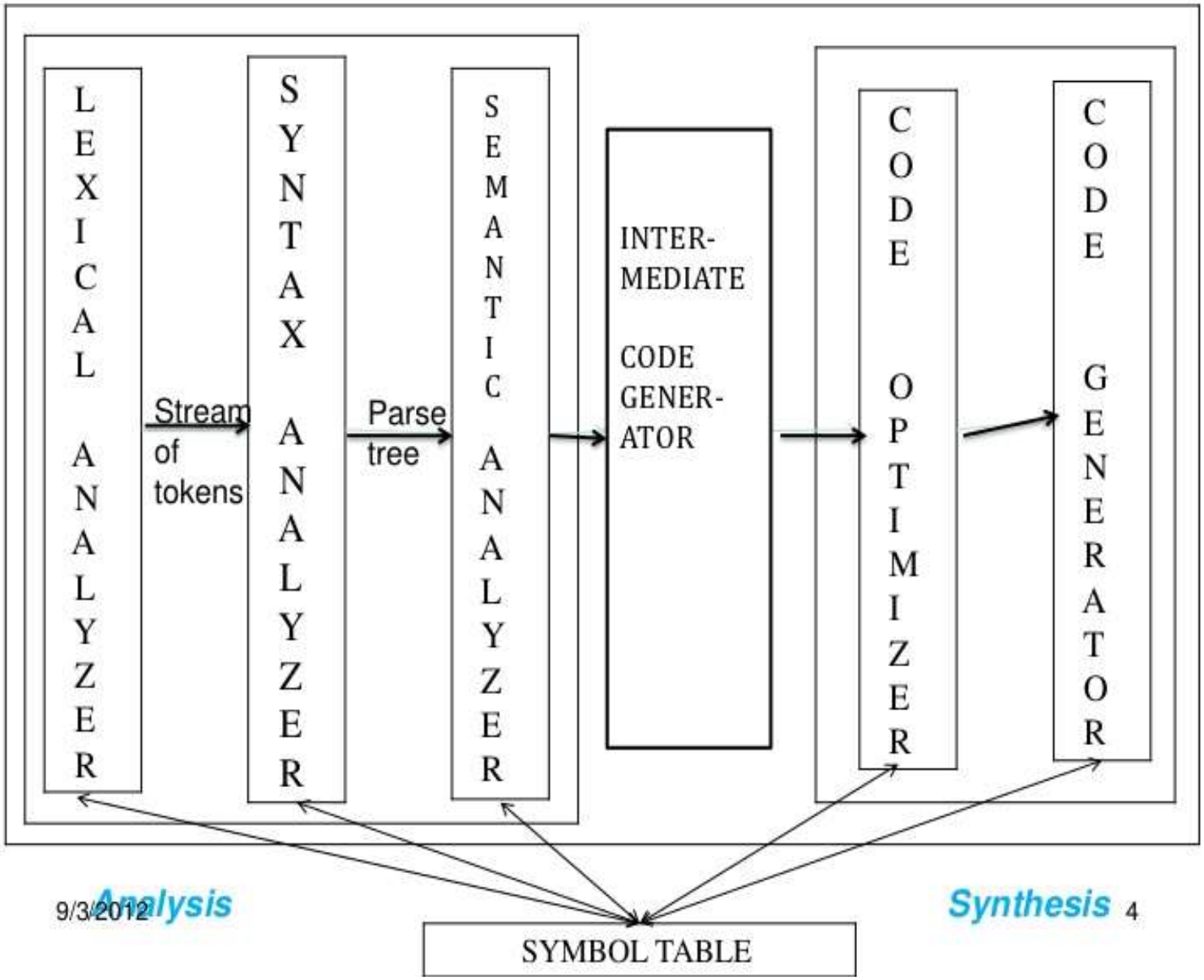
Symbol Table & Hash Table

- Compiler is composed of Analysis and synthesis

Compiler

Analysis

synthesis


LEXICAL ANALYZER
Stream of tokens
SYNTAX ANALYZER
Parse tree
SEMANTIC ANALYZER
INTER-MEDIATE CODE GENER-ATOR
CODE OPTIMIZER
CODE GENERATOR
SYMBOL TABLE
Analysis
Synthesis

Analysis

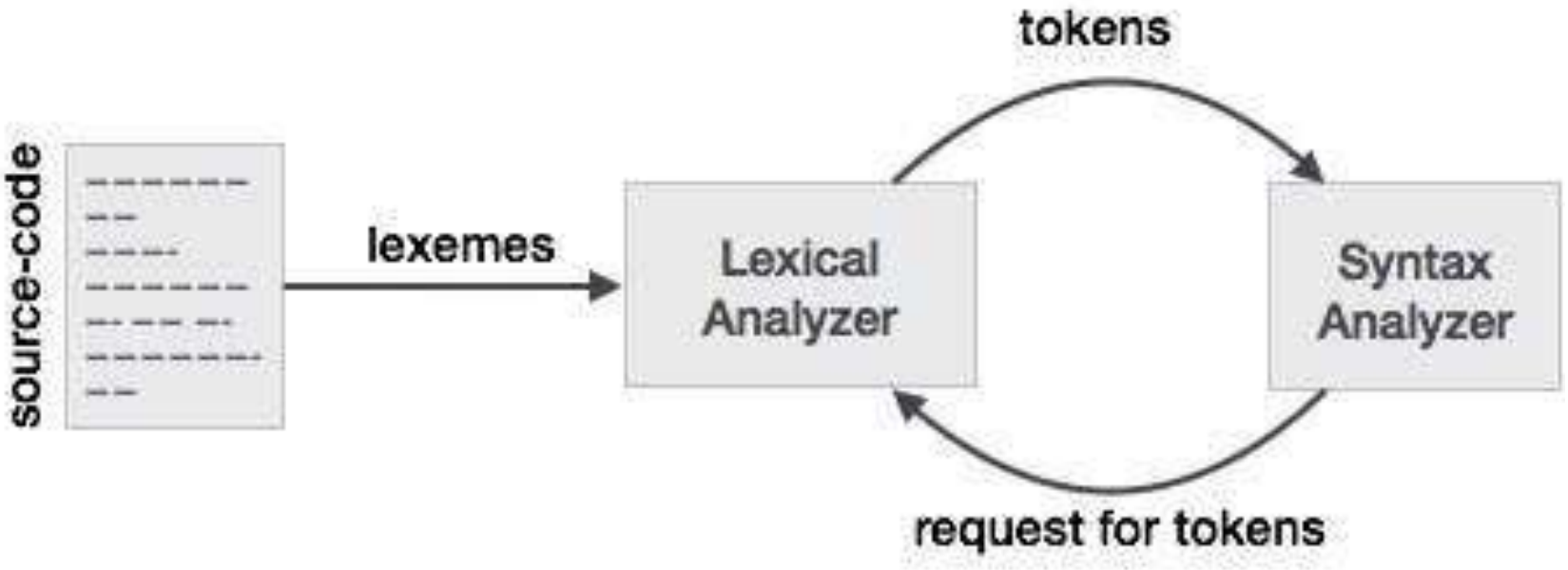
- Lexical Analysis (Stream of Tokens)
- Syntax Analyzer
- Semantic Analyzer
- It does the grammar check create intermediate representation of program
- Syntax ,syntactically correct

- Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences.
- The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.
  - The input string of characters of the source program are divided into sequence of characters that have a logical meaning, and are known as tokens
  - In programming language, keywords, constants, identifiers, strings, numbers, operators and punctuations symbols can be considered as tokens.
  - int value = 100; contains the tokens:

    int (keyword), value (identifier), = (operator),
    100 (constant) and ; (symbol)

- Optimization is a program transformation technique, which tries to improve the code by making it consume less resources (i.e. CPU, Memory) and deliver high speed.

- The final phase of the compiler is the generation of target code, consisting normally of relocatable machine code or assembly code



Lexemes are said to be a sequence of characters (alphanumeric) in a token.

- The lexical analyzer also follows **rule priority** where a reserved word, e.g., a keyword, of a language is given priority over user input. That is, if the lexical analyzer finds a lexeme that matches with any existing reserved word, it should generate an error.
- it involves grouping the tokens of the source program into grammatical phrases that are used by the compiler to synthesize output

- **The semantic analysis phase checks the source program for semantic errors and gathers type information for the subsequent de-generation phase.**

It uses the hierarchical structure determined by the syntax-analysis phase to identify the operators and operands of expressions and statements.

- **An important component of semantic analysis is type checking.**

COMPILERS ROLE??

- An essential function of a compiler –

Record the variable names used in the source program and collect information about various attributes of each name.

- These attributes may provide information about the storage allocated for a name , its type and its scope , procedure names ,number and types of its arguments, the method of passing each argument and the type returned

- **Symbol table** is a data structure used by a language translator such as a compiler or interpreter, where each identifier in a program's source code is associated with information relating to its declaration or appearance in the source, such as its type, scope level and sometimes its location.

- **A data structure called a symbol table is generally** used **to store information about various source language constructs.**

- **The information is collected by the analysis phases of the compiler and used by the synthesis phases to generate the target code**

- A common implementation technique is to use a hash table

- The symbol table is accessed by most phases of a compiler, beginning with the lexical analysis to optimization.

- At that time of accessing variables and allocating memory dynamically, a compiler should perform many works and as such the extended stack model requires the **symbol table**.

- Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces, etc.

- Symbol table is used by both the analysis and the synthesis parts of a compiler.

- Consider the following program written in C

```
// Declare an external function

extern double Abc(double x);
// Define a public function
double fun (int count)
{ double sum = 0.0 ;
// Sum all the values bar(1) to bar(count)
for (int i = 1; i <= count; i++) sum += bar((double) i);
   return sum;
}
```

Symbol name	Type	Scope
Abc	function, double	extern
x	double	function parameter
fun	function, double	global
count	int	function parameter
sum	double	block local
i	int	for-loop statement

In addition to this the symbol table also has dynamic entries suppose in for loop value of any variable ‘i’ changes it is always reflected in table

Scope Rule of Symbol Table

- The rules governing the scope of names in a block-structured language are as follows
    1. A name declared within a block B is valid only within B.
    2. If block B1 is nested within B2, then any name that any name that is valid for B2 is also valid for B1,unless the identifier for that name is re-declared in B1.
- The scope rules required a more complicated symbol table organization than simply a list association between names and attributes.
- Each table is list names and there associated attributes and the tables are organized into a stack.

Operations on Symbol table

- A symbol table, either linear or hash, should provide the following operations.
- **insert()**
- This operation is more frequently used by analysis phase, i.e., the first half of the compiler where tokens are identified and names are stored in the table. This operation is used to add information in the symbol table about unique names occurring in the source code

- **Scope Management :-** A compiler maintains two types of symbol tables: a **global symbol table** which can be accessed by all the procedures and **scope symbol tables** that are created for each scope in the program.

- **lookup()**
- lookup() operation is used to search a name in the symbol table to determine:
- if the symbol exists in the table.
- if it is declared before it is being used.
- if the name is used in the scope.
- if the symbol is initialized.
- if the symbol declared multiple times.

Hash Table

- a **hash table**, or a **hash map**, is a data structure that associates keys with values (attributes).
- A hash table generalizes the simpler notion of an ordinary array.
- The array index is *computed* from the key.
- Directly addressing into an ordinary array makes effective use of our ability to examine an arbitrary position in an array in O(1) time.
- Used for
  - Look-Up Table
  - Dictionary
  - Cache
  - Extended Array

HASHING

- Key-value pairs are stored in a fixed size table called a *hash table*.
  - A hash table is partitioned into many ***buckets***.
  - Each bucket has many ***slots***.
  - Each slot holds ***one record.***
  - A hash function f(x) transforms the identifier (key) into an address in the hash table

- We can insert element in hash table using direct addressing mode
- In direct addressing mode we calculate the slots in bucket
- Where we can store data
- The problem with direct addressing mode is
  - When bucket get completely filled we cannot insert elements
  - There might be situation where same slot is assign to two values or elements

- For this we have a term called as perfect hash function
- Where we calculate **load factor and load density**
- **Let n be the total number of elements in table**
- **T is the total number of possible elements**
- **Density is n/t load factor is alpha=n/sb**
- **S is slot in hash table**

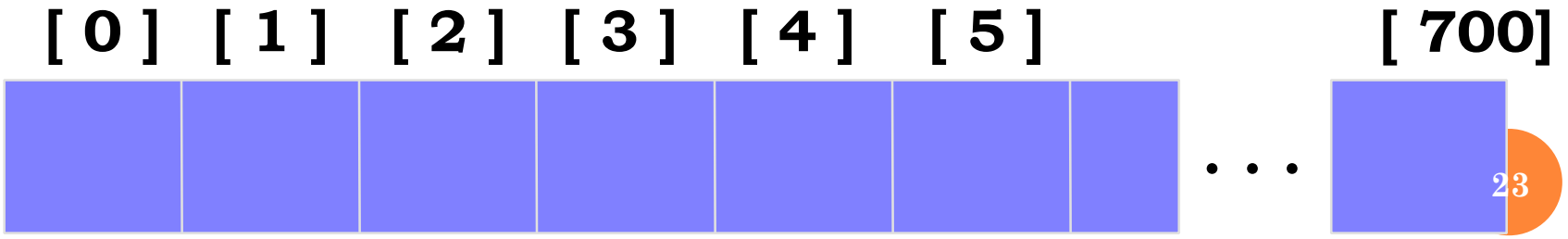
m slots

1			
….			
B-1			

B Buckets

What is a Hash Table ?

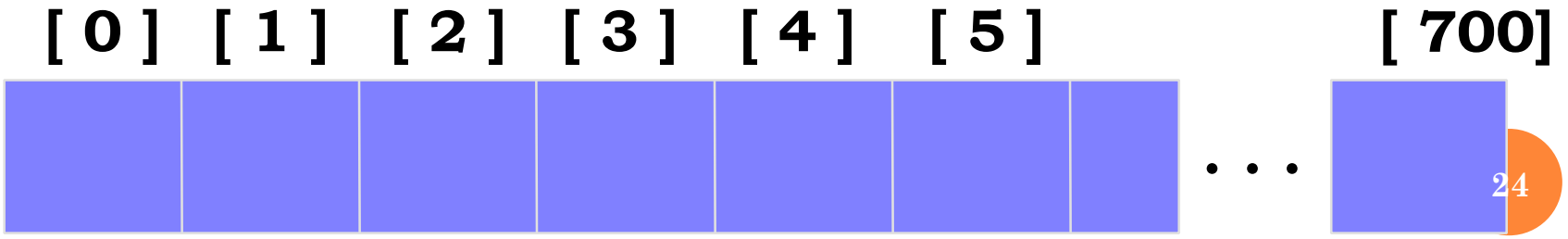
- The simplest kind of hash table is an array of records.
- This example has 701 records.



An array of records

What is a Hash Table ?

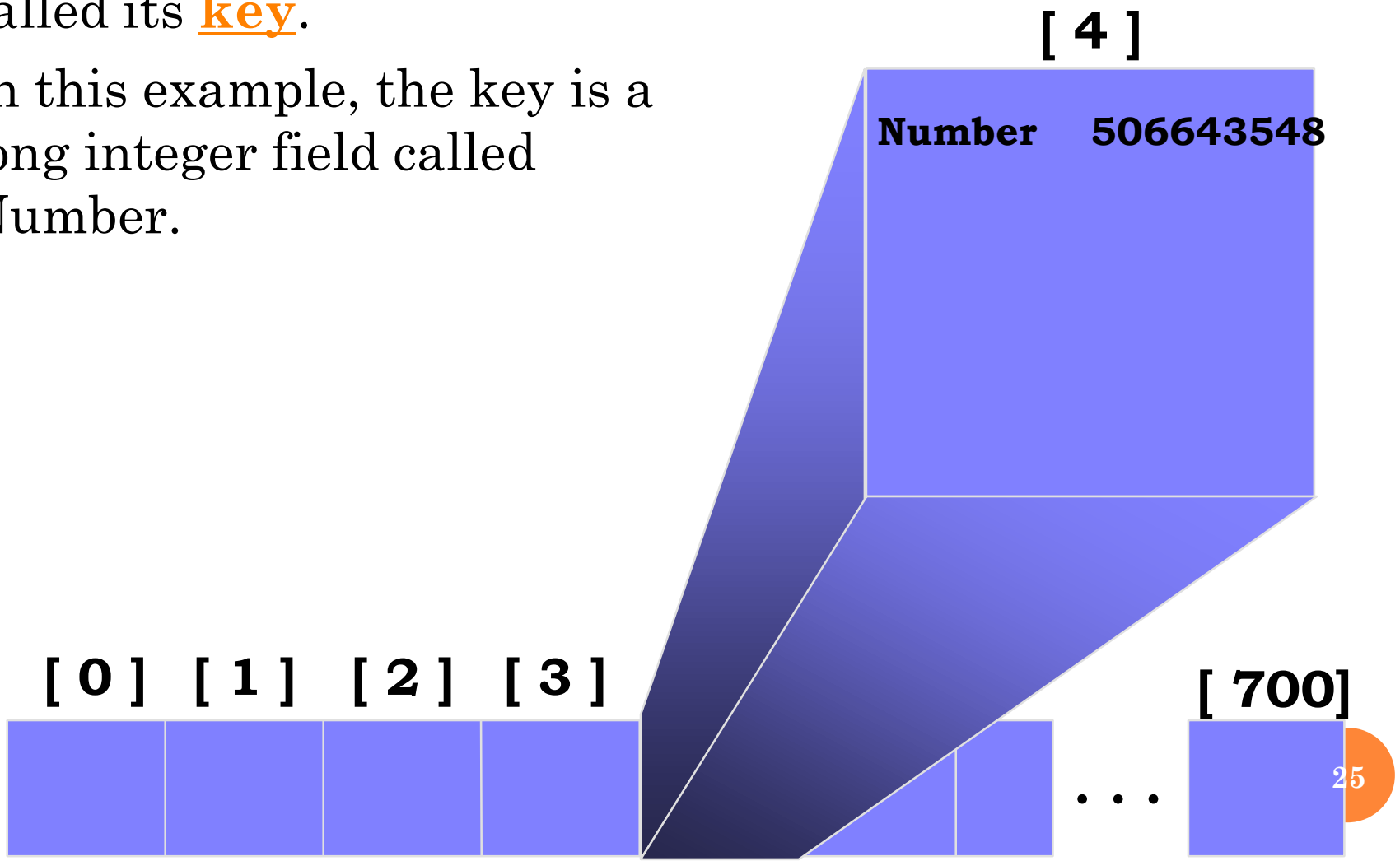
- The simplest kind of hash table is an array of records.
- This example has 701 records.



An array of records

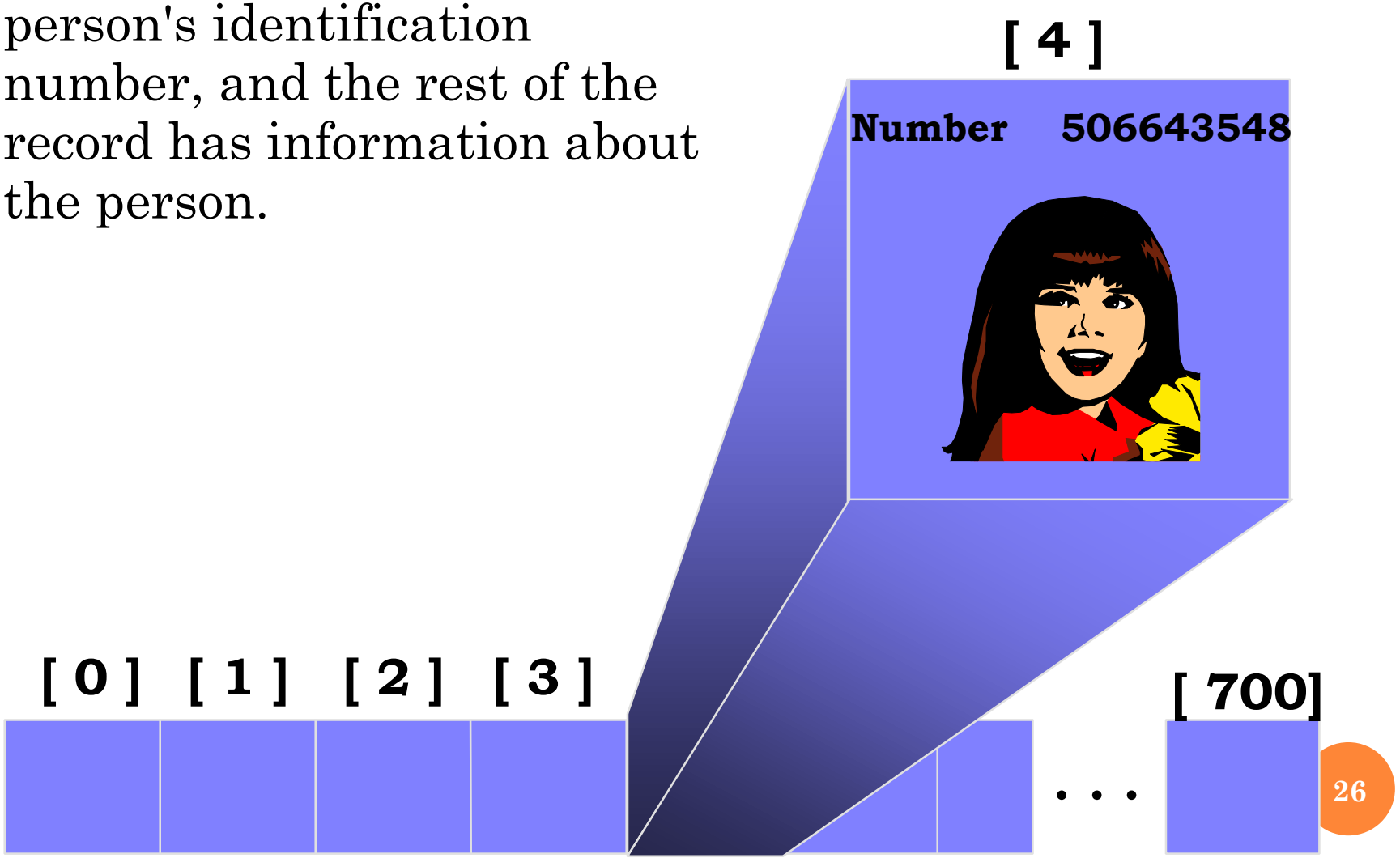
What is a Hash Table ?

