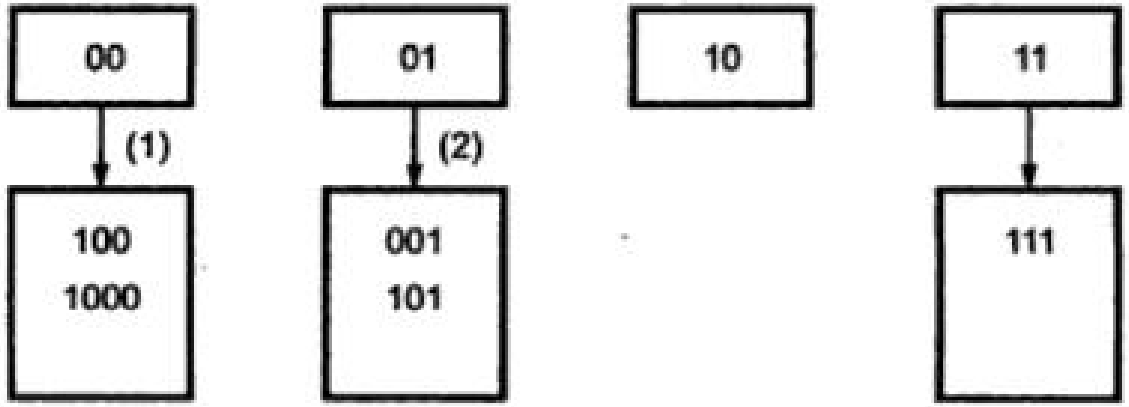
Extensible hashing

- After insertion of 7 consider the last two bits



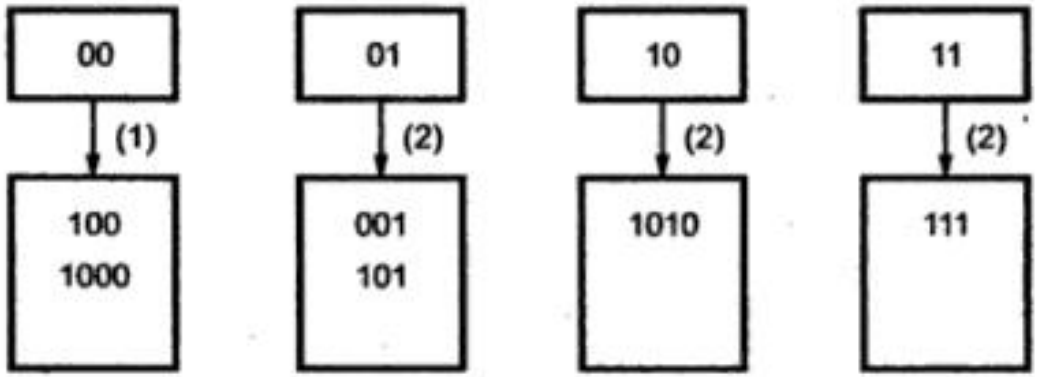
Extensible hashing

- Consider we have to insert 1, 4, 5, 7, 8, 10 assume each page can hold 2 data entries (2 is the depth)
- Step 4: insert 8 i.e 1000



Extensible hashing

- Consider we have to insert 1, 4, 5, 7, 8, 10 assume each page can hold 2 data entries (2 is the depth)
- Step 5: insert 10 i.e 1000

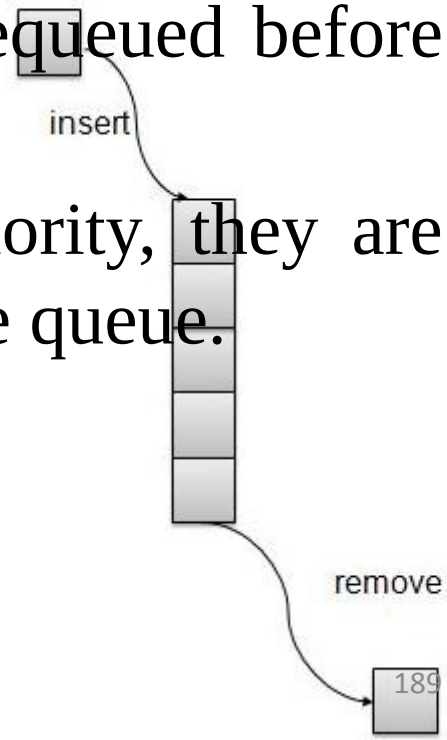


Priority Queue

- Priority Queue is more specialized data structure than Queue. Like ordinary queue, priority queue has same method but with a major difference.
- In Priority queue items are ordered by key value so that item with the lowest value of key is at front and item with the highest value of key is at rear or vice versa.

Priority Queue

- Priority Queue is an extension of queue with following properties.
  - Every item has a priority associated with it.
  - An element with high priority is dequeued before an element with low priority.
  - If two elements have the same priority, they are served according to their order in the queue.



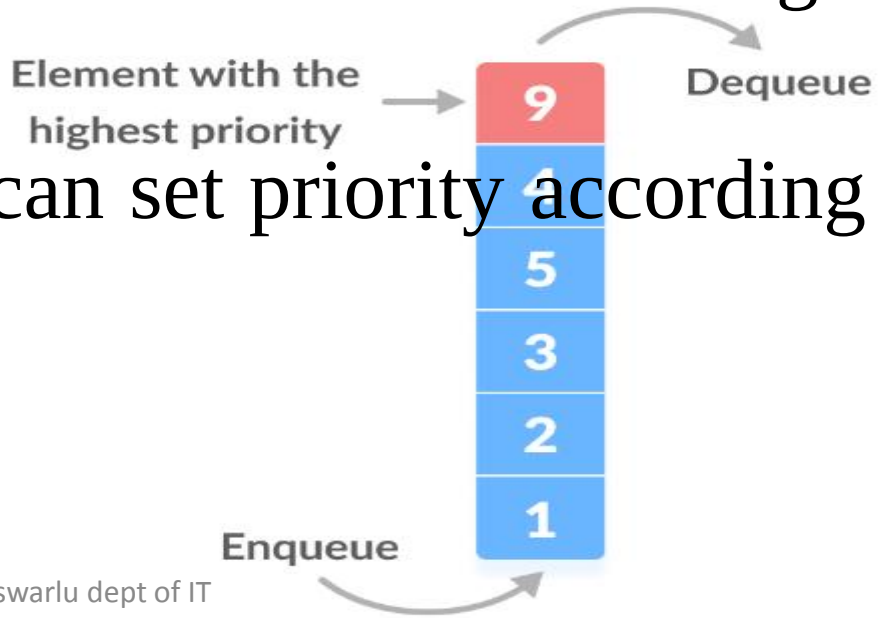
Priority Queue

- A priority queue is a special type of queue in which each element is associated with a priority and is served according to its priority.

- If elements with the same priority occur, they are served according to their order in the queue.
- Generally, the value of the element itself is considered for assigning the priority.

Priority Queue

- The element with the highest value is considered as the highest priority element.
- However, in other case, we can assume the element with the lowest value as the highest priority element.
- In other cases, we can set priority according to our need.



Priority Queue

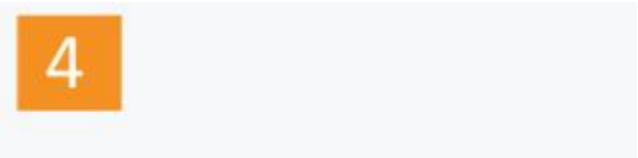
- Priority Queue is similar to queue where we insert an element from the back and remove an element from front, but with a difference that the logical order of elements in the priority queue depends on the priority of the elements.
- The element with highest priority will be moved to the front of the queue and one with lowest priority will move to the back of the queue. Thus it is possible that when you enqueue an element at the back in the queue, it can move to front because of its highest priority.

Priority Queue

- Let's say we have an array of 5 elements :

  {4, 8, 1, 7, 3} and we have to insert all the elements in the max-priority queue.

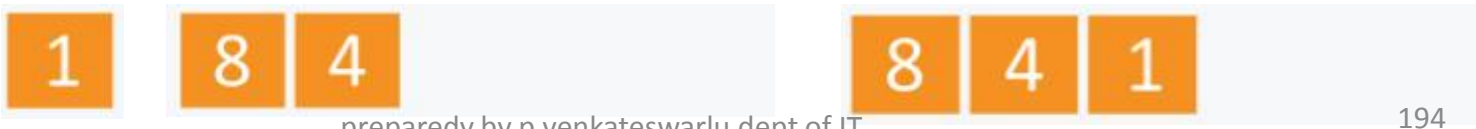
- First as the priority queue is empty, so 4 will be inserted initially.



Priority Queue

- Now when 8 will be inserted it will moved to front as 8 is greater than 4.

- While inserting 1, as it is the current minimum element in the priority queue, it will remain in the back of priority queue.



Priority Queue

- Now 7 will be inserted between 8 and 4 as 7 is smaller than 8.

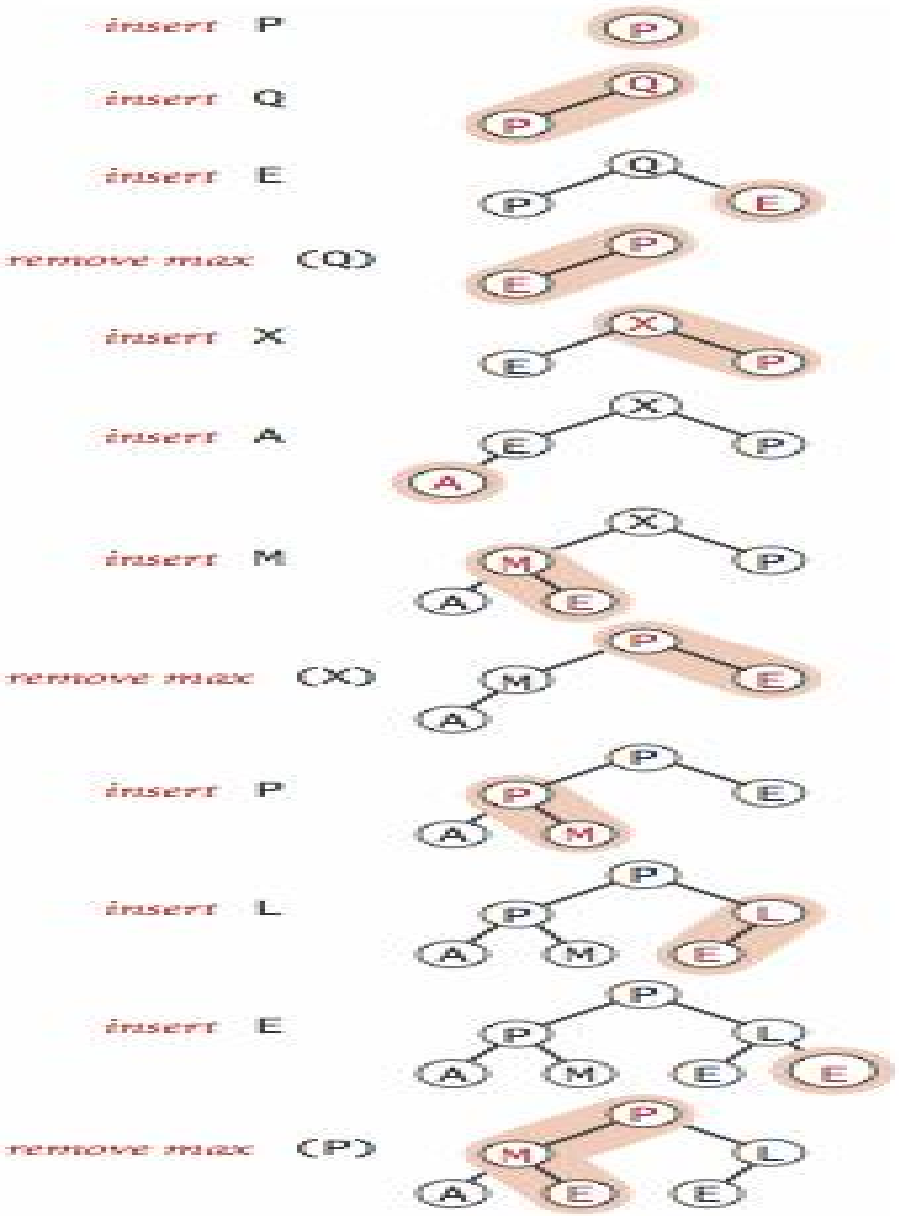
- Now 3 will be inserted before 1 as it is the 2nd minimum element in the priority queue.

Priority Queue

- Now 3 will be inserted before 1 as it is the 2nd minimum element in the priority queue.

operation	argument	return value	size	contents (unordered)							contents (ordered)				
insert	P		1	P							P				
insert	Q		2	P	Q						P	Q			
insert	E		3	P	Q	E					E	P	Q		
remove max		Q	2	P	E						E	P			
insert	X		3	P	E	X					E	P	X		
insert	A		4	P	E	X	A				A	E	P	X	
insert	M		5	P	E	X	A	M			A	E	M	P	X
remove max		X	4	P	E	M	A				A	E	M	P	
insert	P		5	P	E	M	A	P			A	E	M	P	P
insert	L		6	P	E	M	A	P	L		A	E	L	M	P
insert	E		7	P	E	M	A	P	L	E	A	E	E	L	M
remove max		P	6	E	E	M	A	P	L		A	E	E	L	M

A sequence of operations on a priority queue


insert P
insert Q
insert E
remove max (Q)
insert X
insert A
insert M
remove max (X)
insert P
insert L
insert E
remove max (P)

implement the priority queue.

- **Naive Approach:**
  - Suppose we have N elements and we have to insert these elements in the priority queue. We can use list and can insert elements in O(N) time and can sort them to maintain a priority queue in O(NlogN) time.
- **Efficient Approach:**
  - We can use heaps to implement the priority queue. It will take O(logN) time to insert and delete each element in the priority queue.

implement the priority queue.

- Based on heap structure, priority queue also has two types max- priority queue and min - priority queue.

How priority queue differs from a queue?

- In a queue, the **first-in-first-out rule** is implemented whereas, in a priority queue, the values are removed **on the basis of priority**. The element with the highest priority is removed first.

Implementation of Priority Queue

- Priority queue can be implemented using an array, a linked list, a heap data structure or a binary search tree. Among these data structures, heap data structure provides an efficient implementation of priority queues.
- A comparative analysis of different implementations of priority queue is

Priority Queue Operations

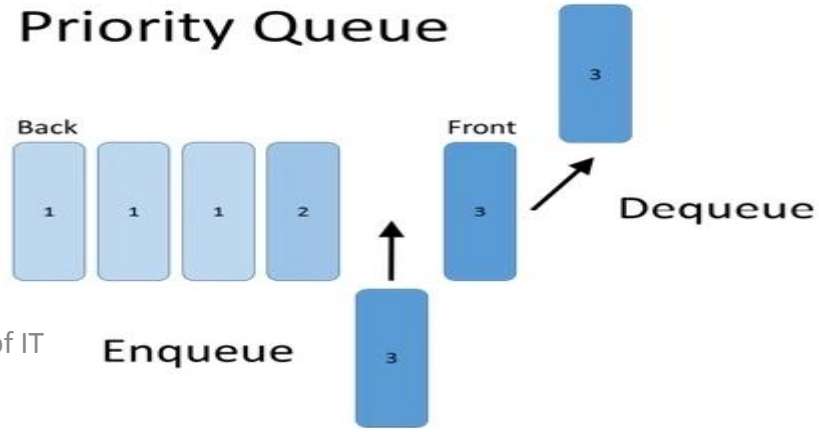
- A priority queue is an abstract data type (ADT) supporting the following three operations:
  - Add an element to the queue with an associated priority
  - Remove the element from the queue that has the highest priority, and return it
  - (optionally) peek at the element with highest priority without removing it

Applications of Priority Queue:

1) CPU Scheduling
2) Graph algorithms like Dijkstra's shortest path algorithm, Prim's Minimum Spanning Tree, etc
3) All queue applications where priority is involved.

Implementation of priority queue using linked list

- A **priority queue** is a very important data structure because it can store data in a very practical way.
- This is a **concept of storing the item with its priority**.
- This way we can **prioritize our concept of a queue**.



Implementation of priority queue using linked list

- Add an element to the queue with an associated priority

```
void PriorityQueue::Insert(int DT)
{
struct Node *newnode;
newnode=new Node;
newnode->Data=DT;
while(ptr->Next!=NULL)
ptr=ptr->Next;
if(ptr->Next==NULL)
{
        newnode->Next=ptr->Next;
        ptr->Next=newnode;
}
NumOfNodes++;
}
```

Implementation of priority queue using linked list

– Remove the element from the queue that has the highest priority, and return it

```
void PriorityQueue::Insert(int DT)
{
struct Node *newnode;
newnode=new Node;
newnode->Data=DT;
while(ptr->Next!=NULL)
ptr=ptr->Next;
if(ptr->Next==NULL)
{
        newnode->Next=ptr->Next;
        ptr->Next=newnode;
}
NumOfNodes++;
}
```

Max Priority Queue

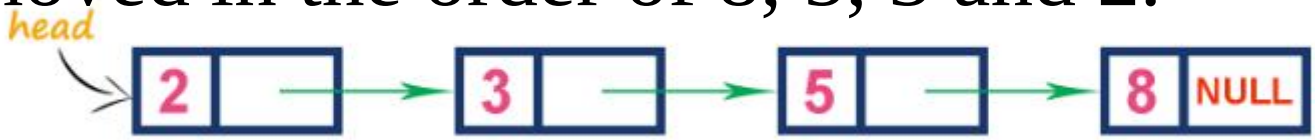
- In a max priority queue, elements are inserted in the order in which they arrive the queue and the maximum value is always removed first from the queue.
- For example, assume that we insert in the order 8, 3, 2 & 5 and they are removed in the order 8, 5, 3, 2.

Max Priority Queue

- The following are the operations performed in a Max priority queue...
  - **isEmpty() - Check whether queue is Empty.**
  - **insert() - Inserts a new value into the queue.**
  - **findMax() - Find maximum value in the queue.**
  - **remove() - Delete maximum value from the queue.**

Using Linked List in Increasing Order

- In this representation, we use a single linked list to represent max priority queue.
- In this representation, elements are inserted according to their value in increasing order and a node with the maximum value is deleted first from the max priority queue.
- For example, assume that elements are inserted in the order of 2, 3, 5 and 8. And they are removed in the order of 8, 5, 3 and 2.

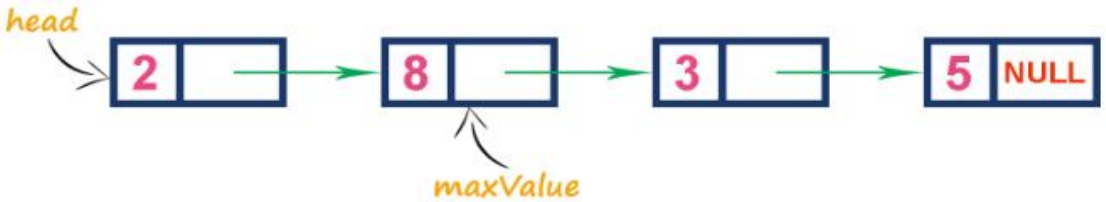


Using Linked List in Increasing Order

- **isEmpty()** - If '**head == NULL**' queue is Empty. This operation requires **O(1)** time complexity which means constant time complexity.
- **insert()** - New element is added at a particular position in the increasing order of elements which requires **O(n)** time complexity. This insert() operation requires **O(n)** time complexity.
- **findMax()** - Finding the maximum element in the queue is very simple because maximum element is at the end of the queue. This findMax() operation requires **O(1)** time complexity.
- **remove()** - Removing an element from the queue is simple because the largest element is last node in the queue. This remove() operation requires **O(1)** time complexity.

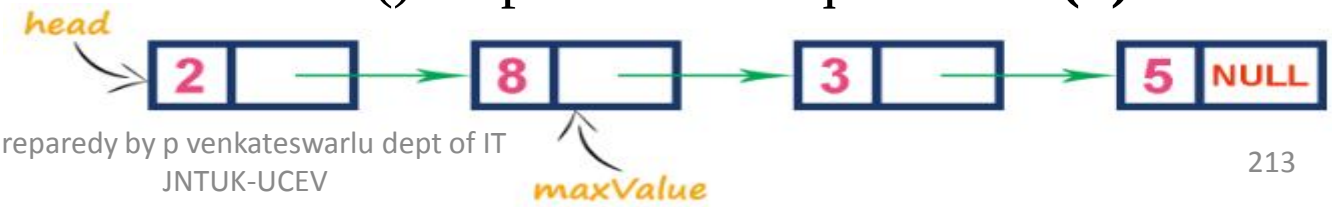
Using Unordered Linked List with reference to node with the maximum value

- In this representation, we use a single linked list to represent max priority queue.
- We always maintain a reference (maxValue) to the node with the maximum value in the queue.
- In this representation, elements are inserted according to their arrival and the node with the maximum value is deleted first from the max priority queue.



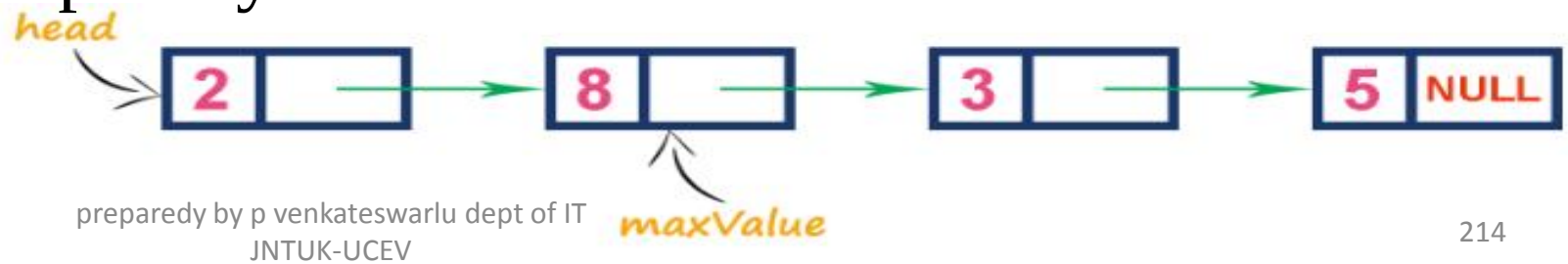
Using Unordered Linked List with reference to node with the maximum value

- let us analyze each operation according to this representation...
- **isEmpty()** - If '**head == NULL**' queue is Empty. This operation requires **O(1)** time complexity which means constant time complexity.
- **insert()** - New element is added at end of the queue which requires **O(1)** time complexity. And we need to update maxValue reference with address of largest element in the queue which requires **O(1)** time complexity. This insert() operation requires **O(1)** time complexity.
- **findMax()** - Finding the maximum element in the queue is very simple because the address of largest element is stored at maxValue. This findMax() operation requires **O(1)** time complexity.



Using Unordered Linked List with reference to node with the maximum value

- **remove()** - Removing an element from the queue is deleting the node which is referenced by maxValue which requires **O(1)** time complexity.
- And then we need to update maxValue reference to new node with maximum value in the queue which requires **O(n) time complexity**.
- This remove() operation requires **O(n)** time complexity.



Min Priority Queue Representations

- Min Priority Queue is similar to max priority queue except for the removal of maximum element first. We remove minimum element first in the min-priority queue.

  The following operations are performed in Min Priority Queue...
- **isEmpty() -** Check whether queue is Empty.
- **insert() -** Inserts a new value into the queue.
- **findMin() -** Find minimum value in the queue.
- **remove() -** Delete minimum value from the queue.

Heap Data structure

- Heap data structure is a specialized binary tree-based data structure. The heap is a binary tree, meaning at the most, each parent has two children.
- Heap is a binary tree with special characteristics. In a heap data structure, nodes are arranged based on their values.
- A heap data structure some times also called as Binary Heap.
- There are two types of heap data structures and they are as follows...
  - **Max Heap**
  - **Min Heap**

Heap Data structure

- Heaps are based on the notion of a **complete tree**, for which we gave an informal definition earlier.
- Formally:
- A binary tree is **completely full** if it is of height, *h*, and has 2^{h+1}-1 nodes.
- A binary tree of height, *h*, is **complete** *iff*
- it is empty *or* its left sub-tree is complete of height *h*-1 and its right sub-tree is completely full of height *h*-2 *or* its left sub-tree is completely full of height *h*-1 and its right sub-tree is complete of height *h*-1.

Heap Data structure

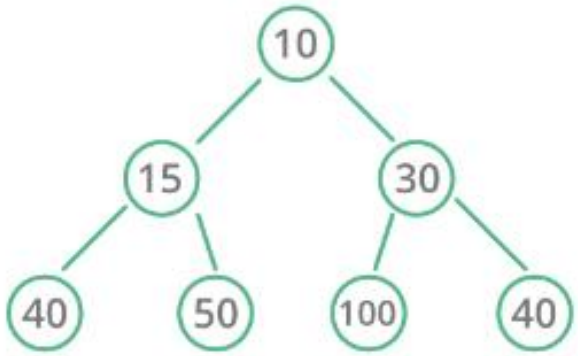
- Provides an efficient implementation for a priority queue
- Every heap data structure has the following properties...
  - **Property #1 (Ordering):** Nodes must be arranged in an order according to their values based on Max heap or Min heap.
  - **Property #2 (Structural):** All levels in a heap must be full except the last level and all nodes must be filled from left to right strictly.

Heap Data structure

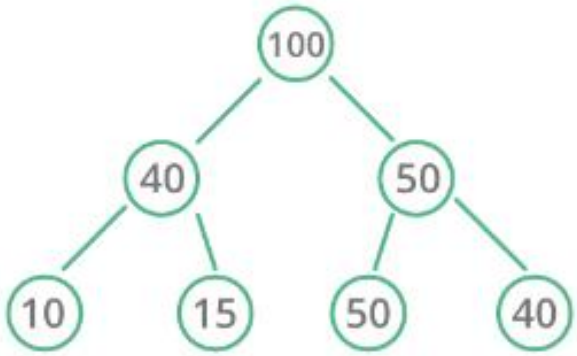
- Can think of heap as a complete binary tree that maintains the heap property:
- Heap Property: Every parent is less-than (if min-heap) or greater-than (if max-heap) both children, but no ordering property between children
- Minimum/Maximum value is always the top element

What is a heap

- Heap is a special case of balanced binary tree data structure where the root-node key is compared with its children and arranged accordingly.
- Heap is a tree-based data structure in which all nodes in the tree are in the specific order.



Min Heap



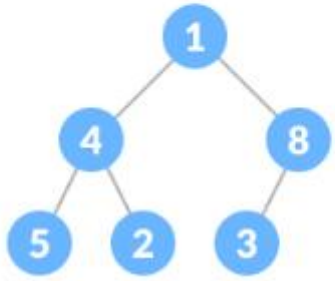
Max Heap

Max Heap

- Max heap data structure is a specialized full binary tree data structure.
- In a max heap nodes are arranged based on node value.
- Max heap is defined as follows...
- Max heap is a specialized full binary tree in which every parent node contains greater or equal value than its child nodes.

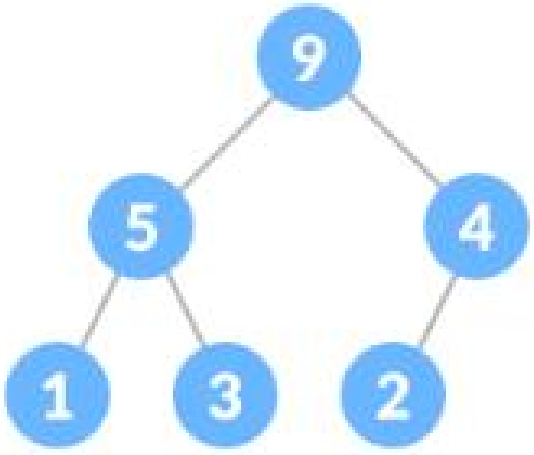
What is a heap?

- Heap data structure is a complete binary tree that satisfies **the heap property**. It is also called as **a binary heap**.
- A complete binary tree is a special binary tree in which
- every level, except possibly the last, is filled
- all the nodes are as far left as possible



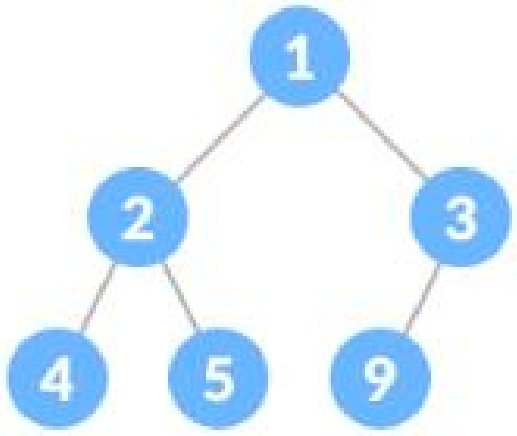
What is a heap?

- Heap Property is the property of a node in which
- (for max heap) key of each node is always greater than its child node/s and the key of the root node is the largest among all other nodes;



What is a heap?

- Heap Property is the property of a node in which
- (for min heap) key of each node is always smaller than the child node/s and the key of the root node is the smallest among all other nodes.



When are Heaps useful?

- Heaps are used when the highest or lowest order/priority element needs to be removed.
- They allow quick access to this item in O(1) time.
- One use of a heap is to implement a priority queue.
- Binary heaps are usually implemented using arrays, which save overhead cost of storing pointers to child nodes.

Basic operations

- insert aka push, add a new node into the heap
- remove aka pop, retrieves and removes the min or the max node of the heap
- examine aka peek, retrieves, but does not remove, the min or the max node of the heap

Heaps

- The heap property of a tree is a condition that must be true for the tree to be considered a heap.
- **Min-heap property:** for min-heaps, requires A[parent(i)] ≤ A[i] So, the root of any sub-tree holds the least value in that sub-tree.
- **Max-heap property:** for max-heaps, requires A[parent(i)] ≥ A[i] The root of any sub-tree holds the greatest value in the sub-tree.

Heaps

- Binary Heap. Min-heap. Max-heap.
- Efficient implementation of heap ADT: use of array
- Basic heap algorithms: ReheapUp, ReheapDown, Insert Heap, Delete Heap, Built Heap.
- Heap Applications:
  - Select Algorithm
  - Priority Queues
  - Heap sort
- Advanced implementations of heaps: use of pointers
  - Leftist heap
  - Skew heap
  - Binomial queues

Heaps

A **heap** is a certain kind of complete binary tree.

Heaps

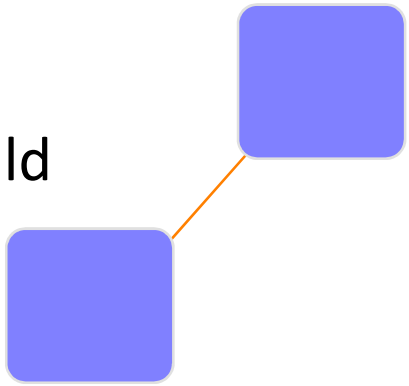
A **heap** is a certain kind of complete binary tree.

When a complete binary tree is built, its first node must be the root.

Heaps

Complete binary tree.

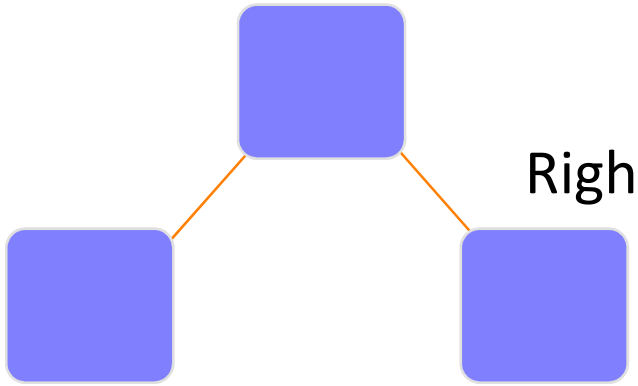
Left child of the root



The second node is always the left child of the root.

Heaps

Complete binary tree.

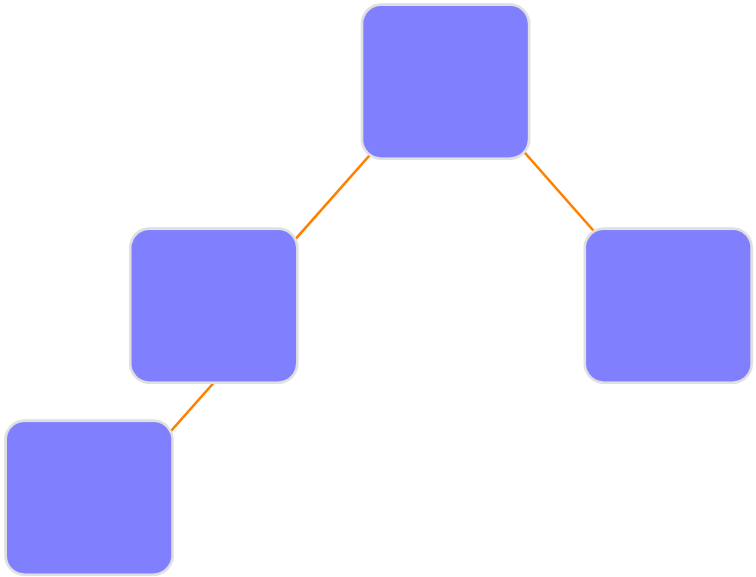


Right child of the root

The third node is always the right child of the root.

Heaps

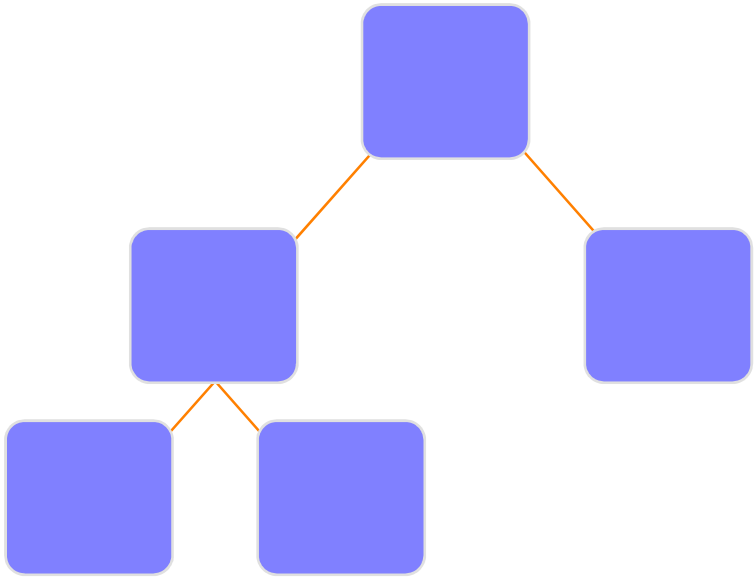
Complete binary tree.



The next nodes always fill the next level from left-to-right.

Heaps

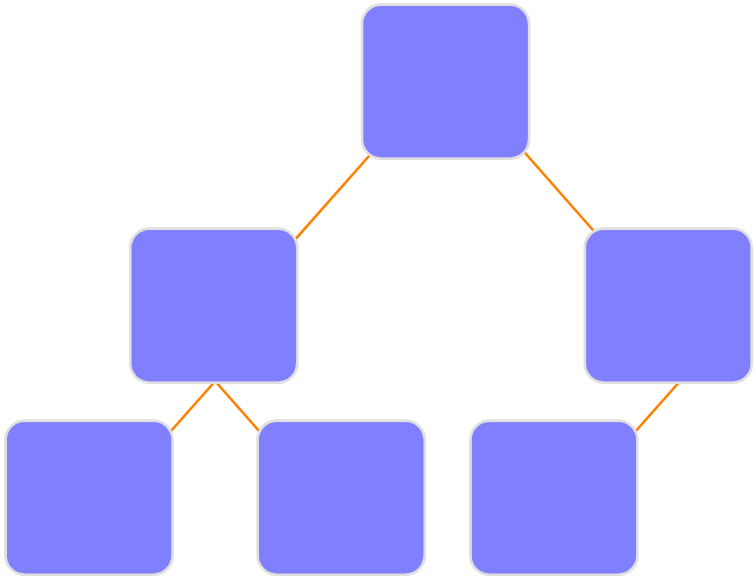
Complete binary tree.



The next nodes always fill the next level from left-to-right.

Heaps

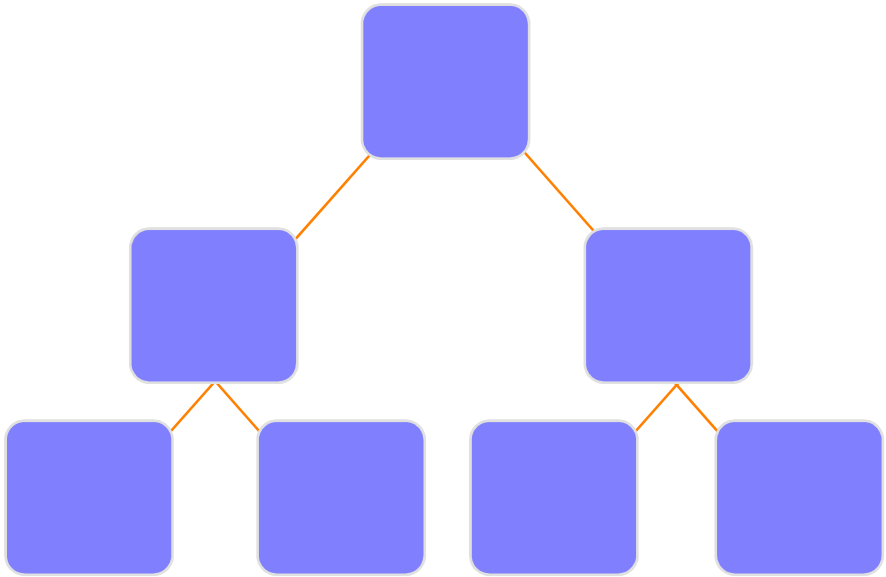
Complete binary tree.



The next nodes always fill the next level from left-to-right.

Heaps

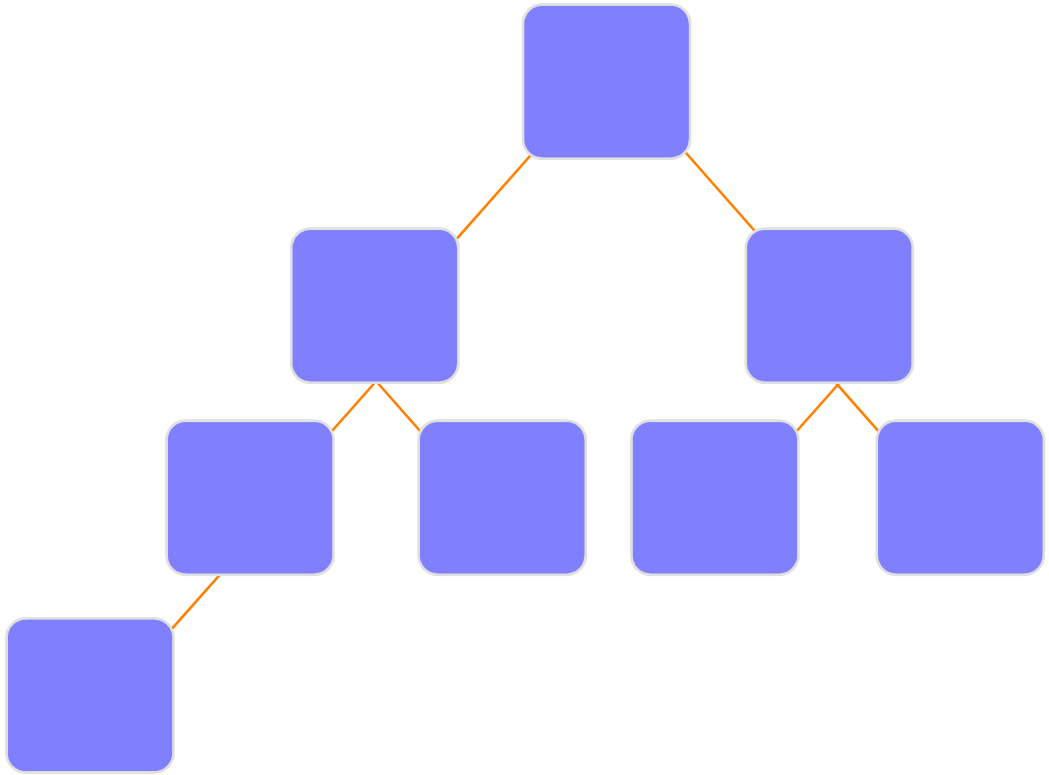
Complete binary tree.



The next nodes always fill the next level from left-to-right.

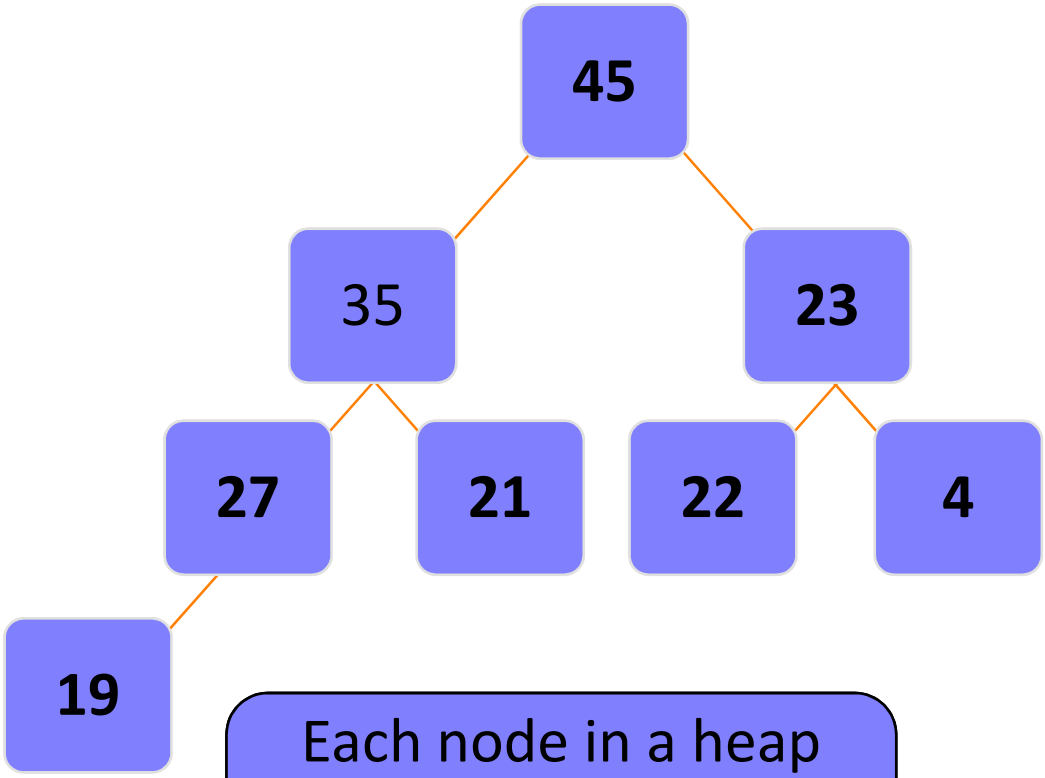
Heaps

Complete binary tree.



Heaps

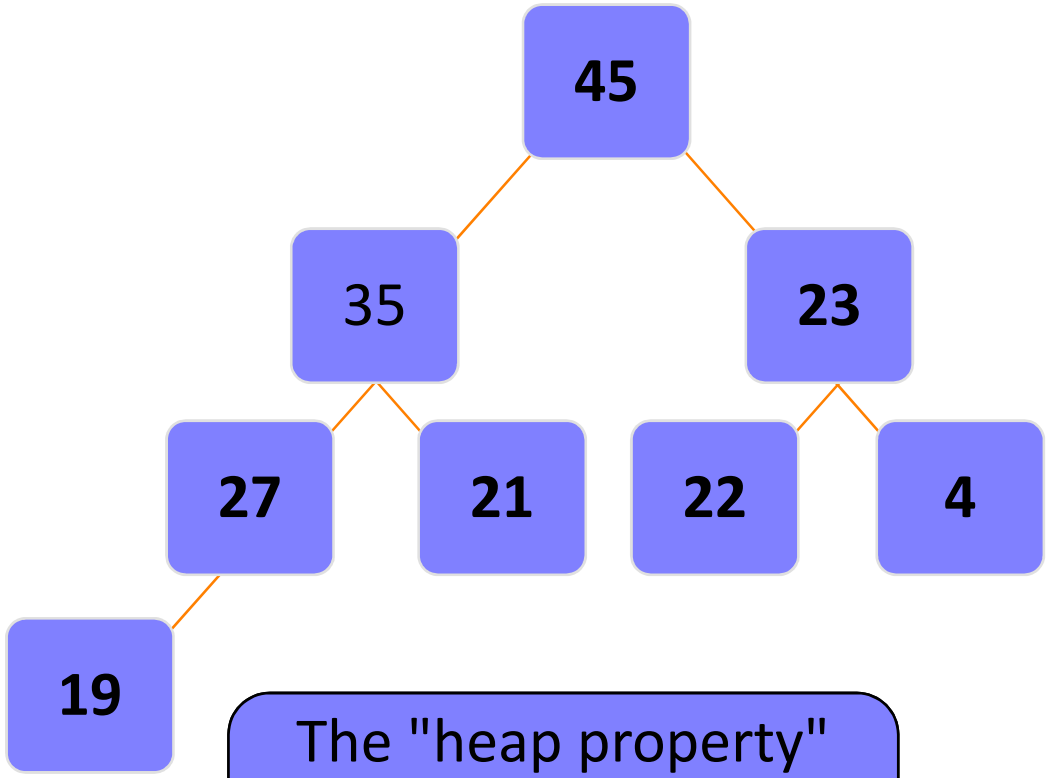
A heap is a **certain** kind of complete binary tree.



Each node in a heap contains a key that can be compared to other nodes' keys.

Heaps

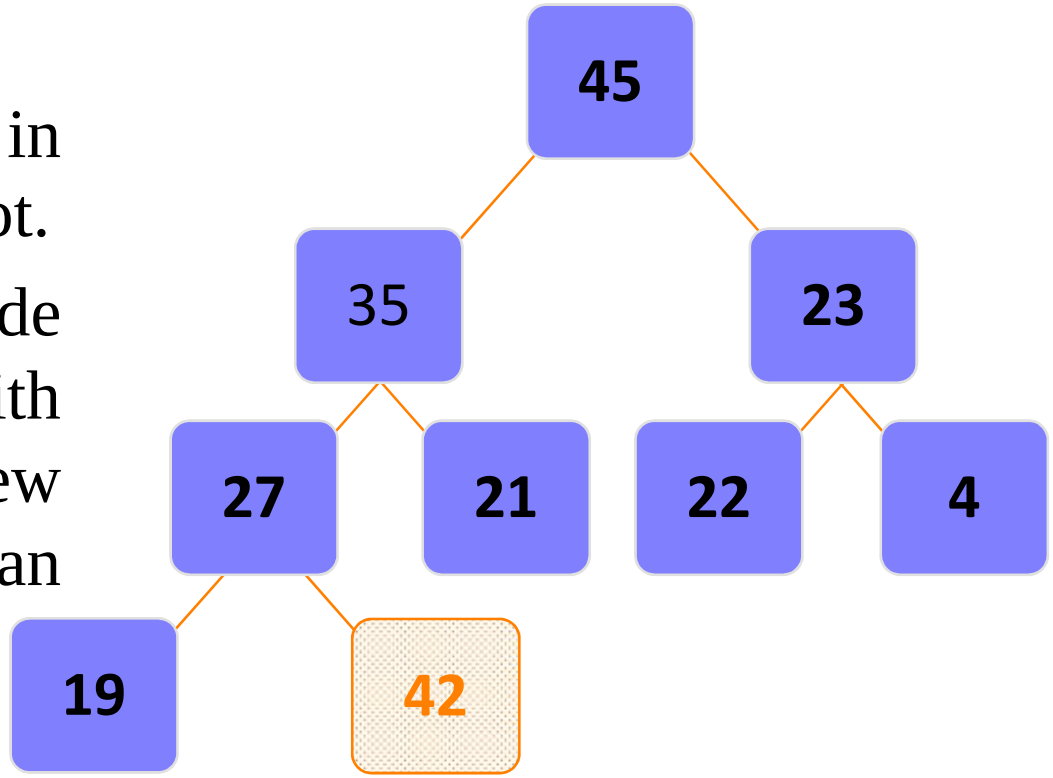
A heap is a **certain** kind of complete binary tree.



The "heap property" requires that each node's key is >= the keys of its children

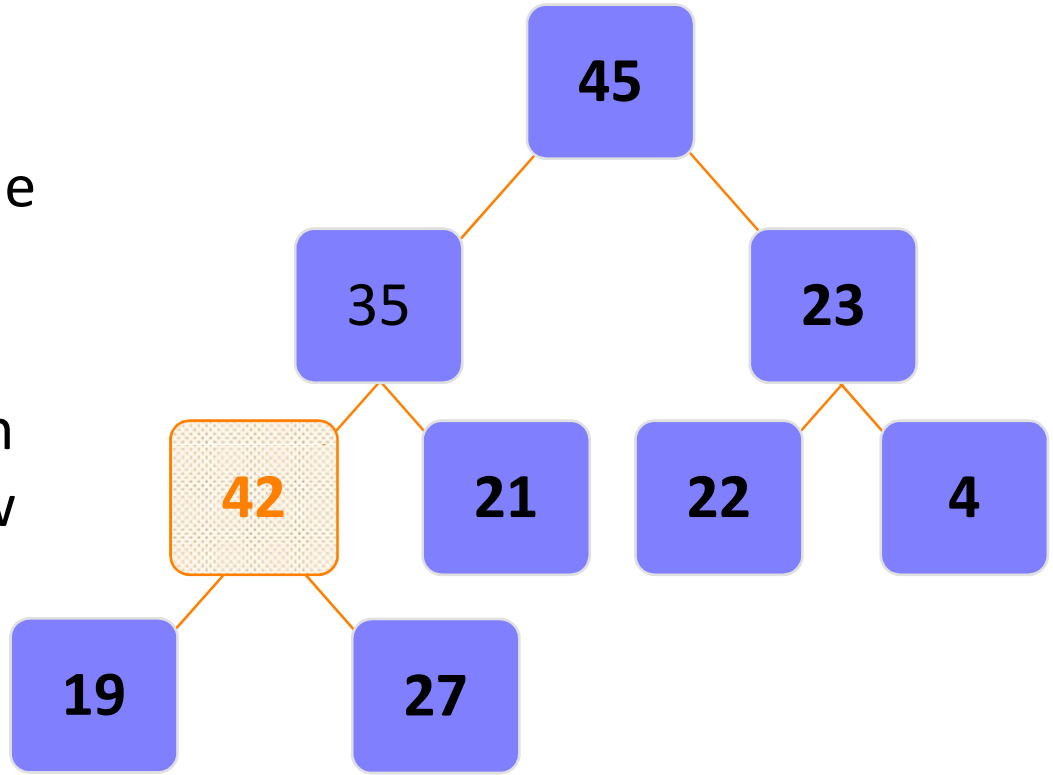
Adding a Node to a Heap

- Put the new node in the next available spot.
- Push the new node upward, swapping with its parent until the new node reaches an acceptable location.



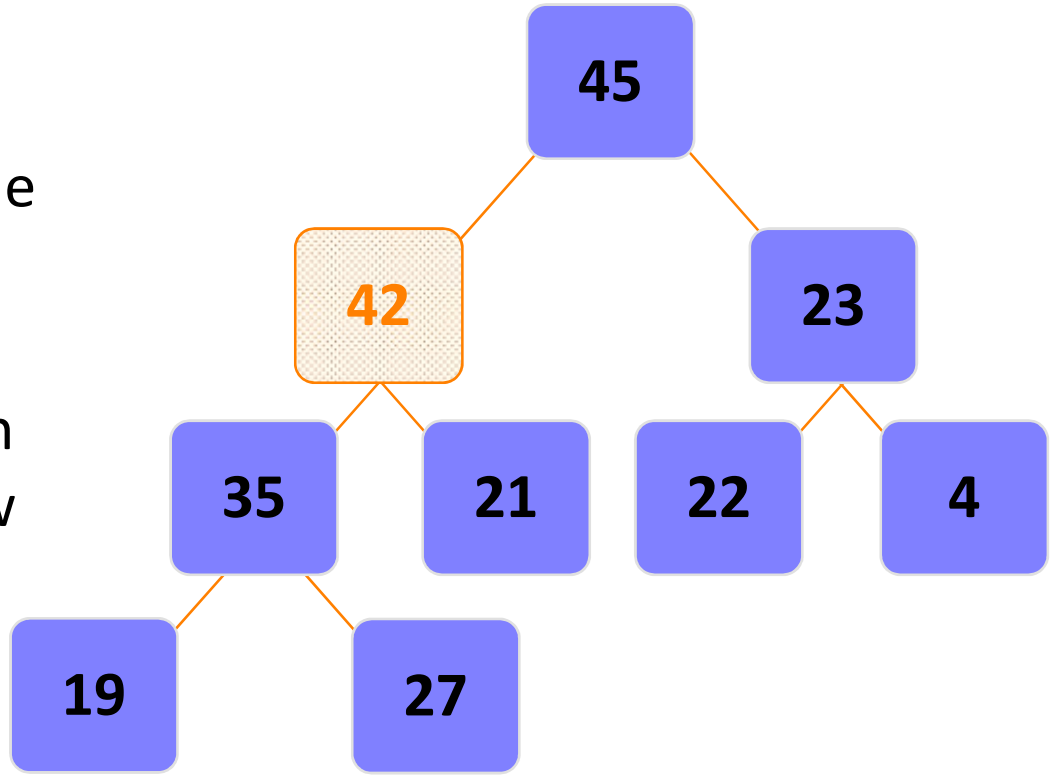
Adding a Node to a Heap

- Put the new node in the next available spot.
- Push the new node upward, swapping with its parent until the new node reaches an acceptable location.



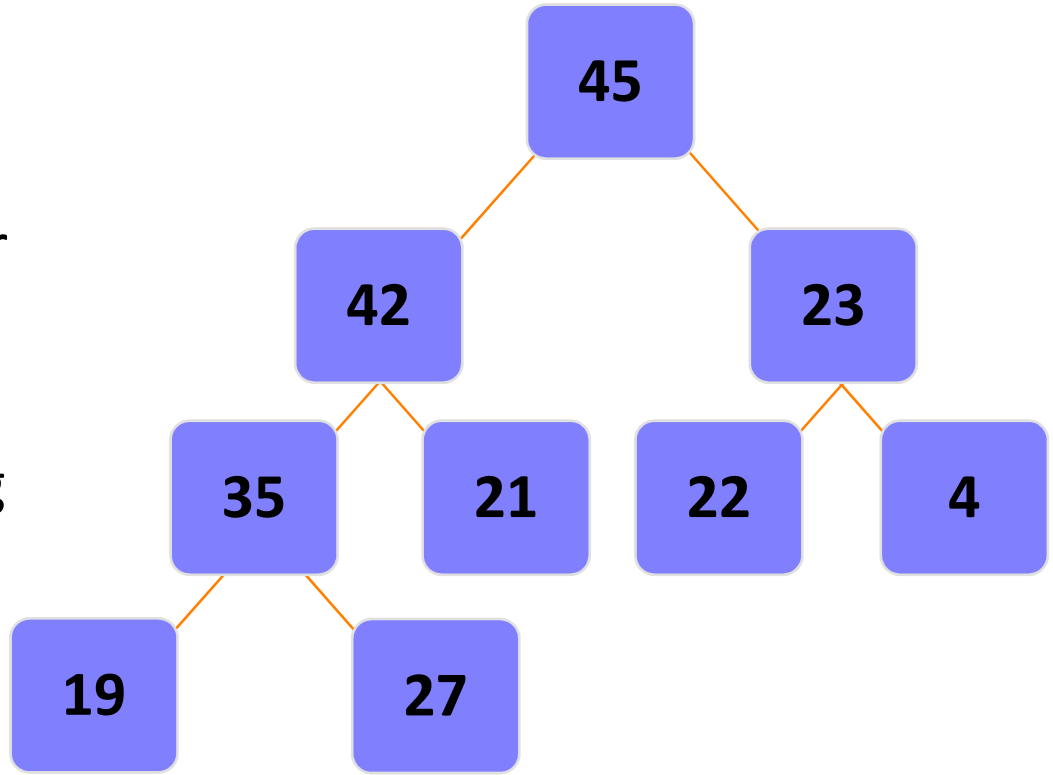
Adding a Node to a Heap

- Put the new node in the next available spot.
- Push the new node upward, swapping with its parent until the new node reaches an acceptable location.



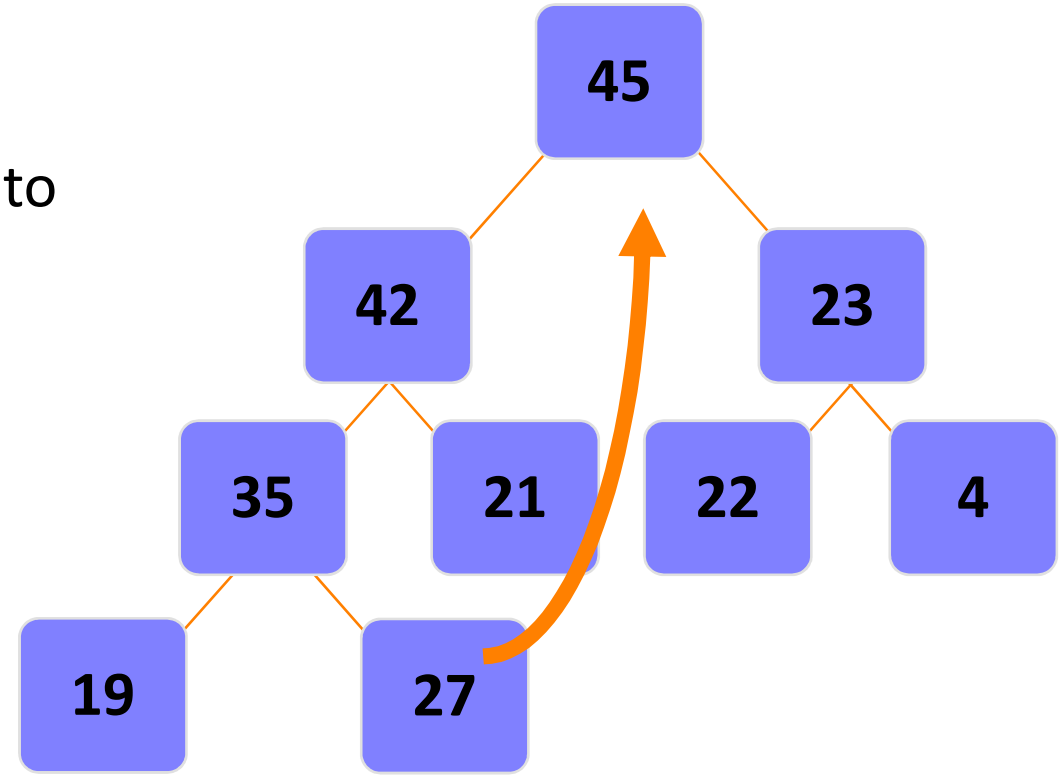
Adding a Node to a Heap

- The parent has a key that is >= new node, or
- The node reaches the root.
- The process of pushing the new node upward is called **reheapification upward**.



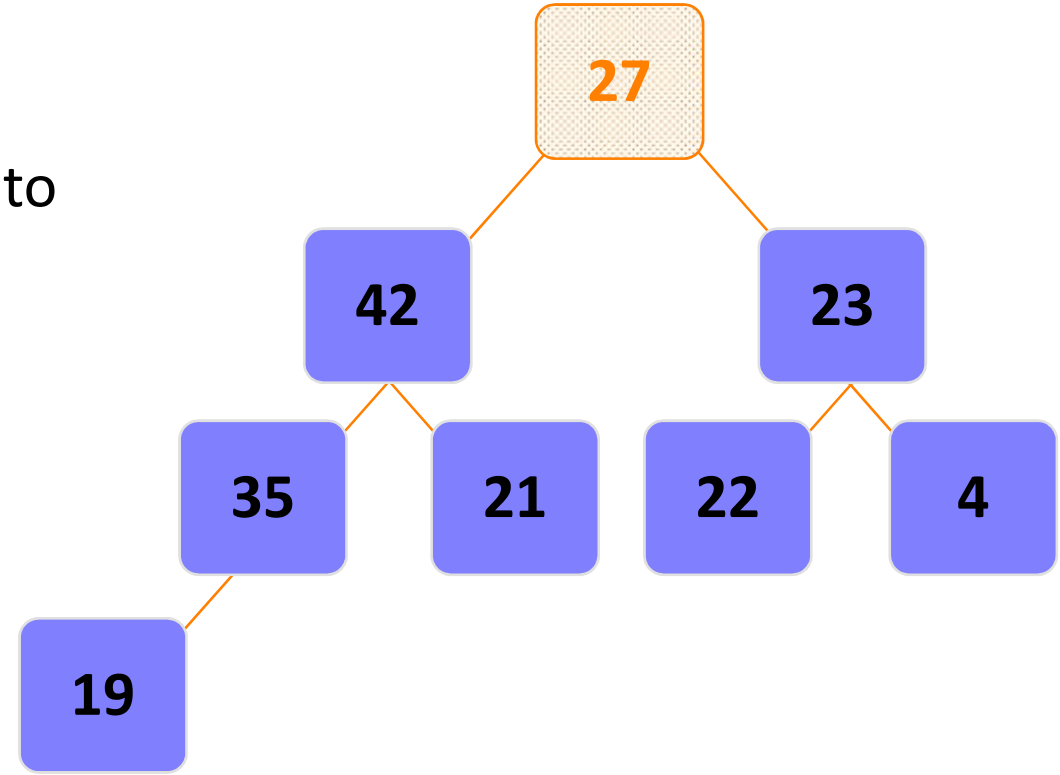
Removing the Top of a Heap

- Move the last node onto the root.



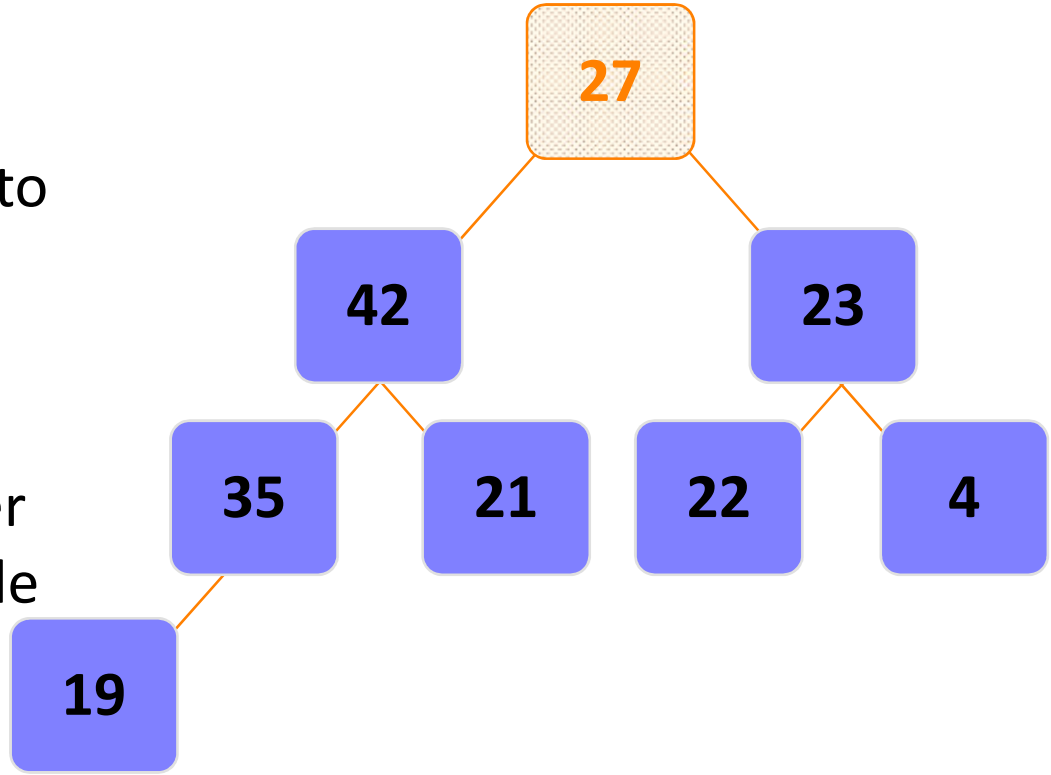
Removing the Top of a Heap

- Move the last node onto the root.



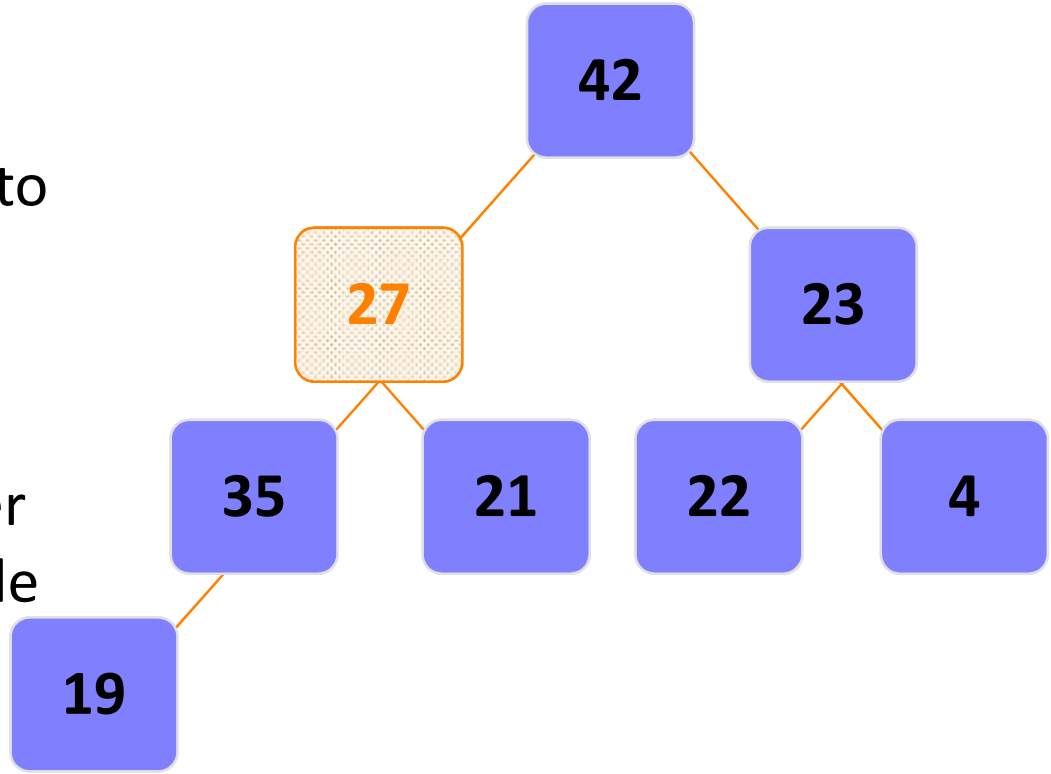
Removing the Top of a Heap

- Move the last node onto the root.
- Push the out-of-place node downward, swapping with its larger child until the new node reaches an acceptable location.



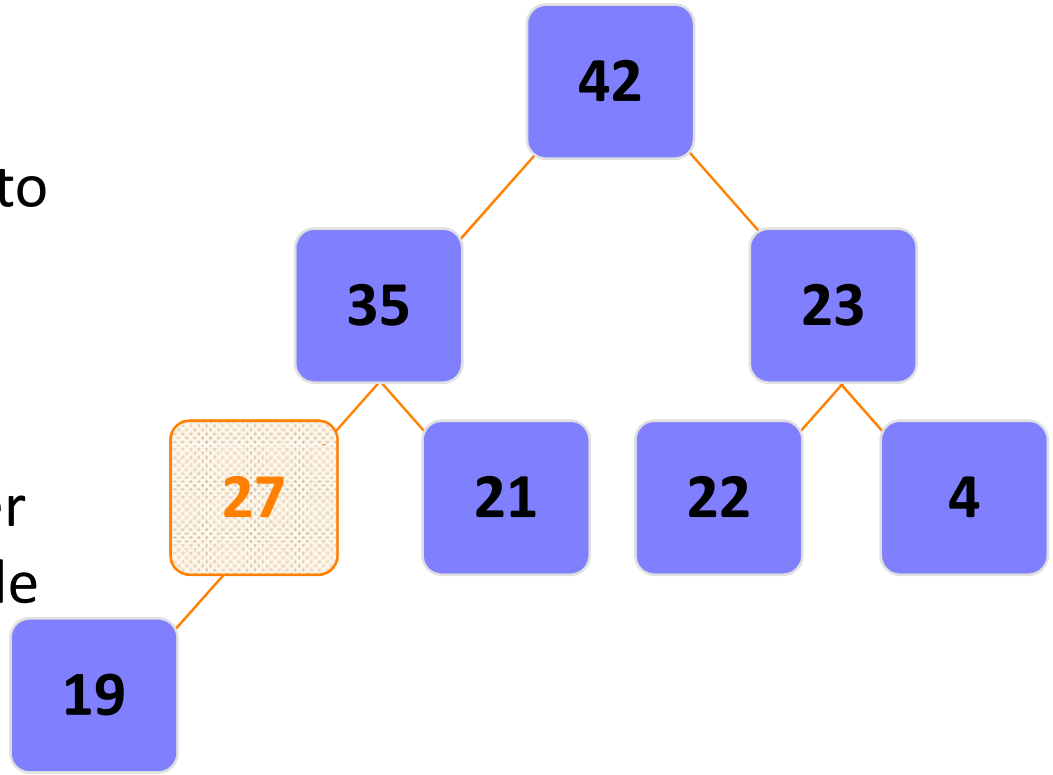
Removing the Top of a Heap

- Move the last node onto the root.
- Push the out-of-place node downward, swapping with its larger child until the new node reaches an acceptable location.



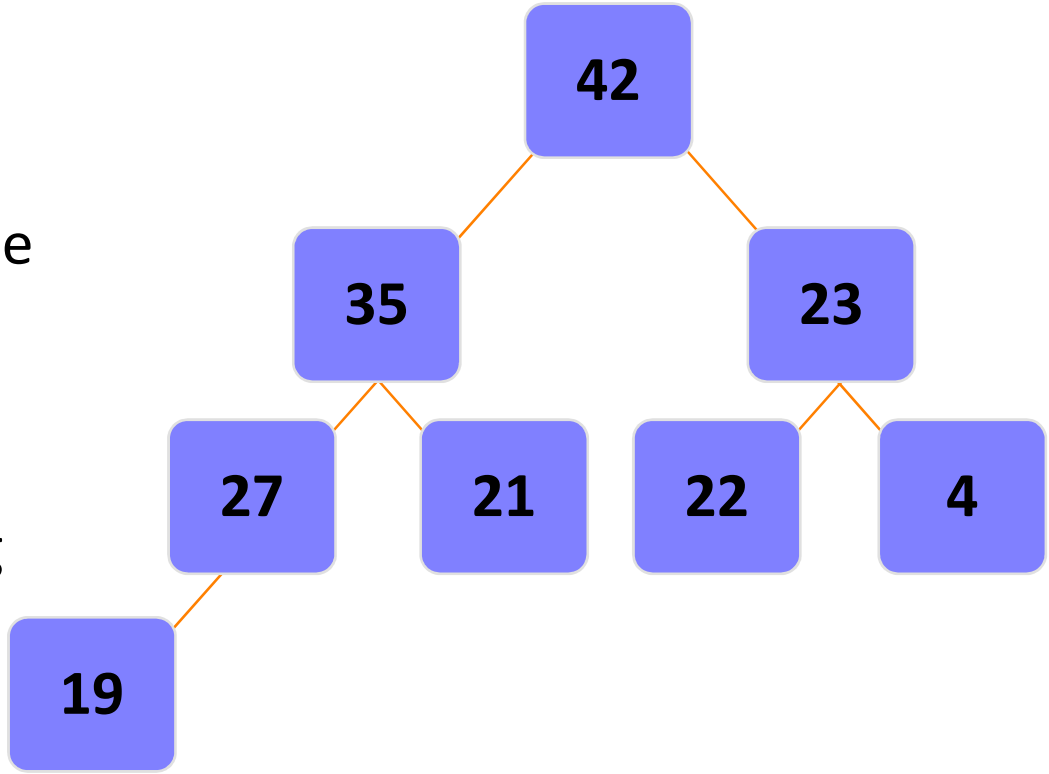
Removing the Top of a Heap

- Move the last node onto the root.
- Push the out-of-place node downward, swapping with its larger child until the new node reaches an acceptable location.



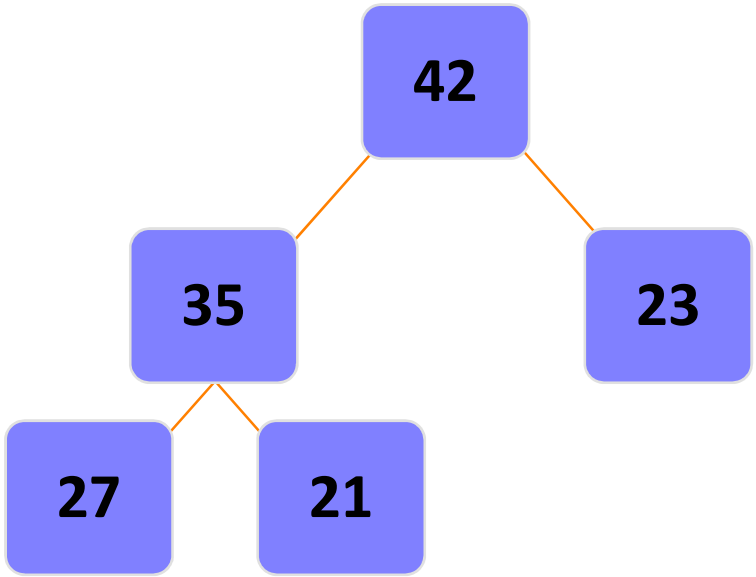
Removing the Top of a Heap

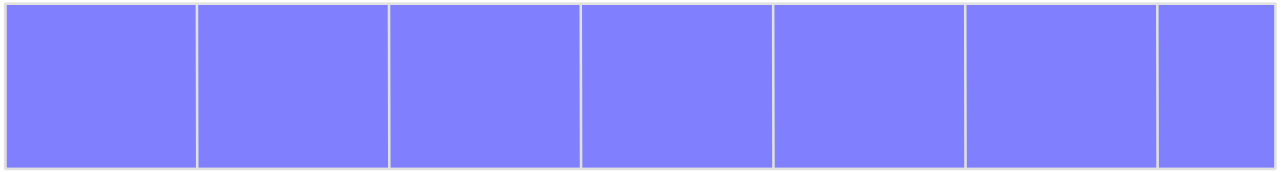
- The children all have keys <= the out-of-place node, or
- The node reaches the leaf.
- The process of pushing the new node downward is called **reheapification downward**.