- Each record has a special field, called its **key**.
- In this example, the key is a long integer field called Number.



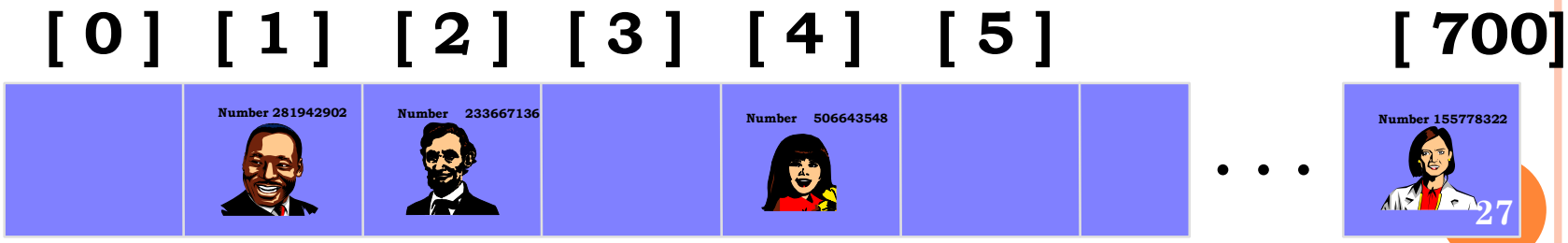
What is a Hash Table ?

- The number might be a person's identification number, and the rest of the record has information about the person.



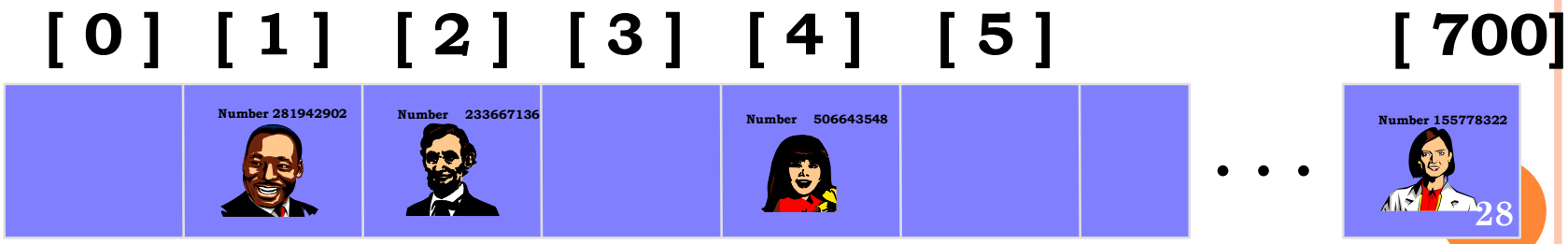
What is a Hash Table ?

- When a hash table is in use, some spots contain valid records, and other spots are "empty".



Inserting a New Record

- In order to insert a new record, the **key** must somehow be **converted to** an array **index**.
- The index is called the **hash value** of the key.

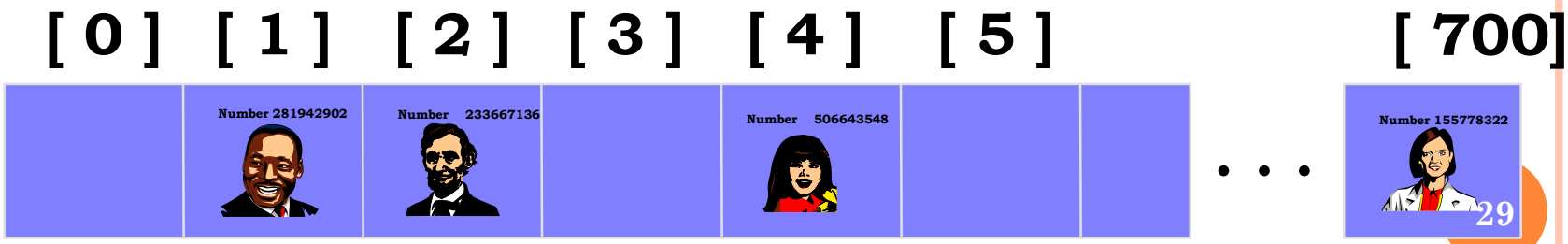


Inserting a New Record

- Typical way create a hash value:

(Number mod 701)

What is (580625685 mod 701) ?

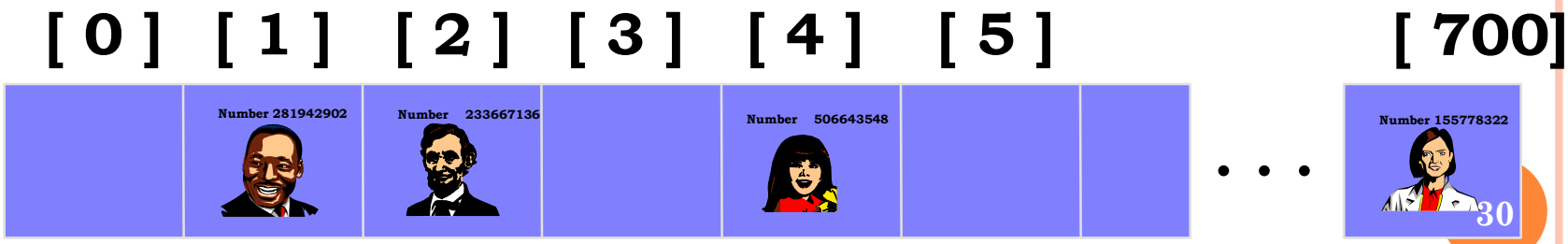


Inserting a New Record

- Typical way to create a hash value:

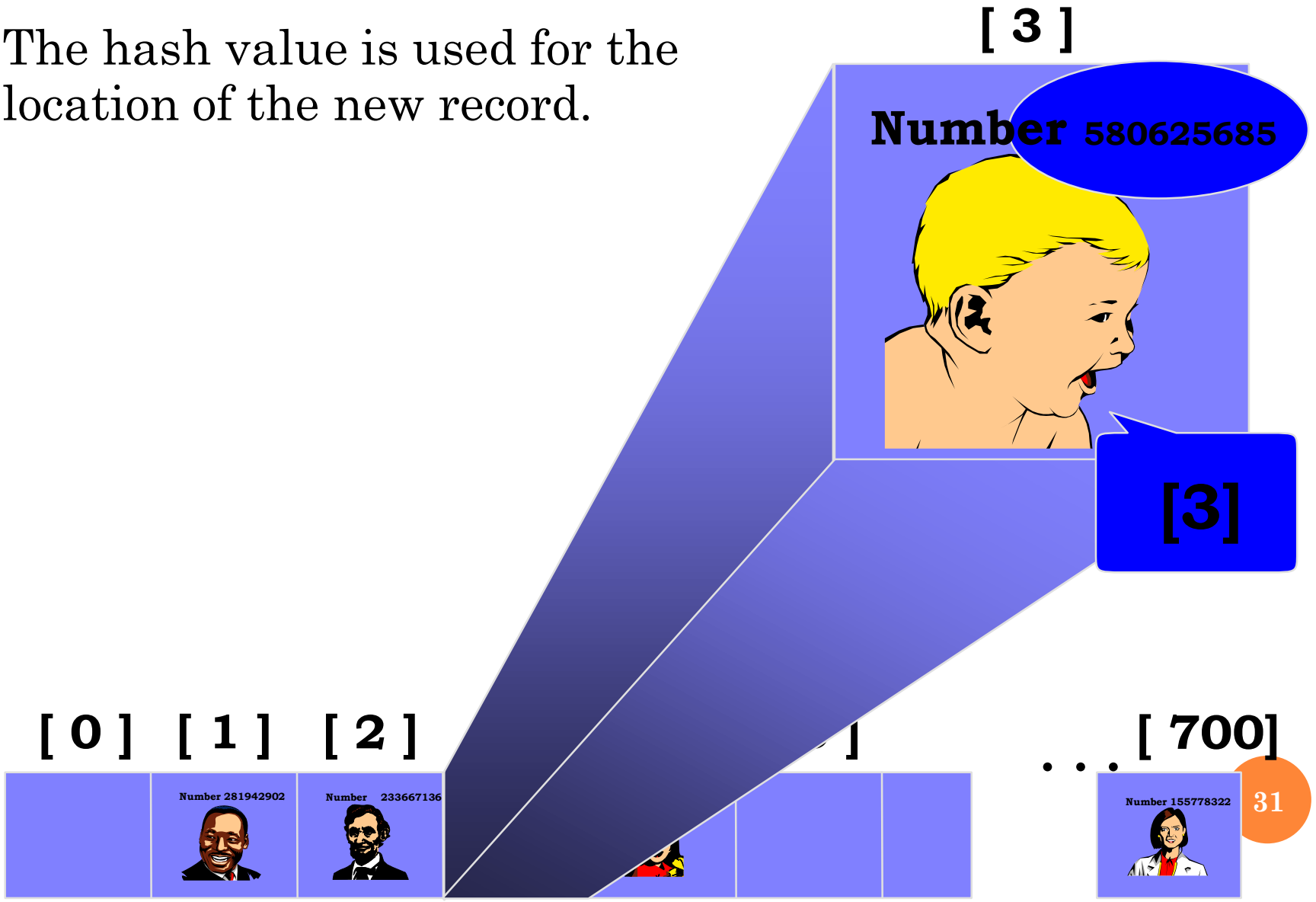
(Number mod 701)

What is (580625685 mod 701)?



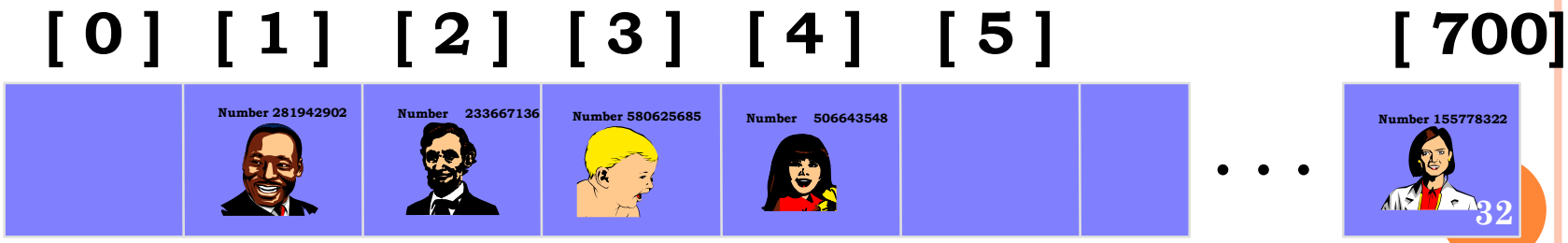
Inserting a New Record

- The hash value is used for the location of the new record.



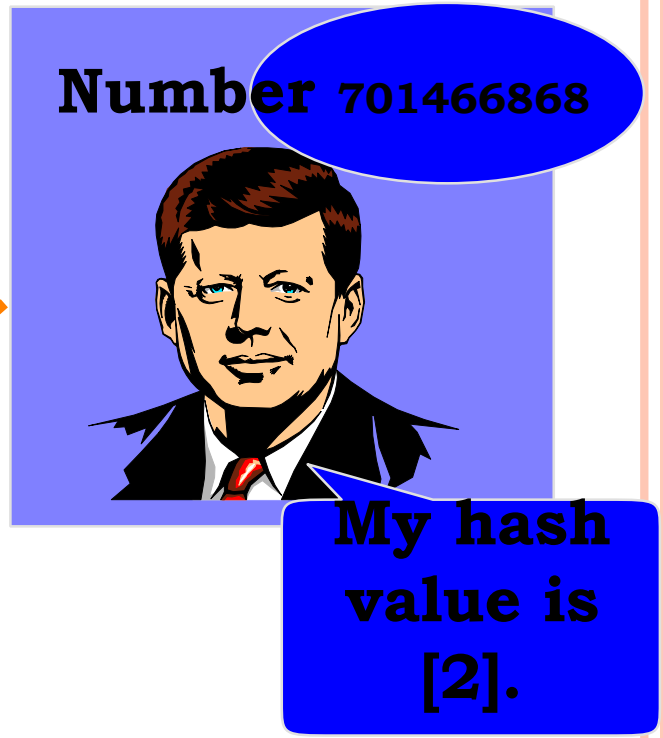
Inserting a New Record

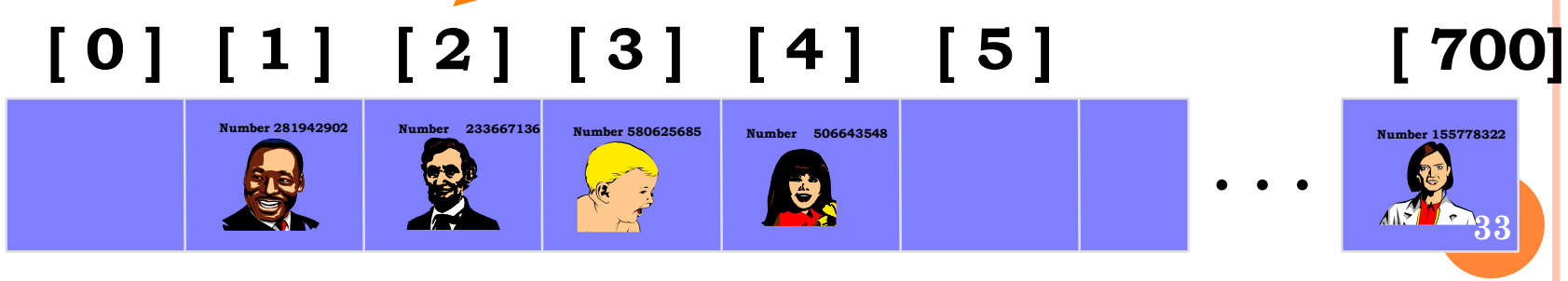
- The hash value is used for the location of the new record.



Collisions

- Here is another new record to insert, with a hash value of 2.





Collision Resolution Policies

- Two classes
  - **Open hashing**
    - In open hashing, keys are stored in linked lists attached to cells of a hash table. Each list contains all the keys hashed to its cell.
  - **Closed hashing**
    - After collisions records are stored somewhere in other empty slot in the table itself
      - Linear Probing
      - Quadratic Probing
      - Double Hashing

Linear Probing

- **Linear probing (linear hashing, sequential probing): f(i) = i**
- **Insert:** When there is a collision we just probe the next slot in the table. If it is unoccupied – we store the key there. If it is occupied – we continue probing the next slot
- One main disadvantage of linear probing is it forms a cluster such clustering increases time to search for a record

Consider an example where we want to insert elements.

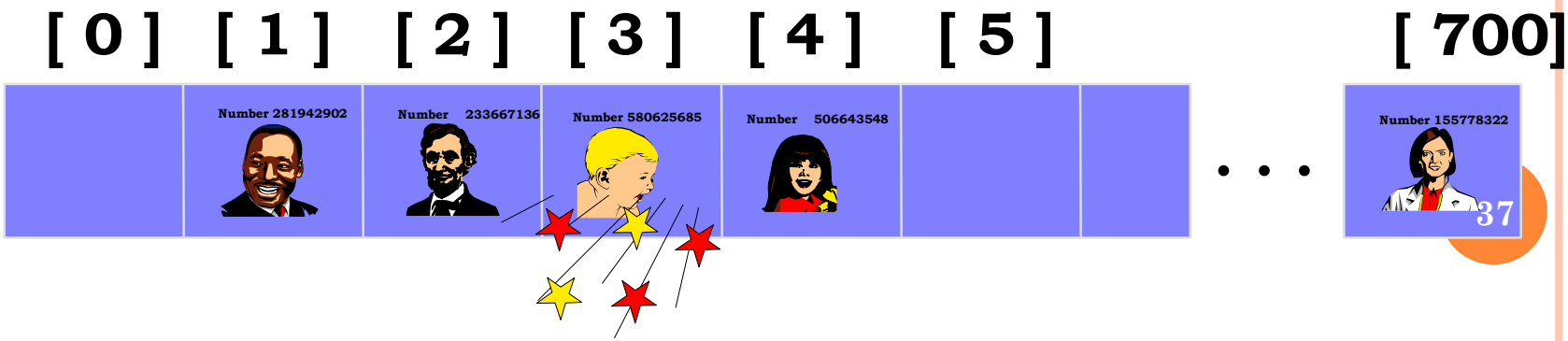
131, 22 , 31 , 11 , 121 , 61 , 7 , 8 and size of hash table is-10

Hash key is 131%10 = 1
22%10=2
31%10 = 1
Index 1 is Occupied

Index	Data
0	
1	131
2	22
3	**31**
4	**11**
5	**121**
6	**61**
7	7
8	8
9	
10	

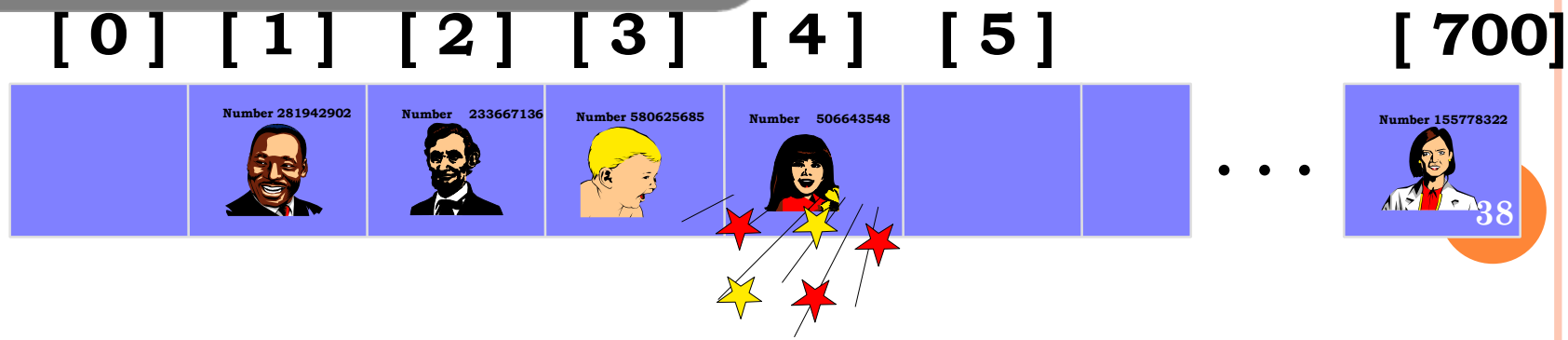
- This is called a **collision**, because there is already another valid record at [2].

When a collision occurs, move forward until you find an empty spot.



- This is called a **collision**, because there is already another valid record at [2].

When a collision occurs, move forward until you find an empty spot.