Implementing a Heap

- We will store the data from the nodes in a partially-filled array.

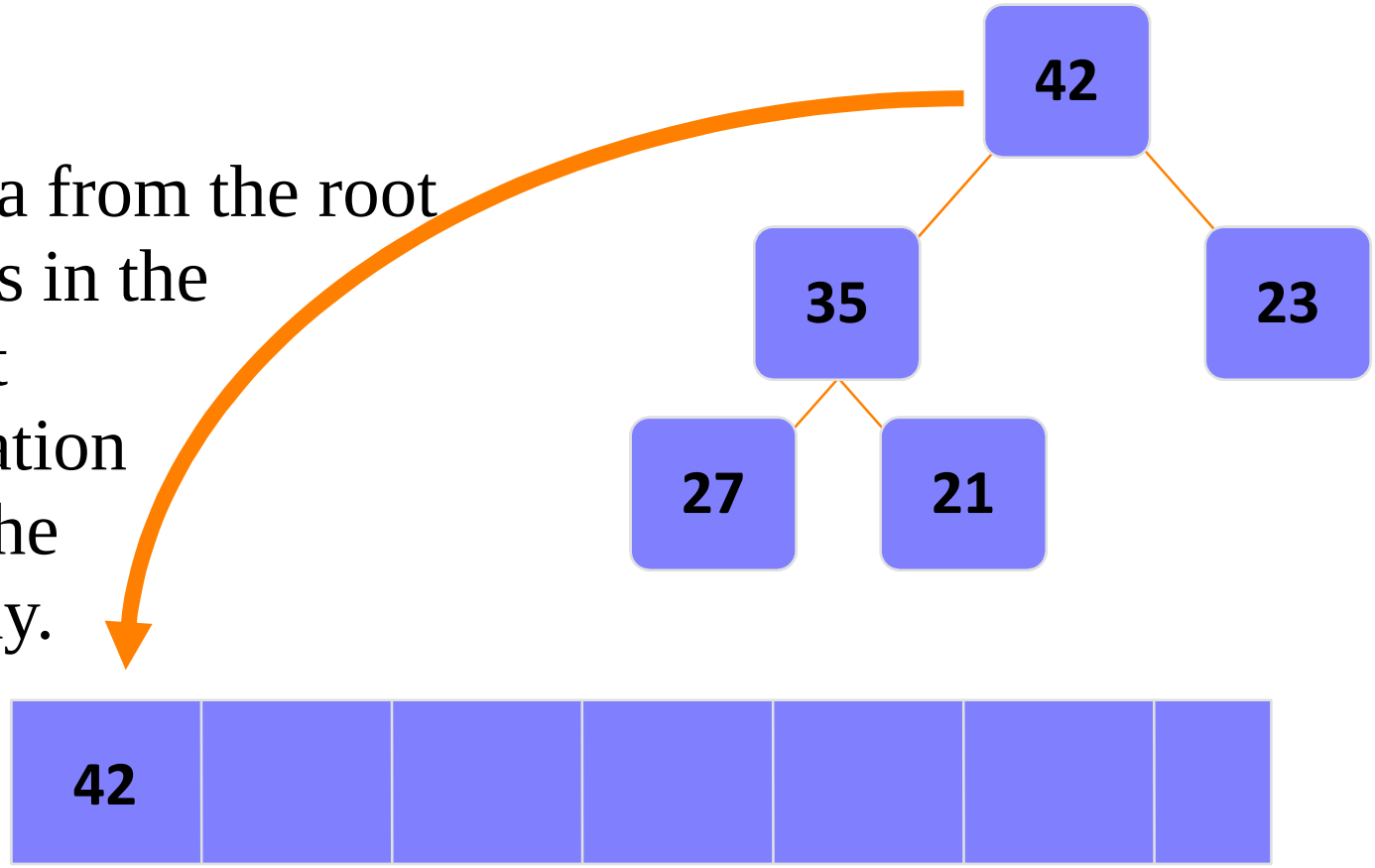




An array of data

Implementing a Heap

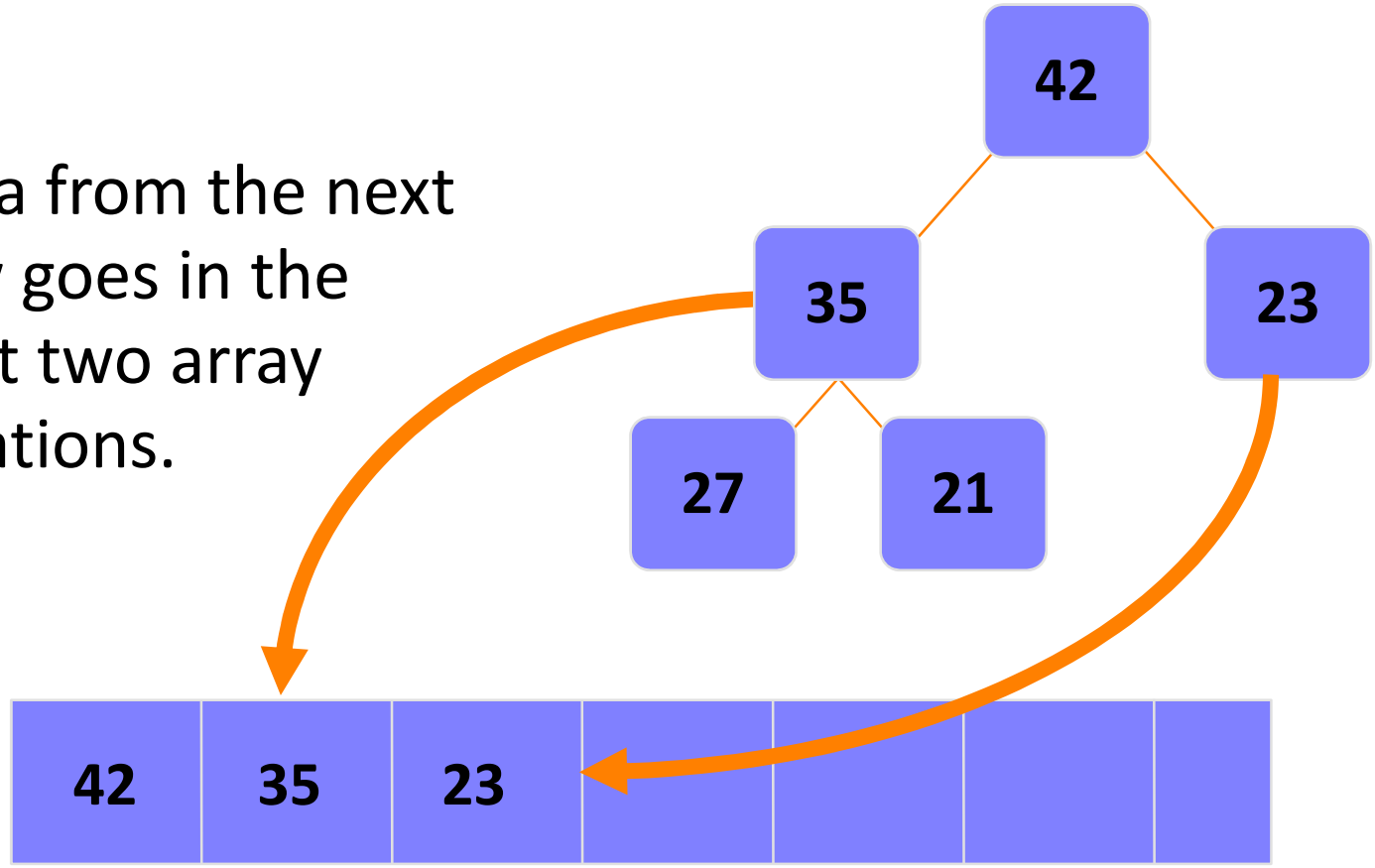
- Data from the root goes in the first location of the array.



An array of data

Implementing a Heap

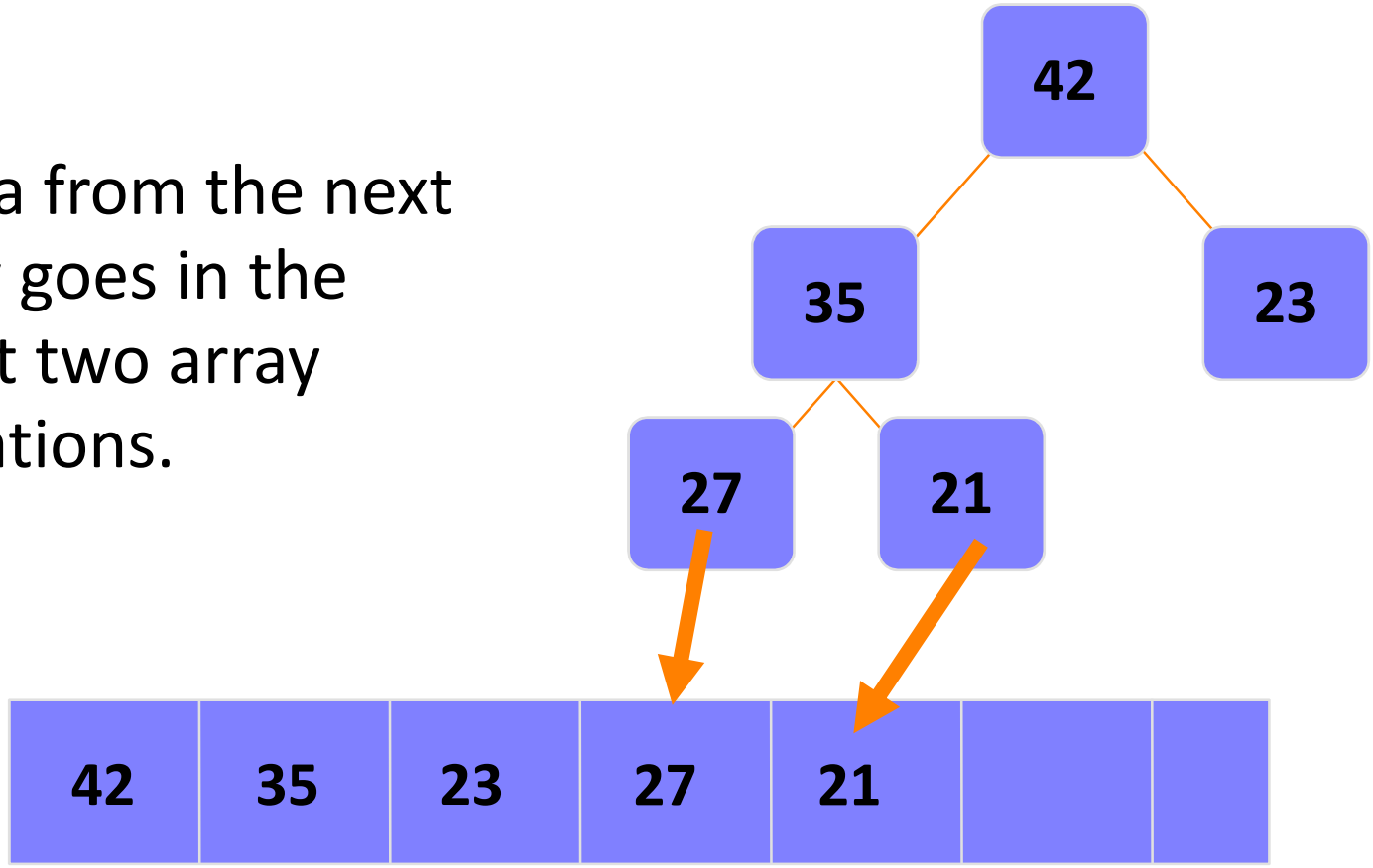
- Data from the next row goes in the next two array locations.



An array of data

Implementing a Heap

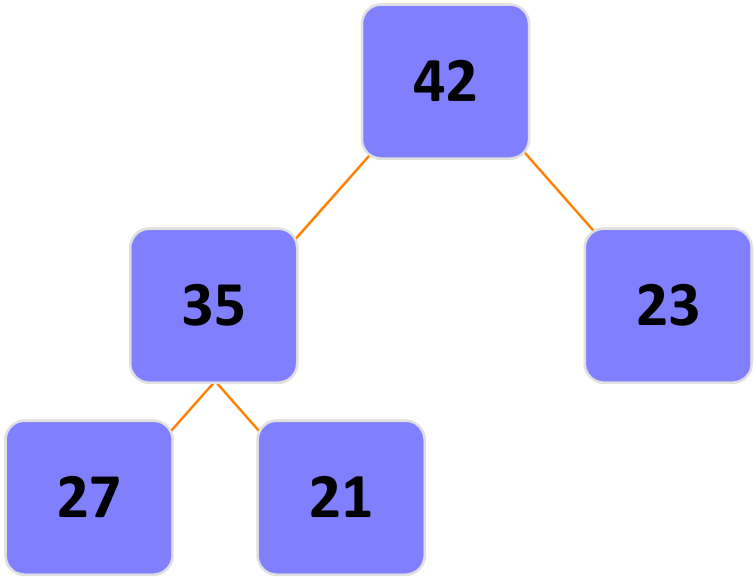
- Data from the next row goes in the next two array locations.



An array of data

Implementing a Heap

- Data from the next row goes in the next two array locations.



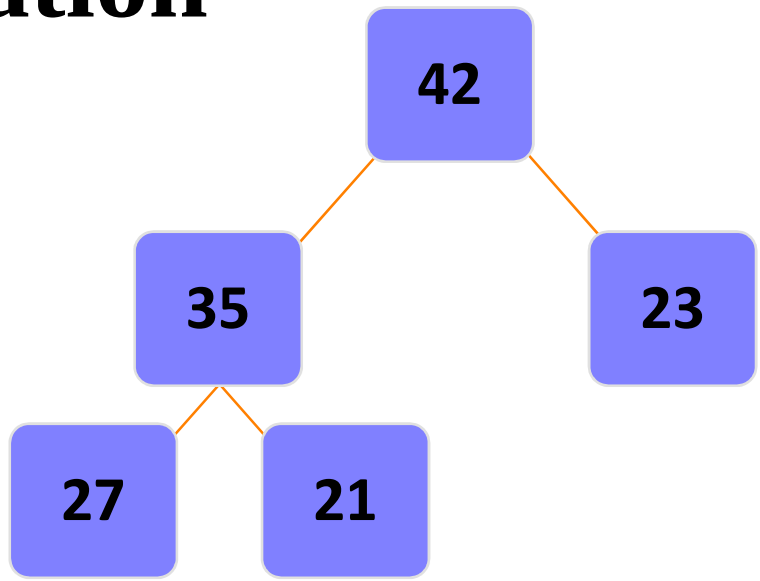
42	35	23	27	21		

An array of data

We don't care what's in this part of the array.

Important Points about the Implementation

- The links between the tree's nodes are not actually stored as pointers, or in any other way.
- The only way we "know" that "the array is a tree" is from the way we manipulate the data.

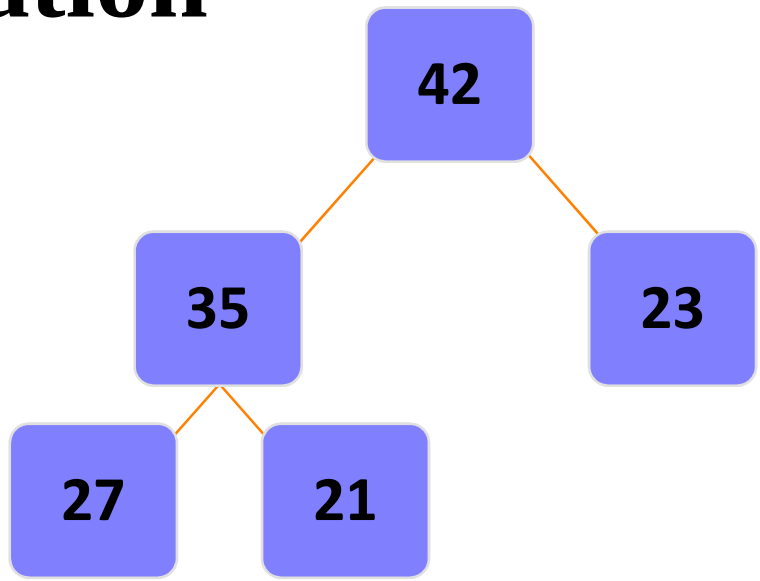


42	35	23	27	21		

An array of data

Important Points about the Implementation

- If you know the index of a node, then it is easy to figure out the indexes of that node's parent and children. Formulas are given in the book.



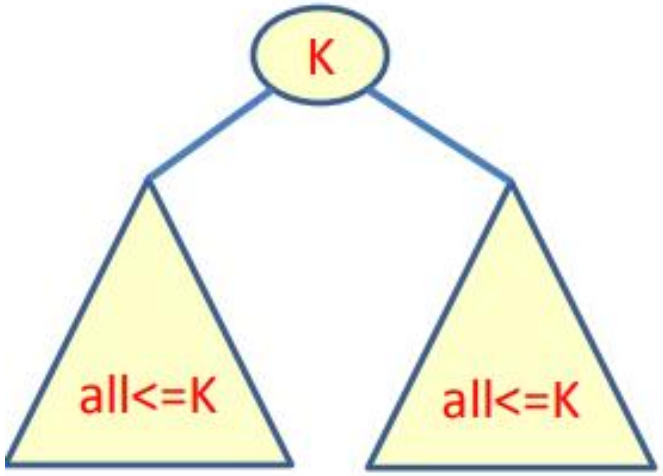
42	35	23	27	21		
[1]	[2]	[3]	[4]	[5]		

Summary

- A heap is a complete binary tree, where the entry at each node is greater than or equal to the entries in its children.
- To add an entry to a heap, place the new entry at the next available spot, and perform a reheapification upward.
- To remove the biggest entry, move the last node onto the root, and perform a reheapification downward.

Binary Heaps

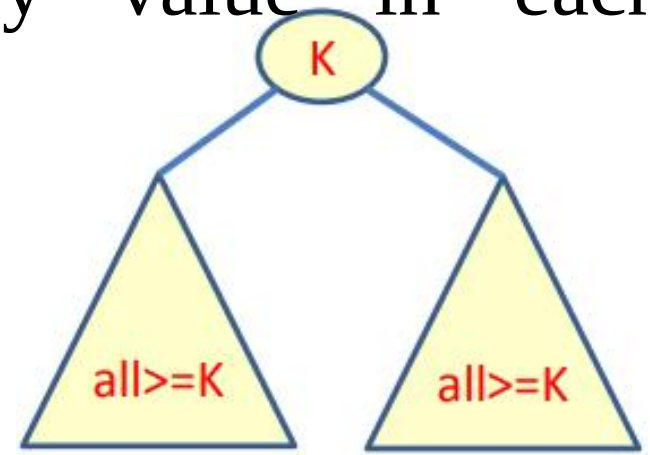
- DEFINITION: A max-heap is a binary tree structure with the following properties:
- The tree is complete or nearly complete.
- The key value of each node is greater than or equal to the key value



max-heap

Binary Heaps

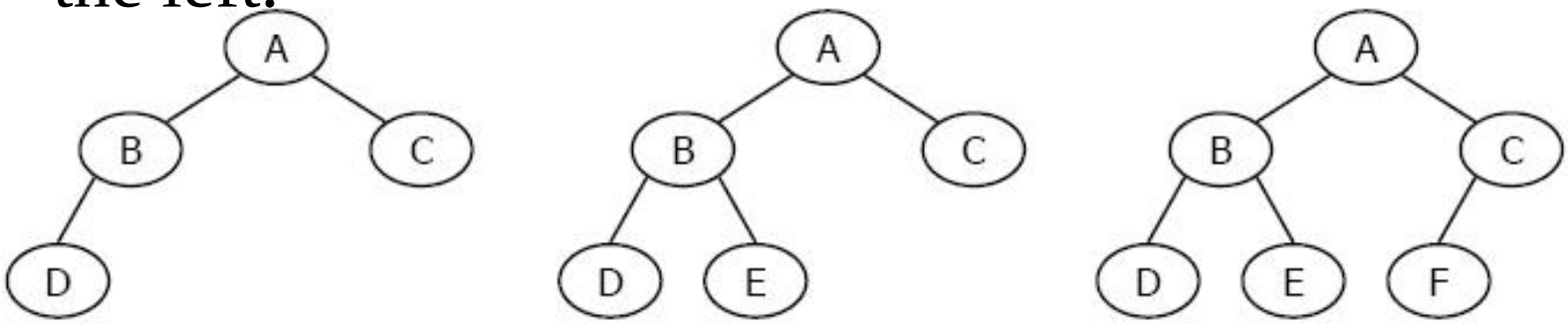
- DEFINITION: A min-heap is a binary tree structure with the following properties:
- The tree is complete or nearly complete.
- The key value of each node is less than or equal to the key value in each of its descendents.



min-heap

Properties of Binary Heaps

- Structure property of heaps
  - A complete or nearly complete binary tree.
  - If the height is h, the number of nodes n is between 2^{h-1} and $(2^h - 1)$
  - Complete tree: $n = 2^h - 1$ when last level is full.
  - Nearly complete: All nodes in the last level are on the left.

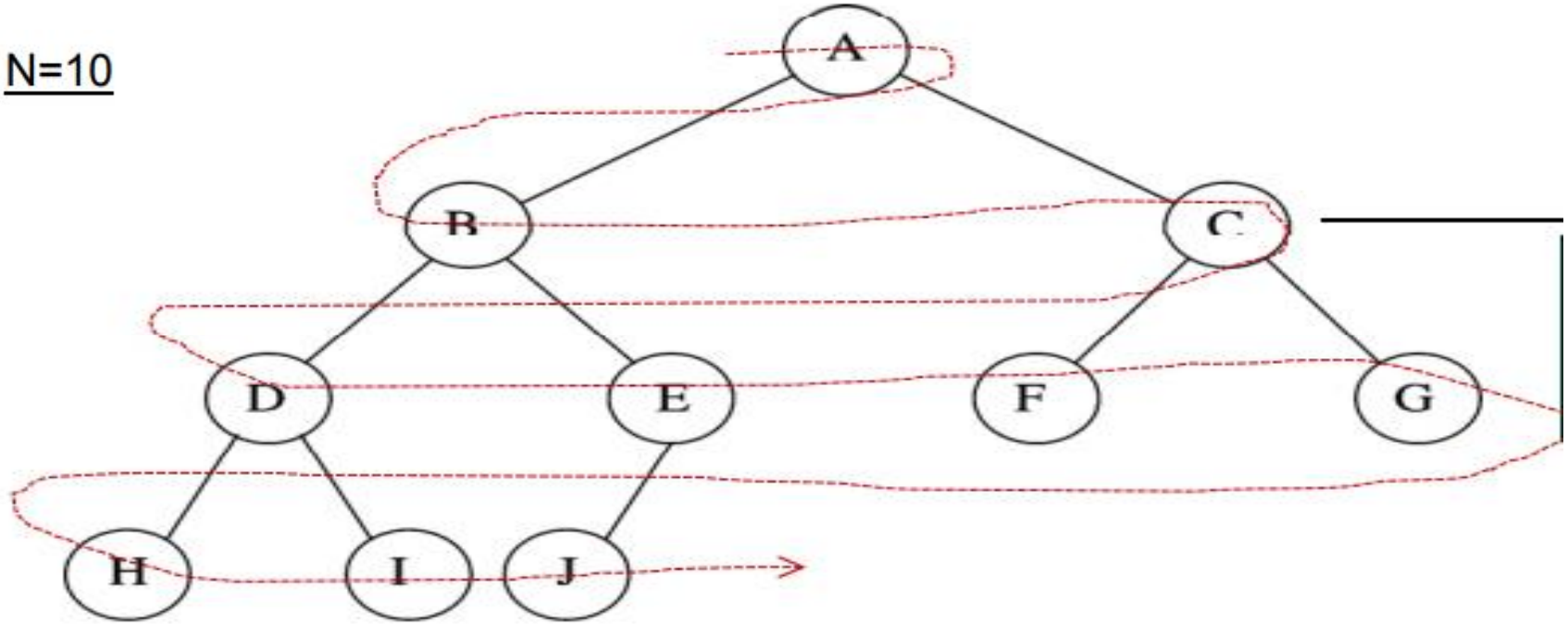


Properties of Binary Heaps

- A binary heap is a complete binary tree
- Each level (except possibly the bottom most level) is completely filled
- The bottom most level may be partially filled (from left to right)
- Height of a complete binary tree with N elements is $\lfloor \log_2 N \rfloor$

Binary Heap Example

N=10



Array representation:

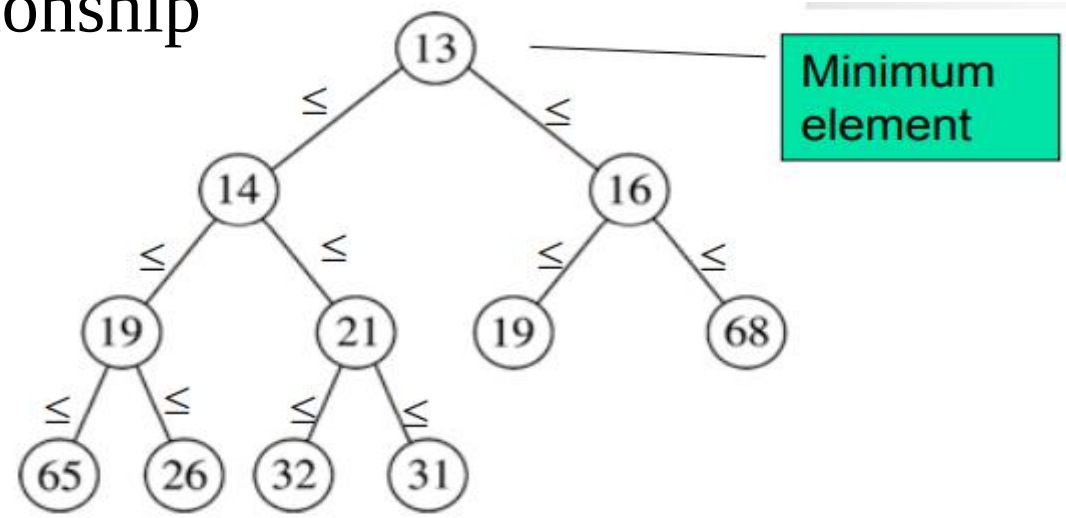
	A	B	C	D	E	F	G	H	I	J			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Properties of Binary Heaps

- Heap-order Property:
  - Heap-order property (for a “MinHeap”)
  - For every node X, key(parent(X)) ≤ key(X)
  - Except root node, which has no parent
- Thus, minimum key always at root
  - Alternatively, for a “MaxHeap”, always keep the maximum key at the root
- Insert and deleteMin must maintain heap - order property

Properties of Binary Heaps

- Heap-order Property:
  - Duplicates are allowed
  - No order implied for elements which do not share ancestor share ancestor -descendant relationship descendant relationship



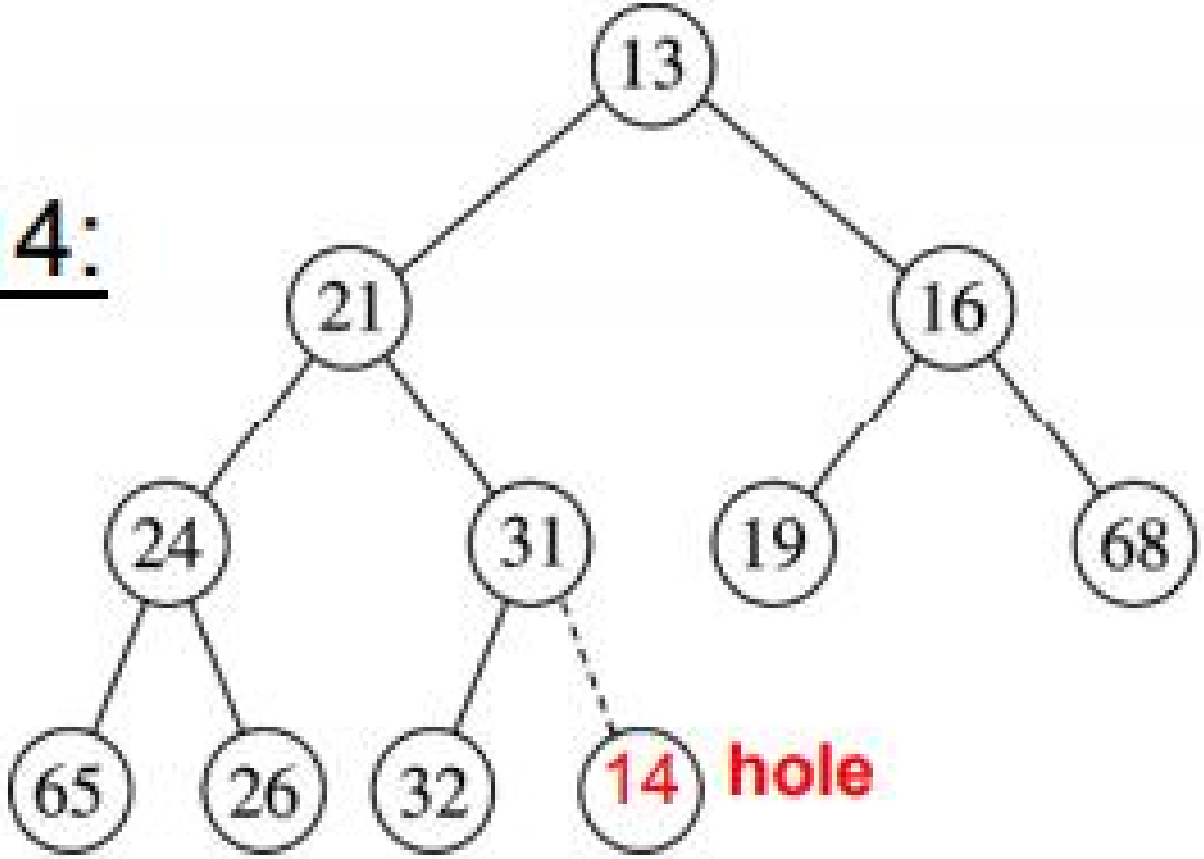
Heap Insert

- Insert new element into the heap at the next available slot (next available slot (hole) “hole”)
- According to maintaining a complete binary tree
- Then, “percolate” the element up the heap while heap heap while heap-order property not order property not satisfied

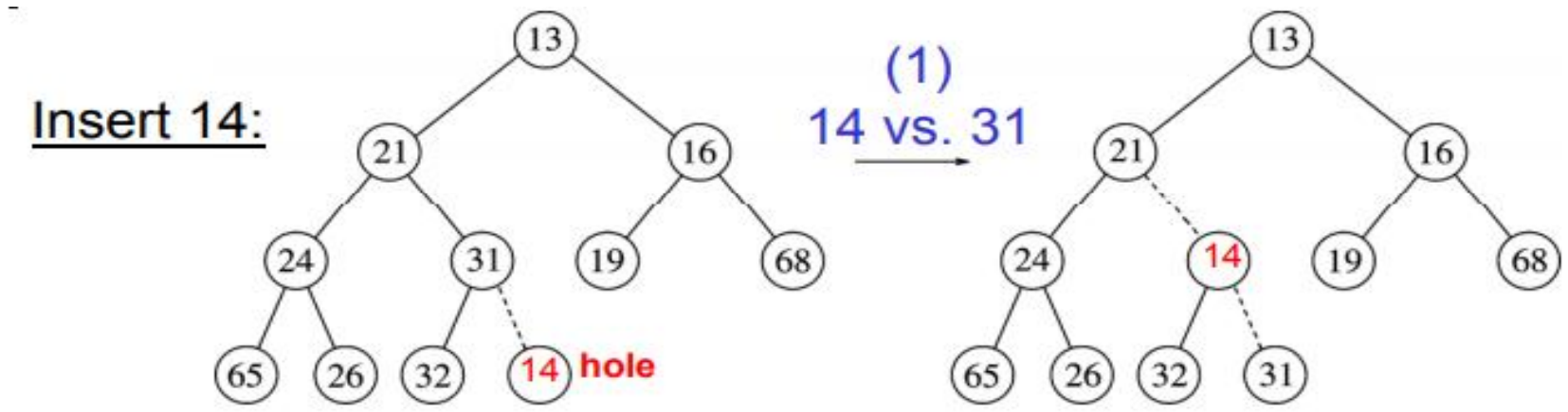
Heap Insert

-

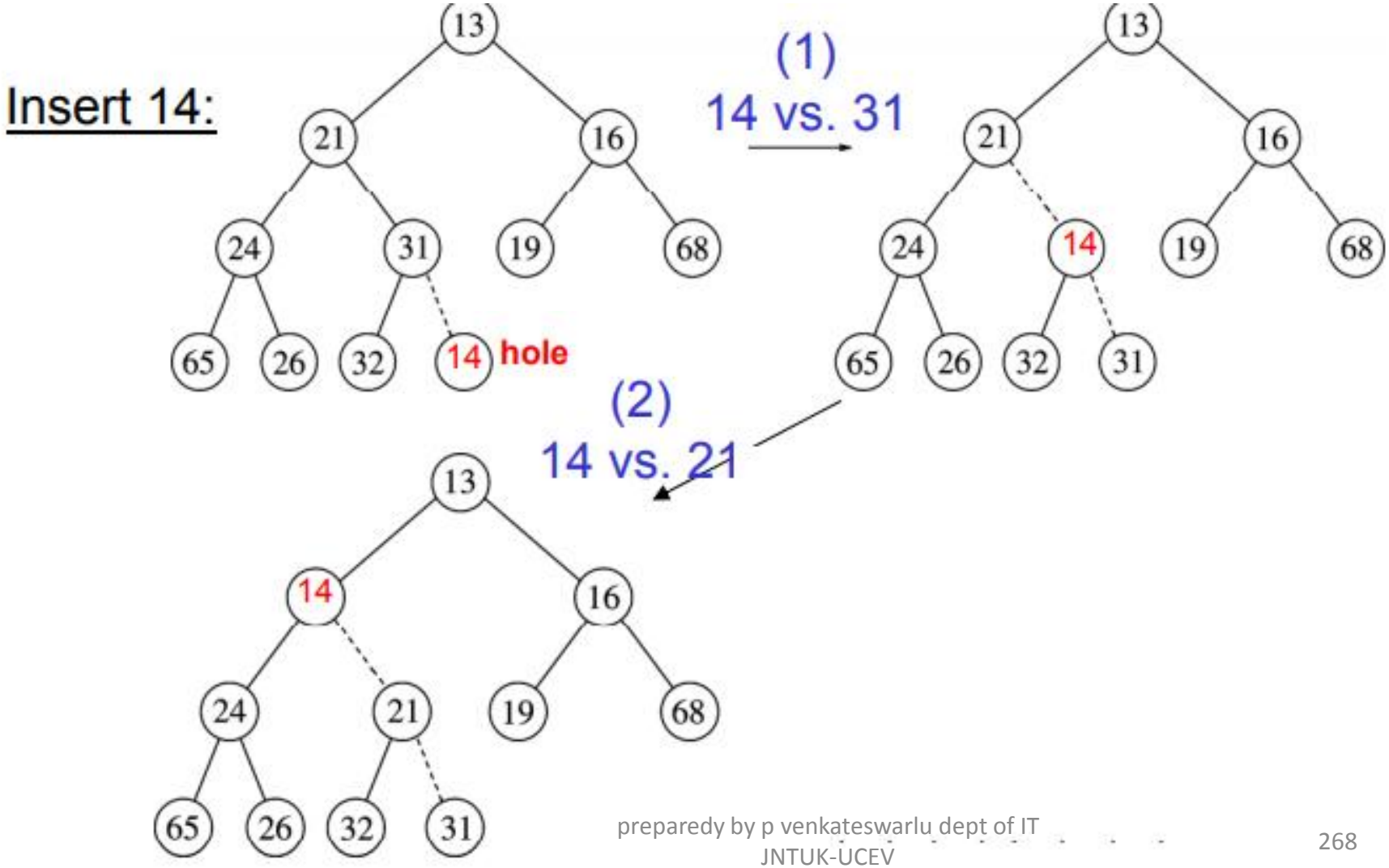
Insert 14:



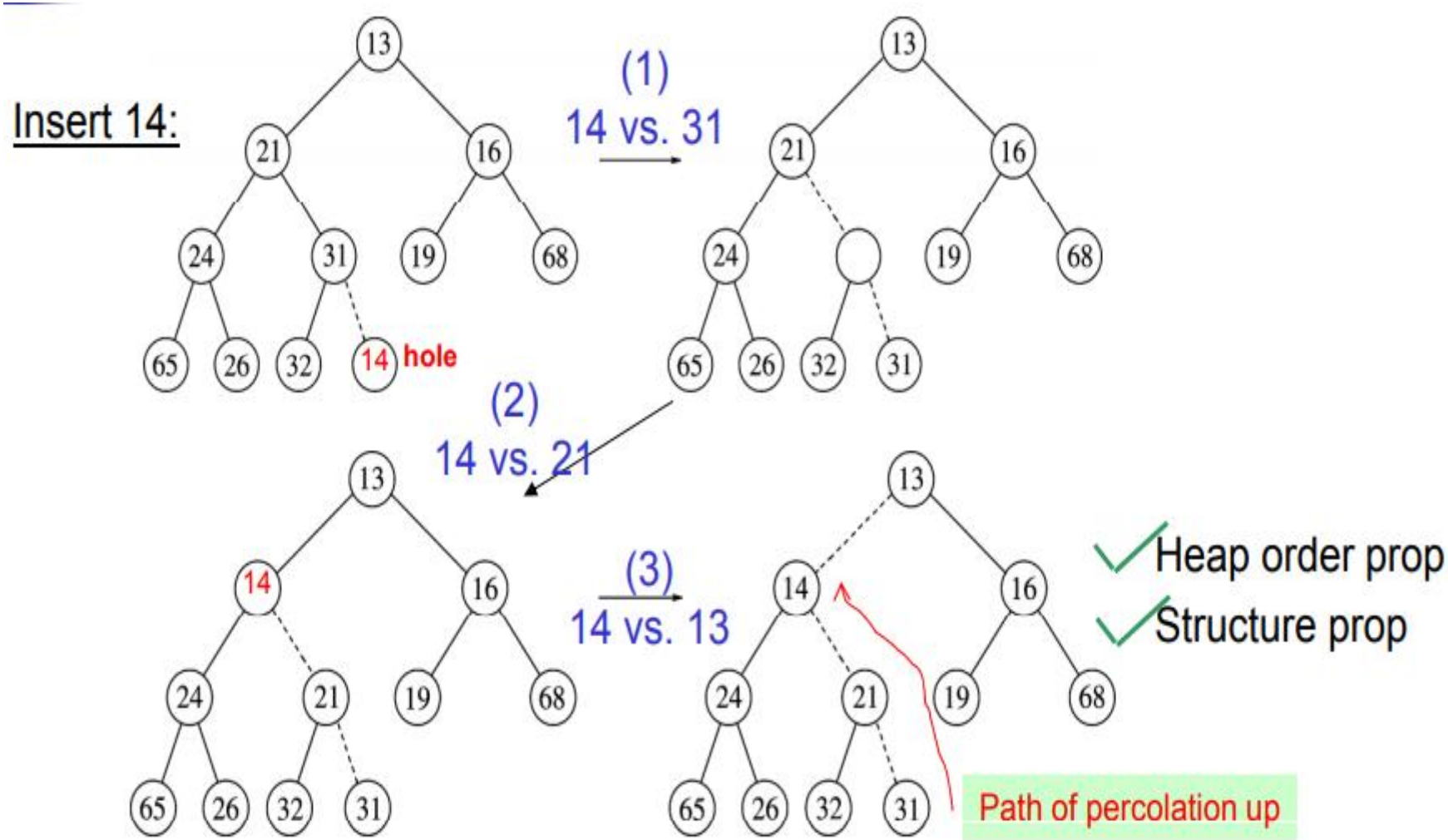
Heap Insert


Insert 14:
13
21
16
24
31
19
68
65
26
32
14
hole
(1)
14 vs. 31
13
21
16
24
14
19
68
65
26
32
31

Heap Insert


Insert 14:
13
21
16
24
31
19
68
65
26
32
14
hole
(1)
14 vs. 31
13
21
16
24
14
19
68
65
26
32
31
(2)
14 vs. 21
13
14
16
24
21
19
68
65
26
32
31

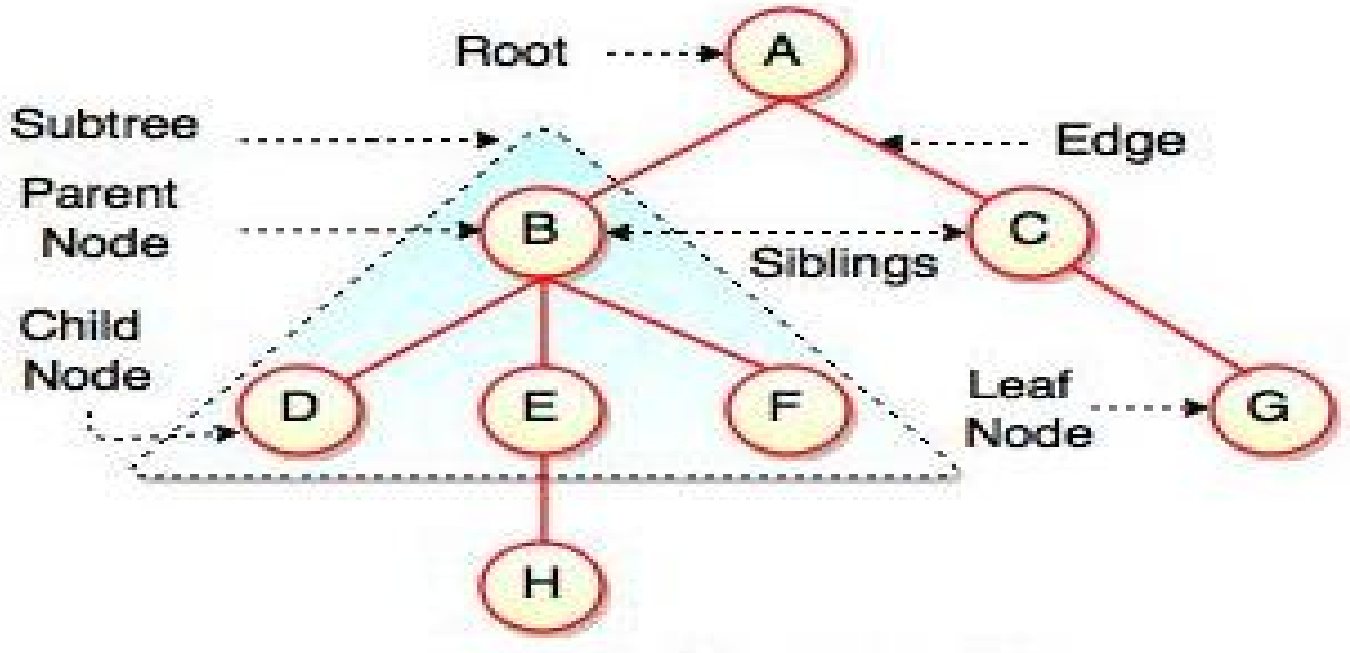
Heap Insert


Insert 14:
(1)
14 vs. 31
hole
(2)
14 vs. 21
(3)
14 vs. 13
Heap order prop
Structure prop
Path of percolation up

What are trees?

- Tree is a hierarchical data structure which stores the information naturally in the form of hierarchy style.
- Tree is one of the most powerful and advanced data structures.
- It is a non-linear data structure compared to arrays, linked lists, stack and queue.
- It represents the nodes connected by edges.

What are trees?



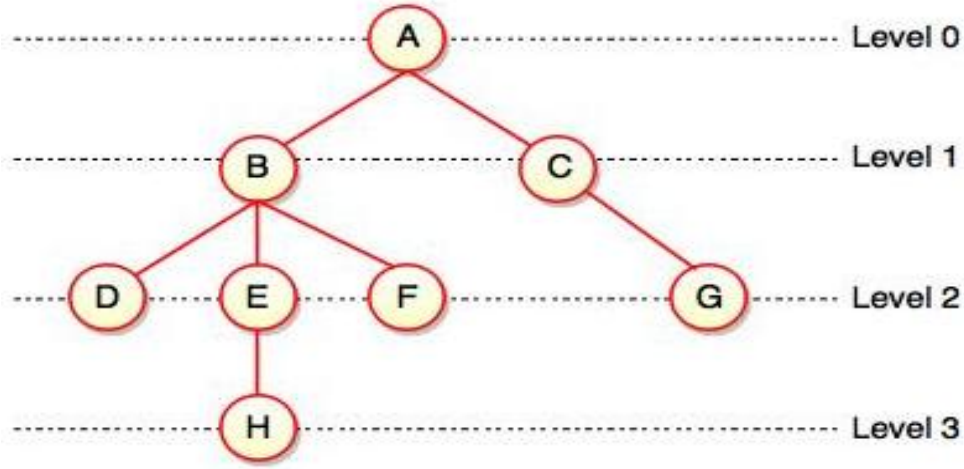
- The above figure represents structure of a tree. Tree has 2 subtrees.
- A is a parent of B and C.
- B is called a child of A and also parent of D, E, F.

What are trees?

Field	Description
Root	Root is a special node in a tree. The entire tree is referenced through it. It does not have a parent.
Parent Node	Parent node is an immediate predecessor of a node.
Child Node	All immediate successors of a node are its children.
Siblings	Nodes with the same parent are called Siblings.
Path	Path is a number of successive edges from source node to destination node.
Height of Node	Height of a node represents the number of edges on the longest path between that node and a leaf.
Height of Tree	Height of tree represents the height of its root node.
Depth of Node	Depth of a node represents the number of edges from the tree's root node to the node.
Degree of Node	Degree of a node represents a number of children of a node.
Edge	Edge is a connection between one node to another. It is a line between two nodes or a node and a leaf.

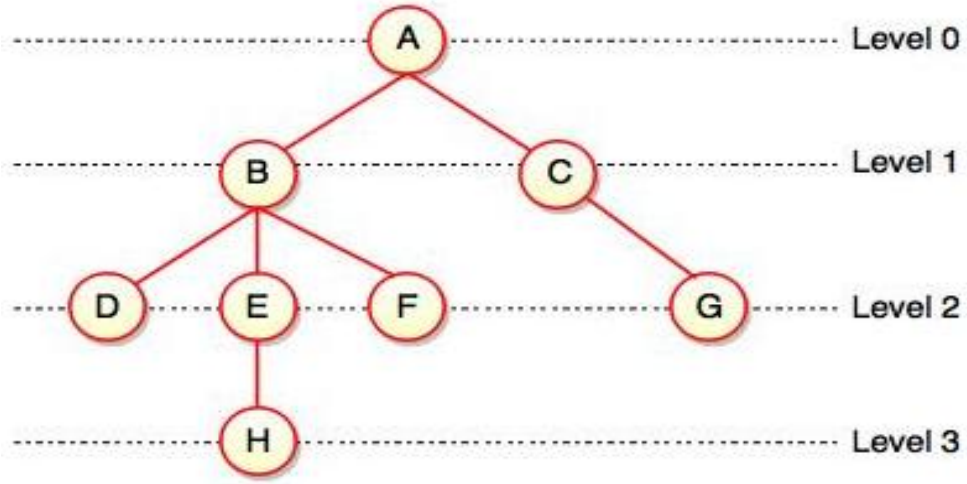
What are trees?

- **Levels of a node:** Levels of a node represents the number of connections between the node and the root. It represents generation of a node. If the root node is at level 0, its next node is at level 1, its grand child is at level 2 and so on. Levels of a node can be shown as follows:



What are trees?

- **Levels of a node:**
  - If node has no children, it is called **Leaves** or **External Nodes.**
  - Nodes which are not leaves, are called **Internal Nodes**. Internal nodes have at least one child.
  - A tree can be empty with no nodes or a tree consists of one node called the **Root**.

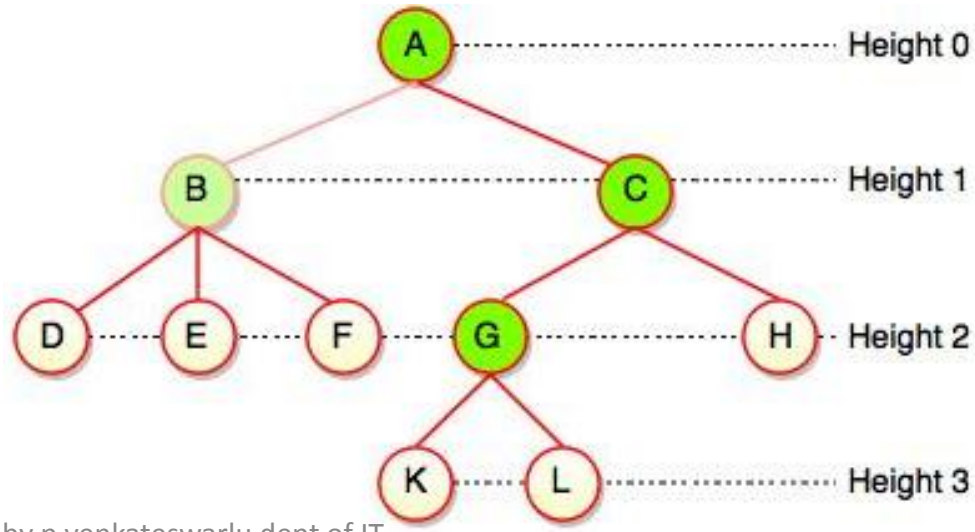


What are trees?

- **Height of a Node**
- height of a node is a number of edges on the longest path between that node and a leaf. Each node has height.
- In the above figure, A, B, C, D can have height. Leaf cannot have height as there will be no path starting from a leaf. Node A's height is the number of edges of the path to K not to D. And its height is 3.

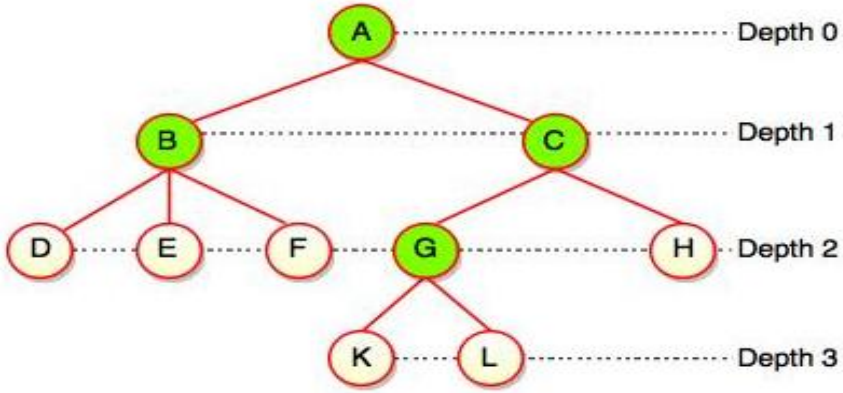
What are trees?

- **Height of a Node:**
  - Height of a node defines the longest path from the node to a leaf.
  - Path can only be downward.



What are trees?

- **Depth of a Node**
  - While talking about the height, it locates a node at bottom where for depth, it is located at top which is root level and therefore we call it depth of a node.
  - In the above figure, Node G's depth is 2. In depth of a node, we just count how many edges between the targeting node & the root and ignoring the directions.

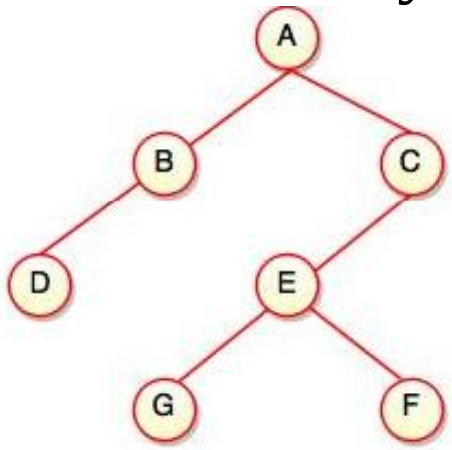


Binary Tree

- Binary tree is a special type of data structure. In binary tree, every node can have a maximum of 2 children, which are known as **Left child** and **Right Child**.
- It is a method of placing and locating the records in a database, especially when all the data is known to be in random access memory (RAM)

Binary Tree

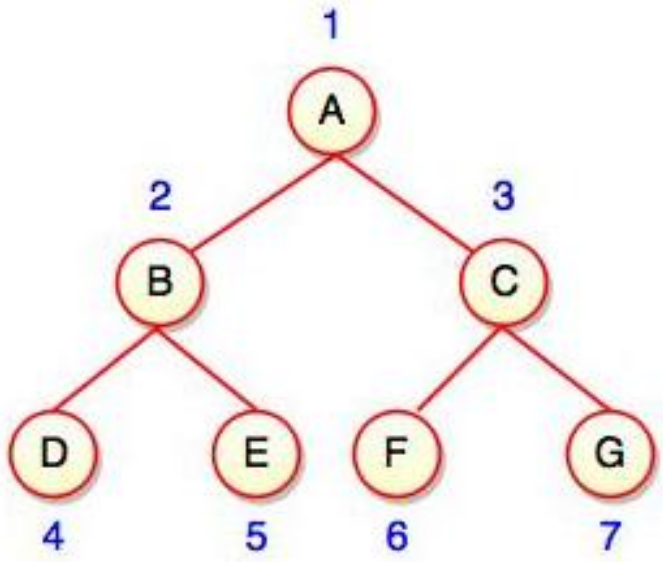
- "A tree in which every node can have maximum of two children is called as Binary Tree.“



- The above tree represents binary tree in which node A has two children B and C. Each children have one child namely D and E respectively.

Binary Tree

- **Representation of Binary Tree using Array:**
- Binary tree using array represents a node which is numbered sequentially level by level from left to right. Even empty nodes are numbered.



Binary Tree

- **Representation of Binary Tree using Array:**
  - Array index is a value in tree nodes and array value gives to the parent node of that particular index or node.
  - Value of the root node index is always -1 as there is no parent for root.
  - When the data item of the tree is sorted in an array, the number appearing against the node will work as indexes of the node in an array.

0	1	2	3	4	5	6	7
7	A	B	C	D	E	F	G

Binary Tree

- **Representation of Binary Tree using Array:**
  - Location number of an array is used to store the size of the tree.
  - The first index of an array that is '0', stores the total number of nodes.
  - All nodes are numbered from left to right level by level from top to bottom.
  - In a tree, each node having an index i is put into the array as its i th element.

0	1	2	3	4	5	6	7
7	A	B	C	D	E	F	G

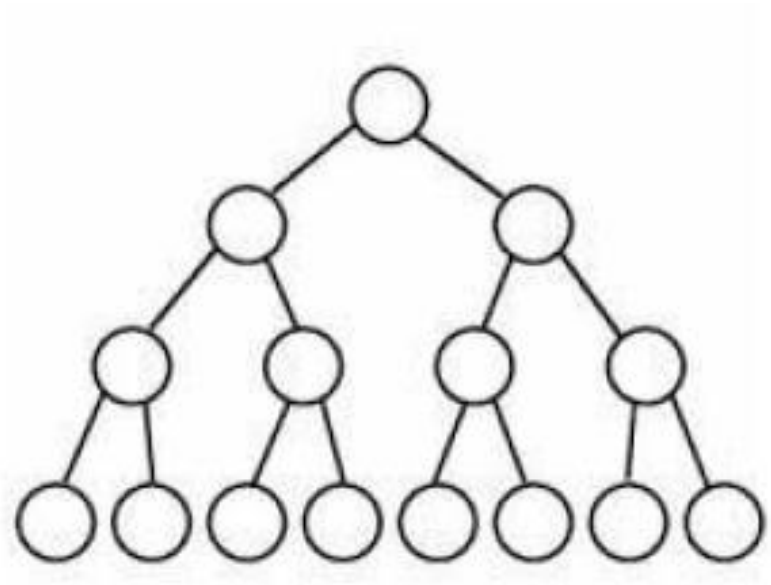
Binary Tree

- **Representation of Binary Tree using Array:**
  - The above figure shows how a binary tree is represented as an array.
  - Value '7' is the total number of nodes. If any node does not have any of its child, null value is stored at the corresponding index of the array..

0	1	2	3	4	5	6	7
7	A	B	C	D	E	F	G

Full Binary Tree or Complete Trees:

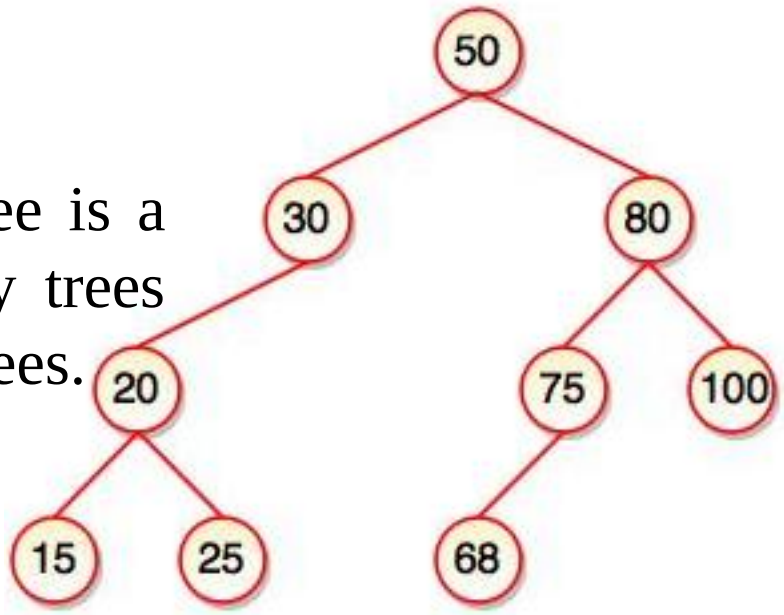
- A binary tree of height is ‘h’ and contains exactly “**2h-1” elements is called full binary tree.**



Binary Search Tree

- "Binary Search Tree is a binary tree where each node contains only smaller values in its left subtree and only larger values in its right subtree."

Note: Every binary search tree is a binary tree, but all the binary trees need not to be binary search trees.



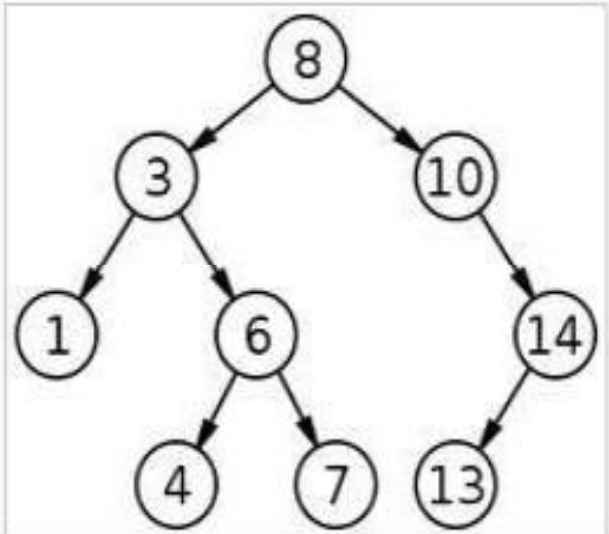
Binary Search Tree

Binary search tree	
Type	Tree
Invented	1960
Invented by	P.F. Windley, A.D. Booth, A.J.T. Colin, and T.N. Hibbard

Time complexity in big O notation

	Average	Worst case
Space	O(n)	O(n)
Search	O(log n)	O(n)
Insert	O(log n)	O(n)
Delete	O(log n)	O(n)

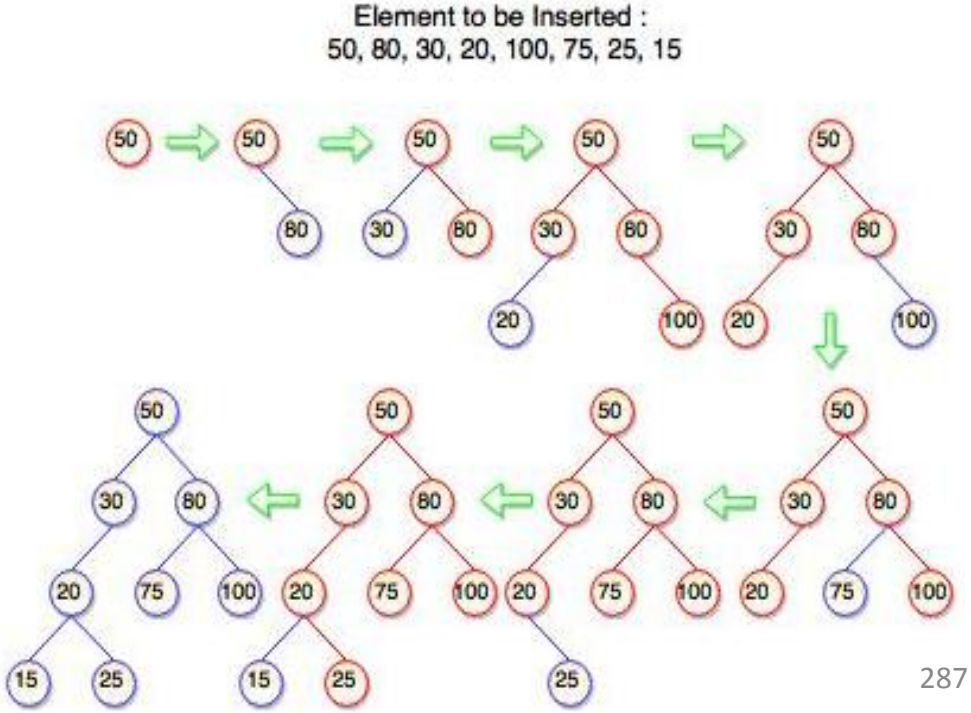
Example



A binary search tree of size 9 and depth 3, with root 8 and leaves 1, 4, 7 and 13

Binary Search Tree

- **Binary Search Tree Operations:**
  - Insert Operation
  - Insert operation is performed with O(log n) time complexity in a binary search tree.
  - Insert operation starts from the root node. It is used whenever an element is to be inserted.



Binary Search Tree

- **Binary Search Tree Operations:**
  - Search Operation
  - Search operation is performed with O(log n) time complexity in a binary search tree.
  - This operation starts from the root node. It is used whenever an element is to be searched.

Binary Search Tree

- **Binary Tree Traversal**
  - **There are three techniques of traversal:**
    1. Preorder Traversal
    2. Postorder Traversal
    3. Inorder Traversal

Binary Search Tree

- Preorder Traversal:
- **Algorithm for preorder traversal**
  Step 1 : Start from the Root.
  Step 2 : Then, go to the Left Subtree.
  Step 3 : Then, go to the Right Subtree.

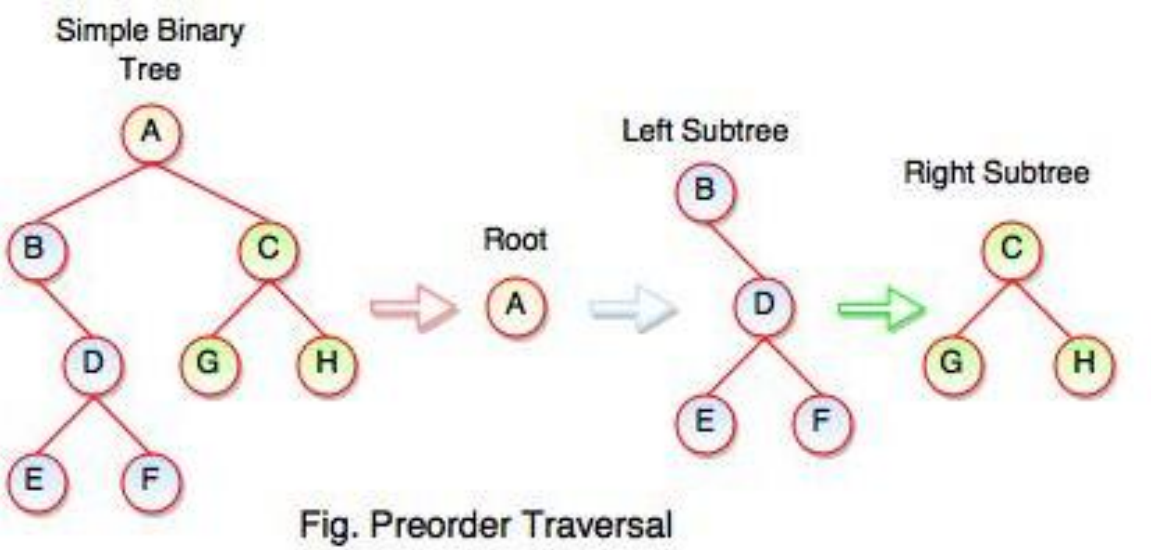


Fig. Preorder Traversal

A + B + D + E + F + C + G + H

Binary Search Tree

- Postorder Traversal
  - **Algorithm for postorder traversal**
    Step 1 : Start from the Left Subtree (Last Leaf).
    Step 2 : Then, go to the Right Subtree.
    Step 3 : Then, go to the Root.

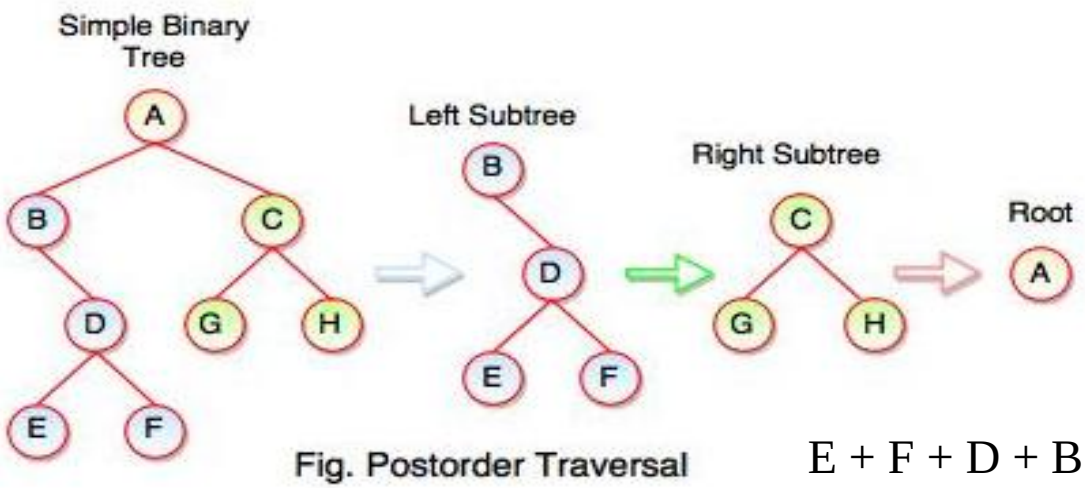


Fig. Postorder Traversal

E + F + D + B + G + H + C + A

Binary Search Tree

- Inorder Traversal:
  - **Algorithm for inorder traversal**
    Step 1 : Start from the Left Subtree.
    Step 2 : Then, visit the Root.
    Step 3 : Then, go to the Right Subtree.

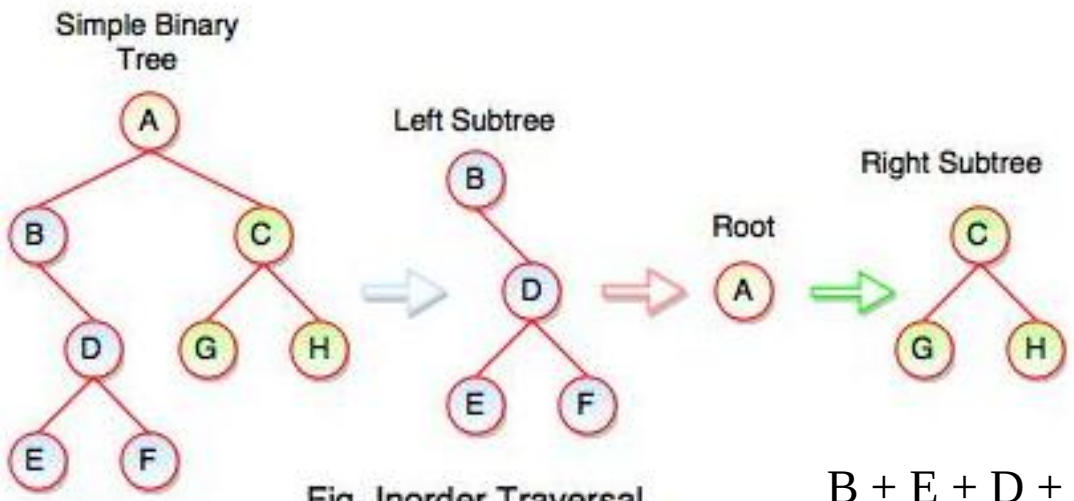


Fig. Inorder Traversal

B + E + D + F + A + G + C + H

Balanced Tree

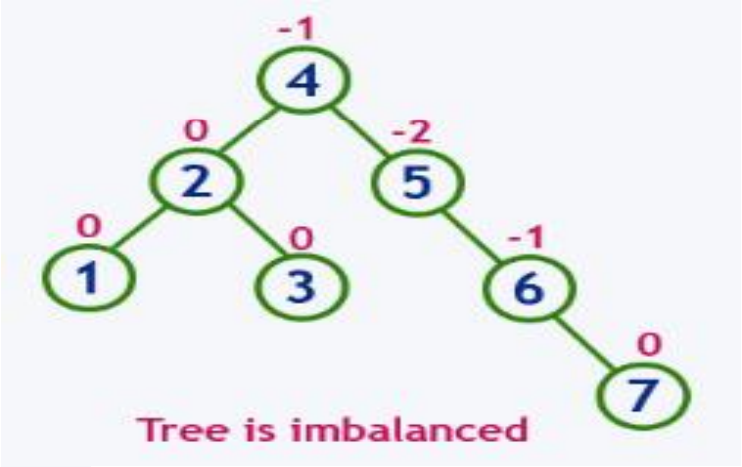
- Balancing or self-balancing (Height balanced) tree is a binary search tree.
- Balanced tree is any node based binary search tree that automatically keeps its height
- (Maximum number of levels below the root) small in the face of arbitrary item insertion and deletion.

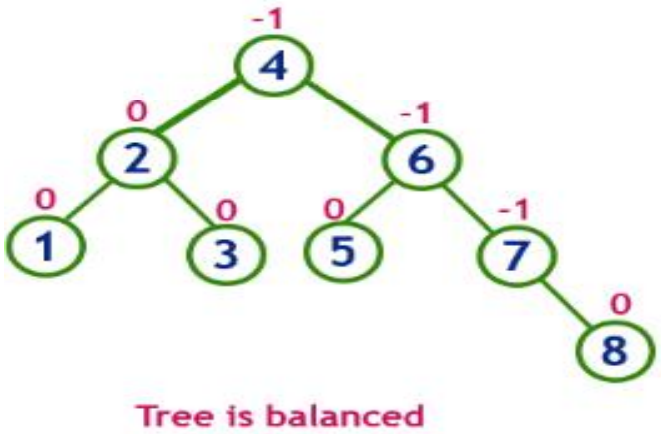
AVL trees

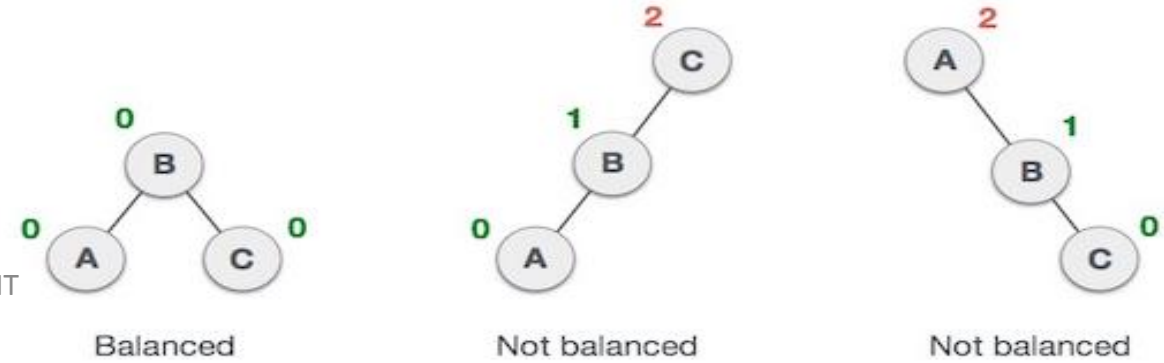
- AVL tree is a binary search tree in which the difference of heights of left and right subtrees of any node is less than or equal to one.
- The technique of balancing the height of binary trees was developed by Adelson, Velskii, and Landi and hence given the short form as AVL tree or Balanced Binary Tree.

AVL trees

- Every AVL Tree is a binary search tree but every Binary Search Tree need not be AVL tree.





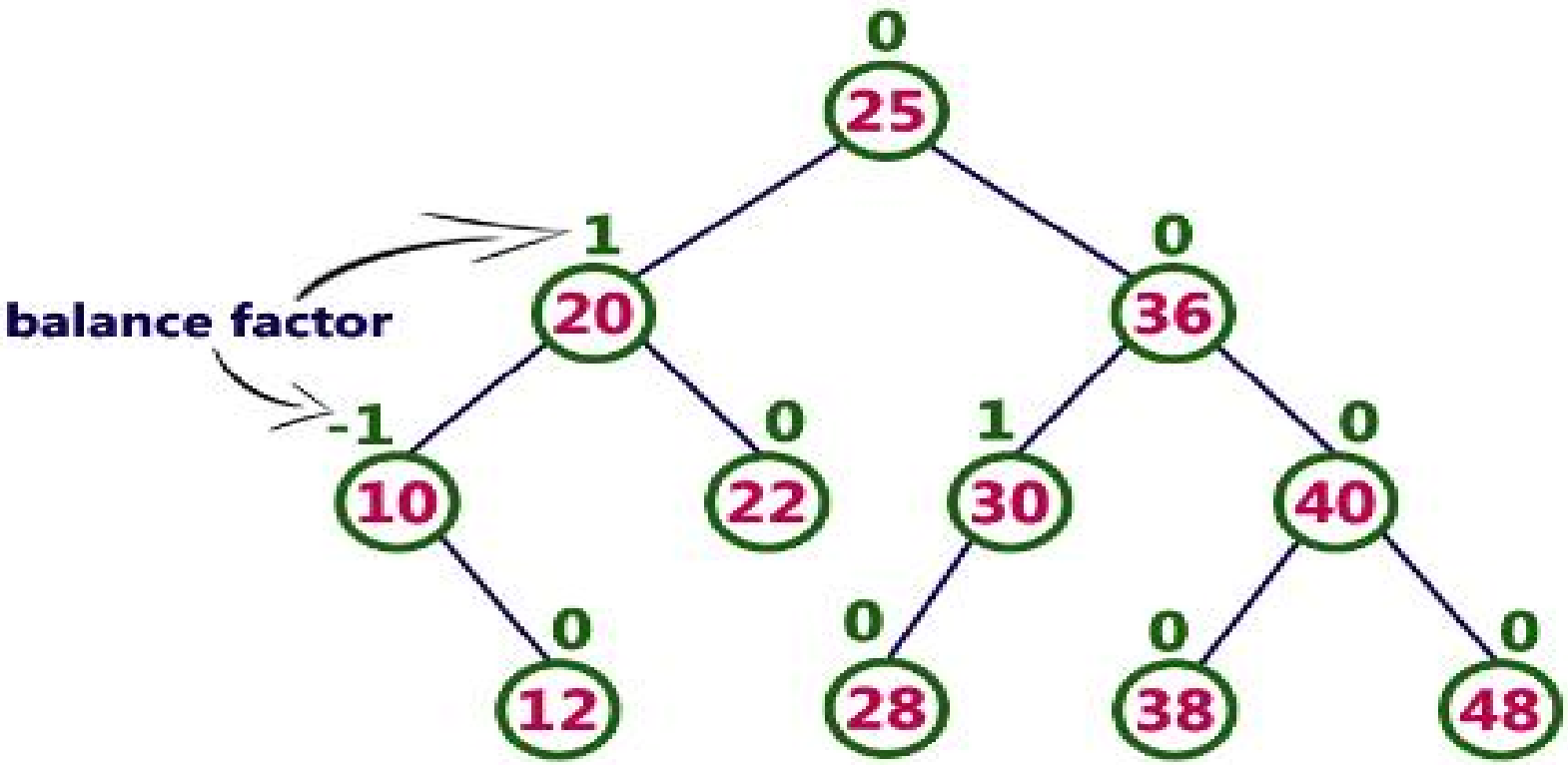


AVL trees

- **Definition:** An AVL tree is a binary search tree in which the balance factor of every node, which is defined as the difference b/w the heights of the node's left & right sub trees is either 0 or +1 or -1 .