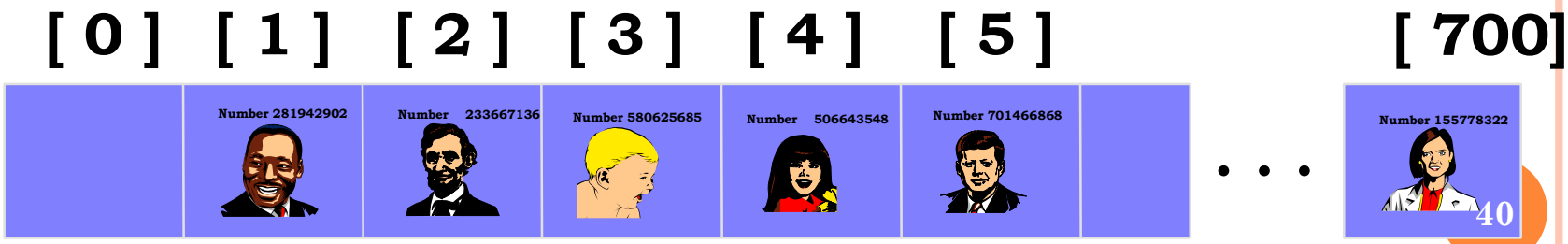


- This is called a collision, because there is already another valid record at [2].

When a collision occurs, move forward until you find an empty spot.

- This is called a **collision**, because there is already another valid record at [2].

The new record goes in the empty spot.



Quadratic Probing

- Quadratic Probing
- **Quadratic probing: $f(i) = i^2$**
- A quadratic function is used to compute the next index in the table to be probed.
- **Example:**
- In linear probing we check the i-th position. If it is occupied, we check the $i+1^{st}$ position, next we check the $i+2^{nd}$, etc.

- In quadric probing, if the i-th position is occupied we check the $i+1^{st}$, next we check the $i+4^{th}$, next - $i + 9^{th}$ etc.
- **The idea** here is to skip regions in the table with possible clusters.

Index	Data
0	90
1	**131**
2	22
3	33
4	
5	**121**
6	
7	7
8	8
9	
10	

Hi (K) = (H(K) + i2) % n

Here we can take values of I as 0,1,2,3…..

For 121 and i=0
121%10=1

Hi(K)=(121+0 Sq)%size → 1
Hi(K)=(121+1 Sq)%size → 2
Hi(K)=(121+2 Sq)%size → 5

Next insert 87 i=0,
Hi(K)=(87+0 Sq)%size →7
Hi(K)=(87+1 Sq)%size →8
Hi(K)=(87+2Sq)%size →1
Hi(K)=(87+3Sq)%size → 6

Inserting a New Record

- Division Method:- The hash function depends on the reminder of division typically the divisor is length of table (10) or Prime Number

- ***H(K)=K (mod)m***
- **4=54%10**
- **2=72%10**
- ***Using prime Number***
- ***Choose prime number close to 99***

so that table size can be increased i.e 97

- ***H(3205)=4***

Index	Value
0	
1	
2	72
3	
4	54
5	

Inserting a New Record

- Mid Square Method:- The key is Squared and then hash function is defined as $H(K)=l$ where l is obtained like

$K=3205 \quad K^2 10272025 \;, \; l=72$

- Folding Method :-the key K is partitioned into Number of parts where each parts has same number of digits except possibly the last

- $H(K)=K1+k2+K3\ldots\ldots Kn$
- $H(3205)= 32+05=37$
- *The record will be placed at 37 loc*

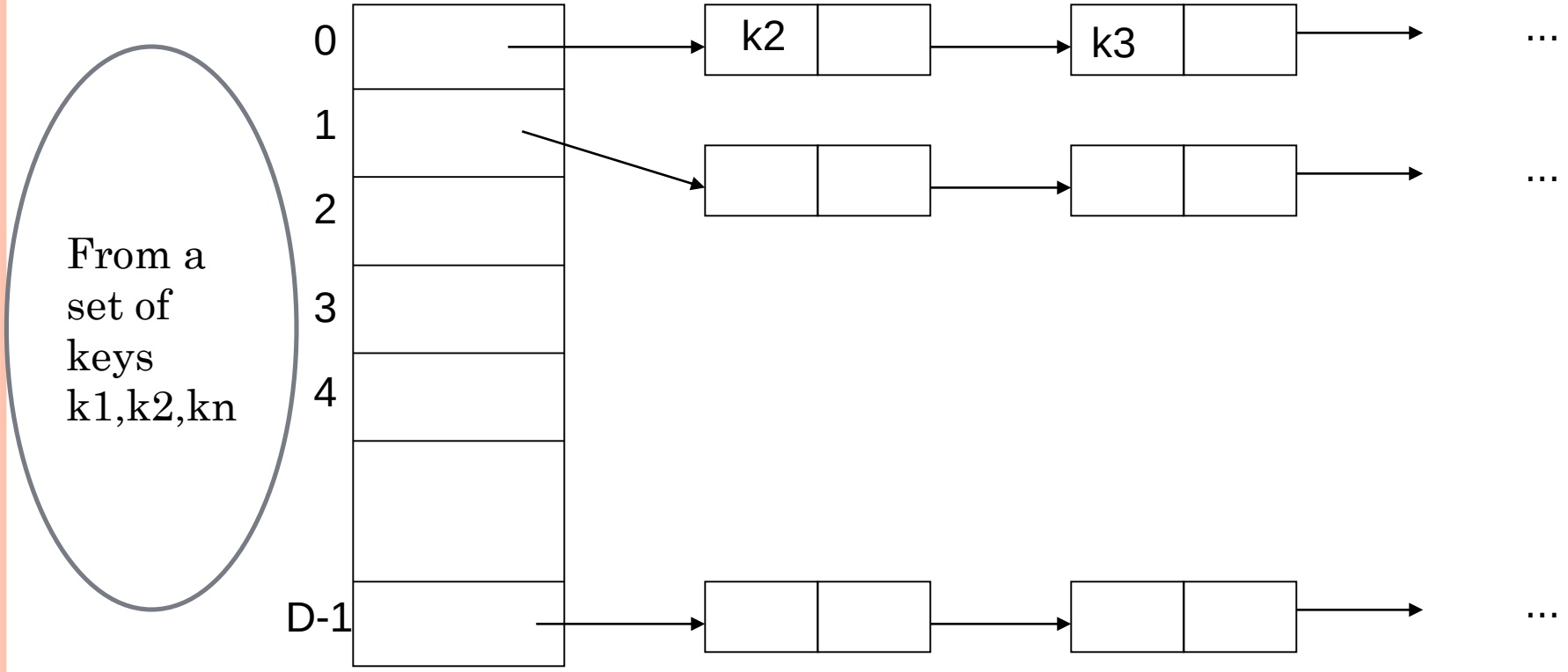
DOUBLE HASHING

- **Double-hash probing**
- A technique which eliminates both primary and secondary clustering is ***double-hashing.***
- The idea is to compute a second hash value of the element to be inserted.
- ***k1 = H(key % size of table(M) or prime no)*** We want 0 < k < capacity.
- Using this value, search this sequence of alternative locations:
- This second hash function needs to generate integers somewhere in the range from 1 (not 0) to capacity - 1.
- ***H(K2) hash2(obj) = 1+ (K1% M')*** where Max is prime number less than or equal to size of table or can be considered as less than 99

Open Hashing

- Each bucket in the hash table is the head of a linked list
- All elements that hash to a particular bucket are placed on that bucket's linked list
- Records within a bucket can be ordered in several ways
  - by order of insertion, by key value order

Open Hashing (Chaining)



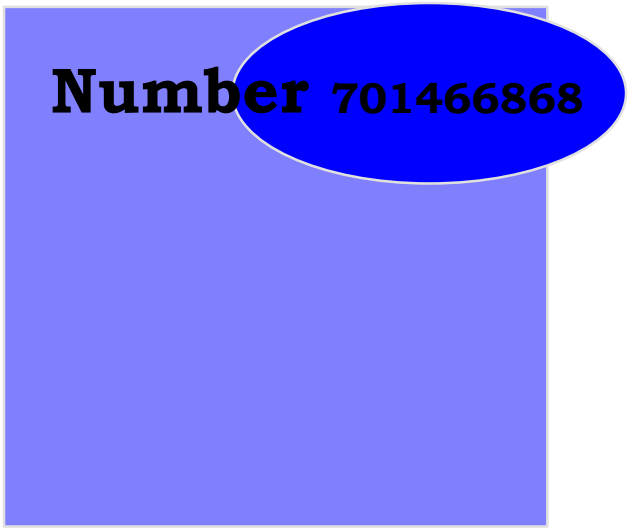
Chaining

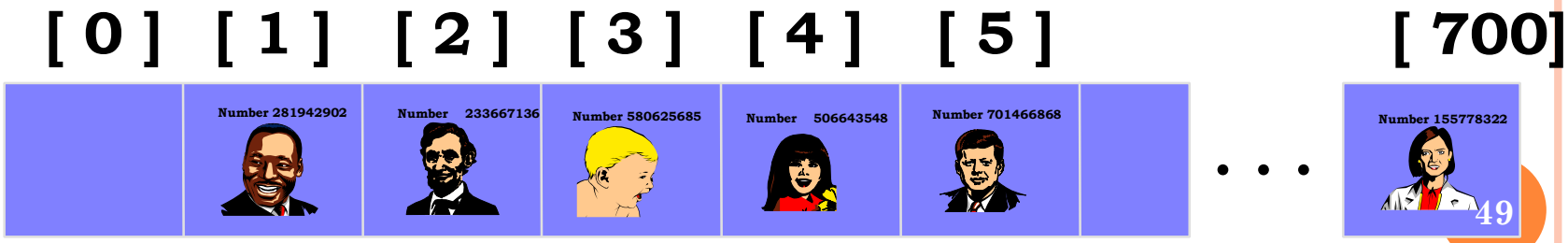
Chaining involves maintaining two table in memory chaining without replacement

Index	Data	Chain
0	-1	-1
1	131	2
2	21	5
3	3	-1
4	64	-1
5	61	7
6	66	-1
7	71	-1

Searching for a Key

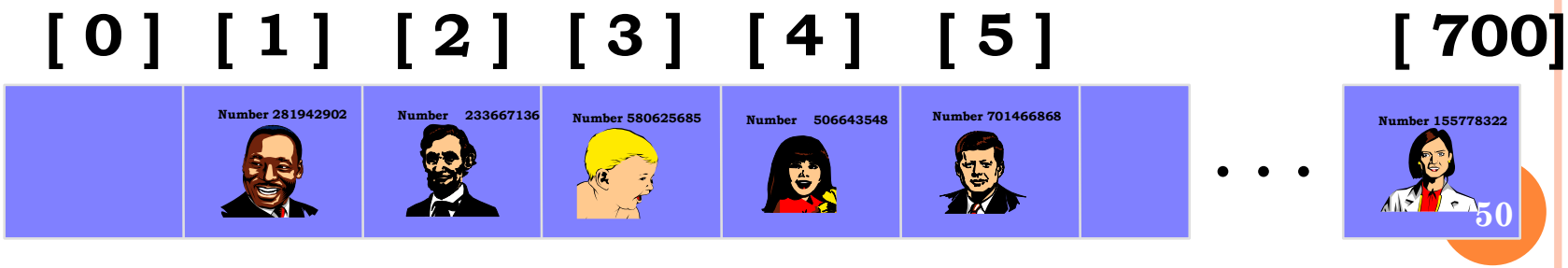
- The data that's attached to a key can be found fairly quickly.





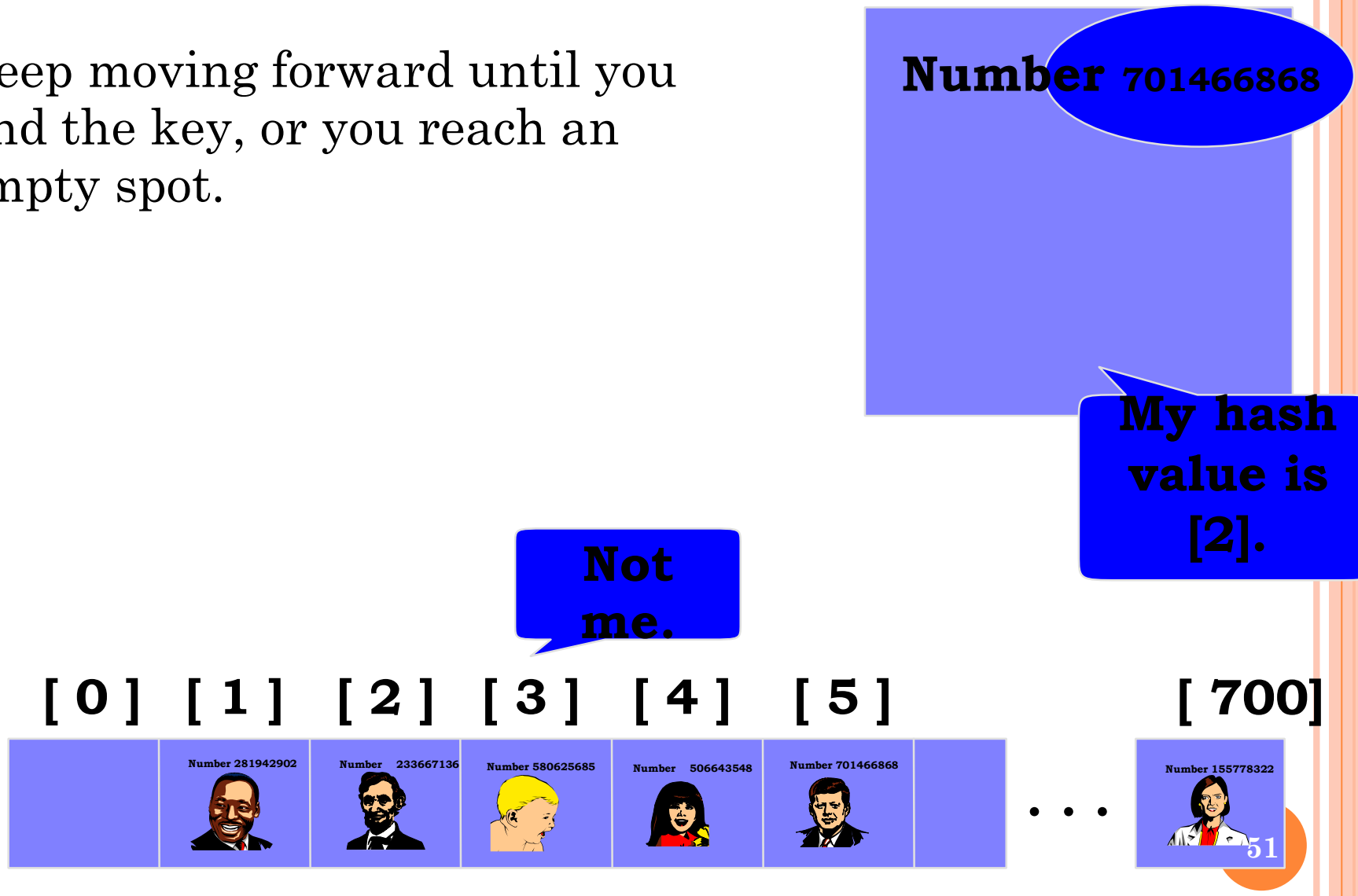
Searching for a Key

- Calculate the hash value.
- Check that location of the array for the key.
- Start by computing the hash value, which is 2 in this case. Then check location 2.
- If location 2 has a different key than the one you are looking for, then move forward…



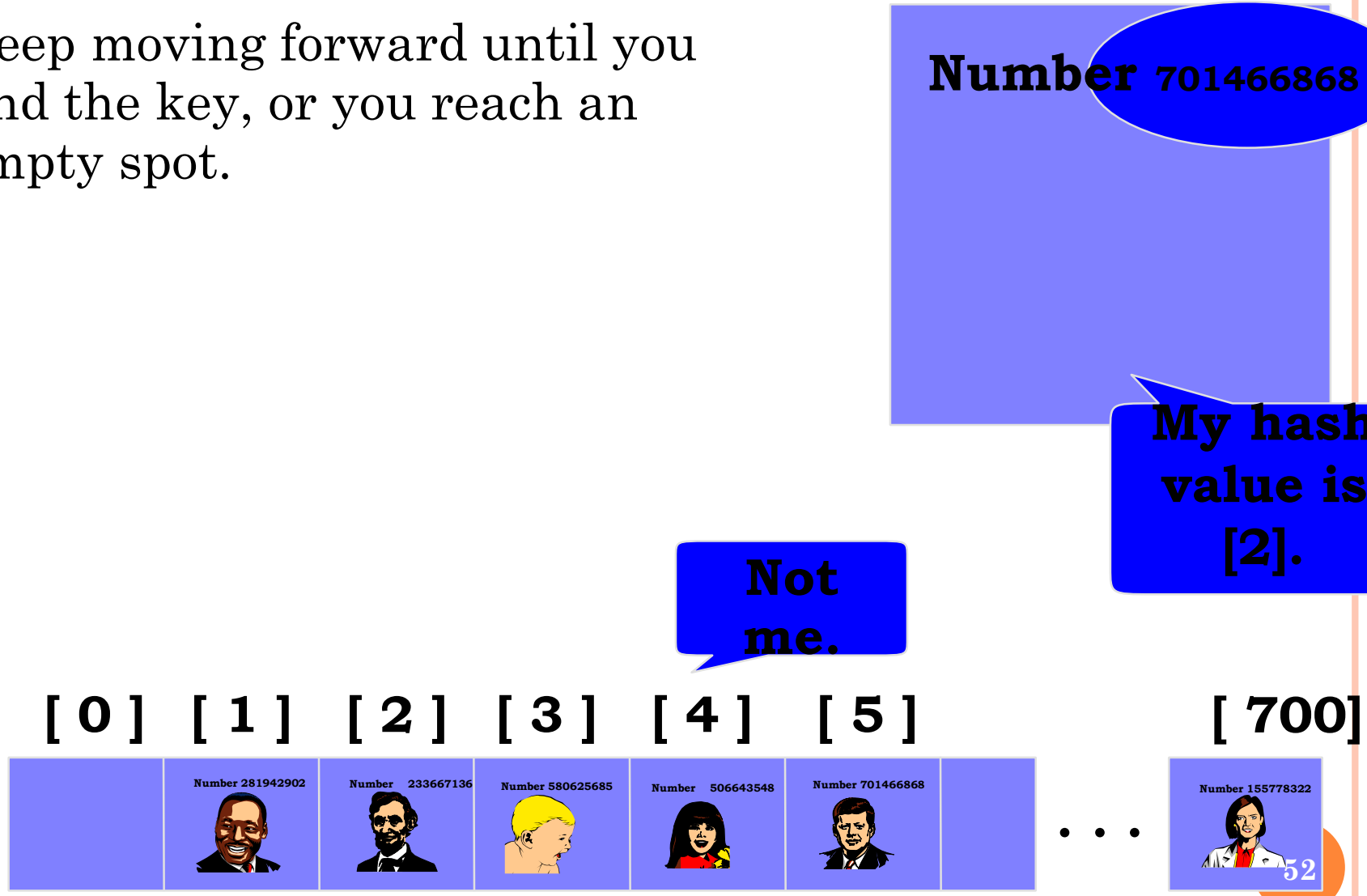
Searching for a Key

- Keep moving forward until you find the key, or you reach an empty spot.



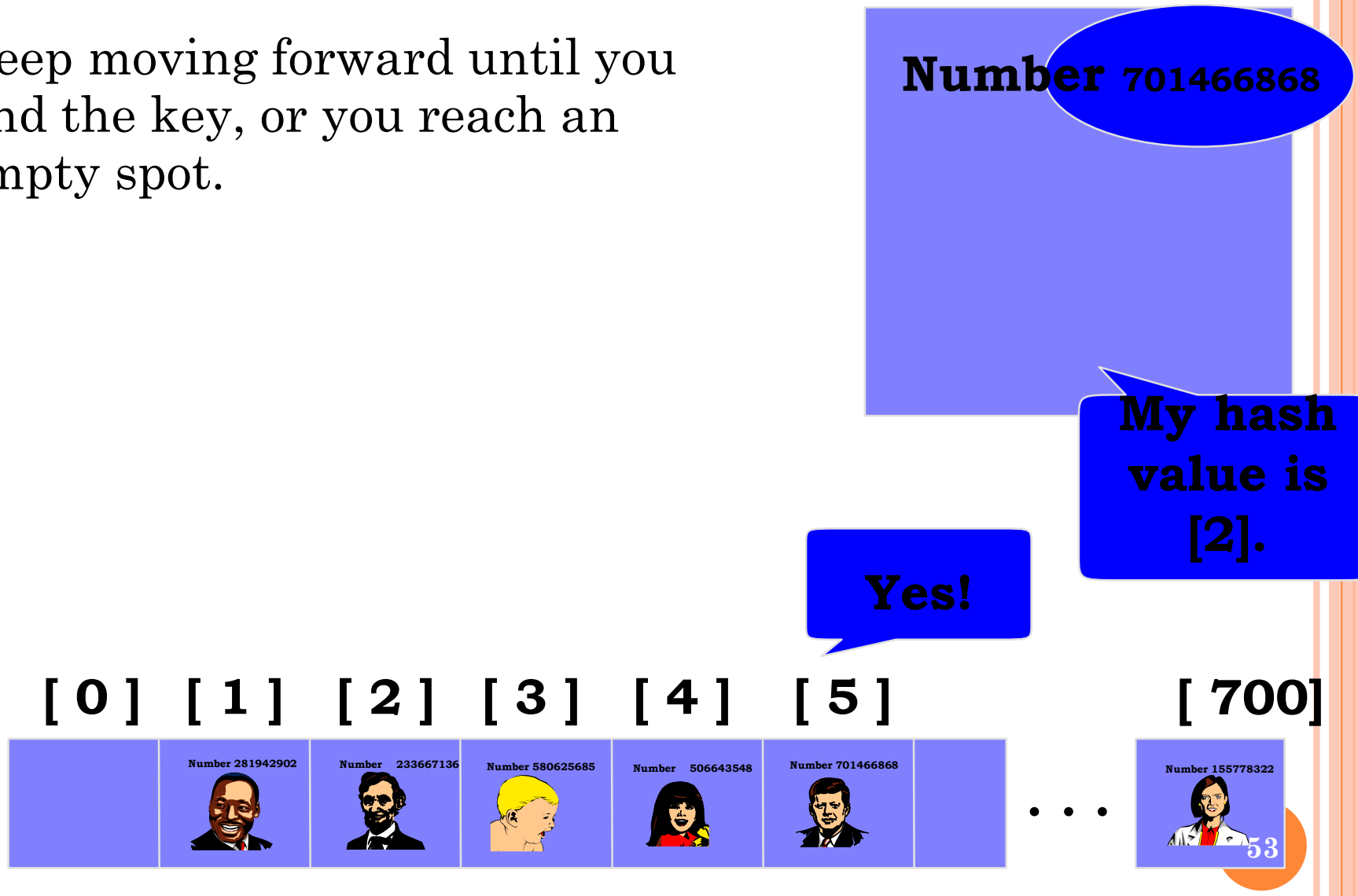
Searching for a Key

- Keep moving forward until you find the key, or you reach an empty spot.



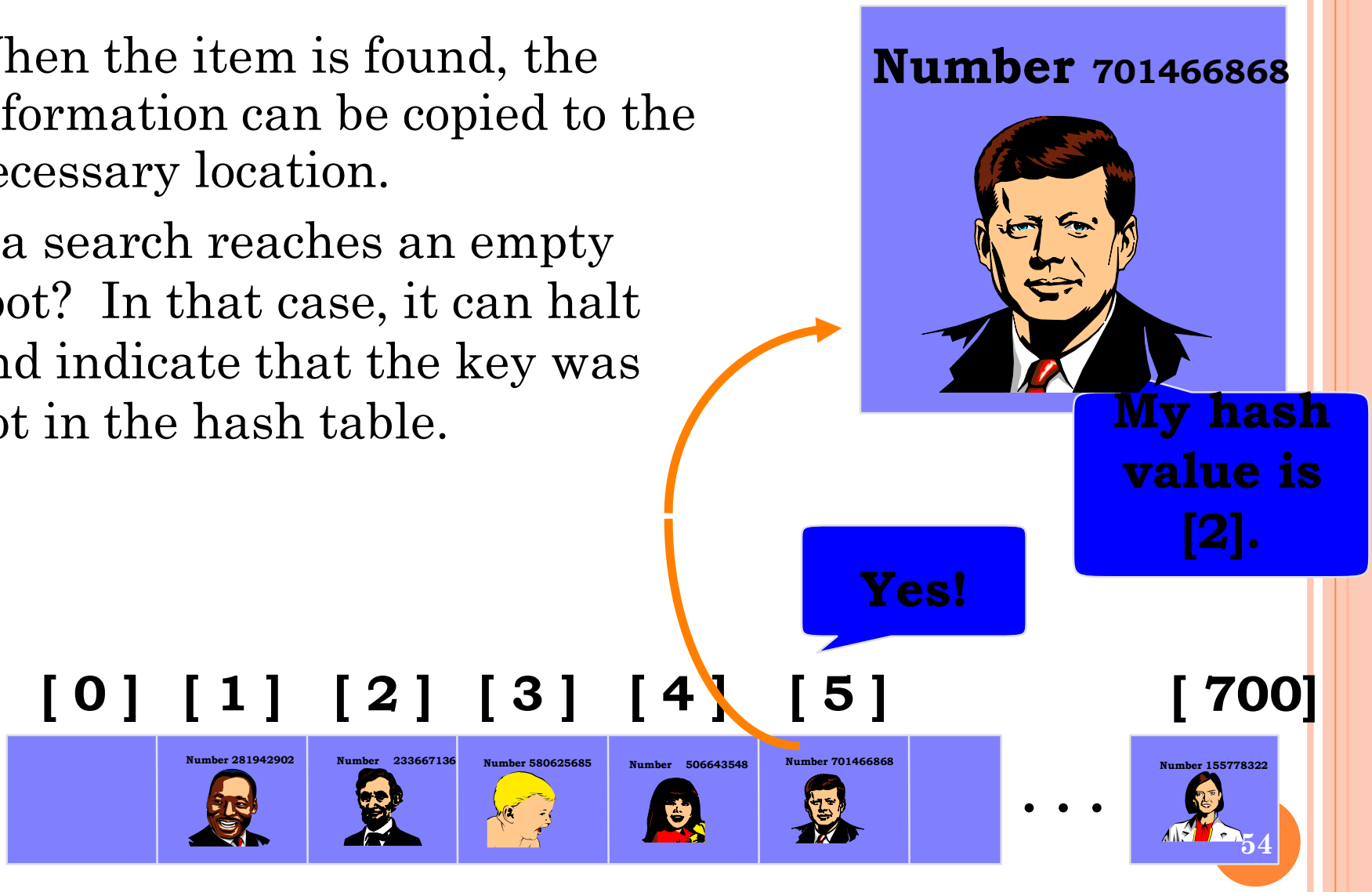
Searching for a Key

- Keep moving forward until you find the key, or you reach an empty spot.



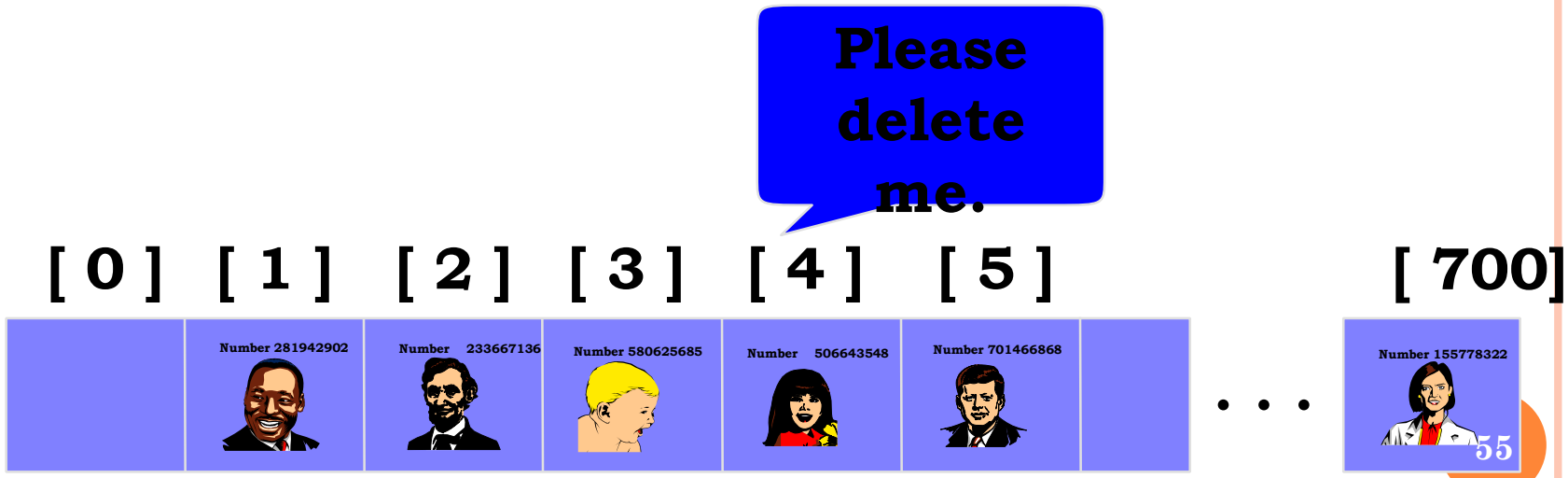
Searching for a Key

- When the item is found, the information can be copied to the necessary location.
- if a search reaches an empty spot? In that case, it can halt and indicate that the key was not in the hash table.



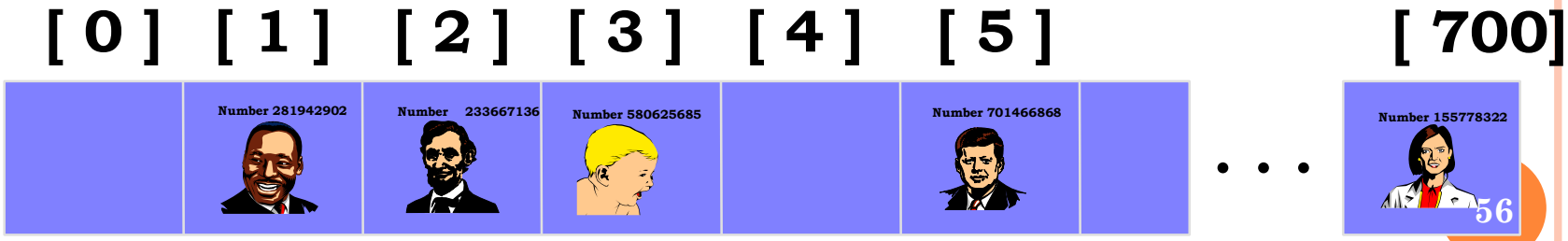
Deleting a Record

- Records may also be deleted from a hash table.



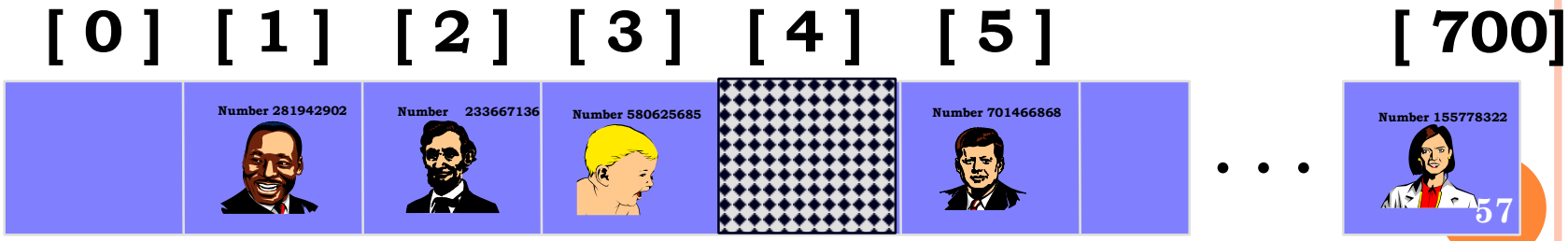
Deleting a Record

- Records may also be deleted from a hash table.
- But the location must not be left as an ordinary "empty spot" since that could interfere with searches. .
- ***(Remember that a search can stop when it reaches an empty spot.)***



Deleting a Record

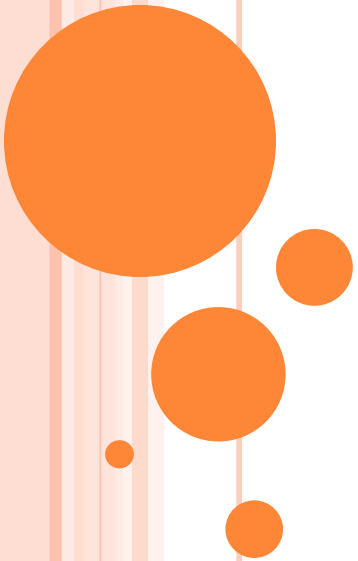
- Records may also be deleted from a hash table.
- But the location must not be left as an ordinary "empty spot" since that could interfere with searches.
- The location must be marked in some special way so that a search can tell that the spot used to have something in it.



- When a empty slot is encountered during insertion, that slot can be used to store the new record.

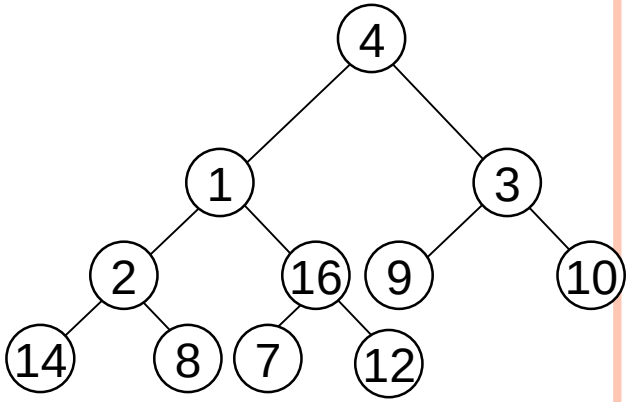
- To avoid inserting duplicate keys, it will still be necessary for the search procedure to follow the probe sequence until a truly empty position has been found

Heap Sort



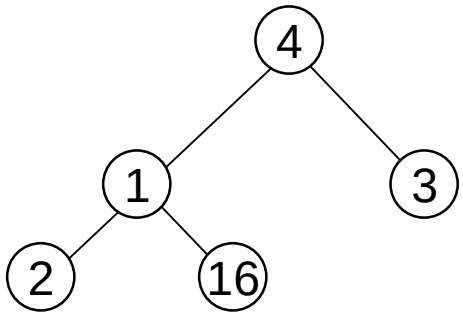
Special Types of Trees

- *Def:* **Full binary tree** = a binary tree in which each node is either a leaf or has degree exactly 2.



Full binary tree

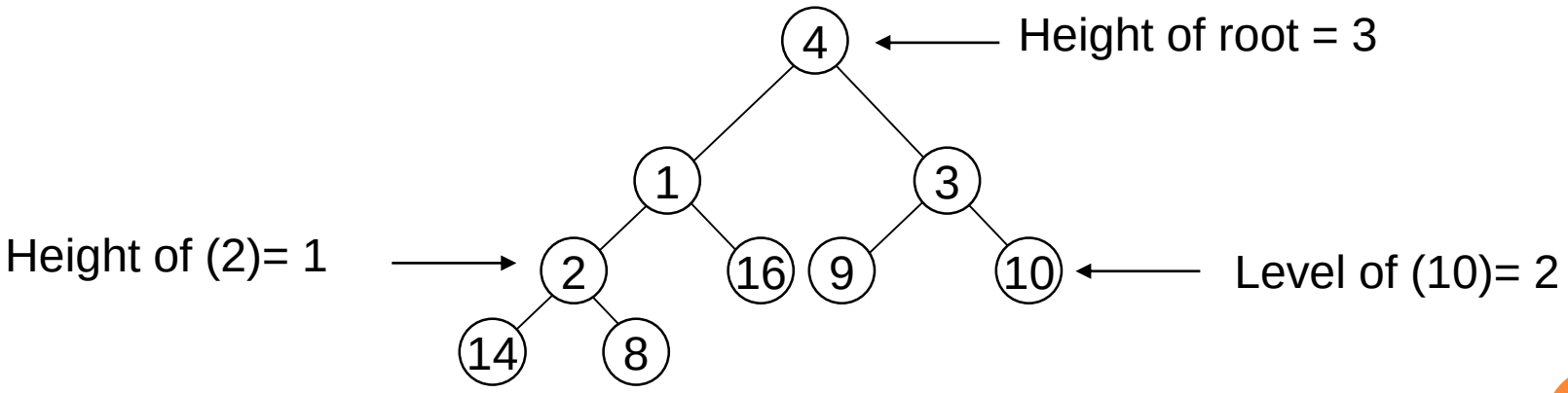
- *Def:* **Complete binary tree** = A binary tree T with n levels is Complete if all levels except possibly the last are completely full, and the last level has all its nodes to the left side.



Complete binary tree

Definitions

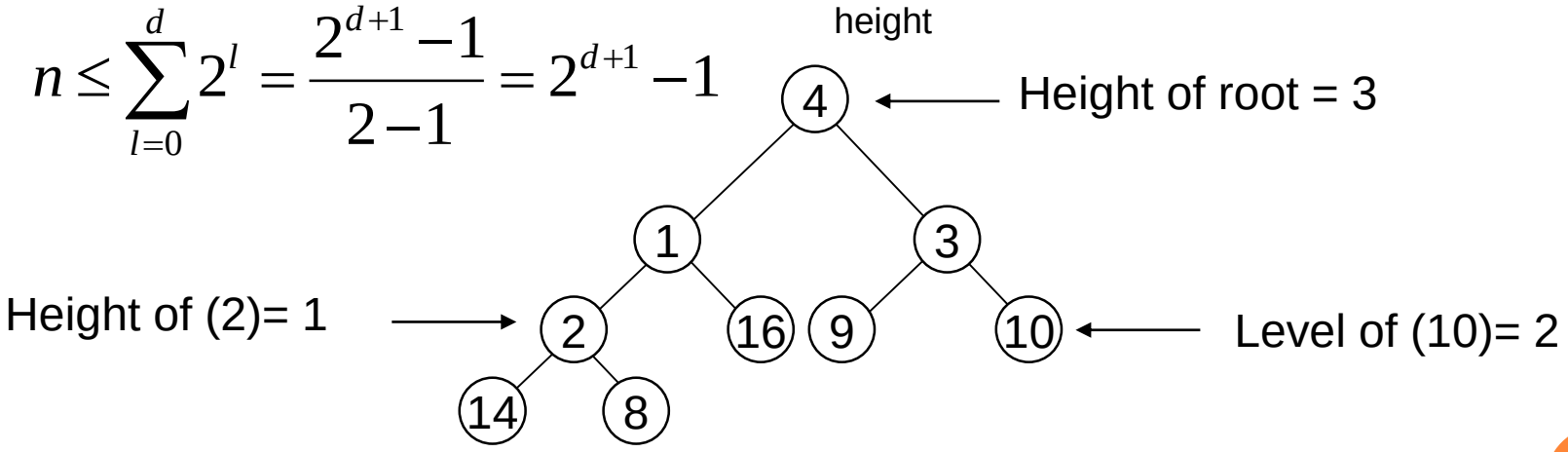
- **Height** of a node = the number of edges on the longest simple path from the node down to a leaf
- **Level** of a node = the length of a path from the root to the node
- **Height** of tree = height of root node



Useful Properties

- There are at most 2^l nodes at level l of binary tree
- A Binary tree with depth (height) d has at most 2^d+1-1 node

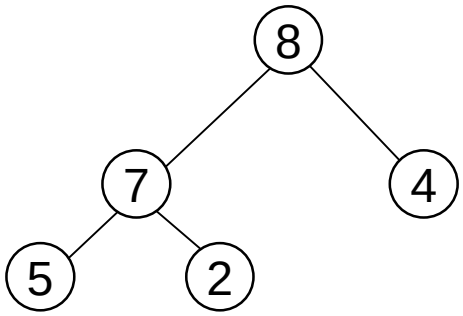
$$n \le \sum_{l=0}^{d} 2^l = \frac{2^{d+1}-1}{2-1} = 2^{d+1}-1$$



The Heap Data Structure

- *Def:* A **heap** is a nearly complete binary tree with the following two properties:
  - **Structural property:** all levels are full, except possibly the last one, which is filled from left to right
  - **Order (heap) property:** for any node x

$$Parent(x) \geq x$$



Heap

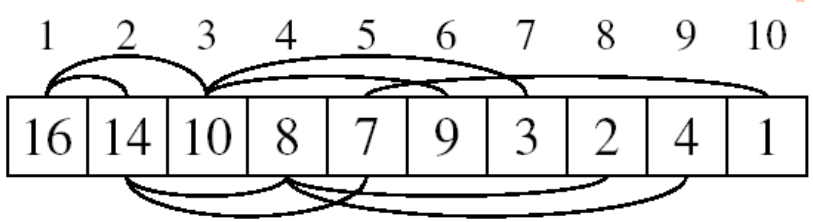
From the heap property, it follows that:

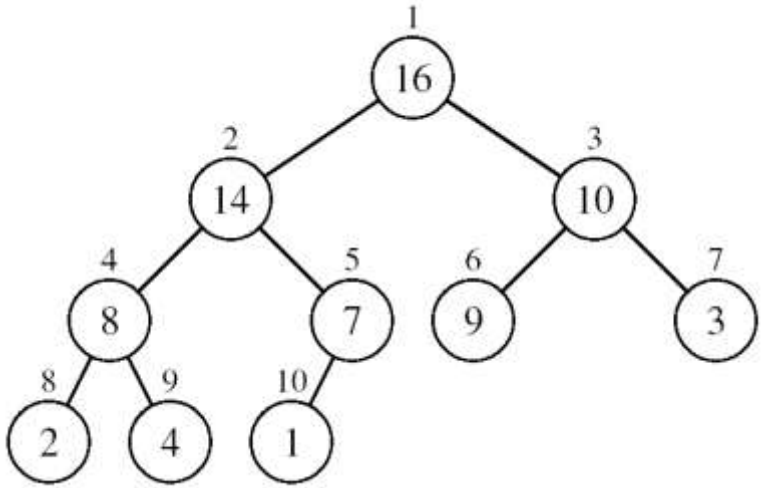
"The root is the maximum element of the heap!"