Balance factor = ht of left sub tree – ht of right sub tree.

AVL trees



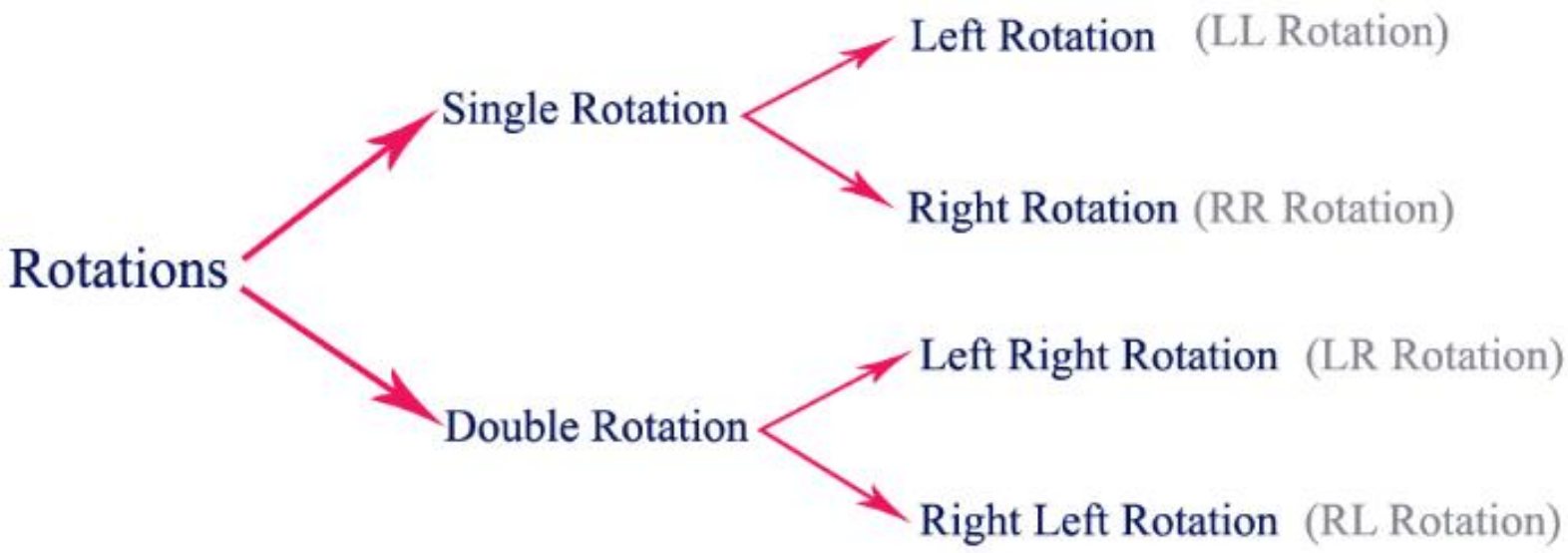
The above tree is a binary search tree and every node is satisfying balance factor condition. So this tree is said to be an AVL tree.

AVL Tree Rotations

- In AVL tree, after performing operations like insertion and deletion we need to check the **balance factor** of every node in the tree.
- If every node satisfies the balance factor condition then we conclude the operation otherwise we must make it balanced.
- Whenever the tree becomes imbalanced due to any operation we use **rotation** operations to make the tree balanced.

AVL Tree Rotations

- Rotation operations are used to make the tree balanced.
- Rotation is the process of moving nodes either to left or to right to make the tree balanced.



AVL Tree Insertion:

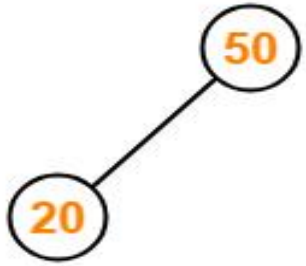
- Insertion in AVL tree is performed in the same way as it is performed in a binary search tree.
- The new node is added into AVL tree as the leaf node. However, it may lead to violation in the AVL tree property and therefore the tree may need balancing.
- The tree can be balanced by applying rotations. Rotation is required only if, the balance factor of any node is disturbed upon inserting the new node, otherwise the rotation is not required.

AVL Tree

- Construct AVL Tree for the following sequence of numbers- 50 , 20 , 60 , 10 , 8 , 15 , 32 , 46 , 11 , 48
- **<u>Step-01: Insert 50</u>**

AVL Tree

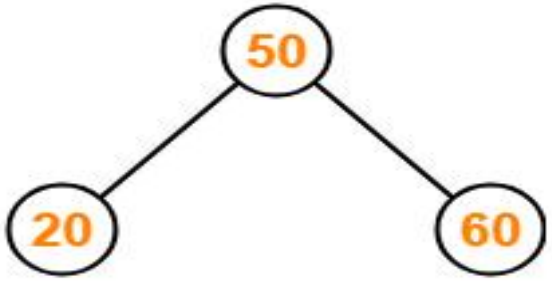
- Construct AVL Tree for the following sequence of numbers- 50 , 20 , 60 , 10 , 8 , 15 , 32 , 46 , 11 , 48
- **<u>Step-02: Insert 20</u>**
- As 20 < 50, so insert 20 in 50's left sub tree.



Tree is Balanced

AVL Tree

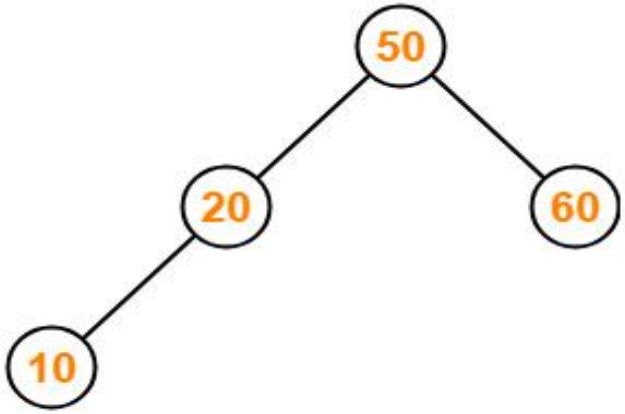
- Construct AVL Tree for the following sequence of numbers- 50 , 20 , 60 , 10 , 8 , 15 , 32 , 46 , 11 , 48
- **<u>Step-03: Insert 60</u>**
- As 60 > 50, so insert 60 in 50's right sub tree.



Tree is Balanced

AVL Tree

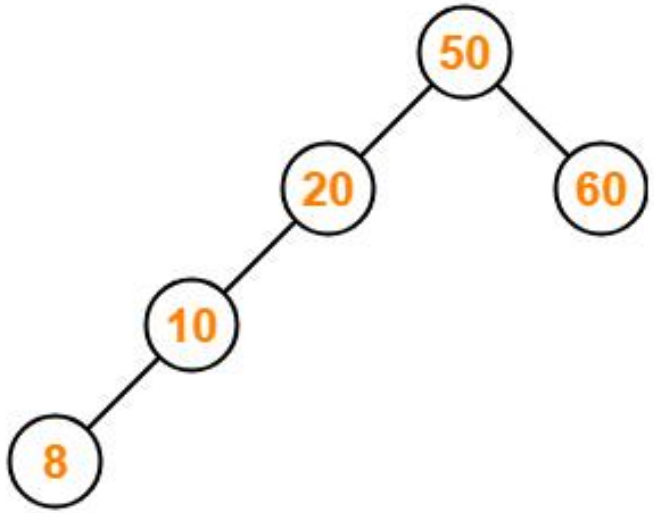
- Construct AVL Tree for the following sequence of numbers- 50 , 20 , 60 , 10 , 8 , 15 , 32 , 46 , 11 , 48
- **<u>Step-04: Insert 10</u>**
  - As 10 < 50, so insert 10 in 50's left sub tree.
  - As 10 < 20, so insert 10 in 20's left sub tree.



Tree is Balanced

AVL Tree

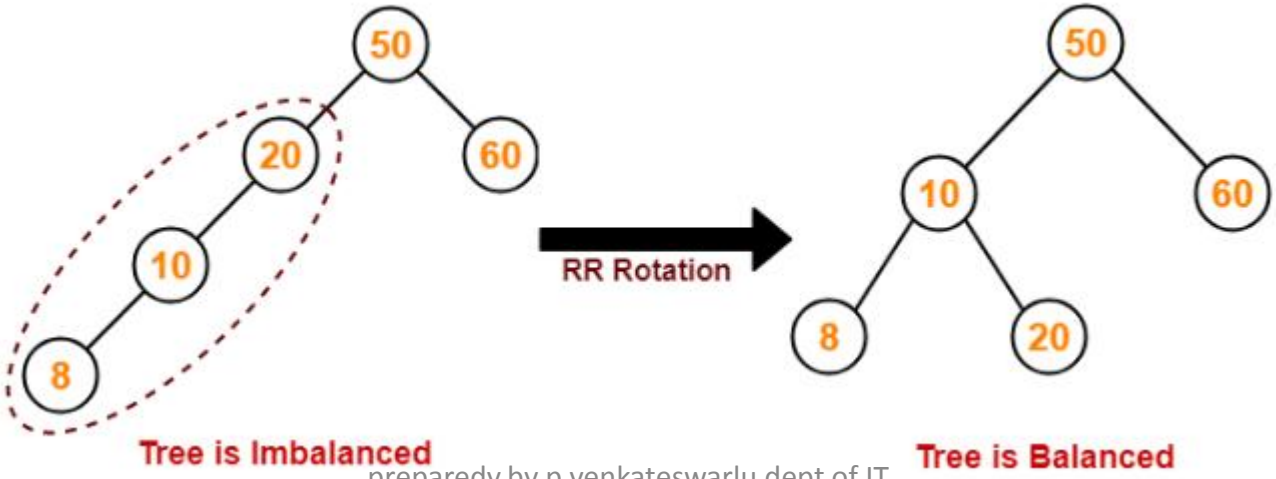
- Construct AVL Tree for the following sequence of numbers-
  50 , 20 , 60 , 10 , 8 , 15 , 32 , 46 , 11 , 48
- **<u>Step-05: Insert 8</u>**
  - As 8 < 50, so insert 8 in 50's left sub tree.
  - As 8 < 20, so insert 8 in 20's left sub tree.
  - As 8 < 10, so insert 8 in 10's left sub tree.



Tree is Imbalanced

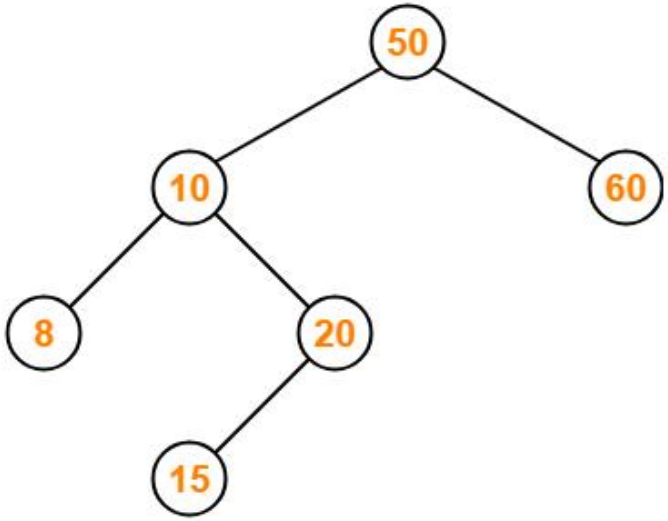
AVL Tree

- To balance the tree,
- Find the first imbalanced node on the path from the newly inserted node (node 8) to the root node.
- The first imbalanced node is node 20.
- Now, count three nodes from node 20 in the direction of leaf node.
- Then, use AVL tree rotation to balance the tree.



AVL Tree

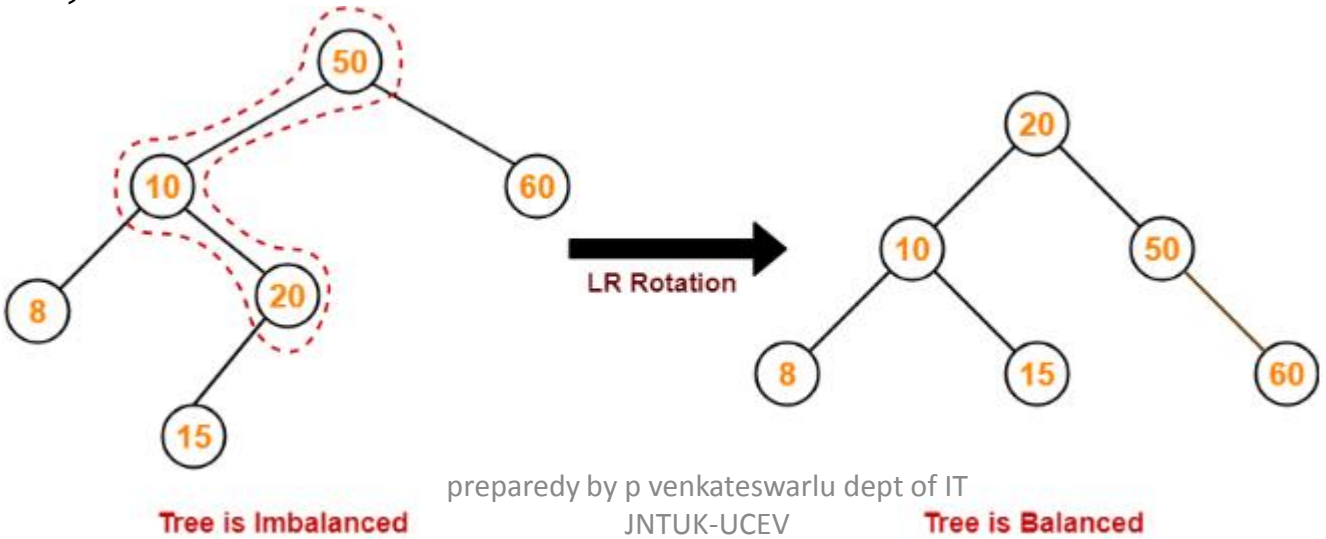
- Construct AVL Tree for the following sequence of numbers-
  50 , 20 , 60 , 10 , 8 , 15 , 32 , 46 , 11 , 48
- **Step-06: Insert 15**
  - As 15 < 50, so insert 15 in 50's left sub tree.
  - As 15 > 10, so insert 15 in 10's right sub tree.
  - As 15 < 20, so insert 15 in 20's left sub tree.



Tree is Imbalanced

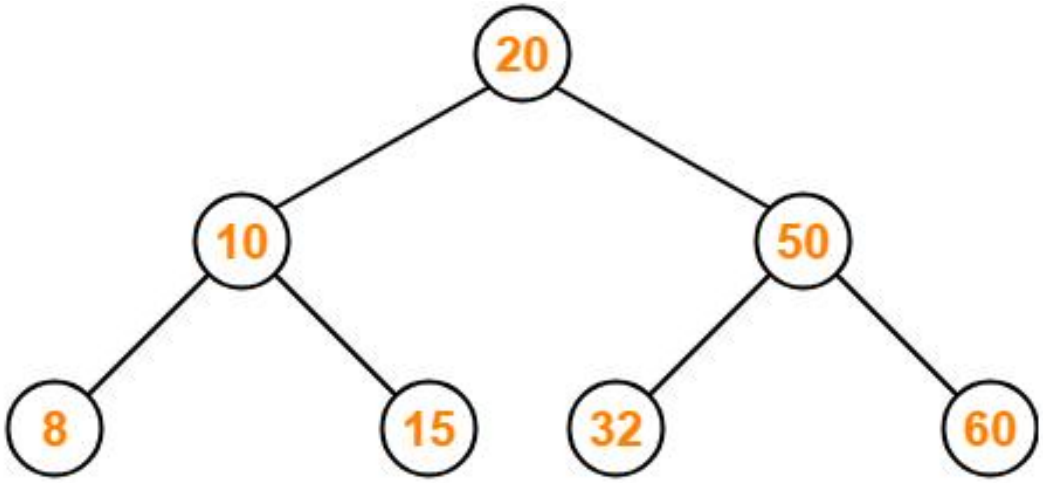
AVL Tree

- To balance the tree,
- Find the first imbalanced node on the path from the newly inserted node (node 15) to the root node.
- The first imbalanced node is node 50.
- Now, count three nodes from node 50 in the direction of leaf node.
- Then, use AVL tree rotation to balance the tree.



AVL Tree

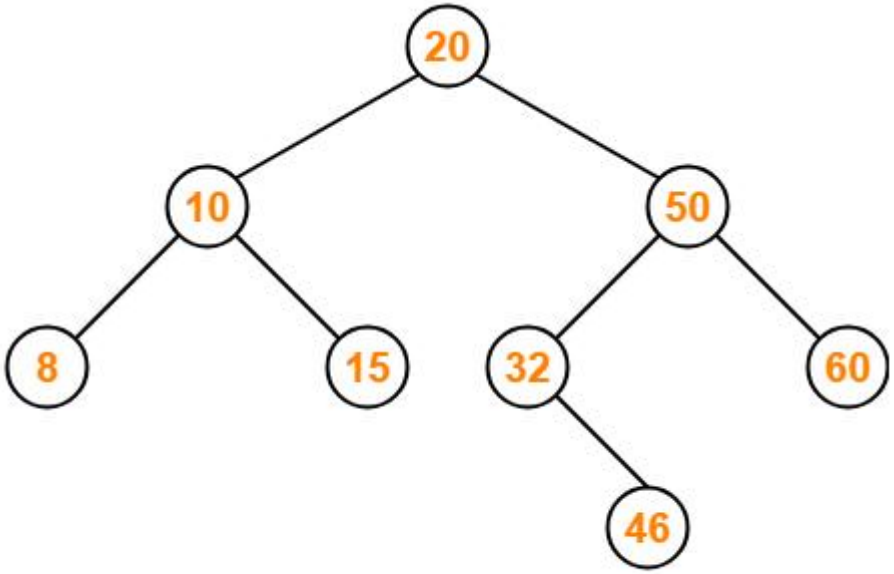
- Construct AVL Tree for the following sequence of numbers-
  50 , 20 , 60 , 10 , 8 , 15 , 32 , 46 , 11 , 48
- **<u>Step-07: Insert 32</u>**
  - As 32 > 20, so insert 32 in 20's right sub tree.
  - As 32 < 50, so insert 32 in 50's left sub tree.



Tree is Balanced

AVL Tree

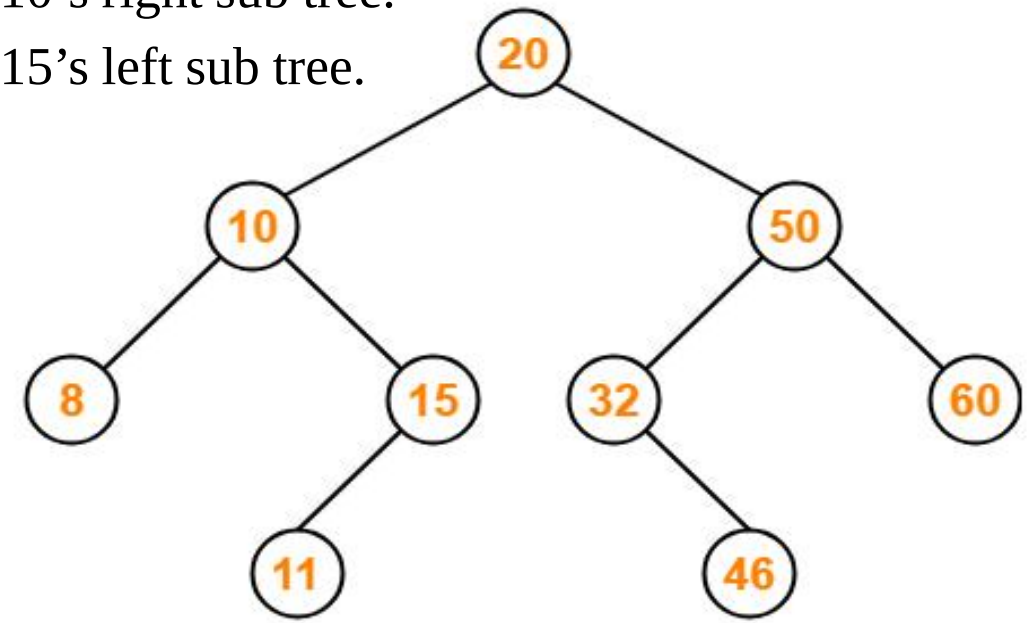
- Construct AVL Tree for the following sequence of numbers-
  50 , 20 , 60 , 10 , 8 , 15 , 32 , 46 , 11 , 48
- **<u>Step-08: Insert 46</u>**
  - As 46 > 20, so insert 46 in 20's right sub tree.
  - As 46 < 50, so insert 46 in 50's left sub tree.
  - As 46 > 32, so insert 46 in 32's right sub tree.



Tree is Balanced

AVL Tree

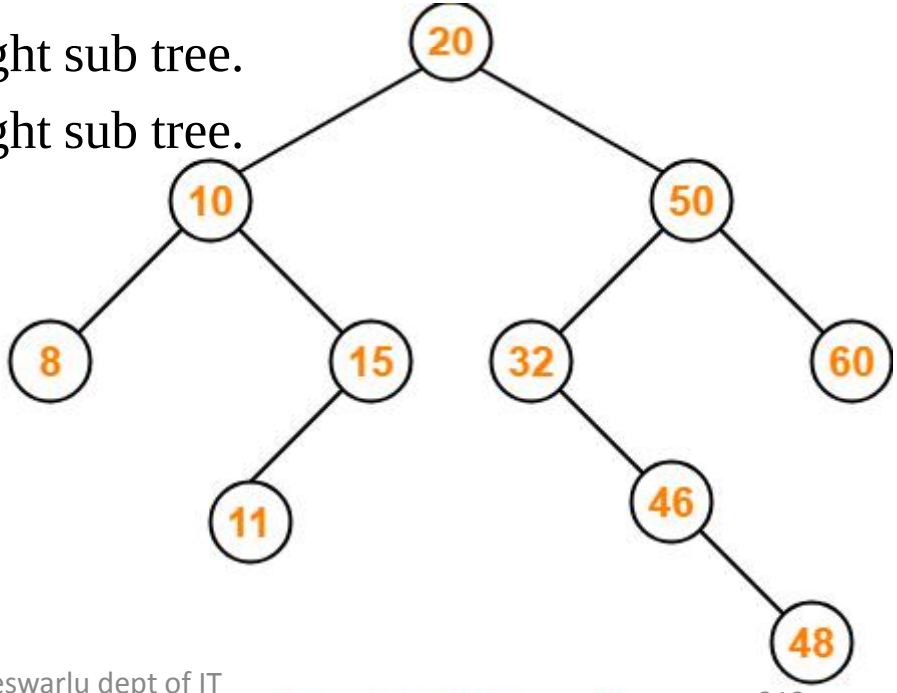
- Construct AVL Tree for the following sequence of numbers-
  50 , 20 , 60 , 10 , 8 , 15 , 32 , 46 , 11 , 48
- **<u>Step-09: Insert 11</u>**
  - As 11 < 20, so insert 11 in 20's left sub tree.
  - As 11 > 10, so insert 11 in 10's right sub tree.
  - As 11 < 15, so insert 11 in 15's left sub tree.



Tree is Balanced

AVL Tree

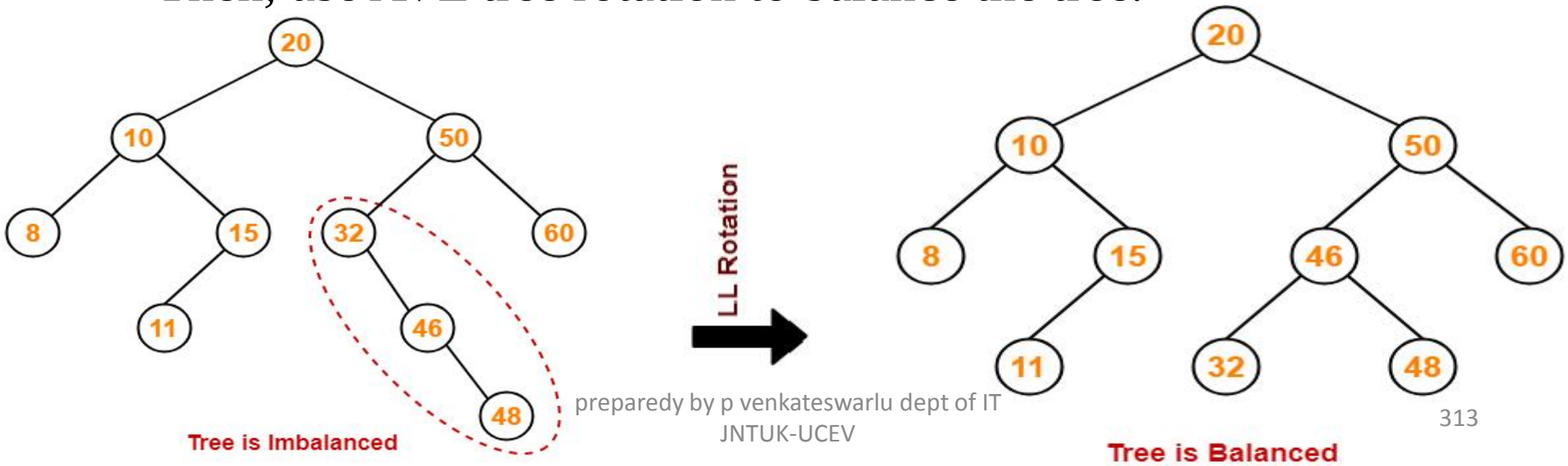
- Construct AVL Tree for the following sequence of numbers-
  50 , 20 , 60 , 10 , 8 , 15 , 32 , 46 , 11 , 48
- **<u>Step-10: Insert 48</u>**
  - As 48 > 20, so insert 48 in 20's right sub tree.
  - As 48 < 50, so insert 48 in 50's left sub tree.
  - As 48 > 32, so insert 48 in 32's right sub tree.
  - As 48 > 46, so insert 48 in 46's right sub tree.



Tree is imbalanced

AVL Tree

- To balance the tree,
- Find the first imbalanced node on the path from the newly inserted node (node 48) to the root node.
- The first imbalanced node is node 32.
- Now, count three nodes from node 32 in the direction of leaf node.
- Then, use AVL tree rotation to balance the tree.



- **AVL Tree Example:**
- **Insert 14, 17, 11, 7, 53, 4, 13 into an empty AVL tree**

splay tree

- A **splay tree** is a self-balancing binary search tree with the additional property that recently accessed elements are quick to access again.
- It performs basic operations such as insertion, look-up and removal in O(log(n)) amortized time.
- splay trees perform better than other search trees, even when the specific pattern of the sequence is unknown.
- The splay tree was invented by Daniel Dominic Sleator and Robert Endre Tarjan in 1985.

splay tree

- All normal operations on a binary search tree are combined with one basic operation, called *splaying*.
- Splaying the tree for a certain element rearranges the tree so that the element is placed at the root of the tree.
- In splay trees, we first search the query item, say a as in the usual binary search trees to compare the query item with the value in the root, if less then recursively search in the left subtree else if higher then, recursively search in the right subtree, and if it is equal then we are done.

Tournament Tree

- Tournament tree is a form of min (max) heap which is a complete binary tree.
- Every external node represents a player and internal node represents winner.
- In a tournament tree every internal node contains winner and every leaf node contains one player.

Tournament Tree

- Winner Trees :
  - Complete binary tree with n external nodes and n - 1 internal nodes.
  - External nodes represent tournament players.
  - Each internal node represents a match played between its two children; the winner of the match is stored at the internal node.
  - Root has overall winner

Properties of Tournament Tree

- It is rooted tree i.e. the links in the tree and directed from parents to children and there is a unique element with no parents.
- The key value of the parent node has less than or equal to that node to general any comparison operators can be used as long as the relative values of parent-child are invariant throughout the tree. The tree is a parent ordering of the keys.
- Trees with a number of nodes not a power of 2 contain holes which is general may be anywhere in the tree.
- Tournament tree is a proper generalization of heaps which restrict a node to at most two children.
- The tournament tree is also called selection tree.
- The root of the tournament tree represents overall winner of the tournament.

Types of tournament Tree

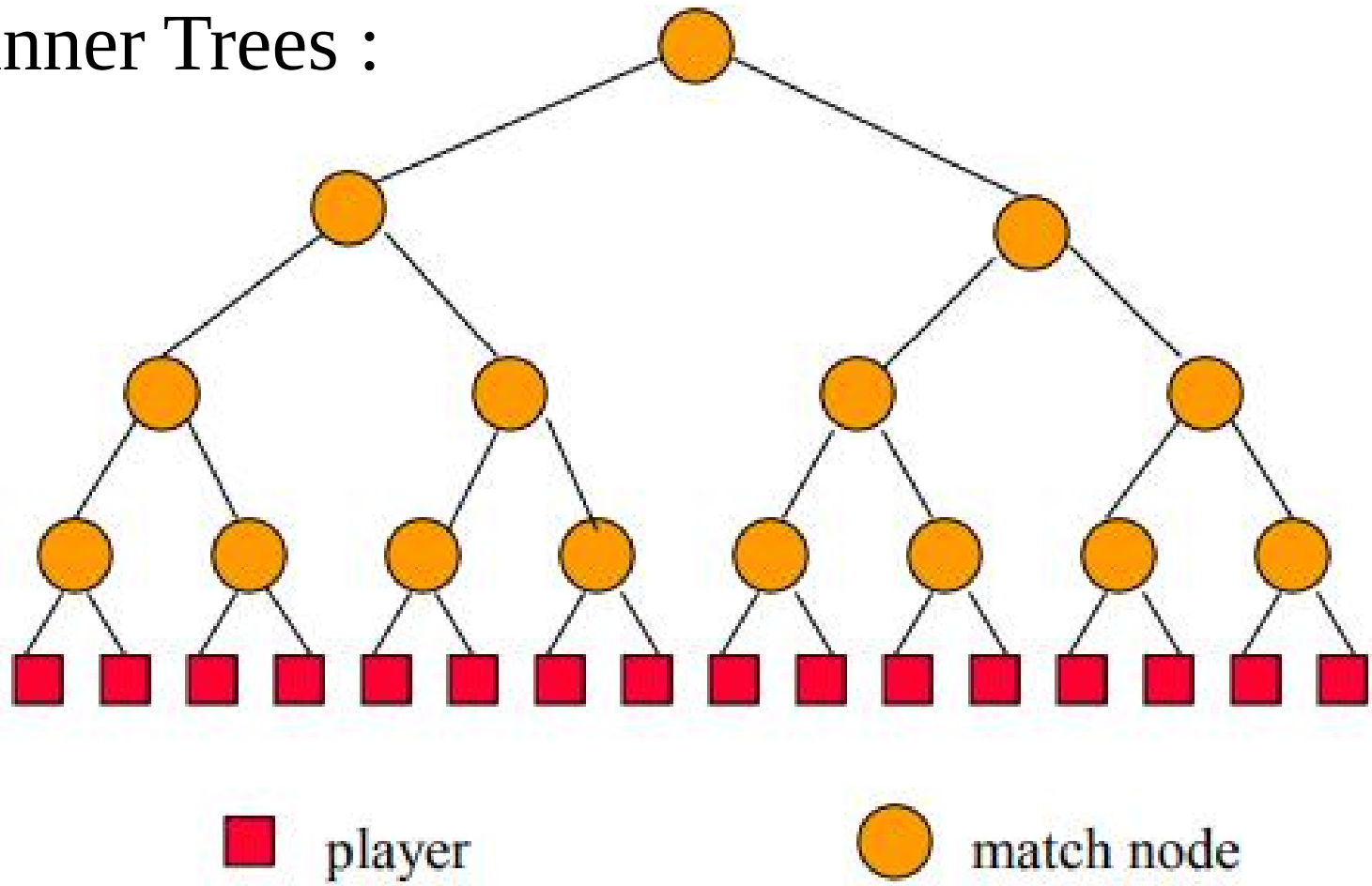
- There are mainly two type of tournament tree,
  - **Winner tree**
  - **Loser tree**

Types of tournament Tree

- Winner tree
  - The complete binary tree in which each node represents the smaller or greater of its two children is called a winner tree.
  - The smallest or greater node in the tree is represented by the root of the tree.
  - The winner of the tournament tree is the smallest or greatest n key in all the sequences.
  - It is easy to see that the winner tree can be computed in **O(logn)** time.

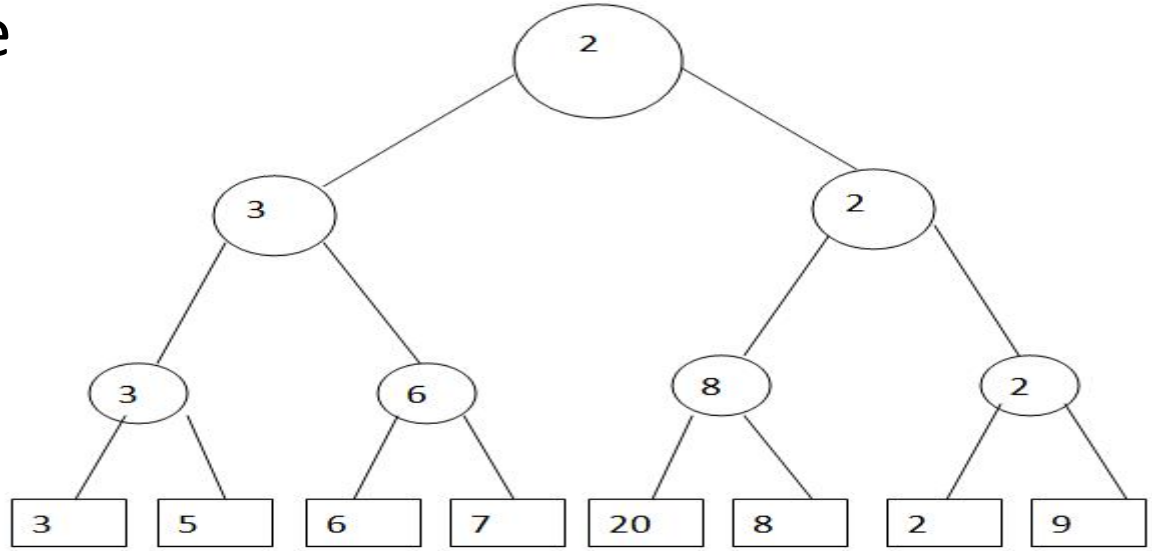
Tournament Tree

- Winner Trees :



Types of tournament Tree

- Winner tree
  - **Example:** Consider some keys 3, 5, 6, 7, 20, 8, 2, 9
  - We try to make minimum or maximum winner tree



Winner tree (minimum)

Types of tournament Tree

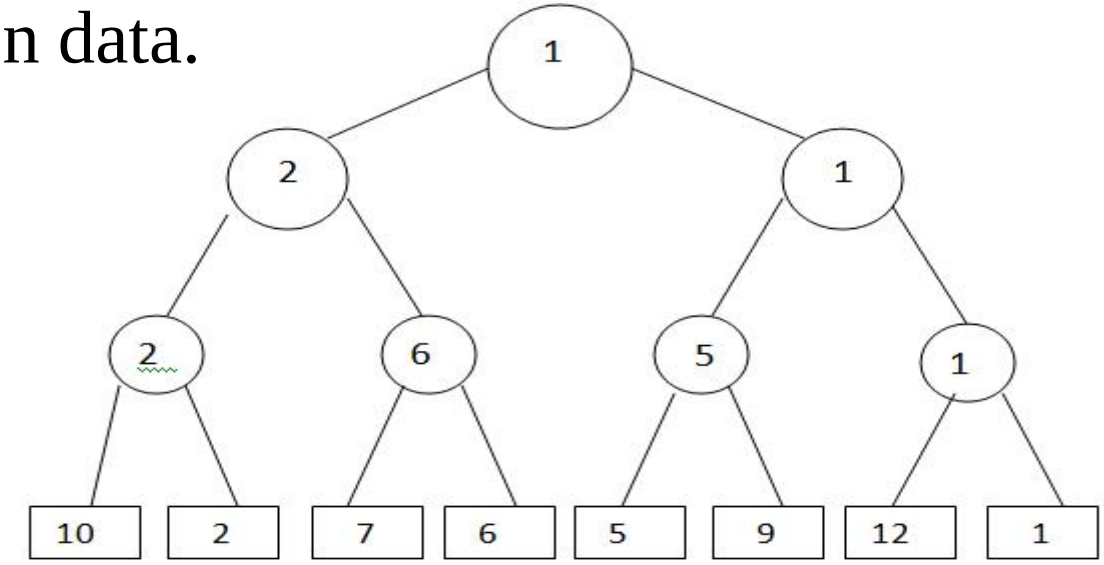
- **Loser Tree**
- The complete binary tree for n players in which there are n external nodes and n-1 internal nodes then the tree is called loser tree.
- The loser of the match is stored in internal nodes of the tree.
- But in this overall winner of the tournament is stored at tree [0].