A heap is a binary tree that is filled in order

Array Representation of Heaps

- A heap can be stored as an array *A*.
  - Root of tree is A[1]
  - Left child of A[i] = A[2i]
  - Right child of A[i] = A[2i + 1]
  - Parent of A[i] = A[$\lfloor i/2 \rfloor$]
  - Heapsize[A] ≤ length[A]
- The elements in the subarray A[($\lfloor n/2 \rfloor$+1) .. n] are leaves.
- Here n=10 so 6 to 10 are leaves.





Heap Types

- **Max-heaps** (largest element at root), have the *max-heap property:*
  - for all nodes i, excluding the root:

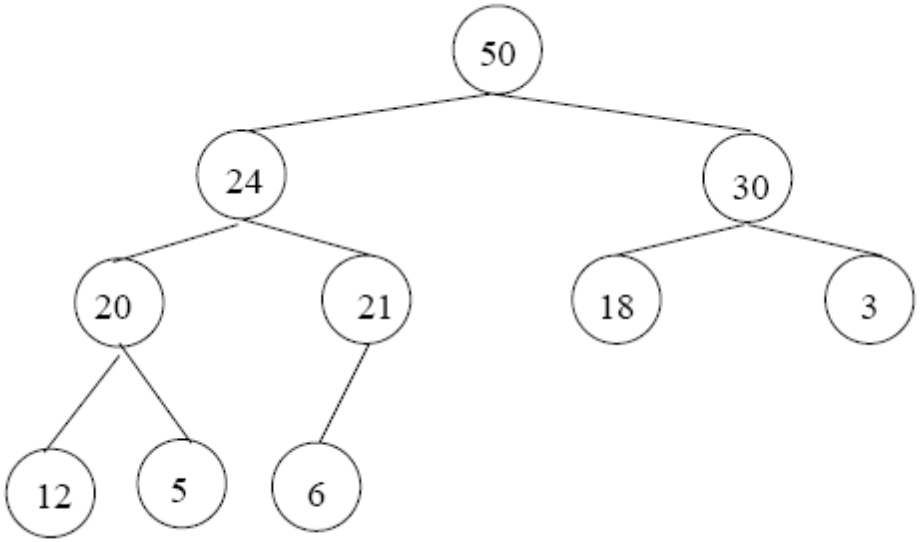
$$A[PARENT(i)] \geq A[i]$$

- **Min-heaps** (smallest element at root), have the *min-heap property:*
  - for all nodes i, excluding the root:

$$A[PARENT(i)] \leq A[i]$$

Adding/Deleting Nodes

- New nodes are always inserted at the bottom level (left to right)
- Nodes are removed from the bottom level (right to left)

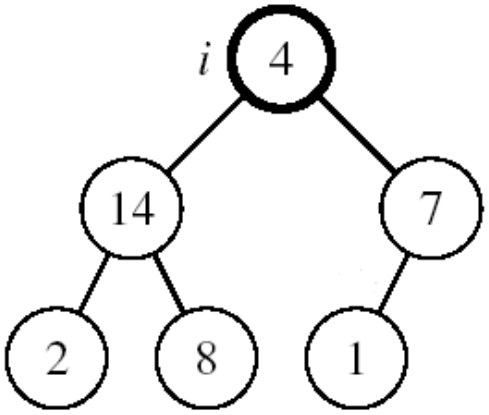


Operations on Heaps

- Maintain/Restore the max-heap property
  - MAX-HEAPIFY
- Create a max-heap from an unordered array
  - BUILD-MAX-HEAP
- Sort an array in place
  - HEAPSORT

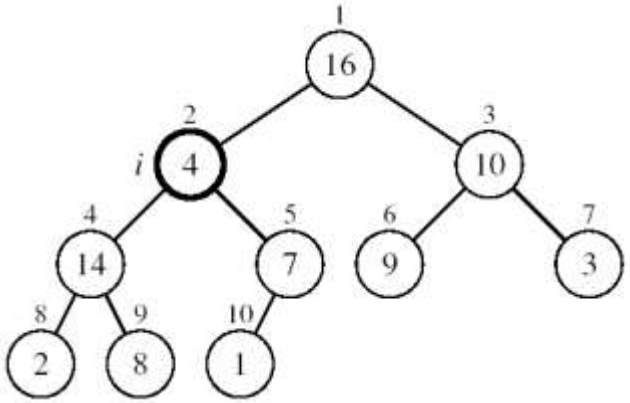
Maintaining the Heap Property

- Suppose a node is smaller than a child
  - Left and Right subtrees of i are max-heaps
- To eliminate the violation:
  - Exchange with larger child
  - Move down the tree
  - Continue until node is not smaller than children



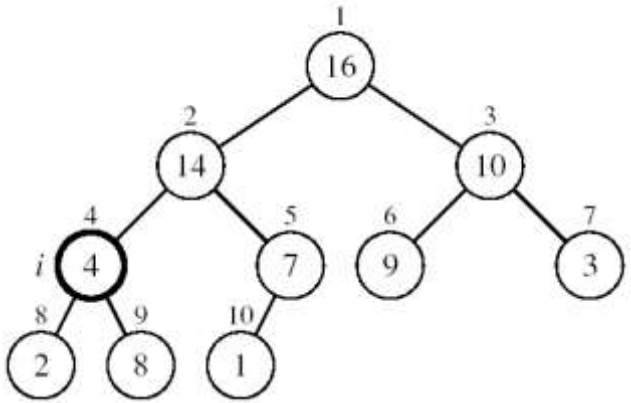
EXAMPLE

MAX-HEAPIFY(A, 2, 10)



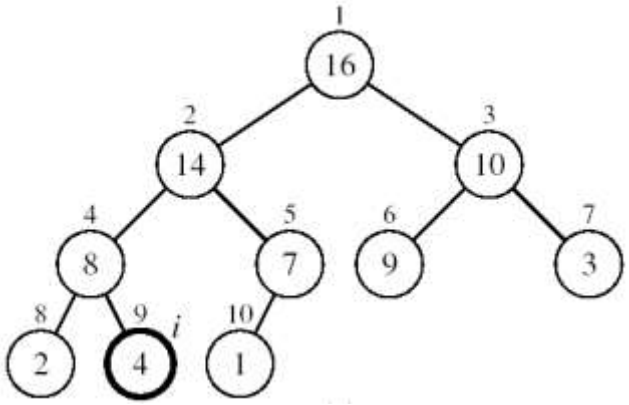
A[2] violates the heap property

A[2] ↔ A[4]



A[4] violates the heap property

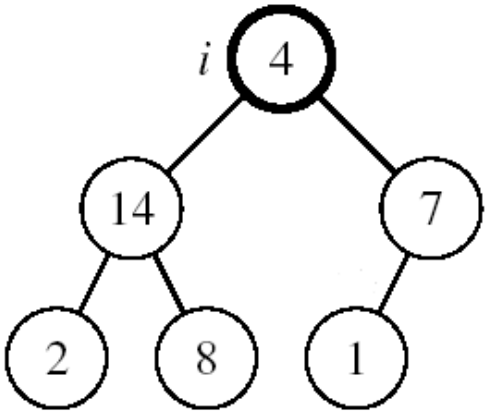
A[4] ↔ A[9]



Heap property restored

Maintaining the Heap Property

- Assumptions:
  - Left and Right subtrees of i are max-heaps
  - A[i] may be smaller than its children



Alg: MAX-HEAPIFY(A, i, n)

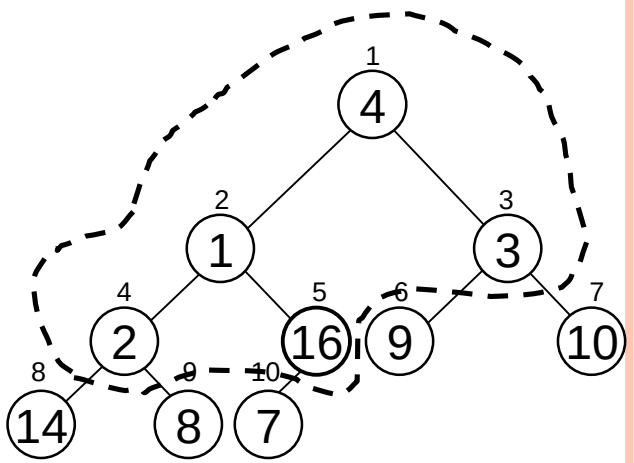
1. l ← LEFT(i)
2. r ← RIGHT(i)
3. **if** l ≤ n and A[l] > A[i]
4. **then** largest ← l
5. **else** largest ← i
6. **if** r ≤ n and A[r] > A[largest]
7. **then** largest ← r
8. **if** largest ≠ i
9. **then** exchange A[i] ↔ A[largest]
10. MAX-HEAPIFY(A, largest, n)

Building a Heap

- Convert an array $A[1 \ldots n]$ into a max-heap ($n = length[A]$)
- The elements in the subarray $A[(\lfloor n/2 \rfloor + 1) .. n]$ are leaves
- Apply MAX-HEAPIFY on elements between 1 and $\lfloor n/2 \rfloor$

Alg: BUILD-MAX-HEAP(A)

1. n = length[A]
2. **for** i ← ⌊n/2⌋ **downto** 1
3. **do** MAX-HEAPIFY(A, i, n)

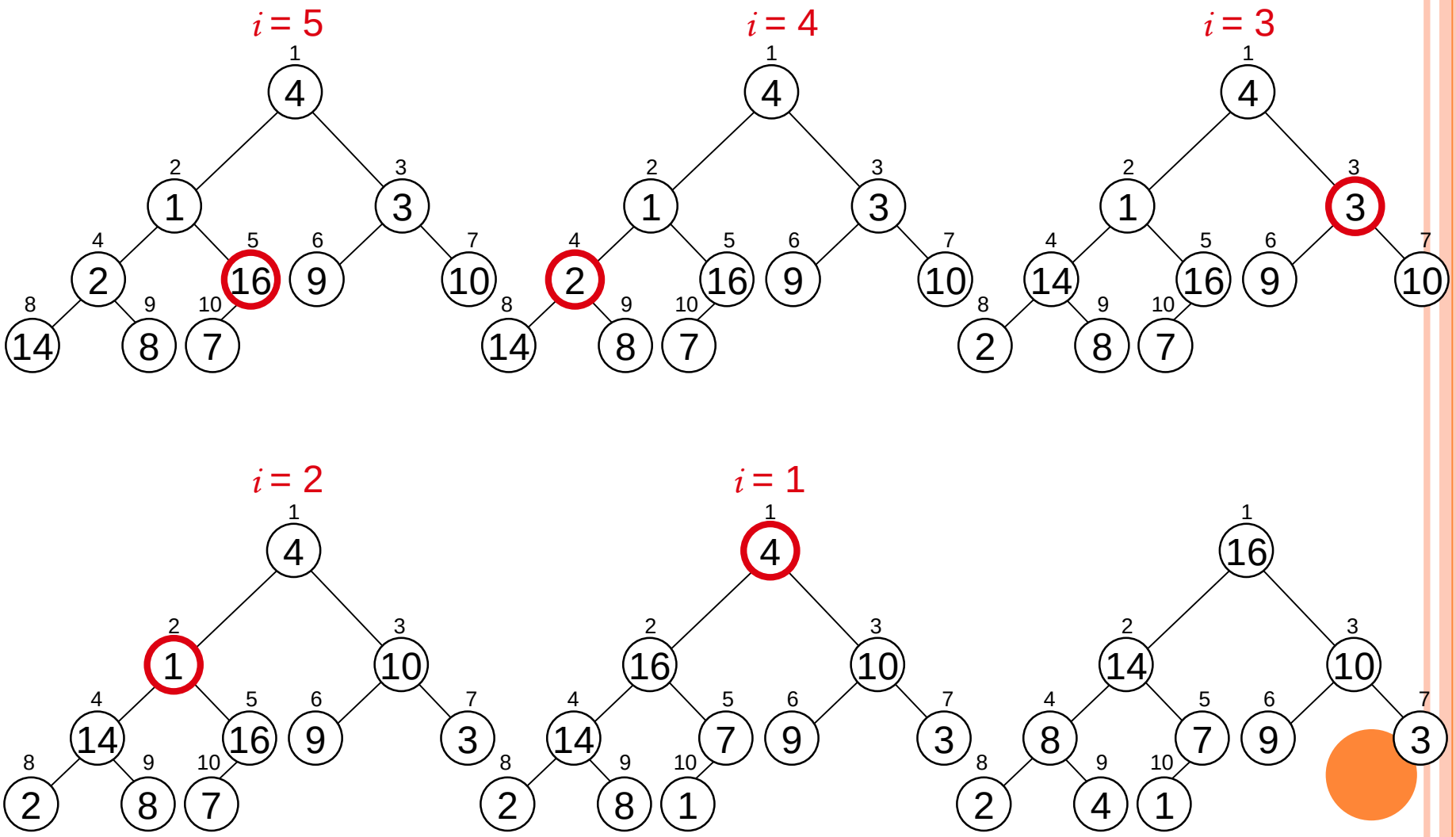


A:

4	1	3	2	16	9	10	14	8	7

EXAMPLE: A

4	1	3	2	16	9	10	14	8	7



Running Time of BUILD MAX HEAP

Alg: BUILD-MAX-HEAP(*A*)

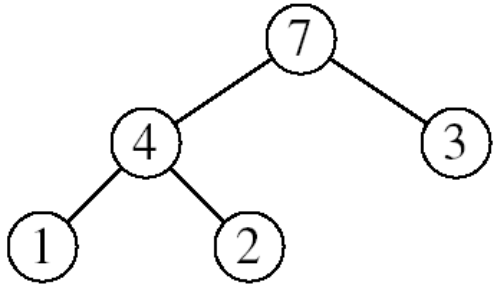
1. n = length[A]
2. **for** i ← ⌊n/2⌋ **downto** 1
3. **do** MAX-HEAPIFY(*A*, i, n)

O(lgn) } O(n)

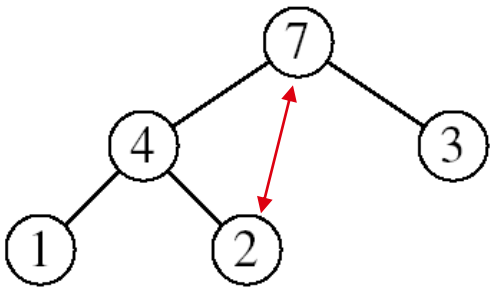
⇒ Running time: O(nlgn)

Heapsort

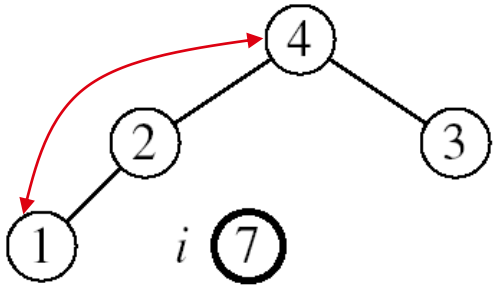
- Goal:
  - Sort an array using heap representations
- Idea:
  - Build a **max-heap** from the array
  - Swap the root (the maximum element) with the last element in the array
  - "Discard" this last node by decreasing the heap size
  - Call MAX-HEAPIFY on the new root
  - Repeat this process until only one node remains



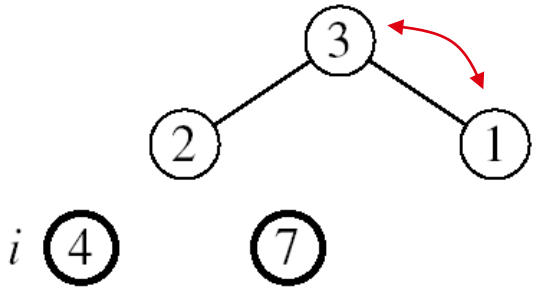
EXAMPLE: A=[7, 4, 3, 1, 2]



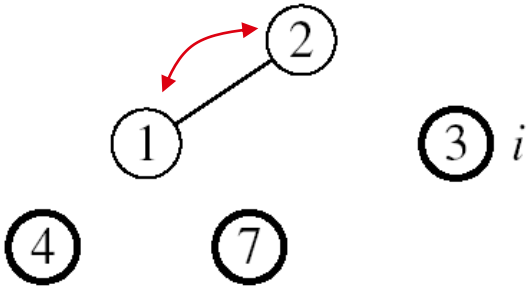
MAX-HEAPIFY(A, 1, 4)

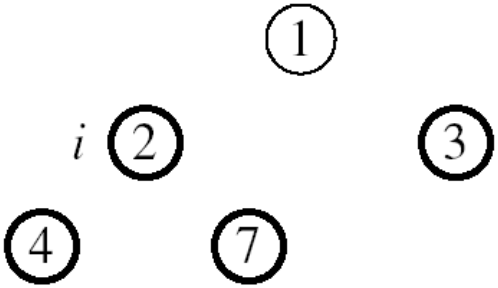


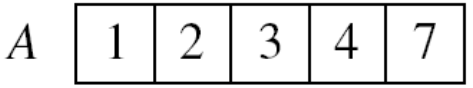
MAX-HEAPIFY(A, 1, 3)



MAX-HEAPIFY(A, 1, 2)







MAX-HEAPIFY(A, 1, 1)

Alg: HEAPSORT(*A*)

1. BUILD-MAX-HEAP(*A*)
2. **for** i ← length[*A*] **downto** 2
3. **do** exchange *A*[1] ↔ *A*[i]
4. MAX-HEAPIFY(*A*, 1, i - 1)

O(n)

n-1 times

O(lgn)

- Running time: O(nlgn) --- Can be shown to be Θ(nlgn)

Problems

- Demonstrate, step by step, the operation of Build-Heap on the array

  A=[5, 3, 17, 10, 84, 19, 6, 22, 9]

ADVANCE DATA STRUCTURES

Mr. A. P. Katade

- Examination Scheme

Theory: 50 Marks

Online: 50 Marks

Practical: 50 Marks

- Teaching Scheme

Lectures: 4 Hrs/week

Practical: 4 Hrs/week

- **Prerequisites: Data Structure and Algorithms**
- **Course Objectives:**

- To develop a logic for graphical modeling of the real life problems.
- To suggest appropriate data structure and algorithm for graphical solutions of the problems.
- To understand advanced data structures to solve complex problems in various domains.
- To operate on the various structured data
- To build the logic to use appropriate data structure in logical and computational solutions.
- To understand various algorithmic strategies to approach the problem solution.

Course Outcomes

- To apply appropriate advanced data structure and efficient algorithms to approach the problems of various domain.
- To design the algorithms to solve the programming problems.
- To use effective and efficient data structures in solving various Computer Engineering domain problems.
- To analyze the algorithmic solutions for resource requirements and optimization
- To use appropriate modern tools to understand and analyze the functionalities confined to the data structure usage.

UNIT1:Trees

- **Syllabus**
- **Tree-**basic terminology, General tree and its representation, representation using sequential and linked organization, Binary tree- properties, converting tree to binary tree, **binary tree traversals-** in-order, preorder, post order, level wise -depth first and breadth first, Operations on binary tree. Binary Search Tree (BST), BST operations, Threaded binary tree- concepts, threading, insertion and deletion of nodes in in-order threaded binary tree, in order traversal of in-order threaded binary tree. **Case Study-** Use of binary tree in expression tree-evaluation and Huffman's coding

Abstract Data Type

- The Data Type which can be represented in the form of its operation is called as ADT
- Abstract means hide
- *How* it is done *what* can it do.

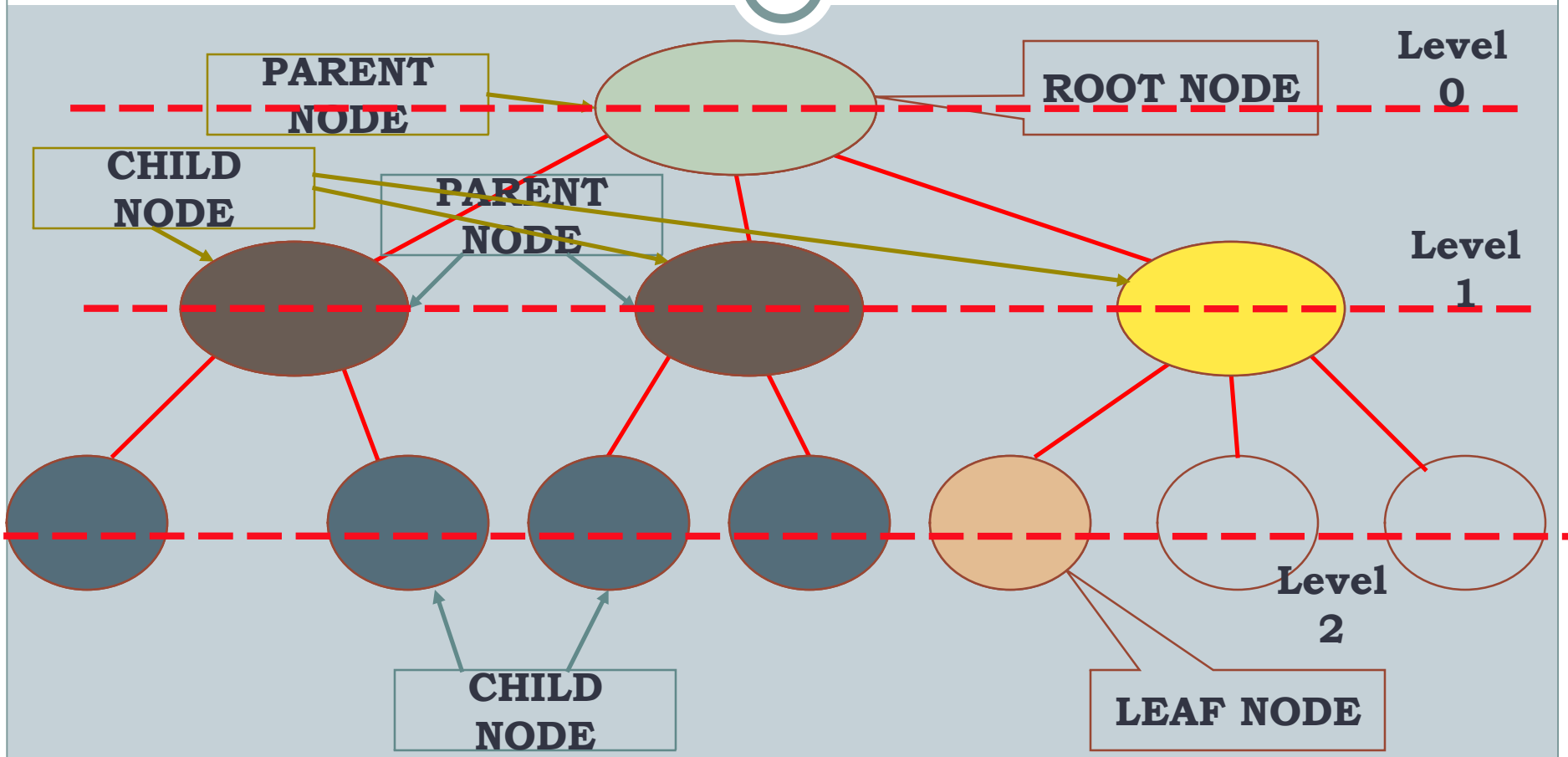
ADT

Declaration of data

Declaration of operation

Encapsulation of data & operation

TREE Representation



Definition of Tree

- A tree is a finite set of one or more nodes such that:
  - There is a specially designated node called the root.
  - The remaining nodes are partitioned into n>=0 disjoint sets T_1, …, T_n, where each of these sets is a tree.
  - We call T_1, …, T_n the subtrees of the root.

Terminology

- Finite set of elements in tree are called **NODES**
- Finite set of lines in tree are called as **BRANCHES**
- Branch directed towards the node is the **INDEGREE** of the node
- Branch directed away from the node is the **OUTDEGREE** of the node
- Sum of INDEGREE and OUTDEGREE of branches is the **DEGREE** of the node
- Number of branches associated with the node is the **DEGREE** of the node

Terminology

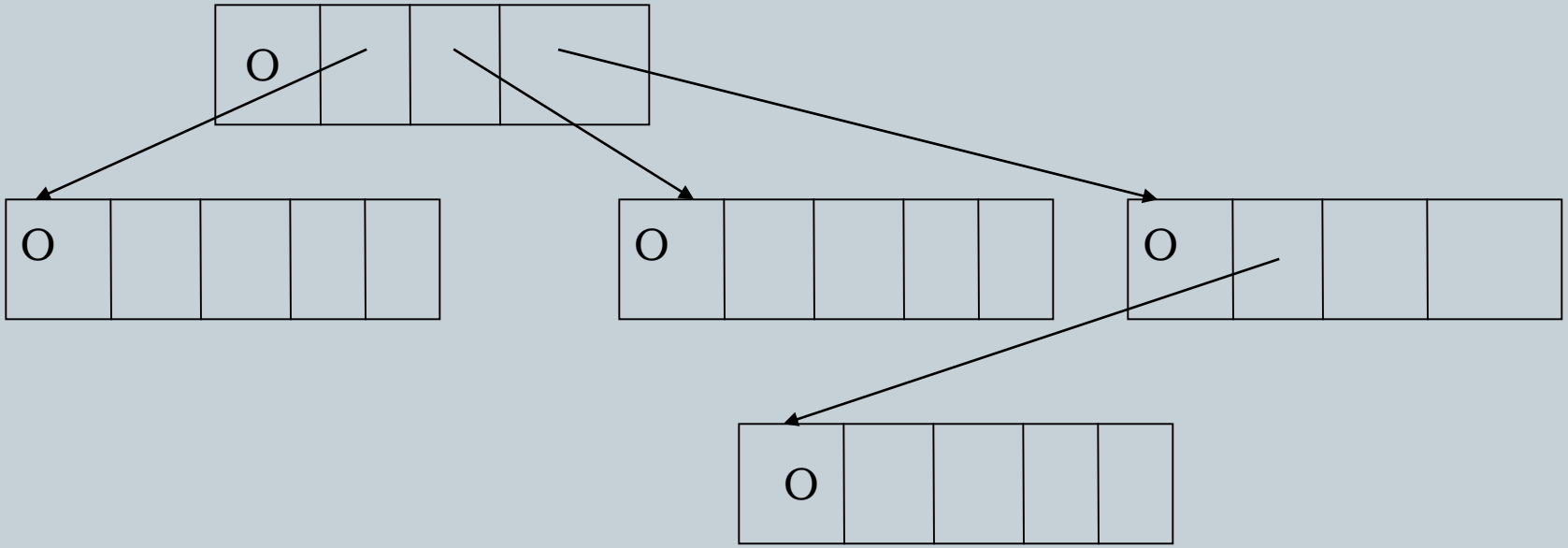
- INDEGREE of ROOT is **ZERO**
- All other nodes of the tree will have in degree one and out degree can be zero or more
- OUTDEGREE of LEAF node is **ZERO**
- A node that is neither ROOT not LEAF is **INTERNAL NODE**
- **HEIGHT** of the tree is the level of the leaf in the longest path from root plus 1
- A node is called **PARENT** node if it has successor nodes i.e. outdegree greater than ZERO

Terminology

- The degree of a node is the number of sub trees of the node
  The degree of root is 3; the degree of child of root is 2.
- The node with degree 0 is a leaf or terminal node.
- A node that has sub trees is the ***parent*** of the roots of the sub trees.
- The roots of these sub trees are the ***children*** of the node.
- Children of the same parent are ***siblings.***
- The ancestors of a node are all the nodes along the path from the root to the node.

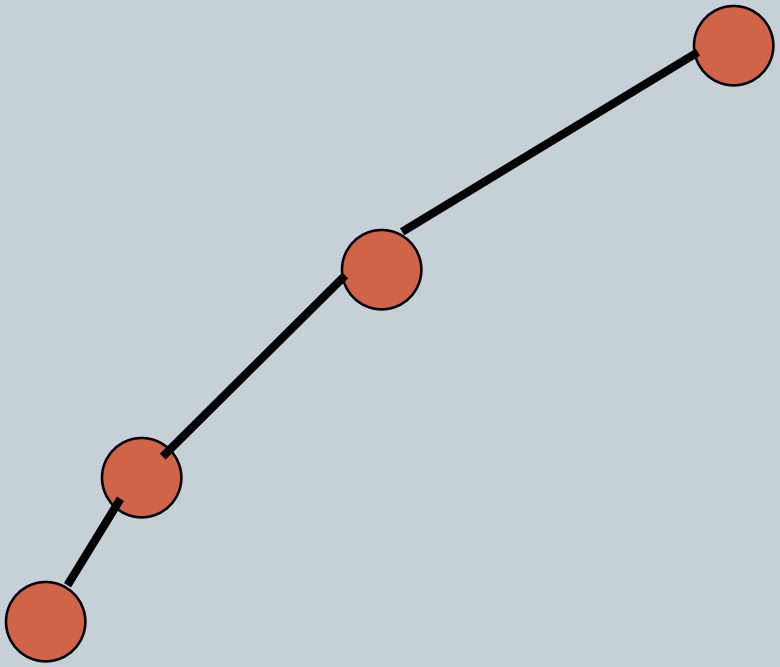
A Tree Node

- Every tree node:
  - object – useful information
  - children – pointers to its children nodes



Minimum Number Of Nodes

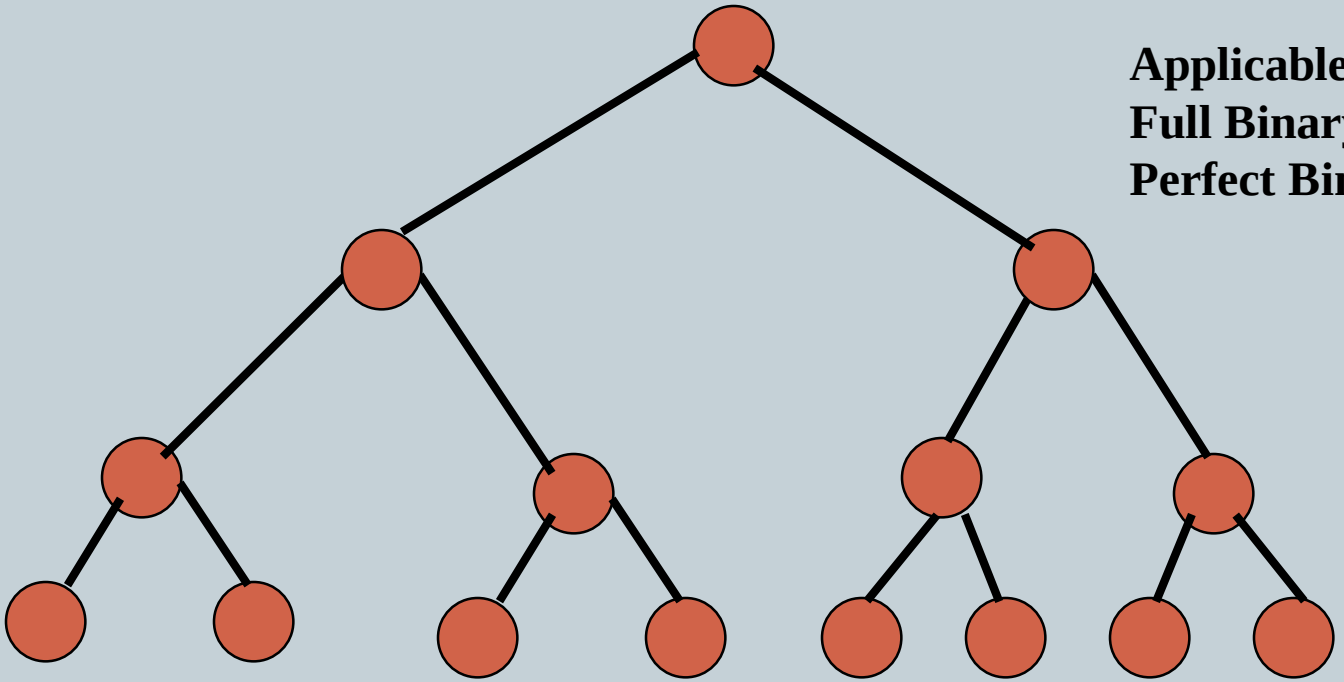
- Minimum number of nodes in a binary tree whose height is h.
- At least one node at each of first h levels.



minimum number of nodes is h

Maximum Number Of Nodes

- All possible nodes at first h levels are present.



Applicable for
Full Binary tree
Perfect Binary tree

Maximum number of nodes

$= 1 + 2 + 4 + 8 + \ldots + 2^{h-1}$

$= 2^{h} - 1$

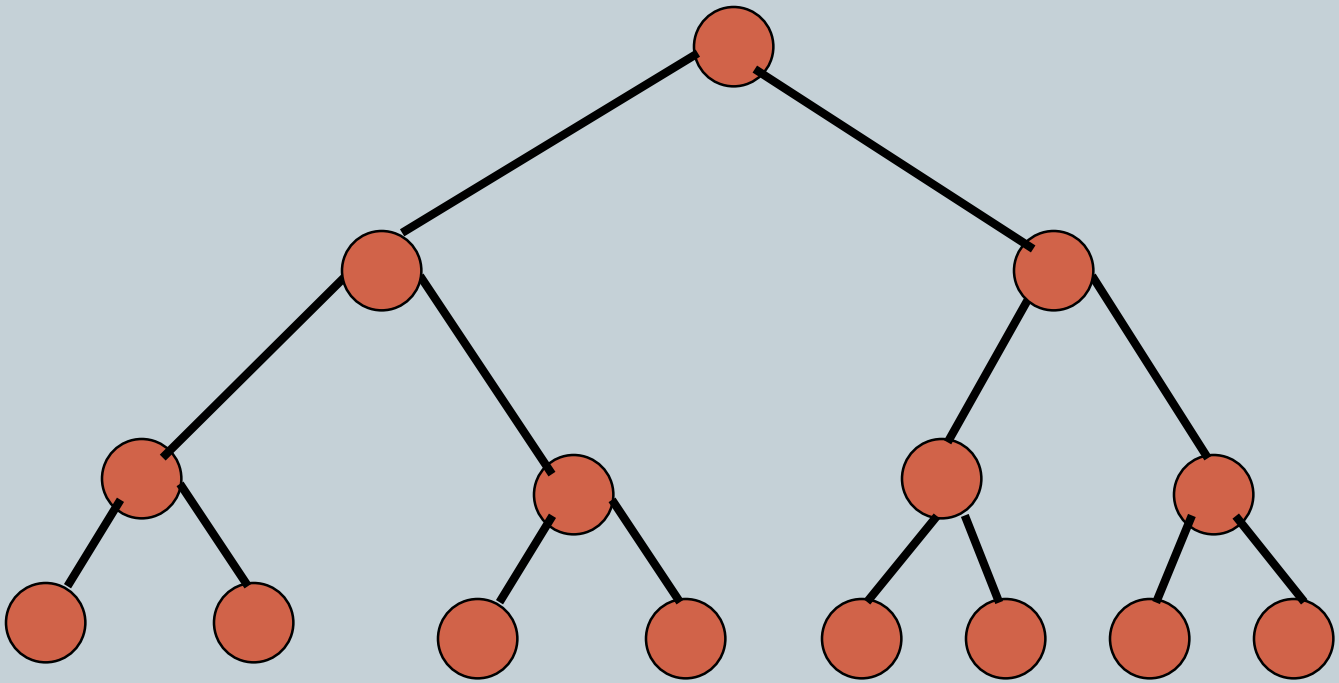
Number Of Nodes & Height

- Let n be the number of nodes in a binary tree whose height is h.
- The **number** of nodes **doubles** every time the **depth** increases by **1**
- $h <= n <= 2^h - 1$
- If height of **a leaf is considered as 0**. In this convention, the above formula becomes $2^{h+1} - 1$
- Number of nodes in any binary tree is $2^{lh} - 1 + 2^{rh} - 1 + 1$

i.e. height of left sub-tree + height of right sub-tree+1

Full Binary Tree

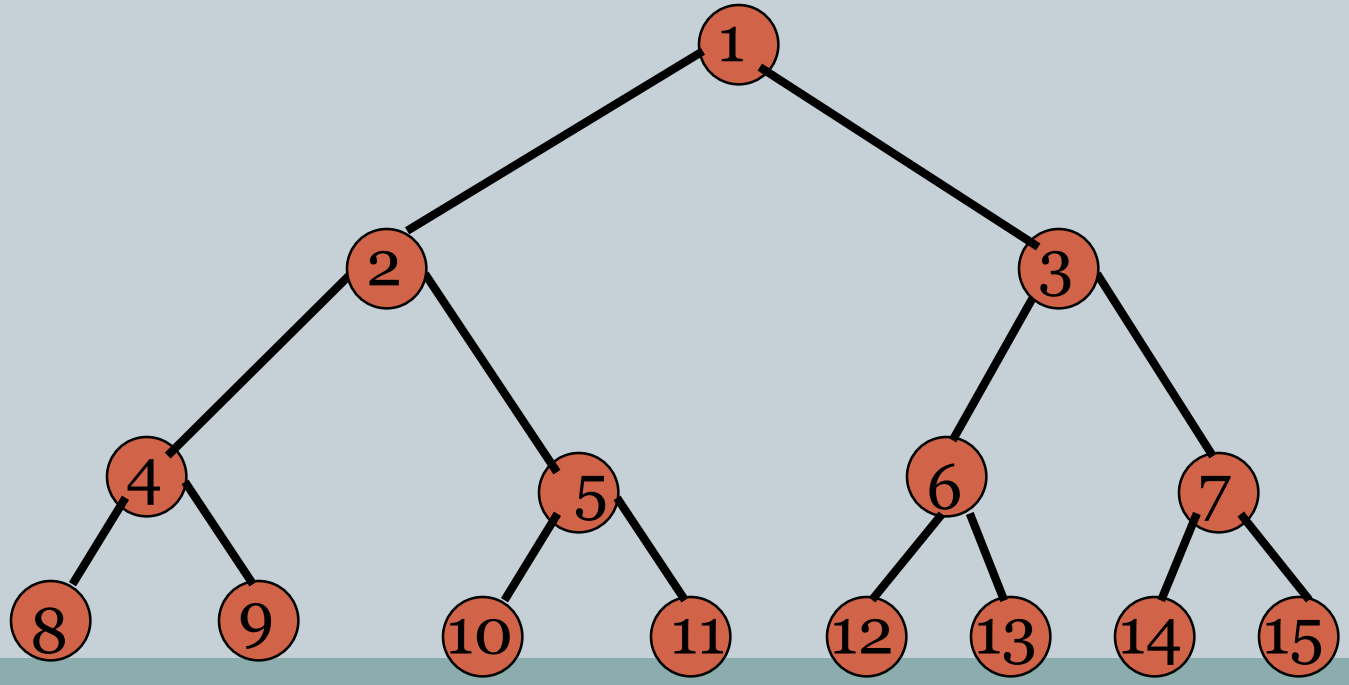
- A full binary tree of a given height h has $2^h - 1$ nodes.



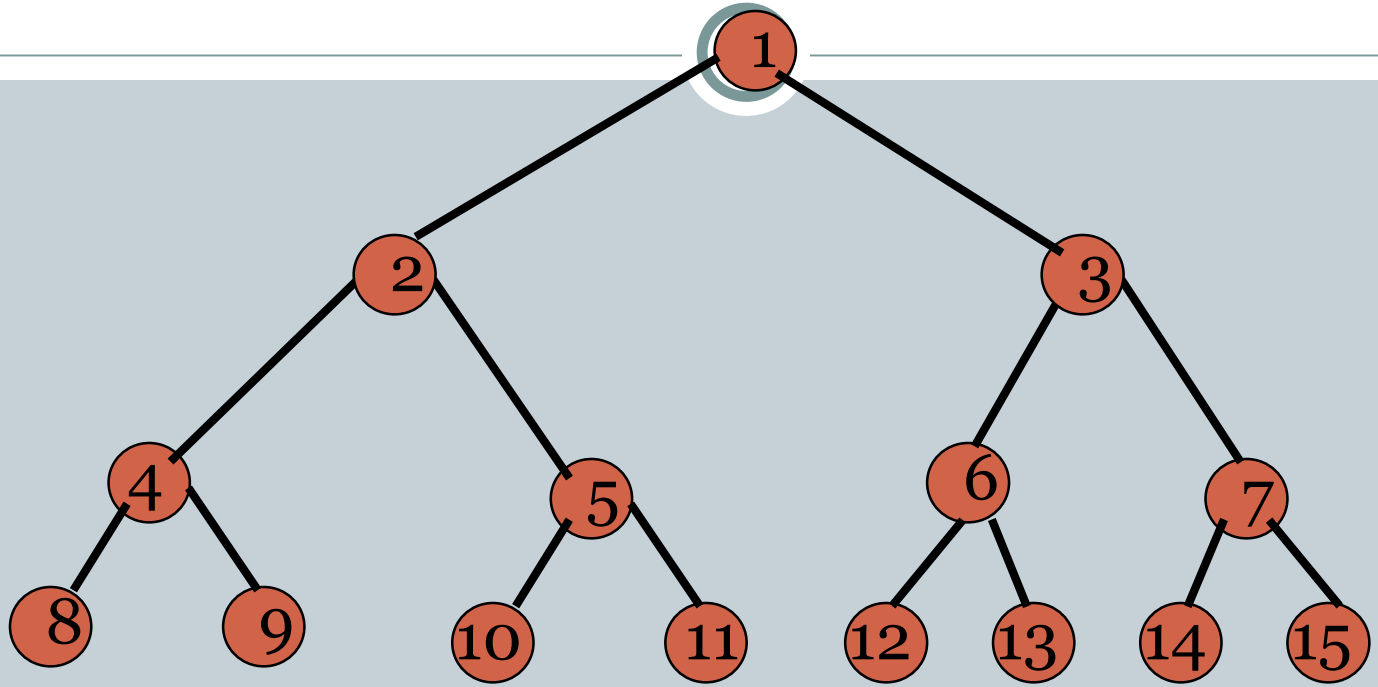
Height 4 full binary tree.