Types of tournament Tree

- **Loser Tree**
- The loser is an alternative representation that stores the loser of a match at the corresponding node.
- An advantage of the loser is that to restructure the tree after a winner tree been output, it is sufficient to examine node on the path from the leaf to the root rather than the sibling of nodes on this path.

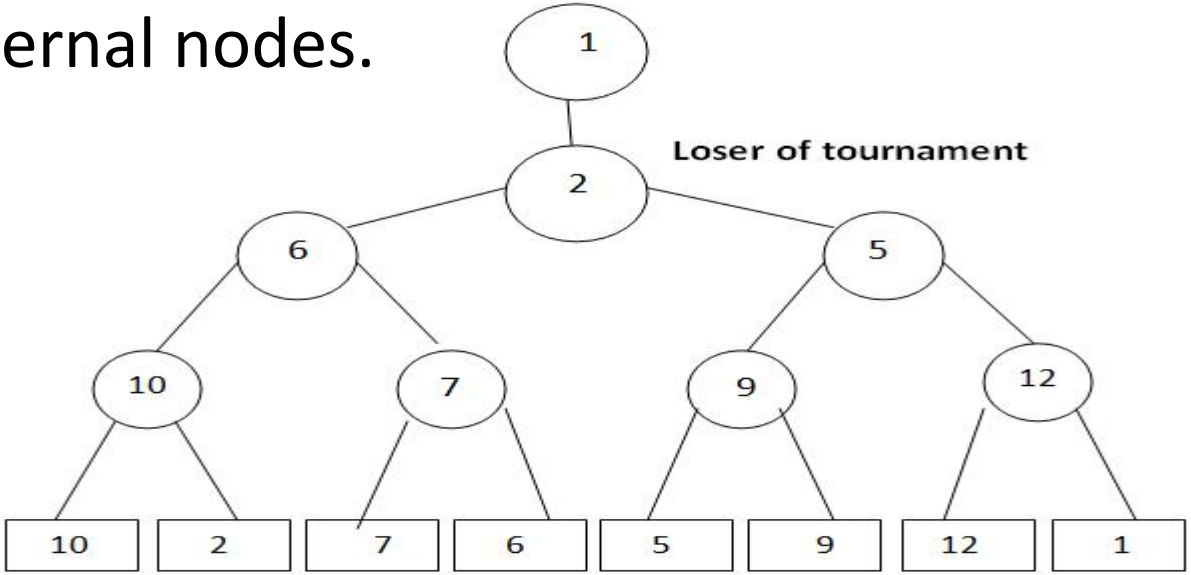
Types of tournament Tree

- **Loser Tree**
  - **Example:** Consider some keys 10, 2, 7, 6, 5, 9, 12, 1
  - **Step 1)** We will first draw min winner tree for given data.



Types of tournament Tree

- **Loser Tree**
  - **Example:** Consider some keys 10, 2, 7, 6, 5, 9, 12, 1
  - **Step 2)** Now we will store losers of the match in each internal nodes.



Application of Tournament Tree

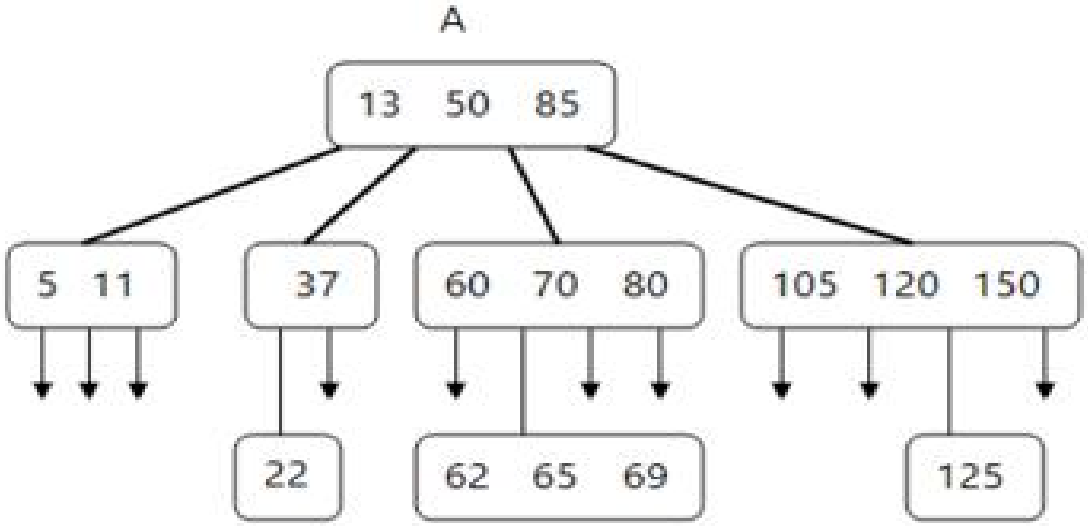
- It is used for finding the smallest and largest element in the array.
- It is used for sorting purpose.
- Tournament tree may also be used in M-way merges.
- Tournament replacement algorithm selection sort is used to gather the initial run for external sorting algorithms.

Complexity of Loser Tree Initialize

- One match at each match node One match at each match node.
- One store of a left child winner.
- Total time is O(n).
- M il ore precisely $\Theta(n)$.

Multiway Trees

- **Multiway Search Trees** allow nodes to store multiple child nodes (greater then 2).
- These differ from binary search trees which can only have a maximum of 2 nodes.



Multiway Trees

- **Characteristics**
  - Nodes may carry multiple keys.
  - Each node may have **N** number of children
  - Each node maintains **N-1** search keys
  - The tree maintains all leaves at the same level

Multiway Trees

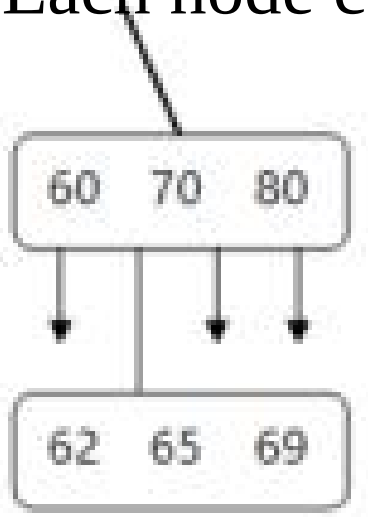
- **Operations**
  - Search: A path is traced starting at the root. The nodes are traversed and a pointer is positioned on the key value being searched. If the key is not found, it returns a **search miss**. If the key is found, it returns a **search hit**.
  - Insert: The pointer searches to make sure a key does not exist. It then creates a link adding the key to the appropriate node.

Multiway Trees

- **2-3-4 Trees**
  - **2-3-4 trees** are a type of Multiway search tree. Each node can hold a maximum of 3 search keys and can hold 2, 3 or 4 child nodes.
  - All leaves are maintained at the same level. 2-3-4 trees are **self-balancing** structures, meaning they rearrange themselves if the structure goes off balance after an insert or delete operation.

Multiway Trees

- **2-3-4 Trees Characteristics**
  - 2-3-4 trees can carry multiple child nodes.
  - Each node maintains **N** child nodes where **N** is equal to 2, 3 or 4 child nodes.
  - Each node can carry (N-1) search keys.



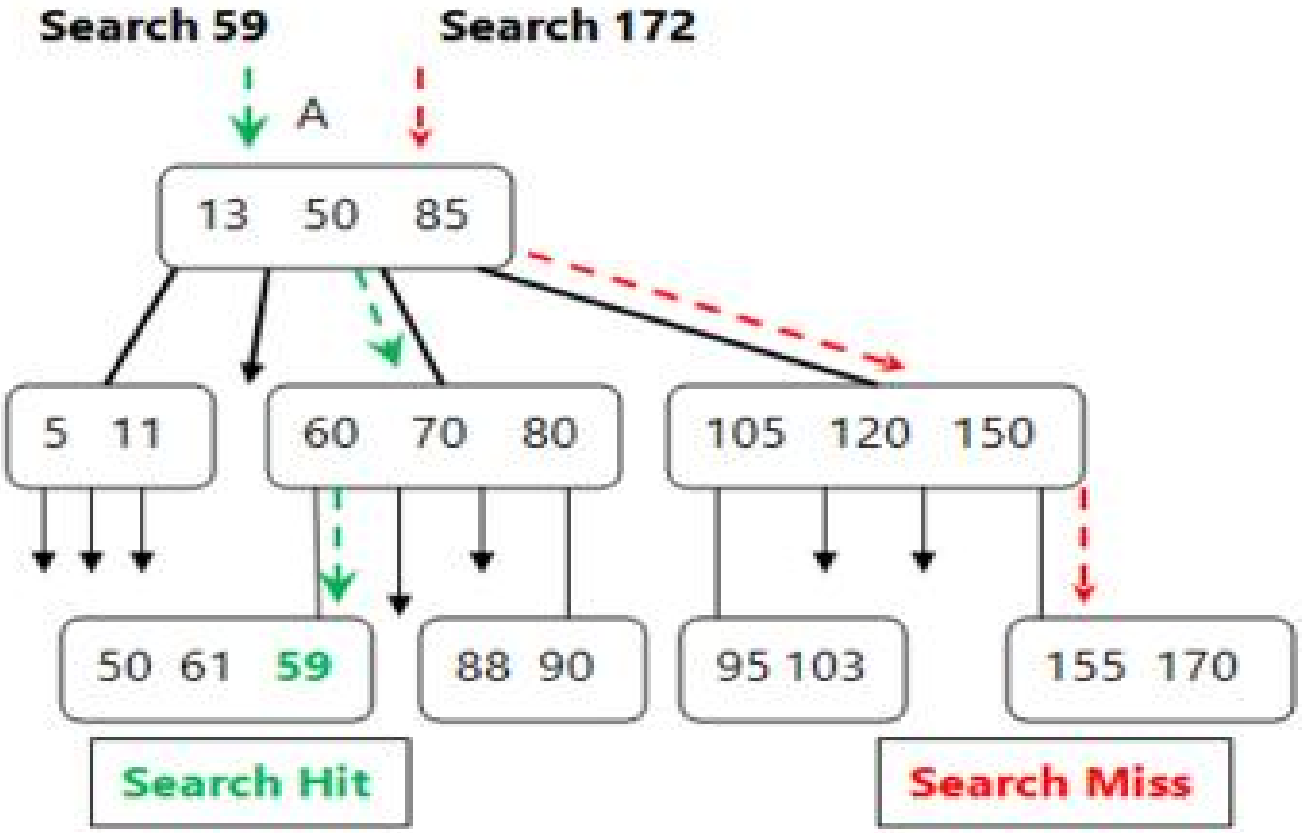
- This node can have maximum N=4 children
- It maintains maximum (N-1)=3 search keys (60, 70, 80)
- The child node is on the second branch with values greater than 60 but less than 70

Multiway Trees

- **2-3-4 Trees Operations**
  - Search: With 2-3-4 trees, searches commence at the root and traverse each node until right node is found.
  - A sequential search is done within the node to locate the correct key value. If the value is found, it returns a search hit. If the value is not found, it returns a search miss.
  - For example, in Figure 3, we search for key 59 and key 172.

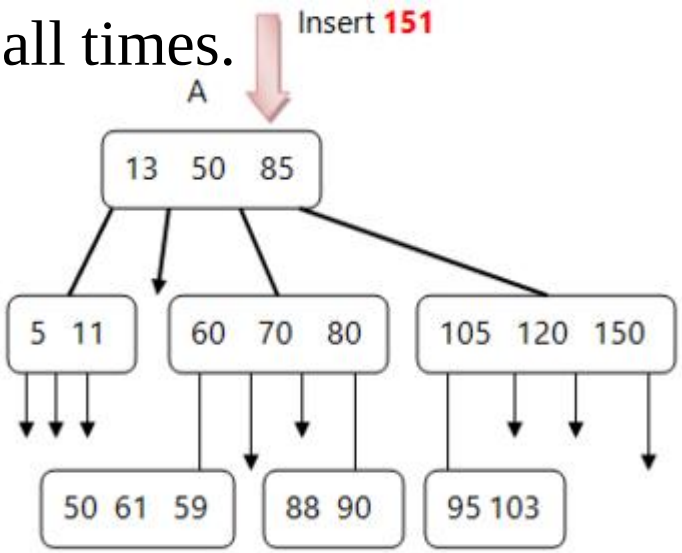
Multiway Trees

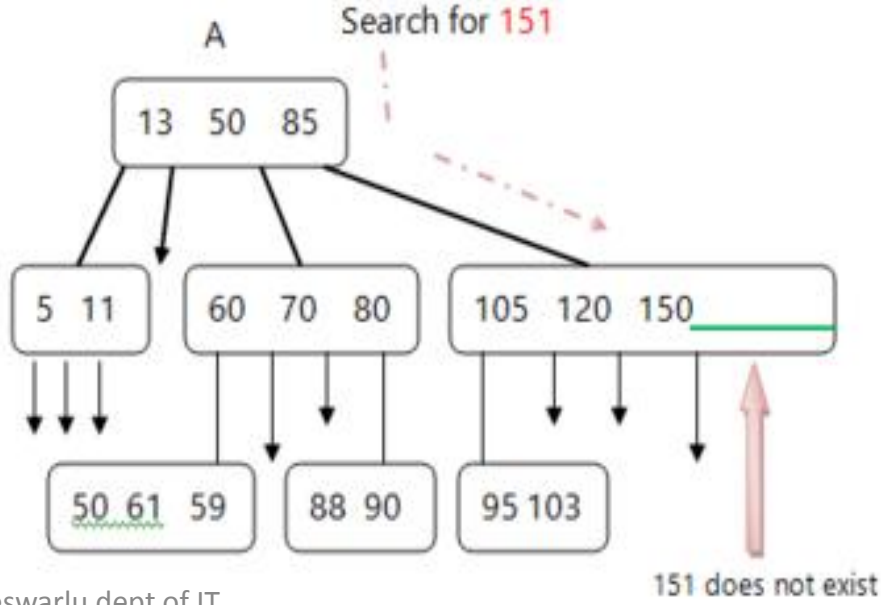
- **2-3-4 Trees Operations**



Multiway Trees

- **2-3-4 Trees Operations**
  - Insert: The tree is first searched to ensure that the key value does not exist.
  - If it doesn't, a link is created in the appropriate node and the search key is inserted.
  - Note that 2-3-4 tree characteristics must be maintained at all times.



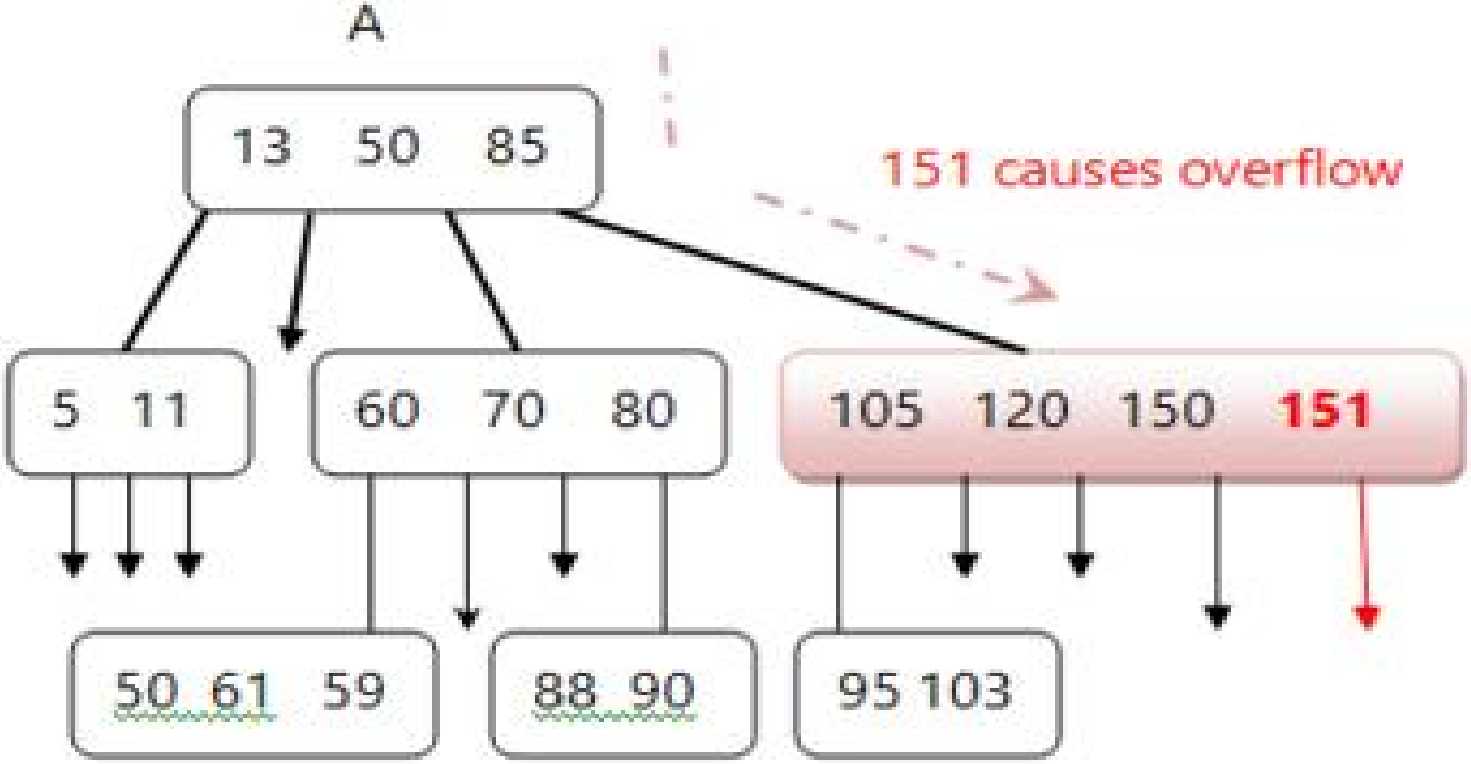


Multiway Trees

- **2-3-4 Trees Operations**
  - an insert is achieved because there is a search miss on that key value.
  - Key 151 does not exist and can therefore be added.
  - a link is created and 151 is inserted in the appropriate node.
  - This, however, results in a violation of the 2-3-4 tree rule that a node can carry no more than N or 4 child nodes and (N-1) or 3 key values. This violation is referred to as an **overflow**.

Multiway Trees

- **2-3-4 Trees Operations**
  - This violation is referred to as an **overflow**.

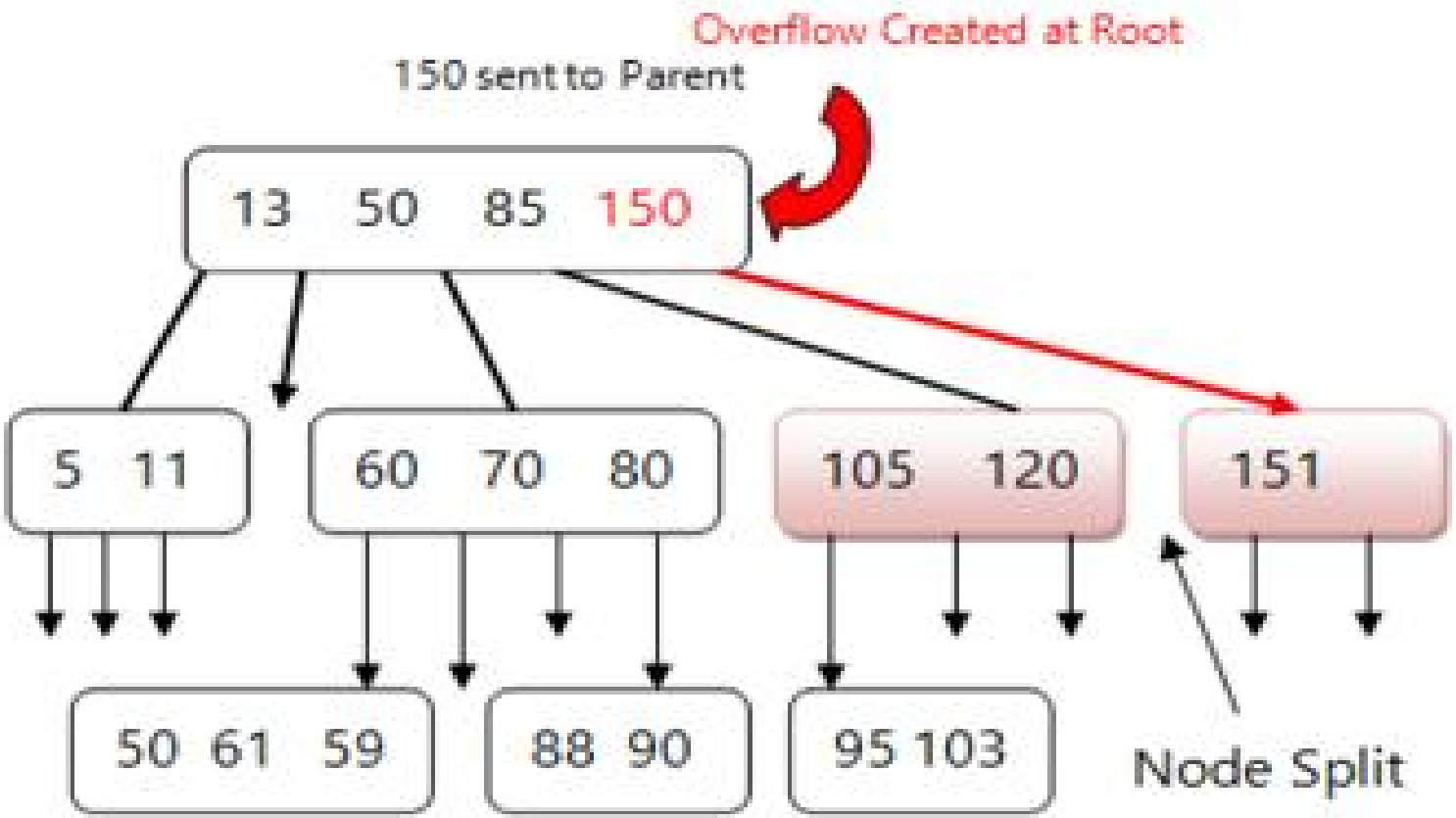


Multiway Trees

- **2-3-4 Trees Operations**
  - To resolve the problem and re-balance the tree, the node with the overflow is split and key value 150 is sent to the parent node, which in this case, is the root.
  - The original node is no longer in overflow as it has been split, but the root node is now in overflow because the key 150 has been inserted

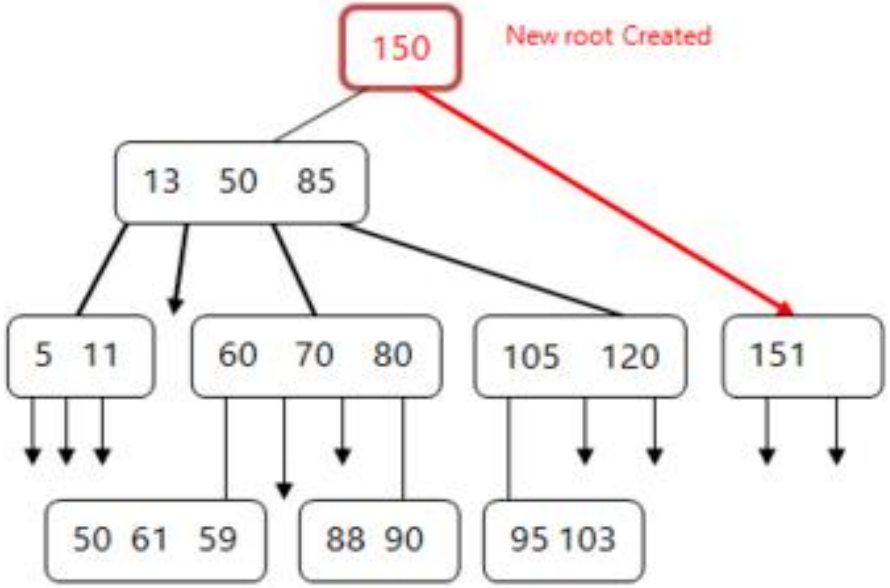
Multiway Trees

- **2-3-4 Trees Operations**



Multiway Trees

- **2-3-4 Trees Operations**
  - To fix this, the root node needs to have a single key with all other nodes emanating from it.
  - key 150 is used to create a new root node and the tree is corrected.



B-Trees

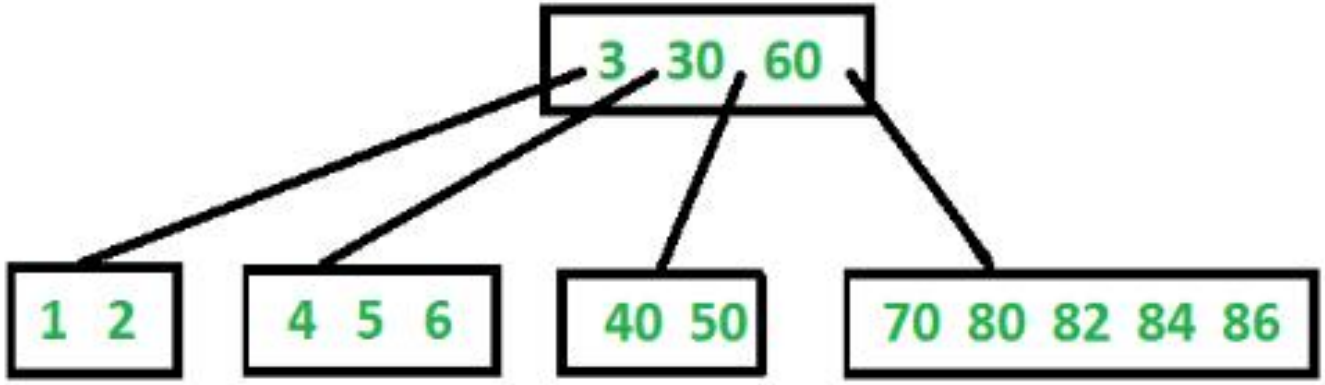
- A B-tree is a tree data structure that keeps data sorted and allows searches, insertions, and deletions in logarithmic amortized time. B-Tree is a self-balancing search tree. In most of the other self-balancing search trees (like AVL and Red-Black Trees), it is assumed that everything is in main memory.

B-Trees

- A B-tree of order m is an m-way tree (i.e., a tree where each node may have up to m children) in which:
  - the number of keys in each non-leaf node is one less than the number of its children and these keys partition the keys in the children in the fashion of a search tree
  - all leaves are on the same level
  - all non-leaf nodes except the root have at least [m / 2]children
  - the root is either a leaf node, or it has from two to m children
  - a leaf node contains no more than m – 1 keys
- The number m should always be odd

B-Trees

- The number m should always be odd



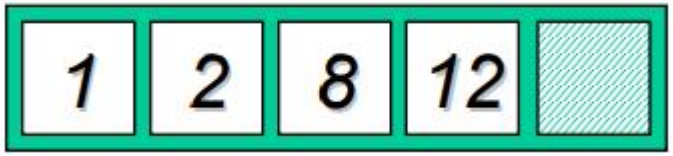
B-Trees

- **Properties of B-Tree**

  1) All leaves are at same level.
  2) A B-Tree is defined by the term *minimum degree* ‘t’. The value of t depends upon disk block size.
  3) Every node except root must contain at least t-1 keys. Root may contain minimum 1 key.
  4) All nodes (including root) may contain at most 2t – 1 keys.
  5) Number of children of a node is equal to the number of keys in it plus 1.
  6) All keys of a node are sorted in increasing order. The child between two keys k1 and k2 contains all keys in the range from k1 and k2.
  7) B-Tree grows and shrinks from the root which is unlike Binary Search Tree. Binary Search Trees grow downward and also shrink from downward.
  8) Like other balanced Binary Search Trees, time complexity to search, insert and delete is O(Logn).

B-Trees

- **Constructing a B-tree**
  - Suppose we start with an empty B-tree and keys arrive in the following order:1 12 8 2 25 6 14 28 17 7 52 16 48 68 3 26 29 53 55 45
  - We want to construct a B-tree of order 5
  - The first four items go into the root:

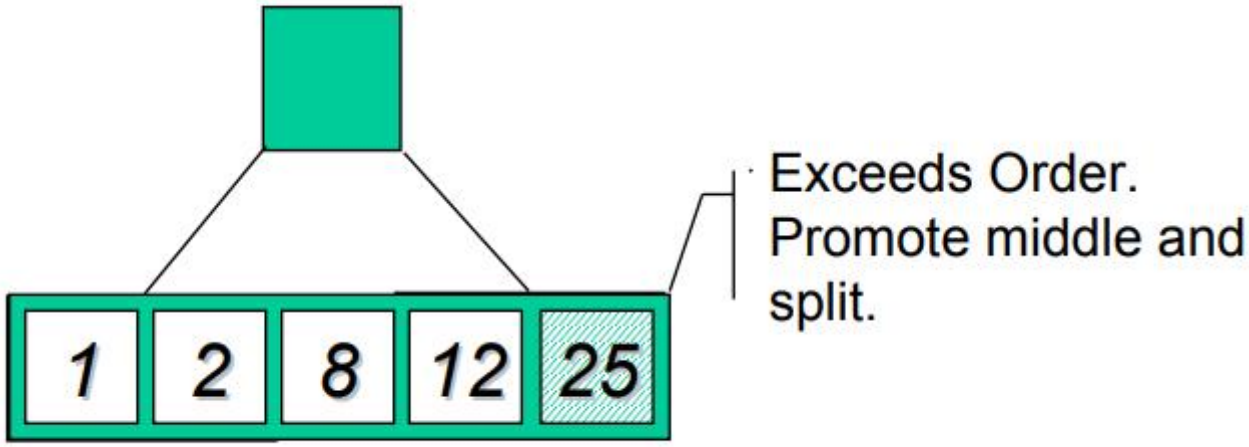


  - To put the fifth item in the root would violate condition 5
  - • Therefore, when 25 arrives, pick the middle key to make a new root

B-Trees

- **Constructing a B-tree**

  Add 25 to the tree



The Advantages of B-Trees

- **Advantages:**
  - Lack of redundant storage (but only marginally different).
  - Some searches are faster (key may be in non-leaf node).
- **Disadvantages:**
  - Leaf and non-leaf nodes are of different size (complicates storage)
  - Deletion may occur in a non-leaf node (more complicated)
- Generally, the structural simplicity of B -tree is preferred.

B+ Tree

- The drawback of B-tree used for indexing, however is that it stores the data pointer corresponding to a particular key value, along with that key value in the node of a B-tree.

B+ Tree

- A **B+ tree** is an N-ary tree with a variable but often large number of children per node.
- A B+ tree consists of a root, internal nodes and leaves.
- The root may be either a leaf or a node with two or more children.

- A B+ tree can be viewed as a B-tree in which each node contains only keys (not key–value pairs), and to which an additional level is added at the bottom with linked leaves.
- The B+-Tree consists of two types of nodes:
  - internal nodes
  - leaf nodes

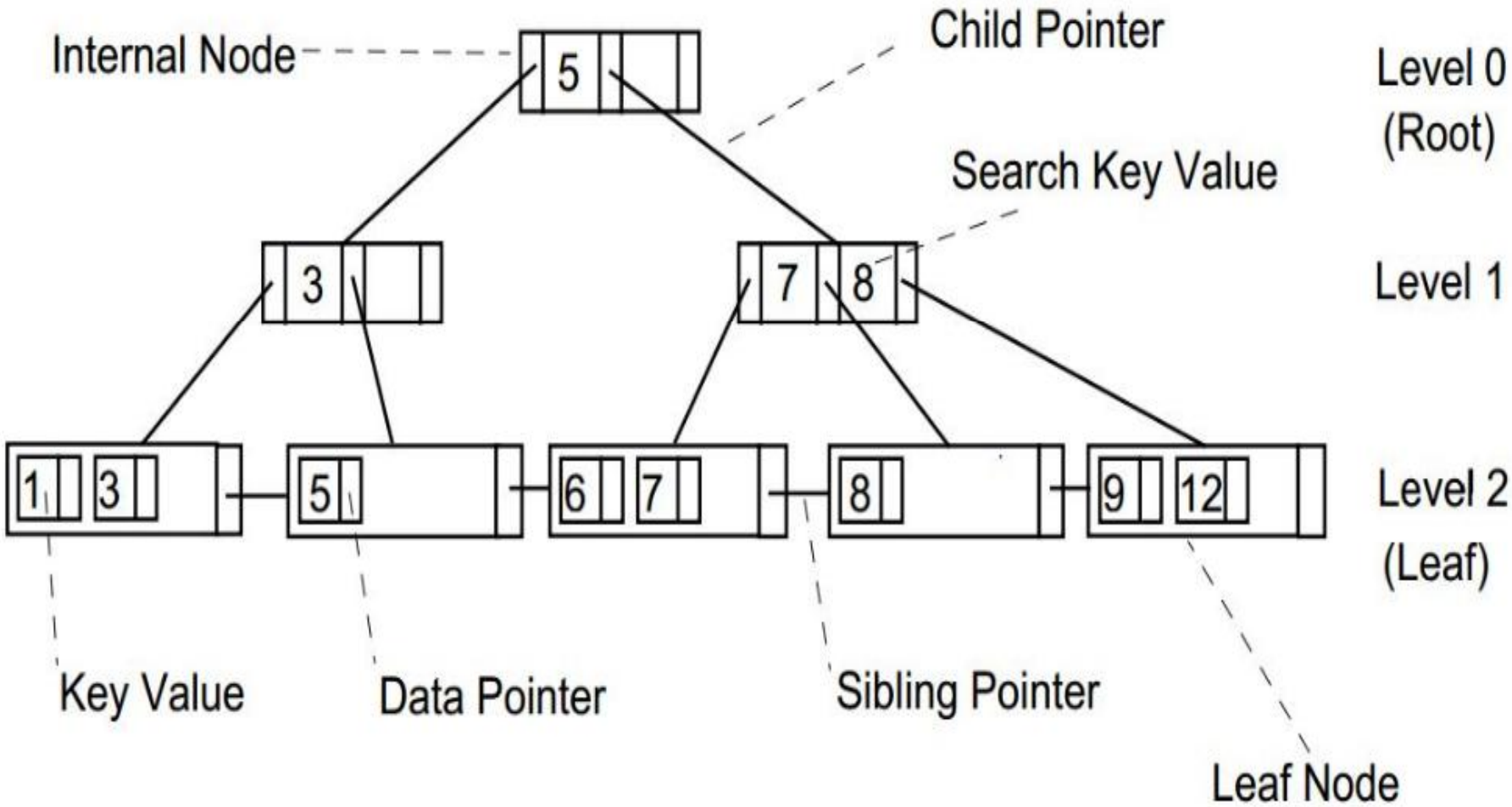
B+ Tree

- **Properties:**
- Internal nodes point to other nodes in the tree.
- Leaf nodes point to data in the database using data pointers. Leaf nodes also contain an additional pointer, called the sibling pointer, which is used to improve the efficiency of certain types of search.
- All the nodes in a B+-Tree must be at least half full except the root node which may contain a minimum of two entries. The algorithms that allow data to be inserted into and deleted from a B+-Tree guarantee that each node in the tree will be at least half full.

B+ Tree

- **Properties:**
- Searching for a value in the B+-Tree always starts at the root node and moves downwards until it reaches a leaf node.
- Both internal and leaf nodes contain key values that are used to guide the search for entries in the index.
- The B+ Tree is called a balanced tree because every path from the root node to a leaf node is the same length. A balanced tree means that all searches for individual values require the same number of nodes to be read from the disc.

B+ Tree


Internal Node
Child Pointer
Level 0
(Root)
Search Key Value
Level 1
Level 2
(Leaf)
Key Value
Data Pointer
Sibling Pointer
Leaf Node

B+ Tree

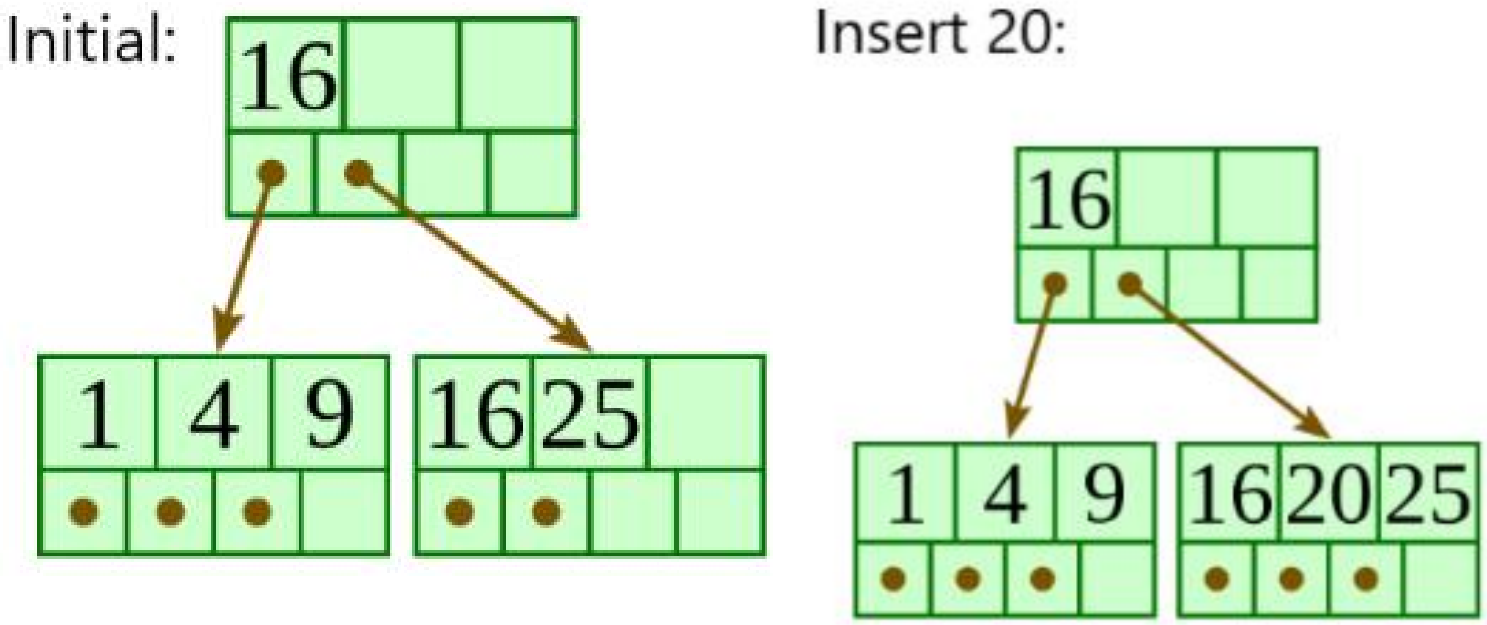
- **Basic operations associated with B+ Tree:**
  - **Searching a node in a B+ Tree**
    - Perform a binary search on the records in the current node.
    - If a record with the search key is found, then return that record.
    - If the current node is a leaf node and the key is not found, then report an unsuccessful search.
    - Otherwise, follow the proper branch and repeat the process.

B+ Tree

- **Insertion of node in a B+ Tree:**
  - Allocate new leaf and move half the buckets elements to the new bucket.
  - Insert the new leaf's smallest key and address into the parent.
  - If the parent is full, split it too.
  - Add the middle key to the parent node.
  - Repeat until a parent is found that need not split.
  - If the root splits, create a new root which has one key and two pointers. (That is, the value that gets pushed to the new root gets removed from the original node)

B+ Tree

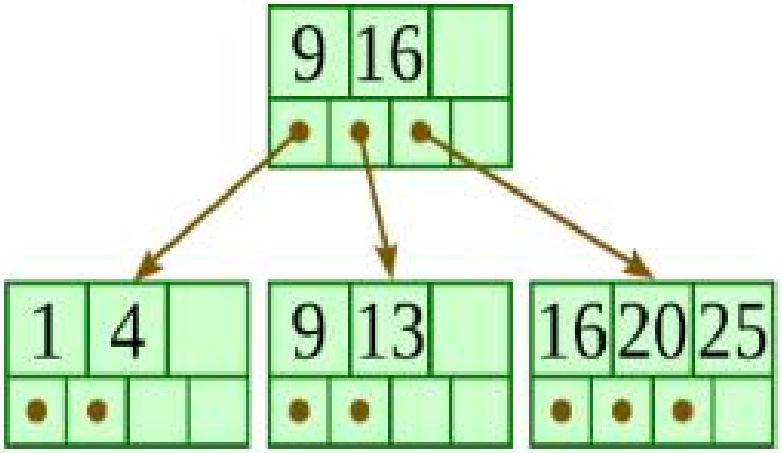
- **Insertion of node in a B+ Tree:**



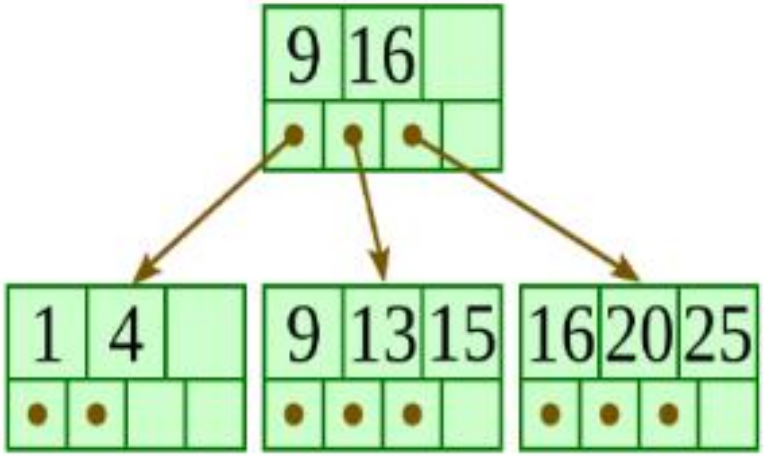
B+ Tree

- **Insertion of node in a B+ Tree:**

Insert 13:

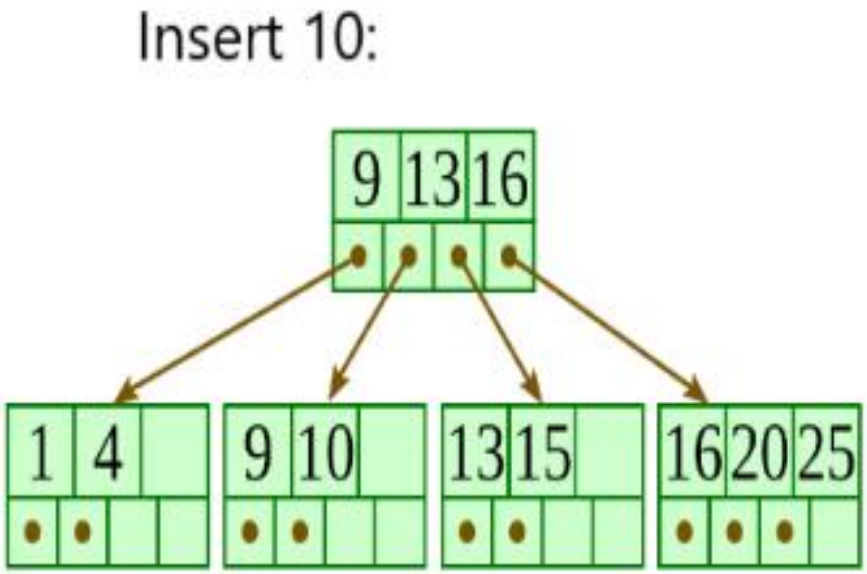


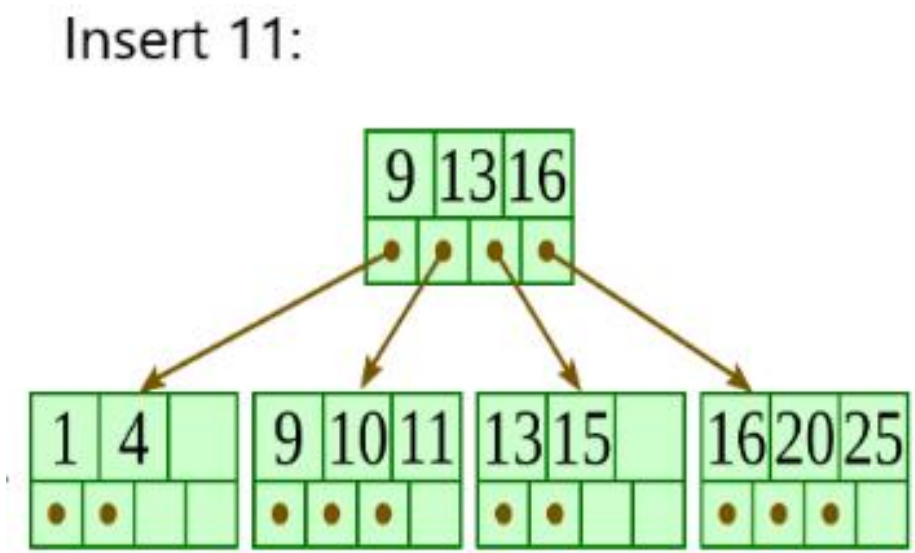
Insert 15:



B+ Tree

- **Insertion of node in a B+ Tree:**





B+ Tree

- **Deletion of a node in a B+ Tree:**
  - Descend to the leaf where the key exists.
  - Remove the required key and associated reference from the node.
  - If the node still has enough keys and references to satisfy the invariants, stop.
  - If the node has too few keys to satisfy the invariants, but its next oldest or next youngest sibling at the same level has more than necessary, distribute the keys between this node and the neighbor. Repair the keys in the level above to represent that these nodes now have a different “split point” between them; this involves simply changing a key in the levels above, without deletion or insertion.

B+ Tree

- **Deletion of a node in a B+ Tree:**
  - If the node has too few keys to satisfy the invariant, and the next oldest or next youngest sibling is at the minimum for the invariant, then merge the node with its sibling; if the node is a non-leaf, we will need to incorporate the “split key” from the parent into our merging.
  - In either case, we will need to repeat the removal algorithm on the parent node to remove the “split key” that previously separated these merged nodes — unless the parent is the root and we are removing the final key from the root, in which case the merged node becomes the new root (and the tree has become one level shorter than before).

External Sorting

- All the internal sorting algorithms require that the input fit into main memory.
- There are, however, applications where the input is much too large to fit into memory.
- For those external sorting algorithms, which are designed to handle very large inputs.

Why We Need New Algorithms

- Most of the internal sorting algorithms take advantage of the fact that memory is directly addressable.
- Shell sort compares elements a[i] and a[i - hk] in one time unit.
- Heap sort compares elements a[i] and a[i * 2] in one time unit.
- Quicksort, with median-of-three partitioning, requires comparing a[left], a[center], and a[right] in a constant number of time units.
- If the input is on a tape, then all these operations lose their efficiency, since elements on a tape can only be accessed sequentially.

Why We Need New Algorithms

- Even if the data is on a disk, there is still a practical loss of efficiency because of the delay required to spin the disk and move the disk head.
- The time it takes to sort the input is certain to be insignificant compared to the time to read the input, even though sorting is an O(n log n) operation and reading the input is only O(n).

Model for External Sorting

- The wide variety of mass storage devices makes external sorting much more device dependent than internal sorting.
- The algorithms that we will consider work on tapes, which are probably the most restrictive storage medium.
- Since access to an element on tape is done by winding the tape to the correct location, tapes can be efficiently accessed only in sequential order

External Sorting

- Used when the data to be sorted is so large that we cannot use the computer's internal storage (main memory) to store it
- We use secondary storage devices to store the data
- The secondary storage devices we discuss here are tape drives. Any other storage device such as disk arrays, etc. can be used

External Sorting

- Sorting large amount of data requires external or secondary memory.
- This process uses external memory such as HDD, to store the data which is not fir into the main memory.
- So, primary memory holds the currently being sorted data only.
- All external sorts are based on process of merging.
- Different parts of data are sorted separately and merged together.

External Sorting

- External Sorting is sorting the lists that are so large that the whole list cannot be contained in the internal memory of a computer.
- Assume that the list(or file) to be sorted resides on a disk. The term block refers to the unit of data that is read form or written to a disk at one time.
- External sorting typically uses a hybrid sort-merge strategy.

External Sorting

- In the sorting phase, chunks of data small enough to fit in main memory are read, sorted, and written out to a temporary file.
- In the merge phase, the sorted sub-files are combined into a single larger file.
- One example of external sorting is the external merge sort algorithm, which sorts chunks that each fit in RAM, then merges the sorted chunks together.

External Sorting

- A block generally consists of several records. For a disk, there are three factors contributing to read/write time:

  (i) Seek time: time taken to position the read/write heads to the correct cylinder. This will depend on the number of cylinders across which the heads have to move.

  (ii) Latency time: time until the right sector of the track is under the read/write head.

  (iii) Transmission time: time to transmit the block of data to/from the disk.

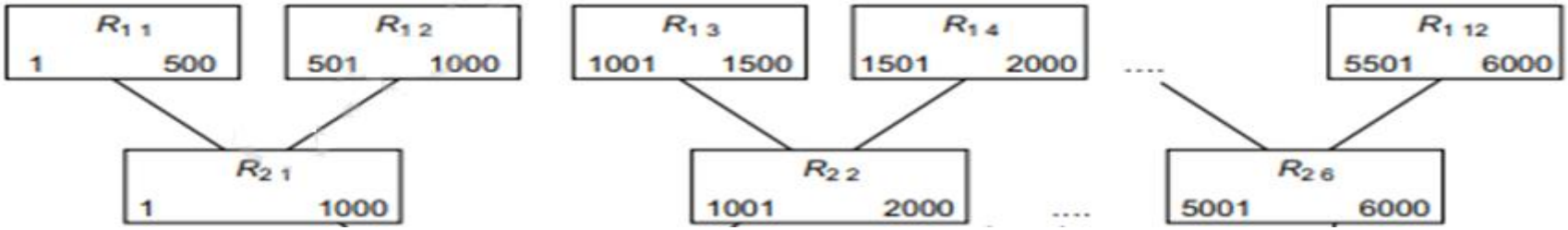
2-Way Merge Sort

- The k–way merge sort where k=2 is a 2–way merge sort.
- In 2–way merge sort 2 runs are merged at a time to generate a single run twice as long.
- The merging process is repeated until a single run is generated.

2-Way Merge Sort

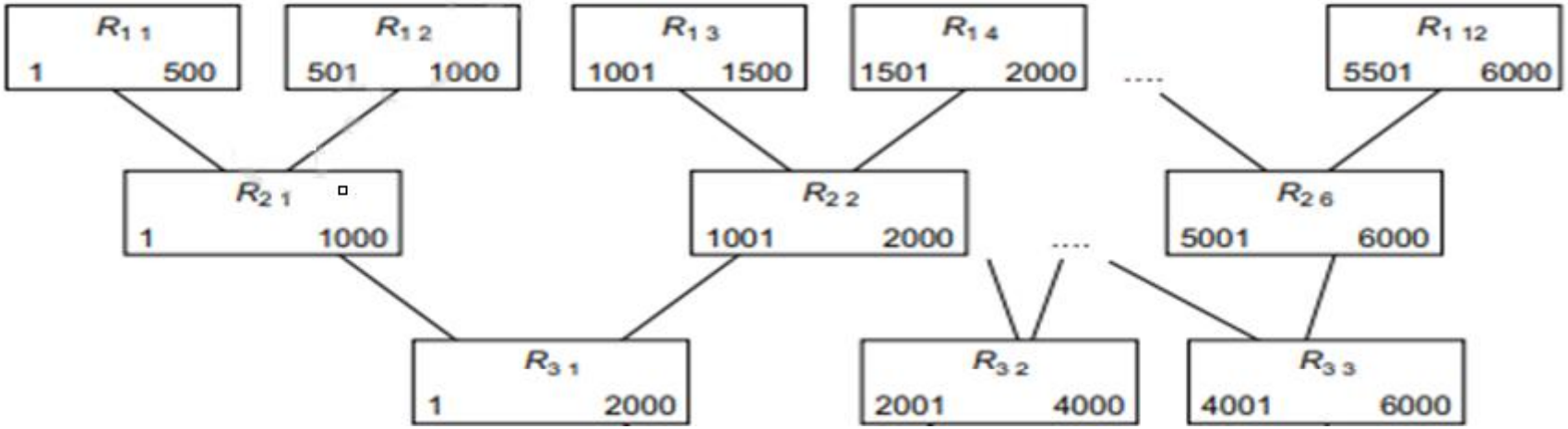
- consider that there are 6000 records to be sorted and the internal memory capacity is 500 records.

- Let Ri j represent the jth run in the ith pass.
- The generated runs in the first pass are R1 1 to R1 12.
- In the first pass, R1 1 and R1 2 are merged resulting in run R2 1 which consists of the sorted list of first 1000 records.



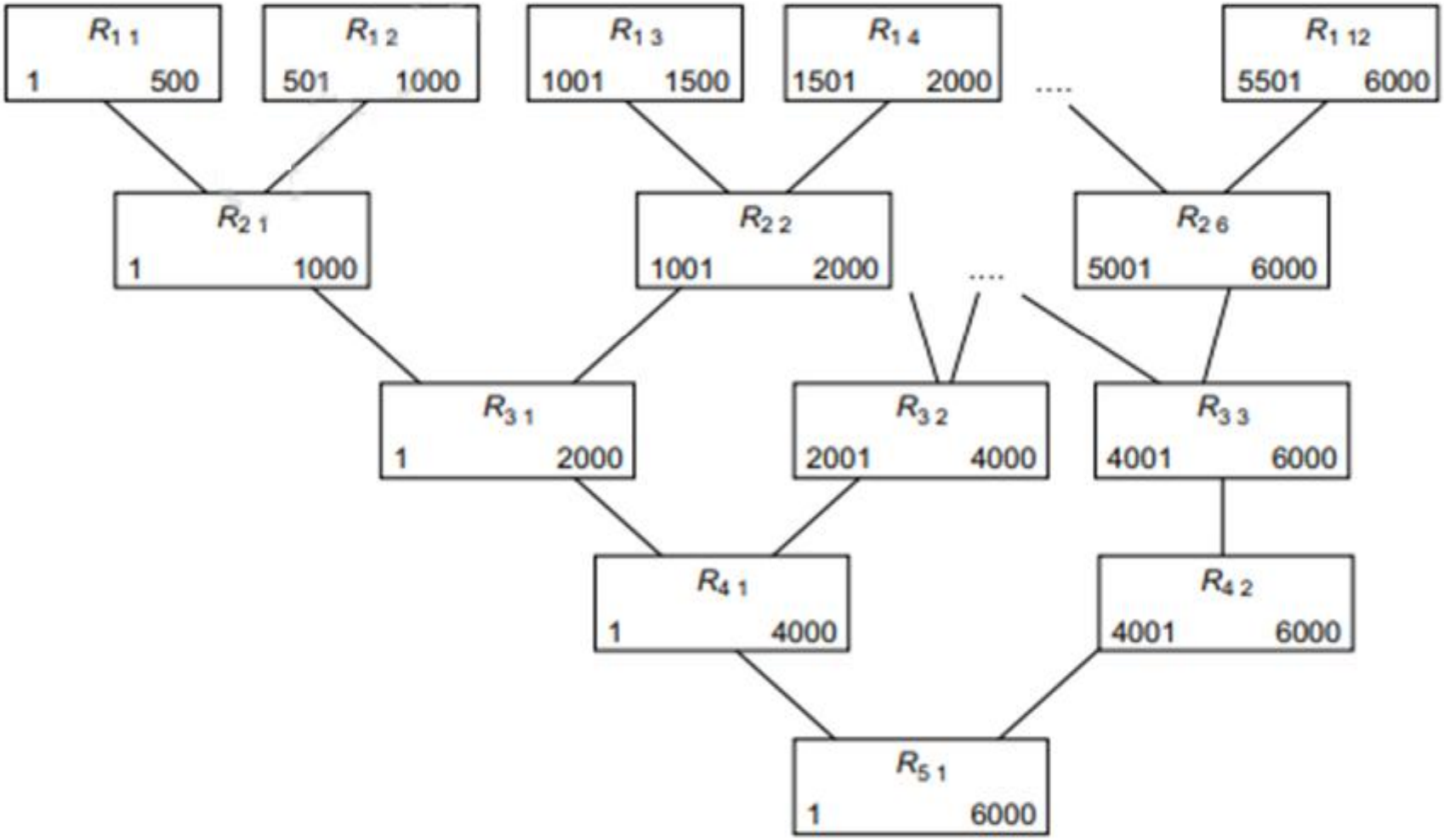
- The next two runs R1 3 and R1 4 are merged resulting in R2 2. Likewise, four other runs will be merged in the second pass resulting in runs R2 1 to R2 6.

2-Way Merge Sort



- Similarly, in the third pass, R2 1 and R2 2 are merged to form R3 1.
- Likewise, two other runs are generated resulting in runs R3 1 to R3 3.
- In the fourth pass, R3 1 and R3 2 are merged to form run R4 1.
- The last run R3 3 will be taken as it is to R4 2.
- In fifth pass, R4 1 and R4 2 runs are merged to form run R5 1, the final sorted file

2-Way Merge Sort

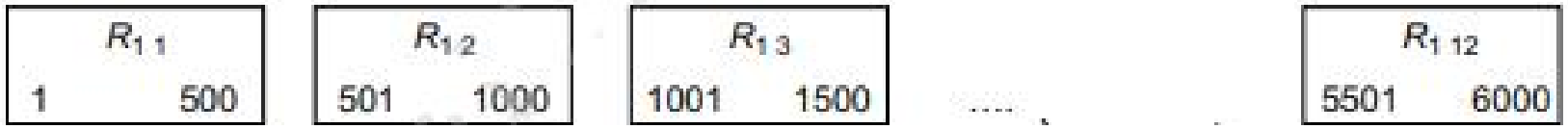


3–Way Merge Sort

- The k–way merge sort where k=3 is a 3–way merge sort.
- In 3–way merge sort, 3 runs are merged at a time to generate a single run thrice as long.
- The merging process is repeated until a single run is generated.

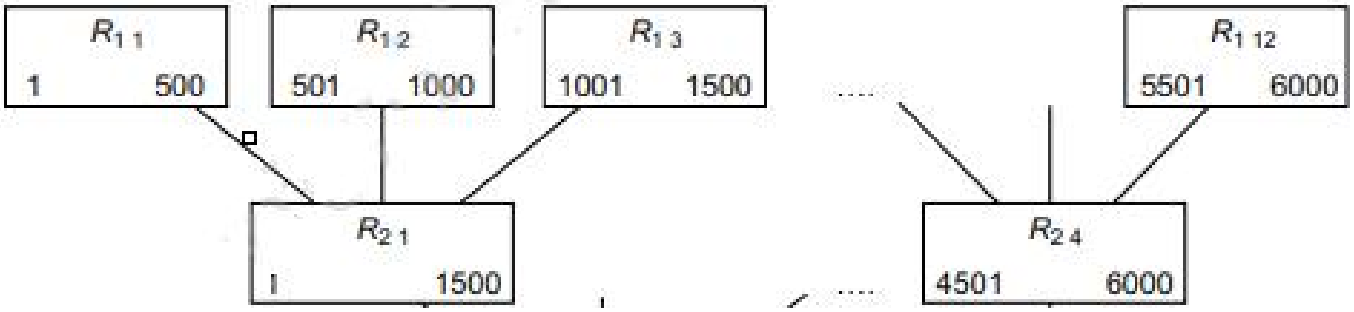
3–Way Merge Sort

- Consider 6000 records are available on a disk which are to be sorted. In the internal memory of the computer, only 500 records can be resided. The block size of the disk is 100 records. Sort the file using 3–way merge sort



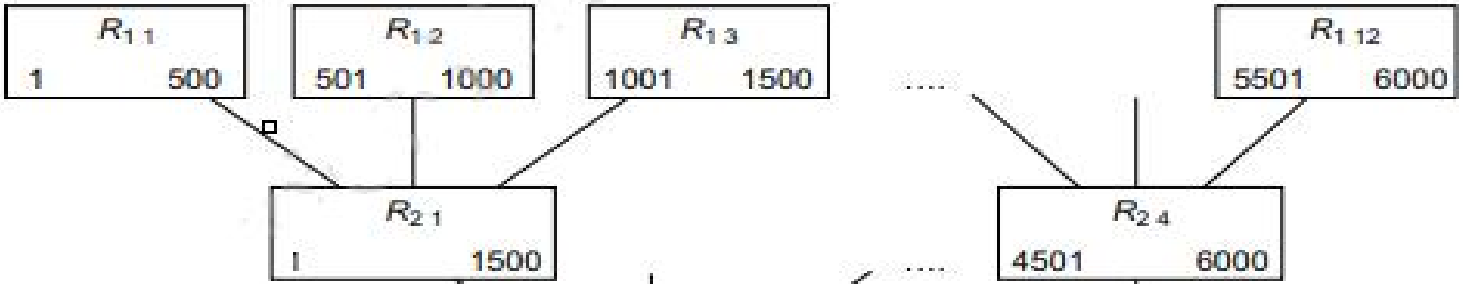
3–Way Merge Sort

- In the first pass, R1 1 to R1 3 are merged resulting in run R2 1, which consists of the sorted list of first 1500 records.



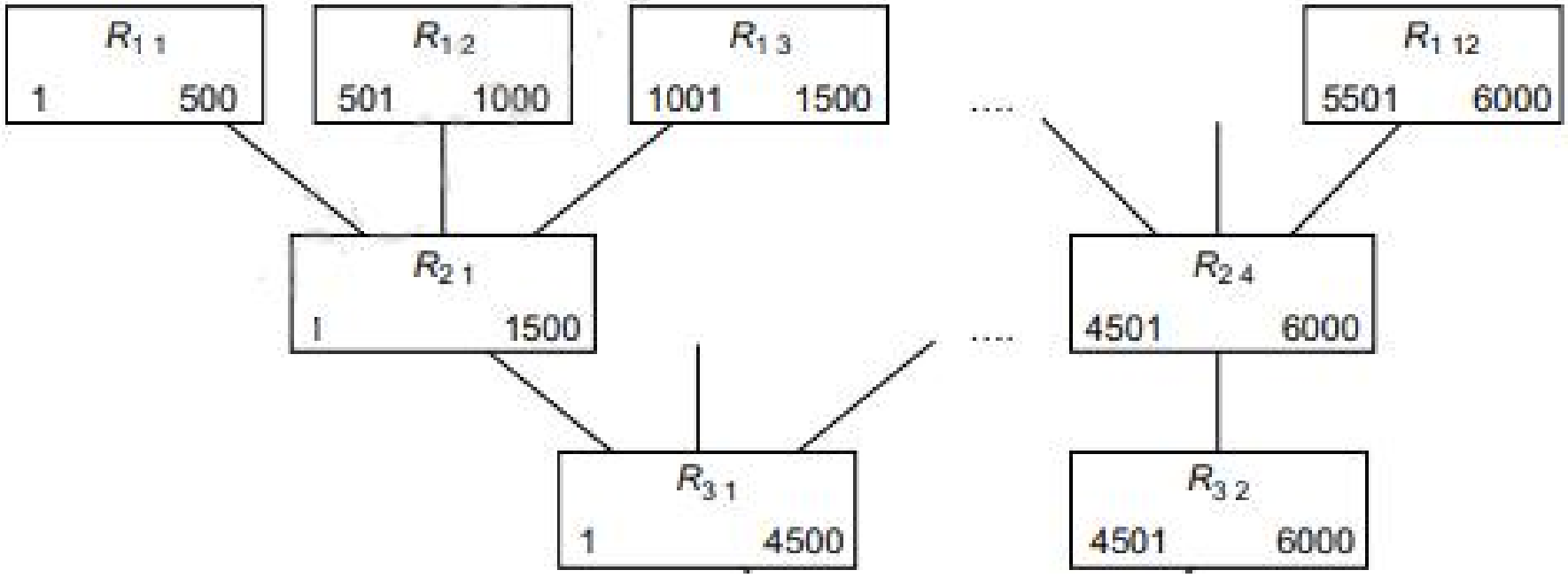
3–Way Merge Sort

- The next three runs R1 4, R1 5, R1 6 are merged resulting in R2 2.
- Likewise, four runs will be emerging in the second pass, i.e., R2 1 to R2 4.
- Similarly, in the third pass, R2 1 to R2 3 are merged to form R3 1.
- The last run R2 4 will be taken as it is to R3 2



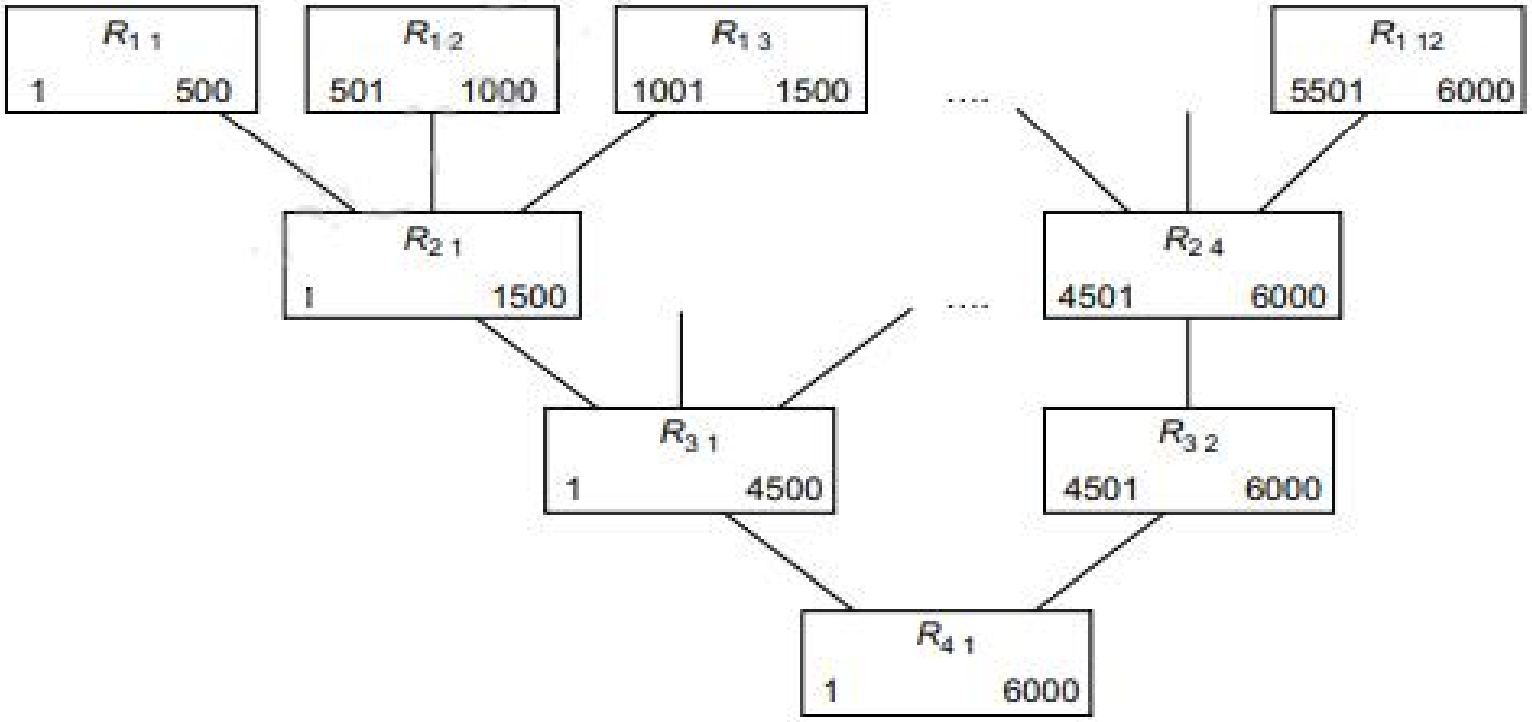
3–Way Merge Sort

- In fourth pass, R3 1 and R3 2 runs are merged to form run R4 1, the sorted output.



3–Way Merge Sort

- In fourth pass, R3 1 and R3 2 runs are merged to form run R4 1, the sorted output.



3–Way Merge Sort

- Implement the 3-way merge sort technique to consider 3 runs with 4 records each .

3	5	12	15

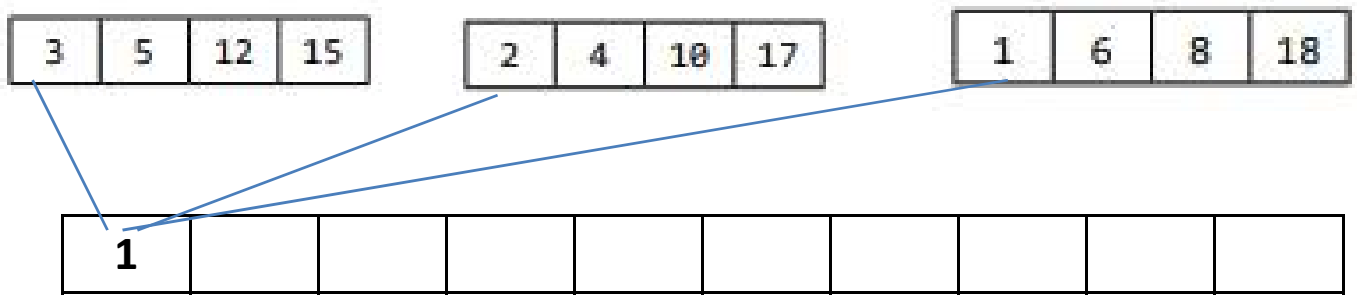
2	4	10	17

1	6	8	18

- Consider the smallest record of each run and add it to the smallest set: {3, 2, 1}.
- Take the smallest record of the smallest set, 1, add it to the output run and delete it from the original run. At this point, the output run is {1}. The step-by- step process of merging the three runs

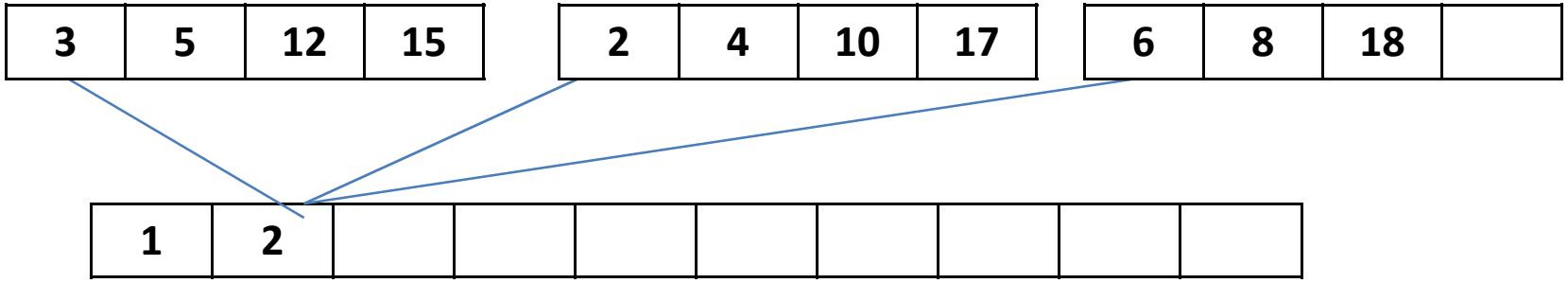
3–Way Merge Sort

- Step 1: The three records in the smallest set are {3, 2, 1}.
- Remove the smallest record1, from the third run and put it in the output run: {1}.
- Move 6 to the smallest set.



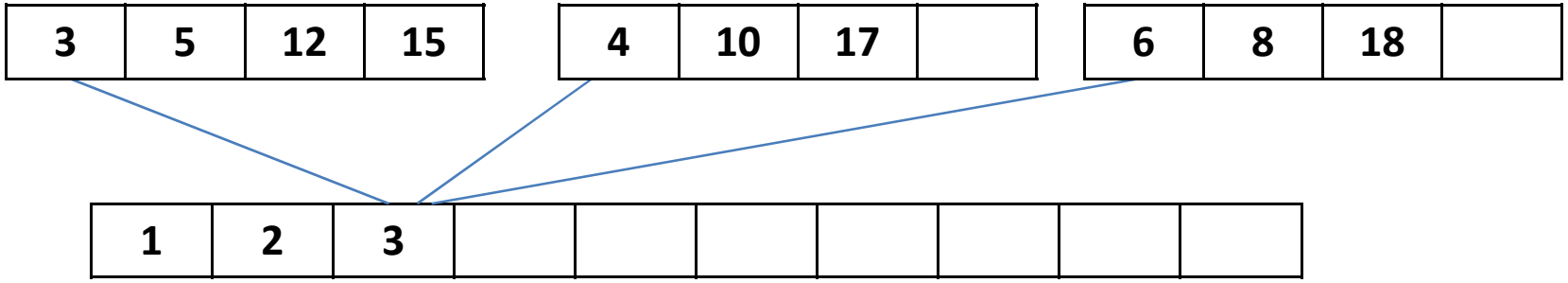
3–Way Merge Sort

- Step 2: The three records in smallest set are {3, 2, 6}.
- Remove 2 from the second run and append it to the output run: {1, 2}.
- Move 4 to the smallest set.



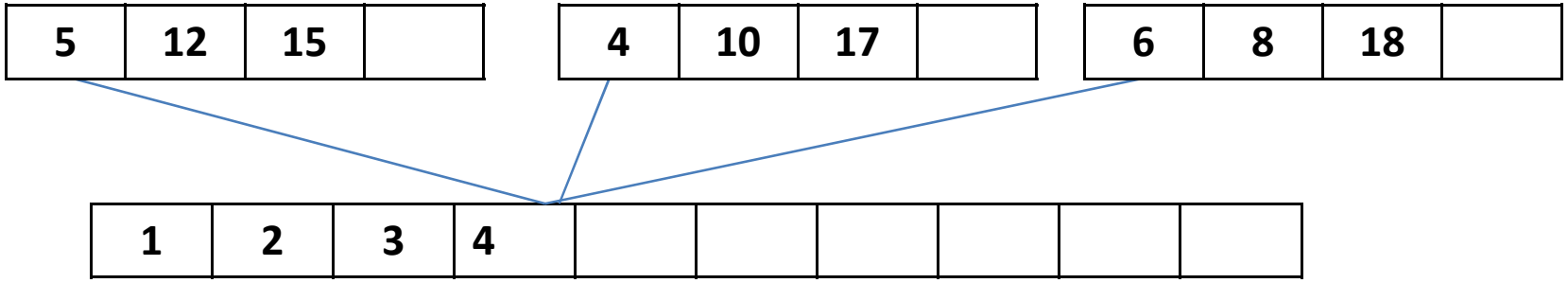
3–Way Merge Sort

- Step 3: The three records in smallest set are {3, 4, 6}.
- Remove 3 from the first run and append it to the output run: {1, 2, 3}.
- Move 5 to the smallest set.