Numbering Nodes In A Full Binary Tree

- Number the nodes 1 through $2^h - 1$.
- Number by levels from top to bottom.
- Within a level number from left to right.

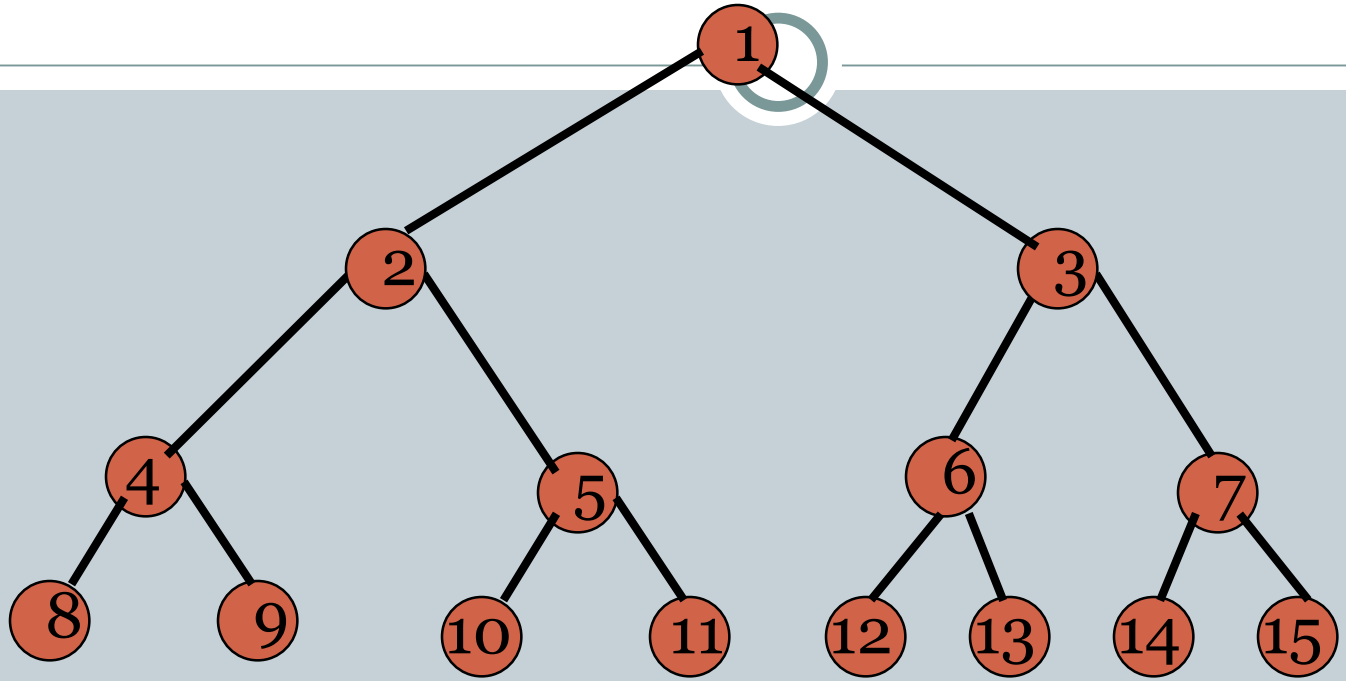


Node Number Properties



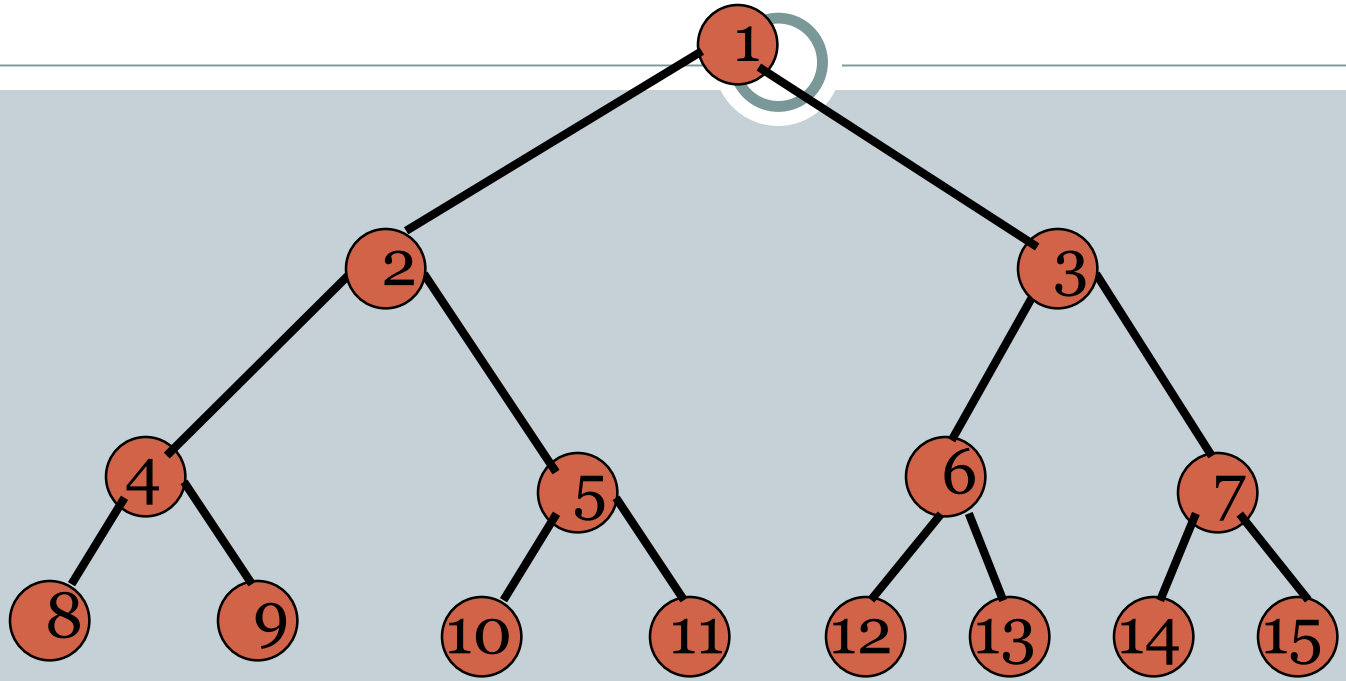
- Parent of node i is node i / 2, unless i = 1.
- Node 1 is the root and has no parent.

Node Number Properties



- Left child of node i is node 2i, unless 2i > n, where n is the number of nodes.
- If 2i > n, node i has no left child.

Node Number Properties



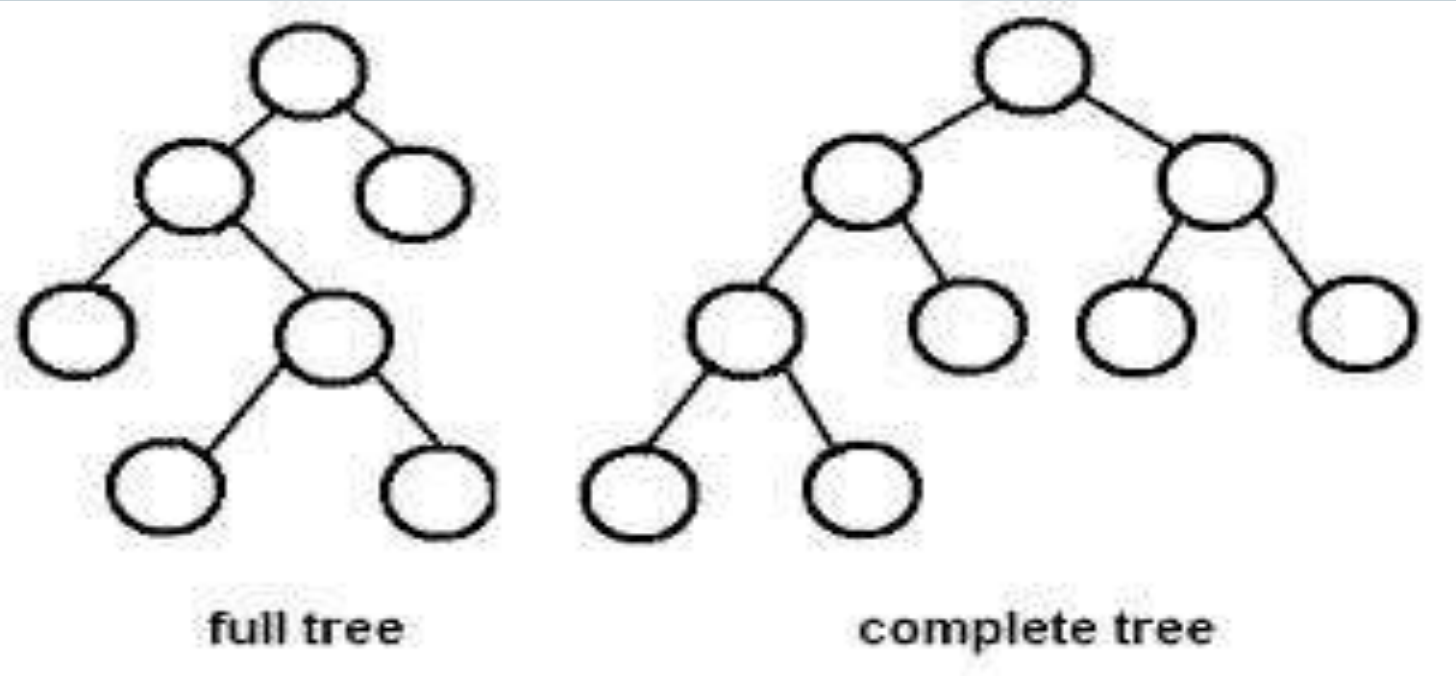
- Right child of node i is node 2i+1, unless 2i+1 > n, where n is the number of nodes.
- If 2i+1 > n, node i has no right child.

Complete Binary Tree With n Nodes

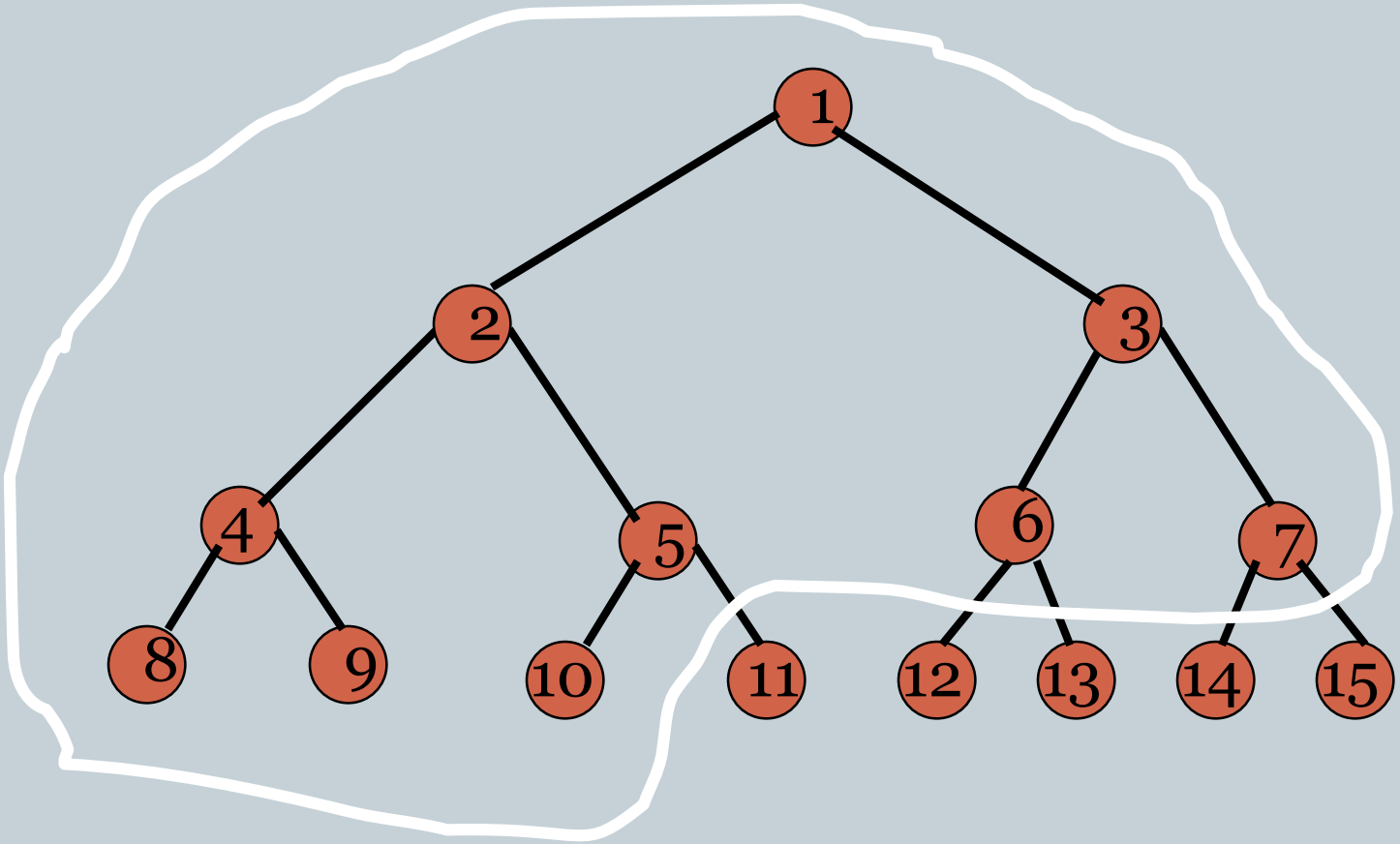
A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

A binary tree T is full If each node is either a **leaf or possesses exactly two child nodes.**

- A binary tree T with n levels is Complete if all levels except possibly the last are completely full, and the last level has all its nodes to the left side.



Example



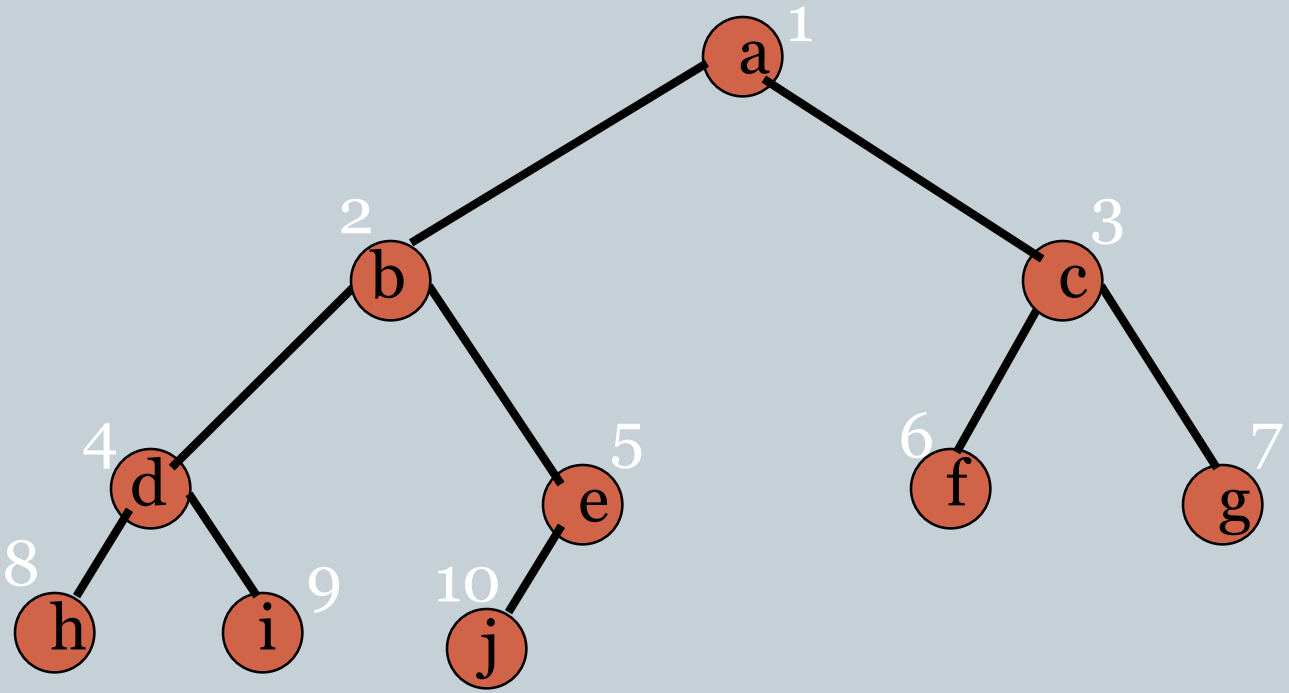
- Complete binary tree with **10** nodes.

Binary Tree Representation

- Array representation.
- Linked representation.

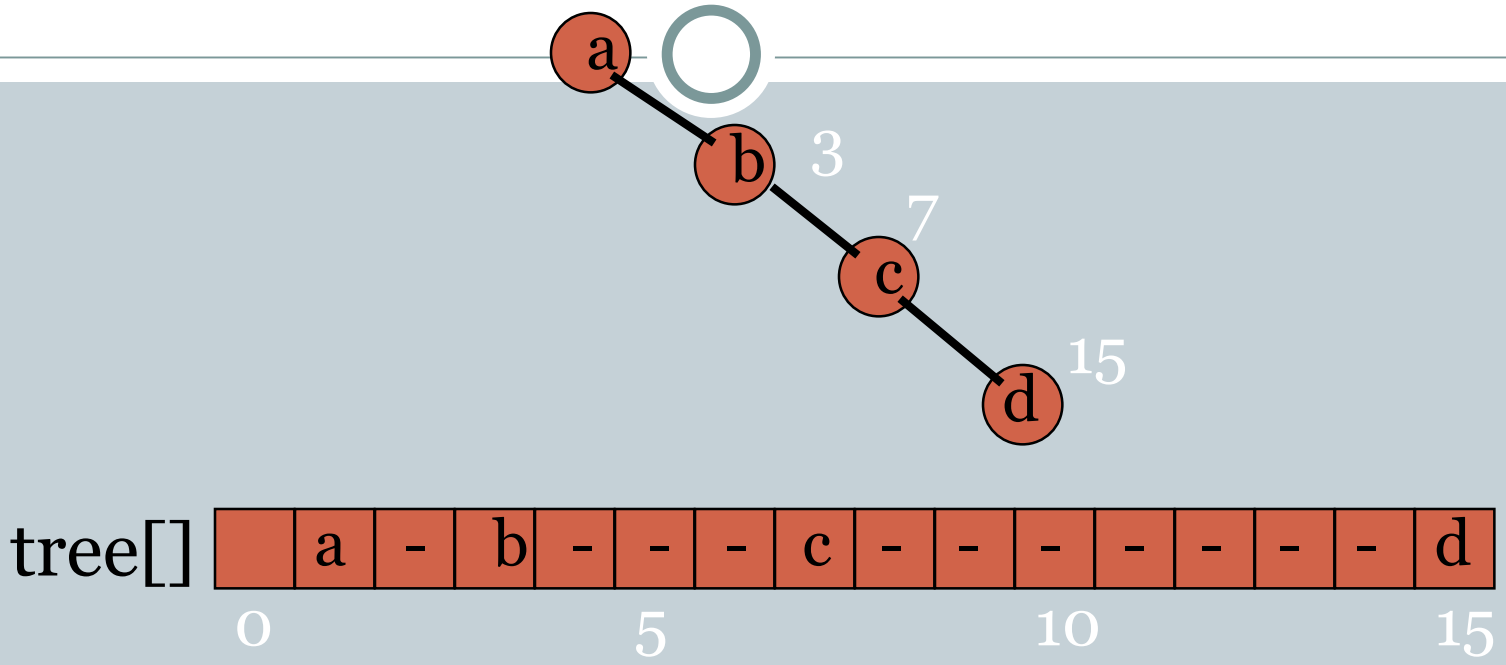
Array Representation

- Number the nodes using the numbering scheme for a full binary tree. The node that is numbered i is stored in tree[i].



tree[]		a	b	c	d	e	f	g	h	i	j
	0					5					10

Right-Skewed Binary Tree



- An n node binary tree needs an array whose length is between n+1 and 2^n.