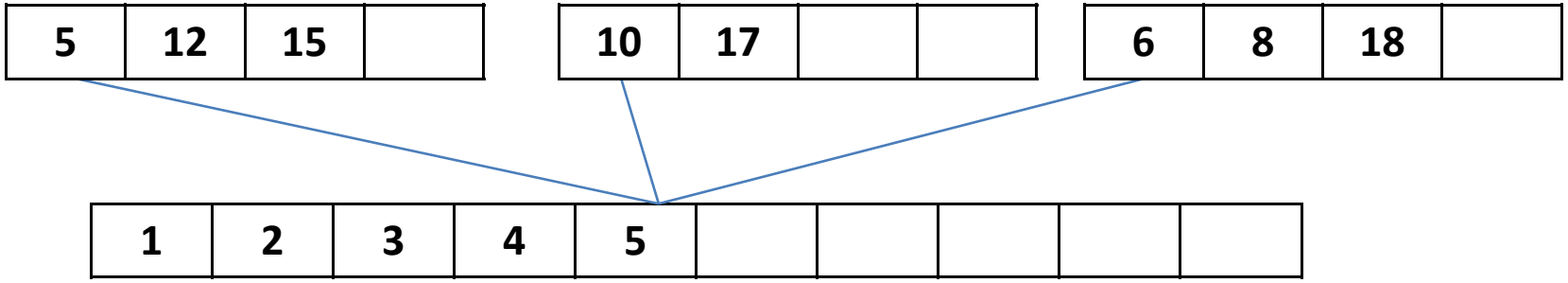
3–Way Merge Sort

- Step 4: The three records in smallest set are {5, 4, 6}.
- Remove 4 from the second run and append it to the output run: {1, 2, 3, 4}.
- Move 10 to the smallest set.



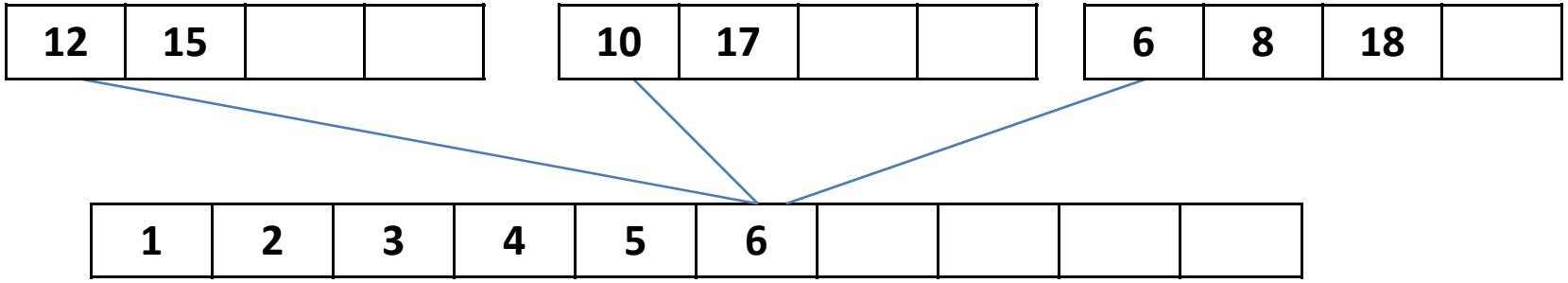
3–Way Merge Sort

- Step 5: The three records in smallest set are {5, 10, 6}.
- Remove 5 from the first run and append it to the output run: {1, 2, 3, 4, 5}.
- Move 12 to the smallest set.



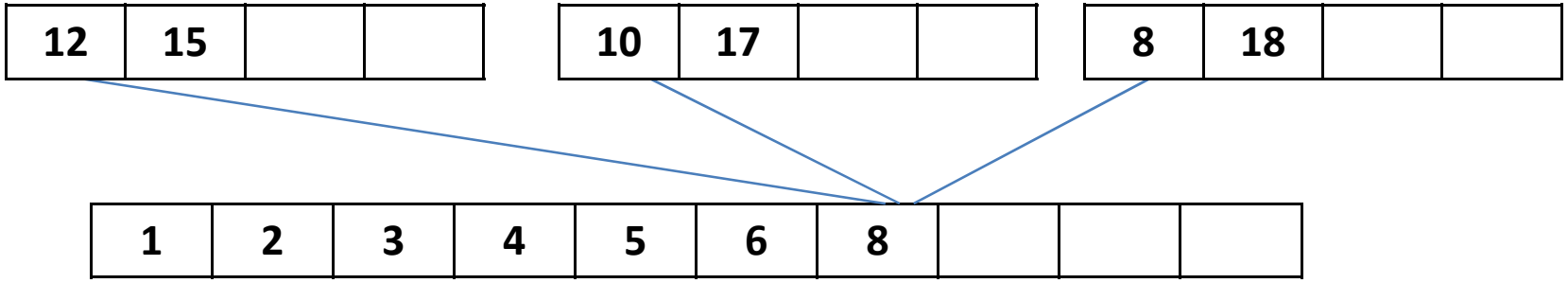
3–Way Merge Sort

- Step 6: The three records in smallest set are {12, 10, 6}.
- Remove 6 from the third run and append it to the output run: {1, 2, 3, 4, 5,6}.
- Move 8 to the smallest set.



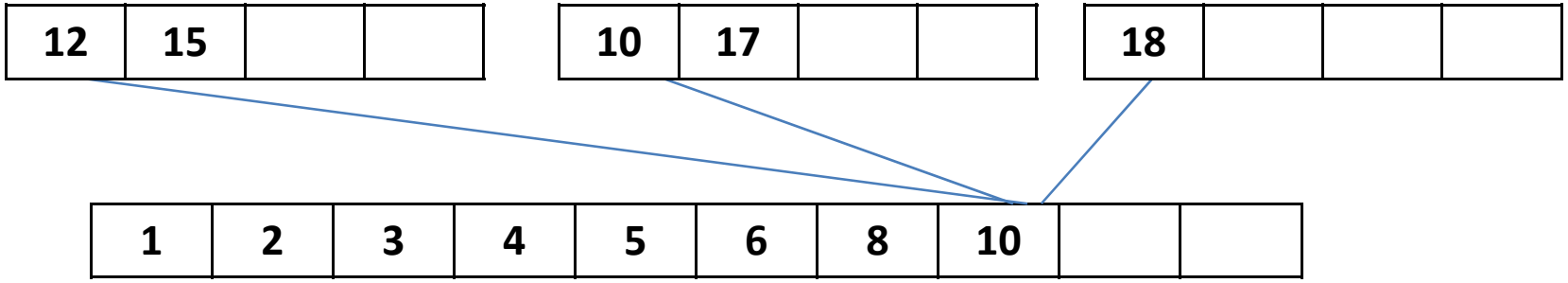
3–Way Merge Sort

- Step 7: The three records in smallest set are {12, 10, 8}.
- Remove 8 from the third run and append it to the output run: {1, 2, 3, 4, 5, 6, 8}.
- Move 18 to the smallest set.



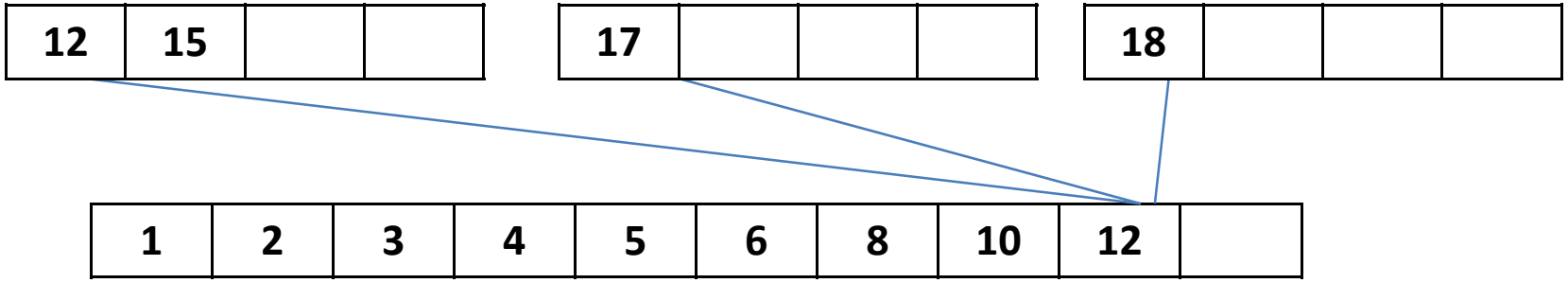
3–Way Merge Sort

- Step 8: The three records in smallest set are {12, 10, 18}.
- Remove 10 from the second run and append it to the output run: {1, 2, 3, 4, 5, 6, 8, 10}.
- Move 17 to the smallest set.



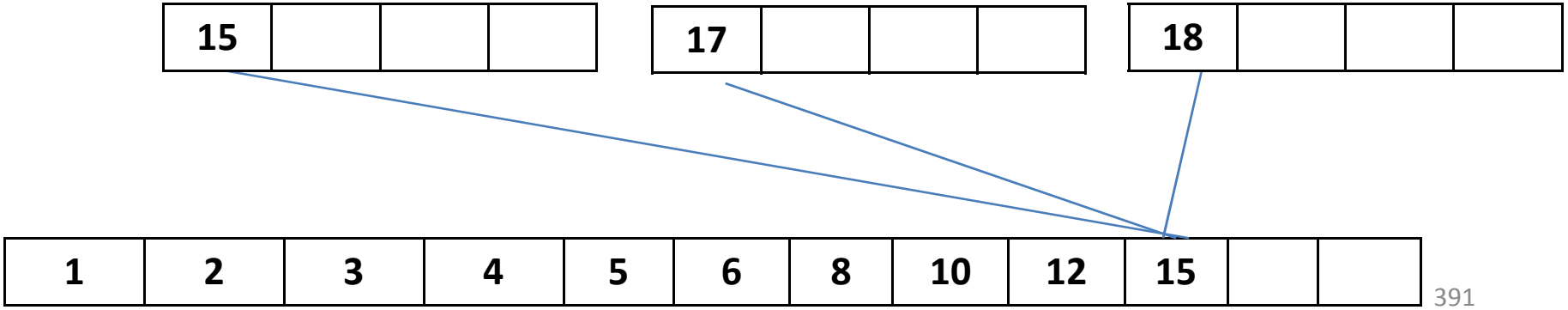
3–Way Merge Sort

- Step 9: The three records in smallest set are {12, 17, 18}.
- Remove 12 from the first run and append it to the output run: {1, 2, 3, 4, 5, 6, 8, 10, 12}.
- Move 15 to the smallest set.



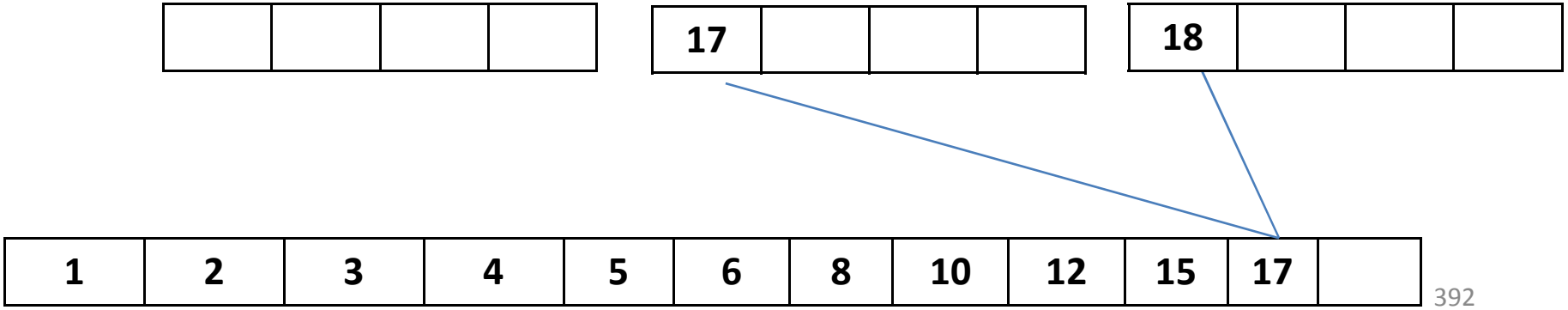
3–Way Merge Sort

- Step 10: The three records in smallest set are {15, 17, 18}.
- Remove 15 from the first run and append it to the output run: {1, 2, 3, 4, 5, 6, 8, 10, 12, 15}.
- The first run is now empty, the merge follows as a 2-way merge instead of a 3-way merge.



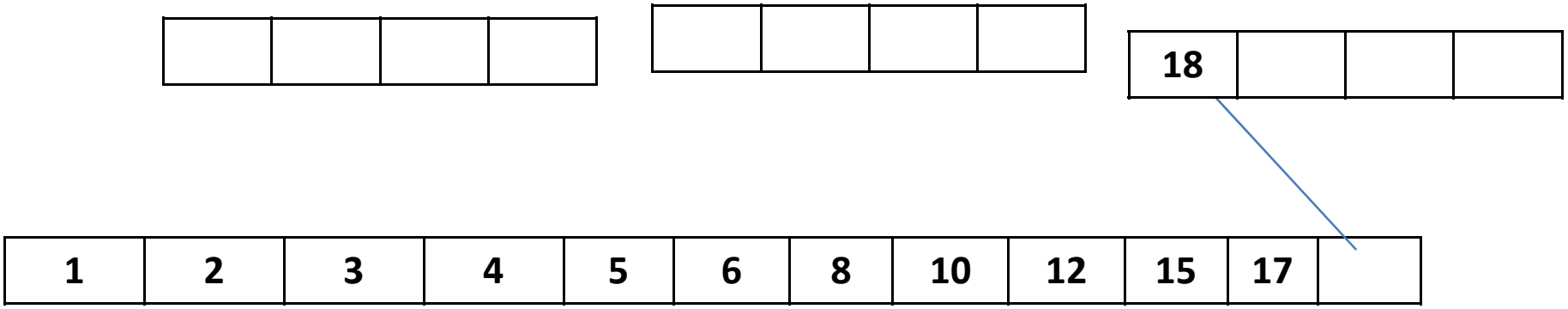
3–Way Merge Sort

- Step 11: The two top records are {17, 18}.
- Remove 17 from the second run and append it to the output run: {1, 2, 3, 4, 5, 6, 8, 10, 12, 15, 17}.
- Now, the second run is also empty, only the third run remains non-empty.



3–Way Merge Sort

- Step 12: The records of the last run are {18} and are appended to the output run and the final run is obtained {1, 2, 3, 4, 5, 6, 8, 10, 12, 15, 17, 18}.



k-way merge sort

- A *merge sort* that sorts a data stream using repeated merges.
- It distributes the input into k streams by repeatedly reading a *block* of input that fits in memory, called a *run*, sorting it, then writing it to the next stream.
- It merges runs from the k streams into an output stream. It then repeatedly distributes the runs in the output stream to the k streams and merges them until there is a single sorted output.

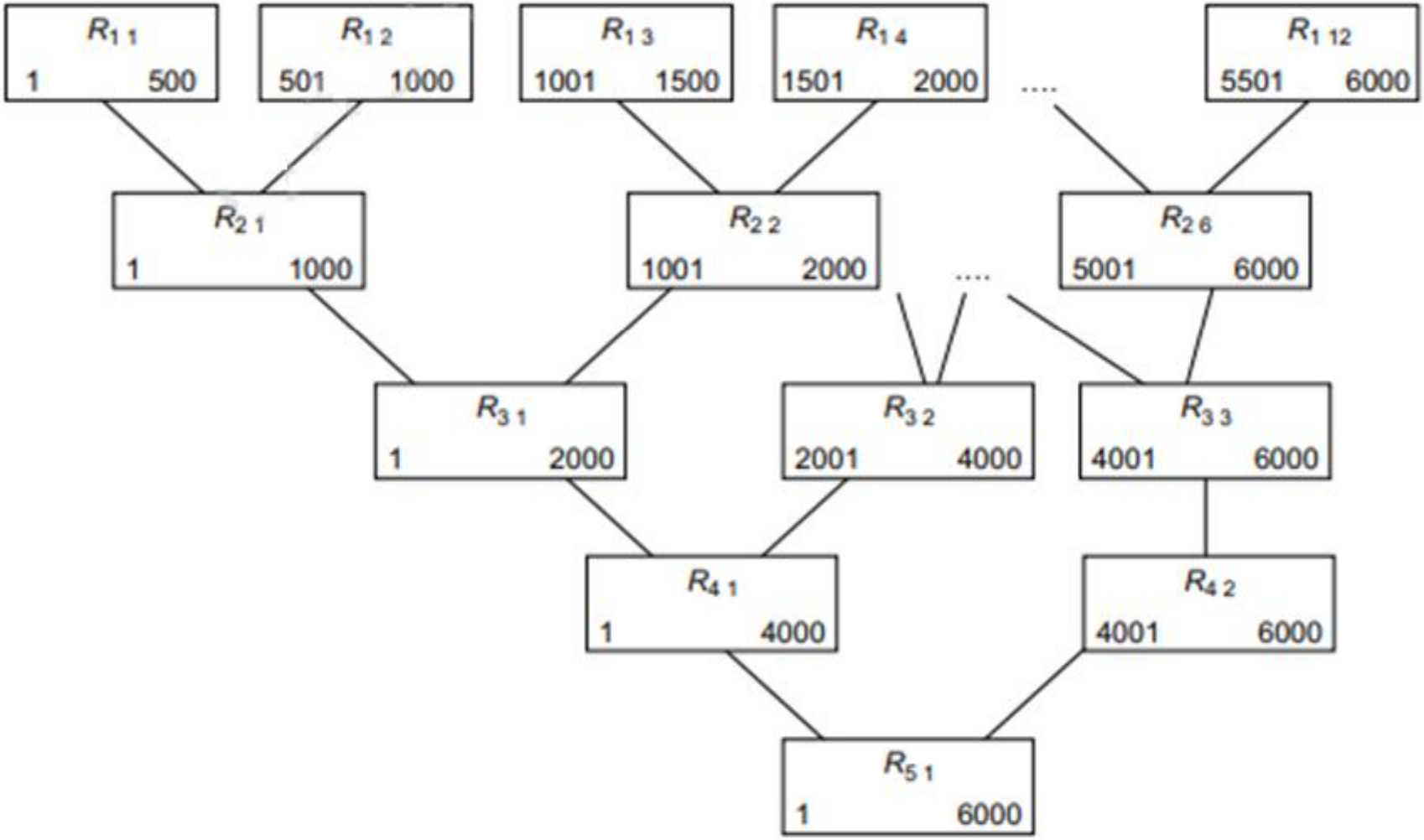
k-way merge sort

- **k-way merge:**
- **Definition:** Combine *k* sorted data streams into a single sorted stream.

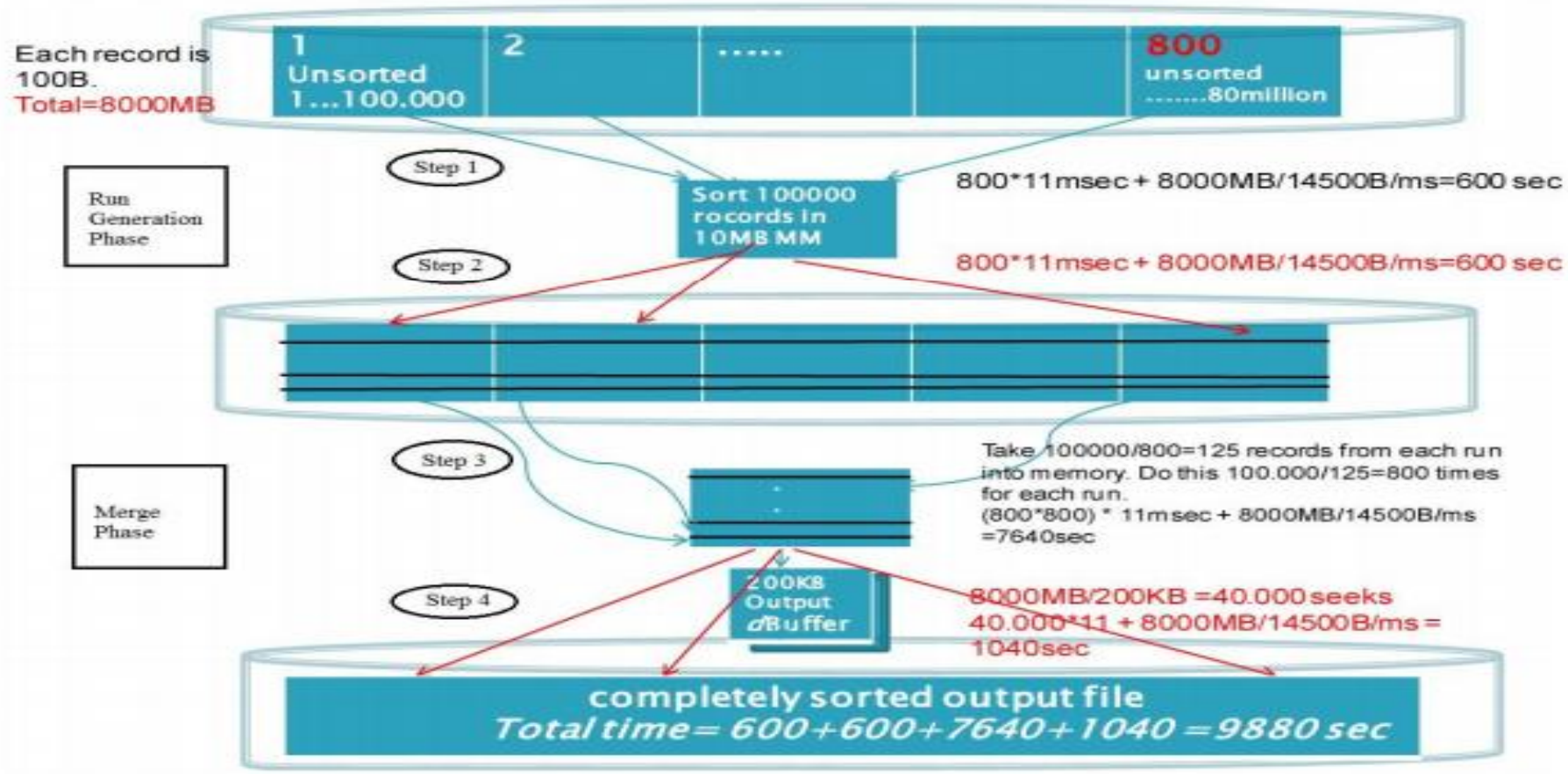
k-way merge sort

- External merge sort is performed in two phases.
- The first phase involves the run generation and the second phase involves the merging of runs to form a larger run.
- This run generation is repeated and merging is continued till a single run is generated with the sorted file as its outcome.
- If k runs are merged at a time, the external merge sort is known as a k–way merge sort.

k-way merge sort



k-way merge sort



Run Generation Phase

- One of the most commonly approaches to external sorting is external merge sort, which consists of two phases, the run generation phase and the merge phase.
- The first phase generates several sorted lists of records, called runs, and the second phase merges the runs into the final sorted list of records.

Run Generation Phase

- In the run generation phase, data is read from the input to generate subsets of ordered records.
- These subsets are called runs.
- Runs are generated using main (internal) memory, and written to external memory (disk).
- After all input records are distributed in runs, the run generation phase ends and the merge phase starts.

Run Generation Phase

- There are several methods used to generate the runs, most of them being based on internal sorting algorithms.
- For example, the main memory can be filled with records from the input and then sorted using any internal sorting algorithm (merge sort, quicksort, etc.) Using this method, called **Load-Sort-Store**, the run length is always equal to the size of the main memory, except for maybe the last run

Run Generation Phase

- Another more advanced algorithm is replacement selection.
- Using replacement selection, the run length is nearly equal to twice the size of the main memory (internal) when the input data is randomly distributed.

Tries

- All the search trees are used to store the collection of numerical values but they are not suitable for storing the collection of words or strings.
- Trie is a data structure which is used to store the collection of strings and makes searching of a pattern in words more easy.
- The term ***trie*** came from the word **retrieval**. Trie data structure makes retrieval of a string from the collection of strings more easily.
- Trie is also called as **Prefix Tree** and some times **Digital Tree**.
- In computer science, a trie, also called digital tree and sometimes radix tree or prefix tree.

Tries

- **Trie is a tree like data structure used to store collection of strings.**
- **Trie is an efficient information storage and retrieval data structure.**
- The trie data structure provides fast pattern matching for string data values.
- Using trie, we bring the search complexity of a string to the optimal limit.
- A trie searches a string in **O(m)** time complexity, where **m** is the length of the string.

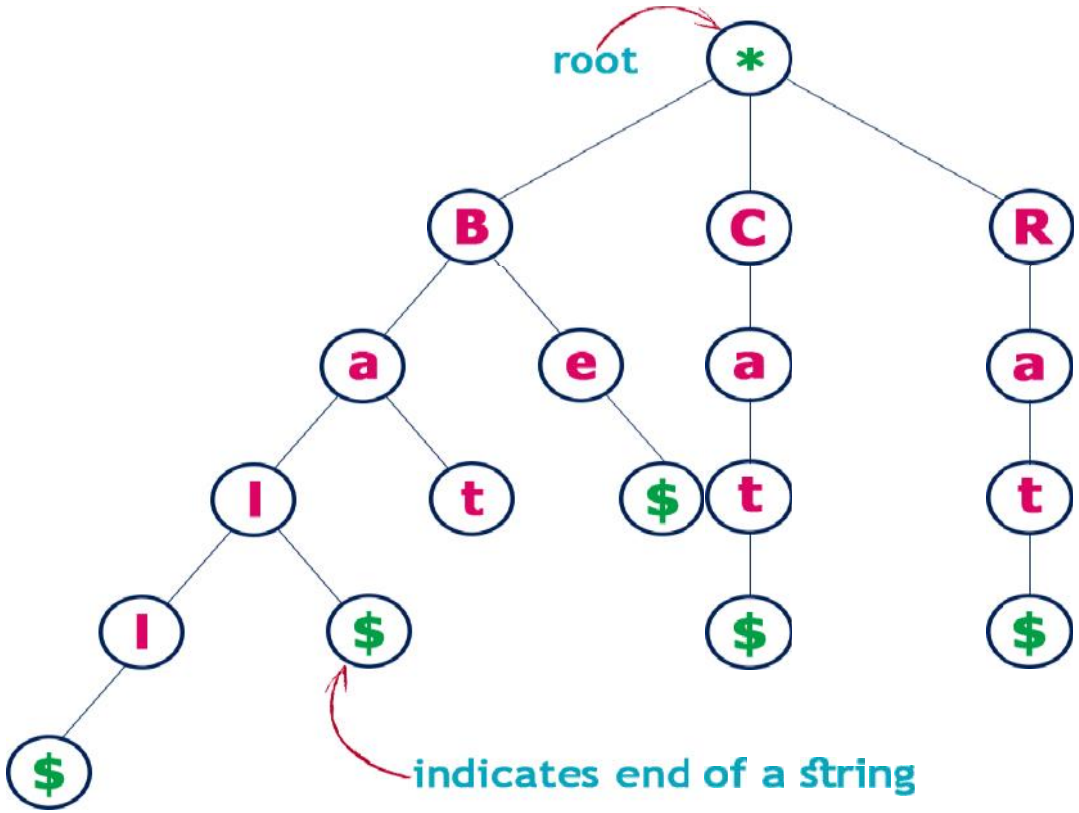
Properties of a tries

- A multi-way tree.
- Each node has from 1 to n children.
- Each edge of the tree is labeled with a character.
- Each leaf nodes corresponds to the stored string, which is a concatenation of characters on a path from the root to this node.

Tries

Consider the following list of strings to construct Trie

Cat, Bat, Ball, Rat, Cap & Be



Different Types of Tries

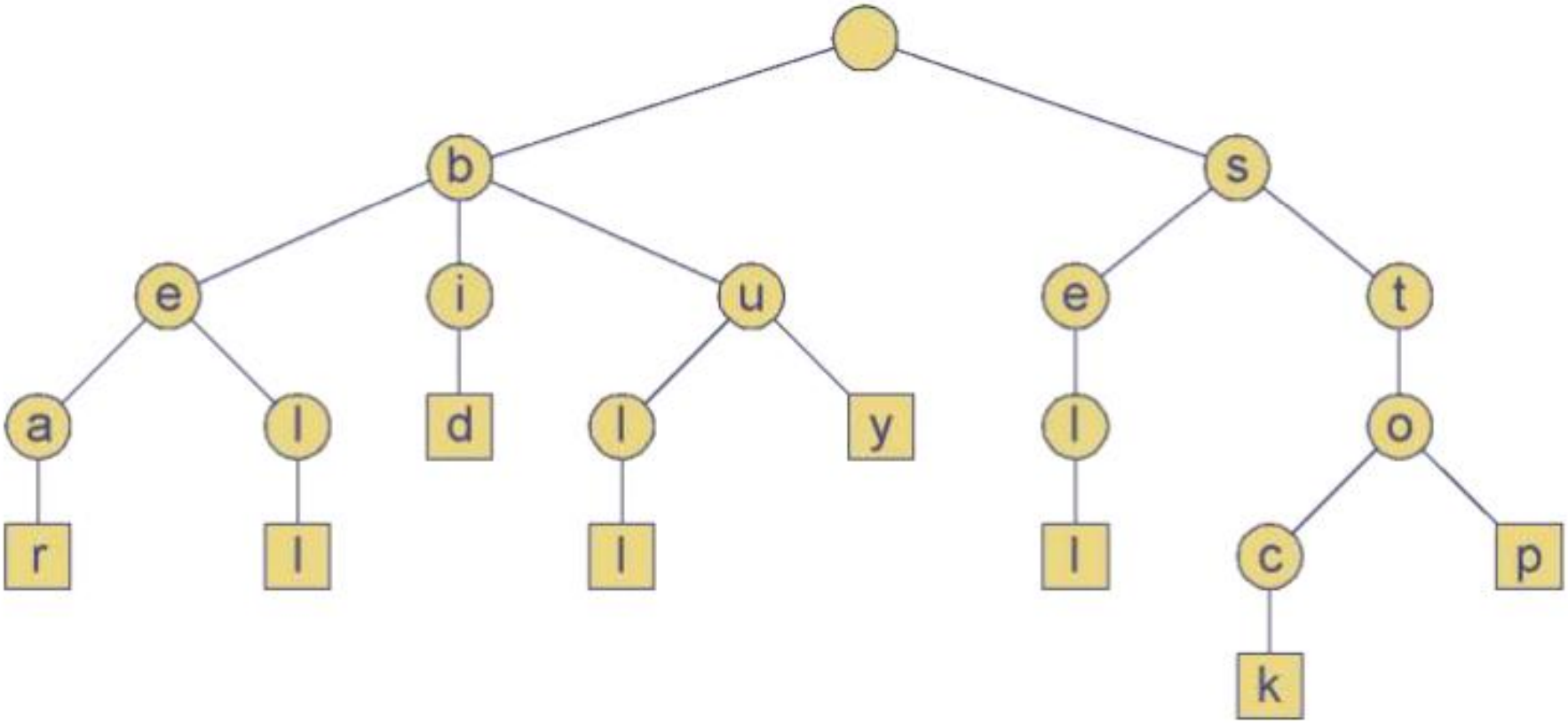
- Standard Tries
- Compressed/Compact Tries
- Suffix Tries

Standard Tries

- Standard Tries
  - The standard trie for a set of strings S is an ordered tree such that:
  - each node but the root is labeled with a character
  - the children of a node are alphabetically ordered
  - the paths from the external nodes to the root yield the strings of S

Standard Tries

- Standard Tries

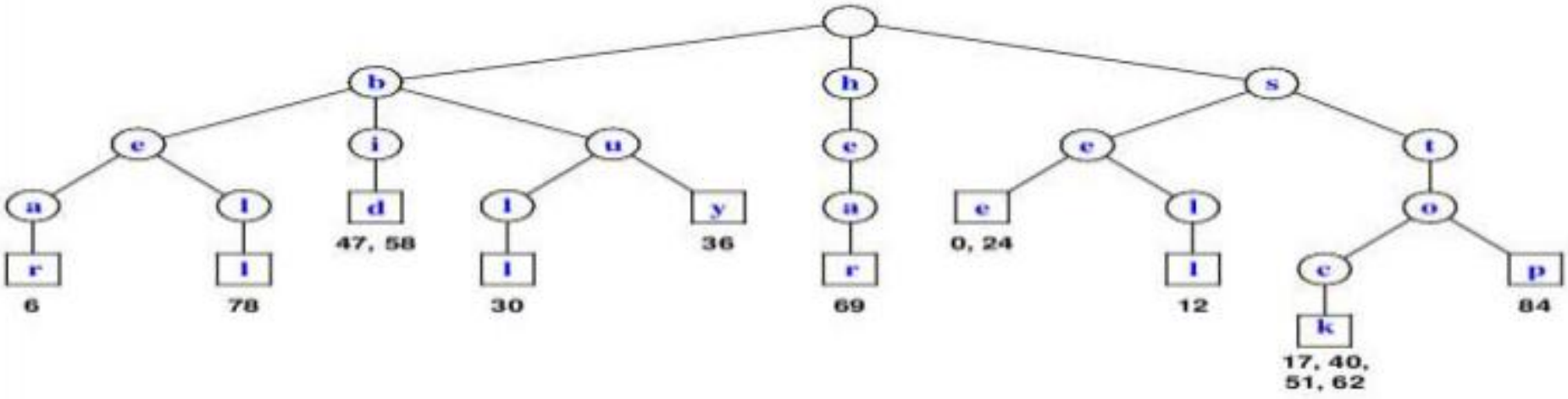


Standard Tries

- Applications of Standard Tries:
  - word matching: find the first occurrence of word X in the text
  - prefix matching: find the first occurrence of the longest prefix of word X in the text

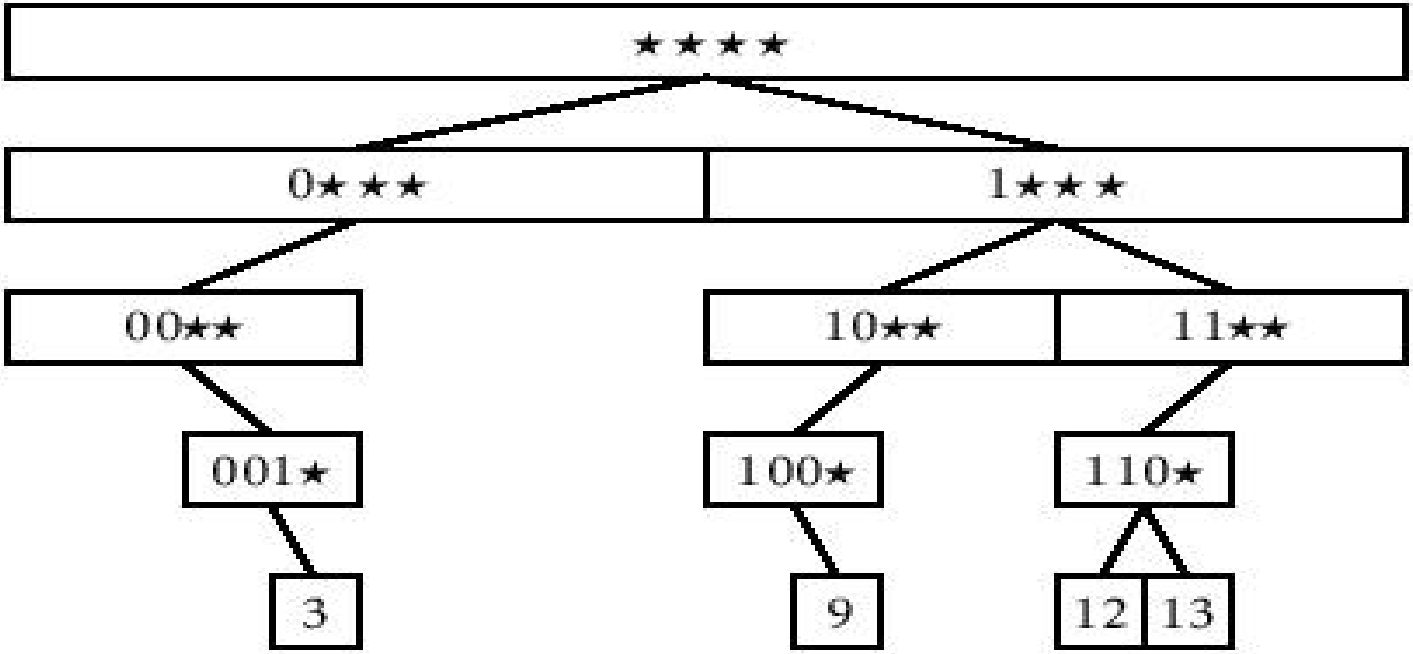
Standard Tries

- Applications of Standard Tries:
  - word matching: find the first occurrence of word X in the text
  - prefix matching: find the first occurrence of the longest prefix of word X in the text



Binary Trie

- A Binary Trie encodes a set of bit integers in a binary tree.
- All leaves in the tree have depth and each integer is encoded as a root-to-leaf path.
- The path for the integer turns left at level i if the ith most significant bit of x is a 0 and turns right if it is a 1.

Binary Trie

- an example for the case , in which the trie stores the integers 3(0011), 9(1001), 12(1100), and 13(1101).