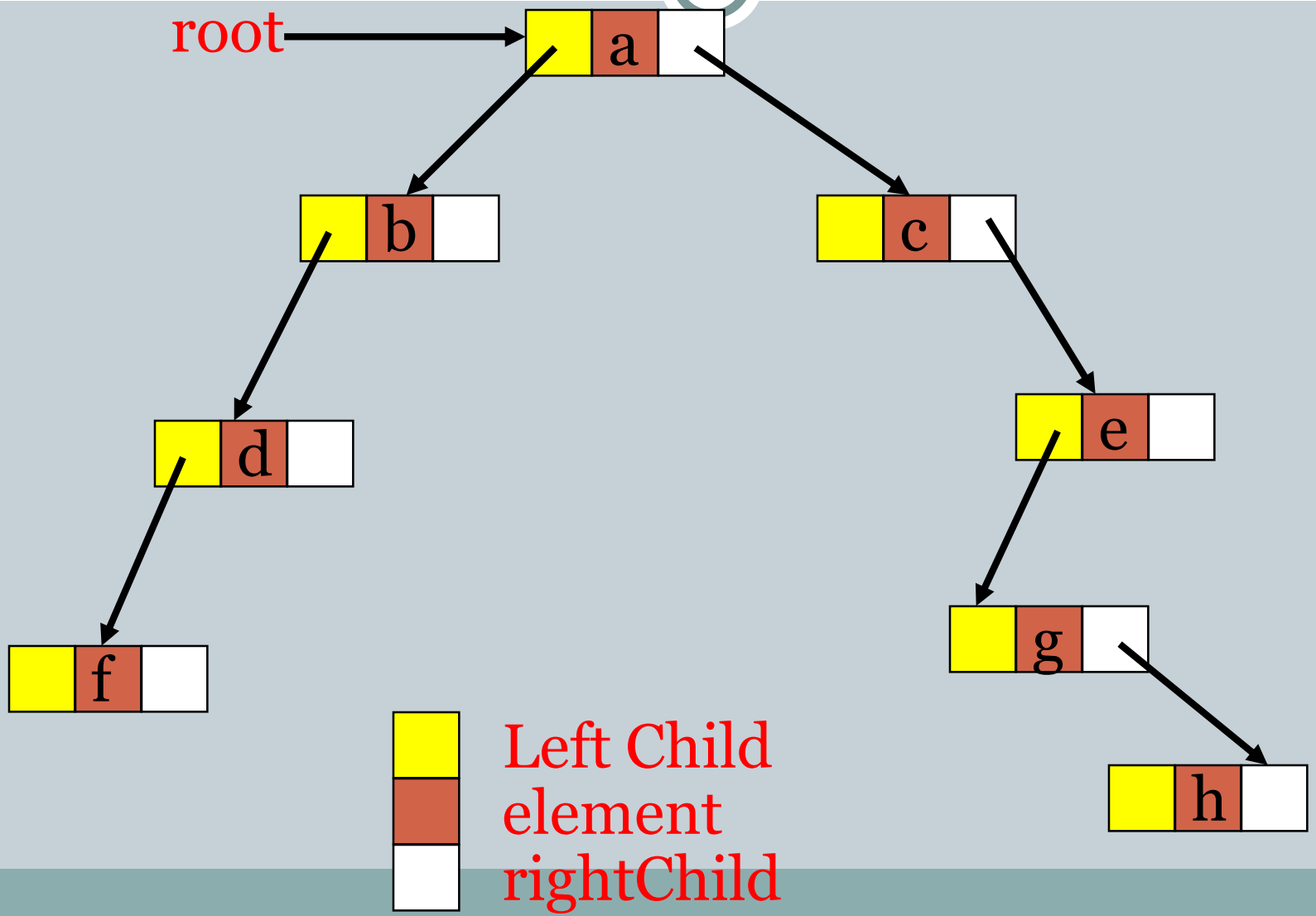
Linked Representation

- Each binary tree node is represented as an object whose data type is BinaryTreeNode.
- The space required by an n node binary tree is n * (space required by one node).

The Class BinaryTreeNode

```
Class BinaryTreeNode
{
<type of data>   Object element;
   BinaryTreeNode leftChild; // left subtree
   BinaryTreeNode rightChild;// right subtree
   // constructors and any other methods
   // come here
};
```

Linked Representation Example



Some Binary Tree Operations

- Determine the height.
- Determine the number of nodes.
- Make a clone.
- Determine if two binary trees are clones.
- Display the binary tree.
- Evaluate the arithmetic expression represented by a binary tree.
- Obtain the infix form of an expression.
- Obtain the prefix form of an expression.
- Obtain the postfix form of an expression.

Binary Tree Traversal

- Many binary tree operations are done by performing a traversal of the binary tree.
- In a traversal, each element of the binary tree is visited exactly once.
- During the visit of an element, all action (make a clone, display, evaluate the operator, etc.) with respect to this element is taken.

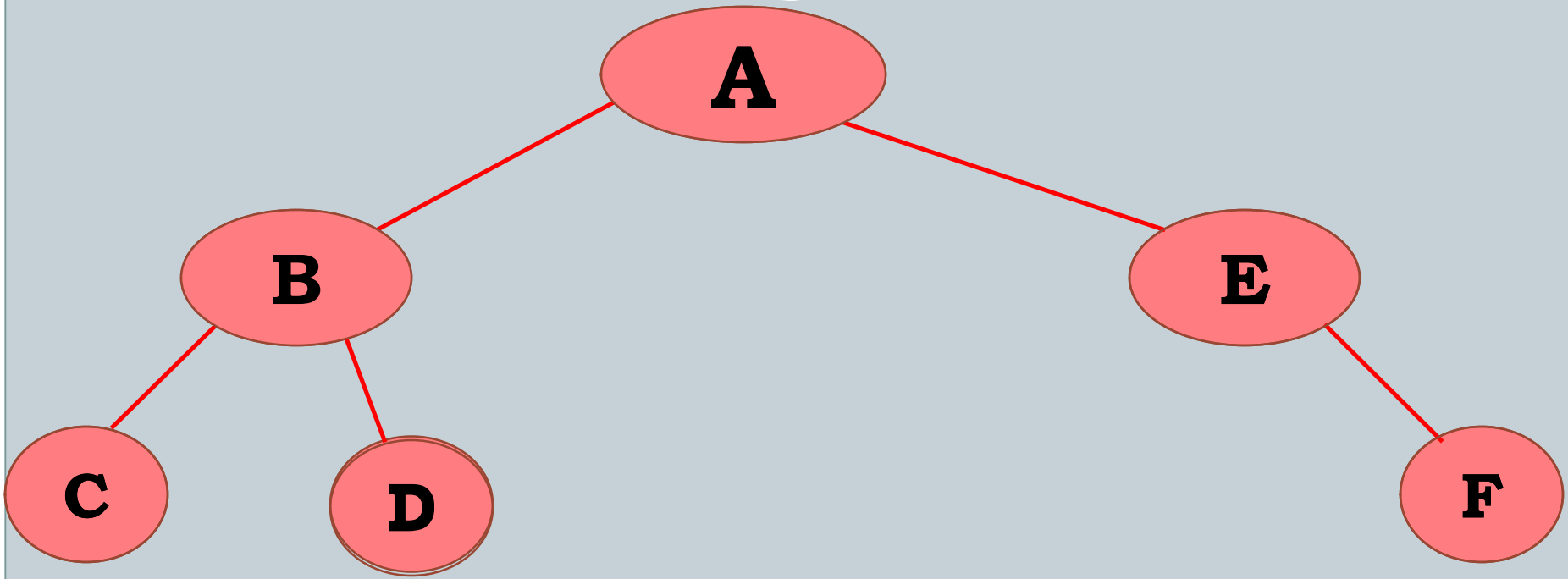
Tree Traversal

- Two main methods:
  - Depth first Traversal
  - Breadth first Traversal
- Three methods of Depth first Traversal
  - Preorder
  - Postorder
  - Inorder
- PREorder:
  - visit the root
  - Traverse left subtree
  - Traverse right subtree

Tree Traversal

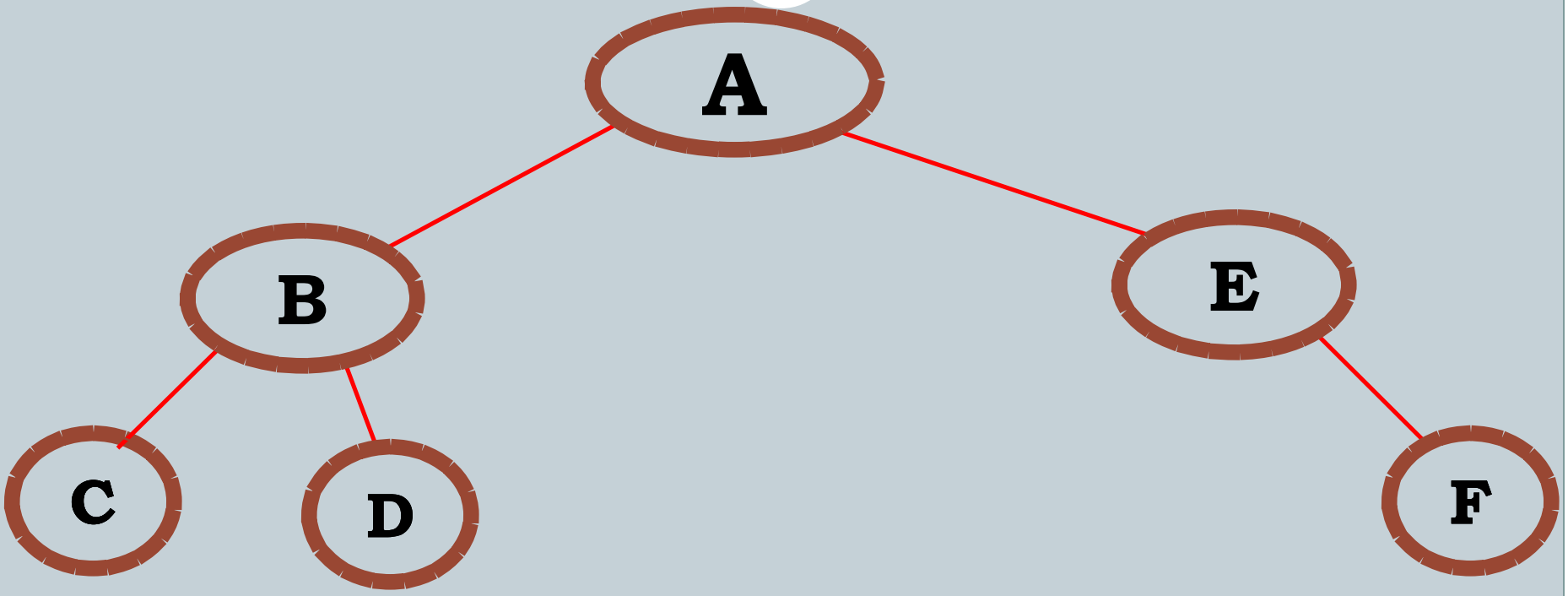
- POSTorder
  - Traverse left subtree
  - Traverse right subtree
  - visit the root
- Inorder
  - Traverse left subtree
  - visit the root
  - Traverse right subtree

Pre-Order Traversal



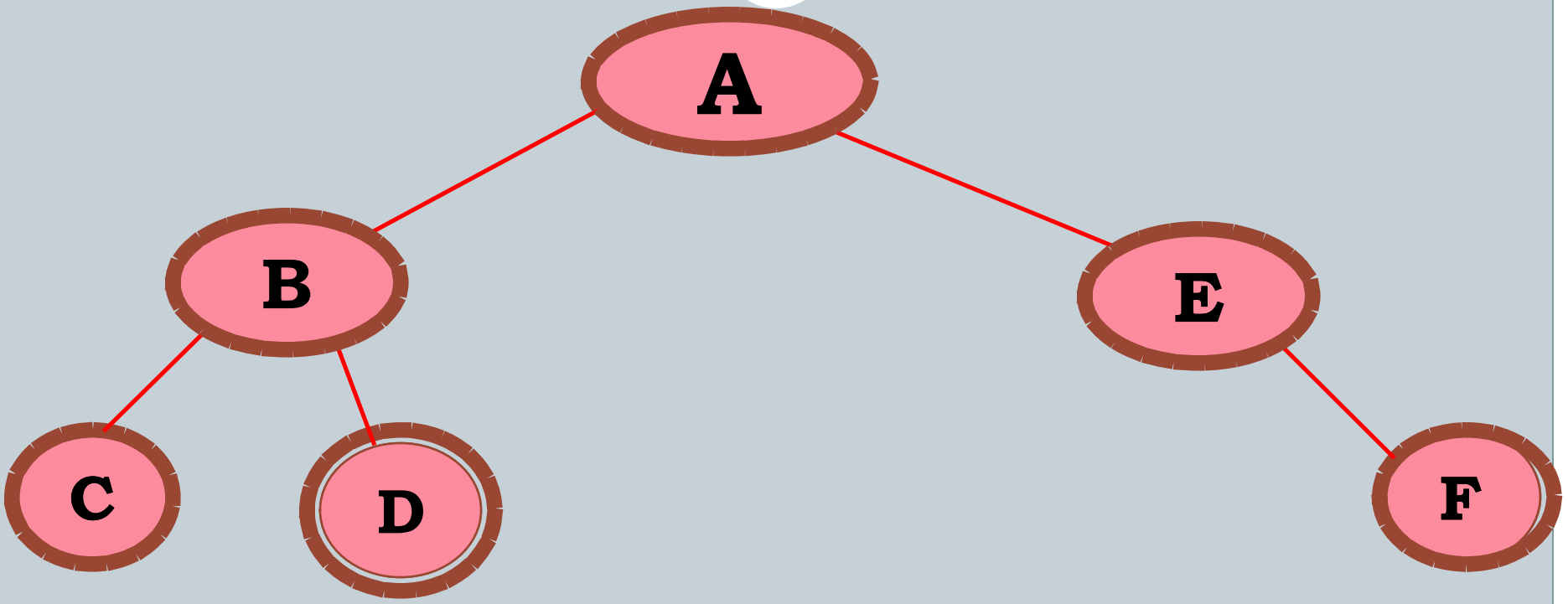
A B C D E F

In-Order Traversal



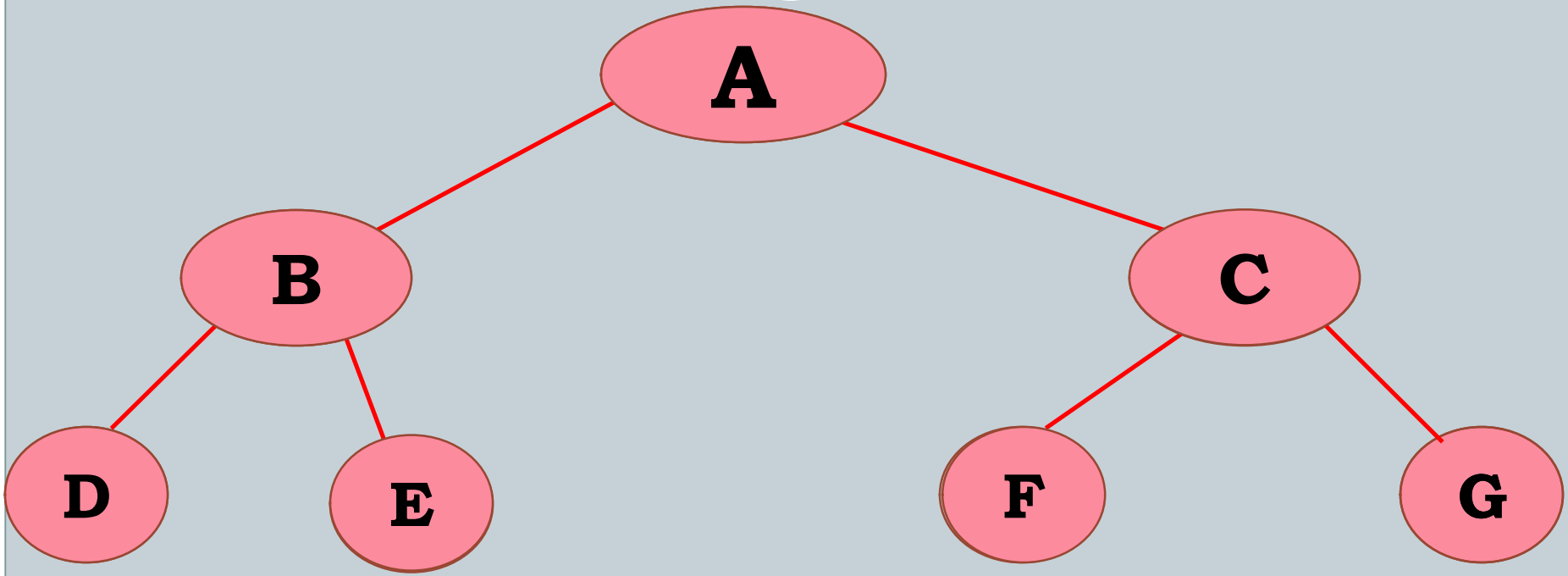
C B D A E F

Post-Order Traversal



C D B F E A

Breadth first Traversal



A B C D E F G

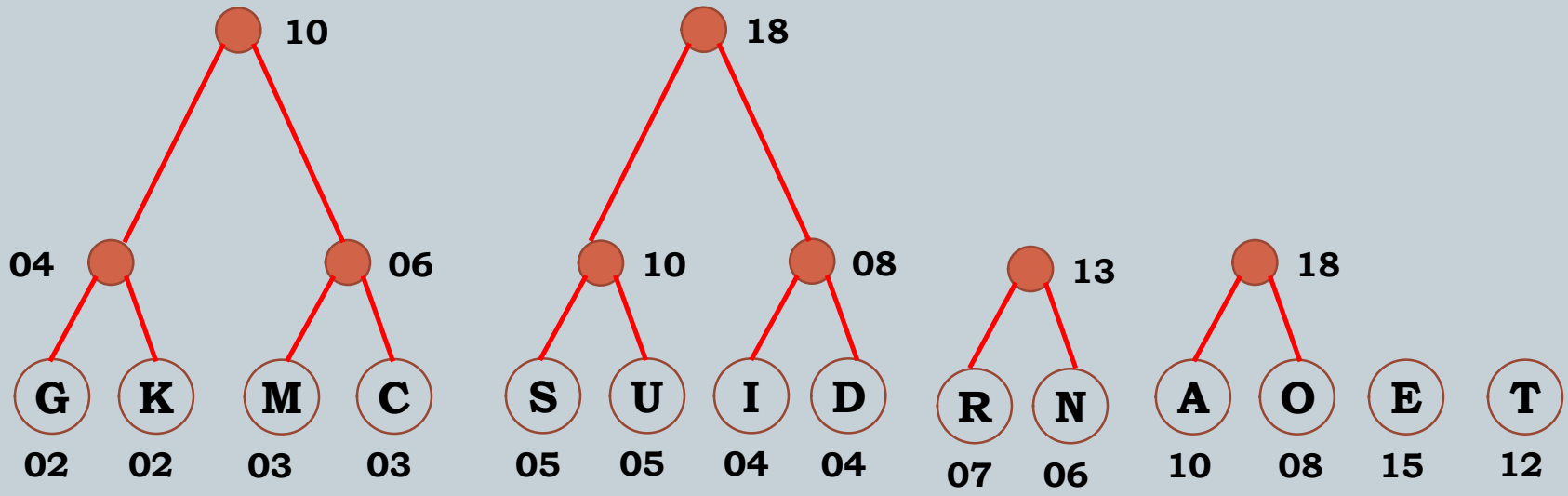
Huffman Code

- ASCII is a fixed length code of 7 bits
- But there are some characters having very high frequency of occurrence and some with very low frequency of occurrence
- In huffman code characters that occur more frequently will be assigned shorter codes and with lower frequency will be assigned longer codes
- Popular for data compression algorithm

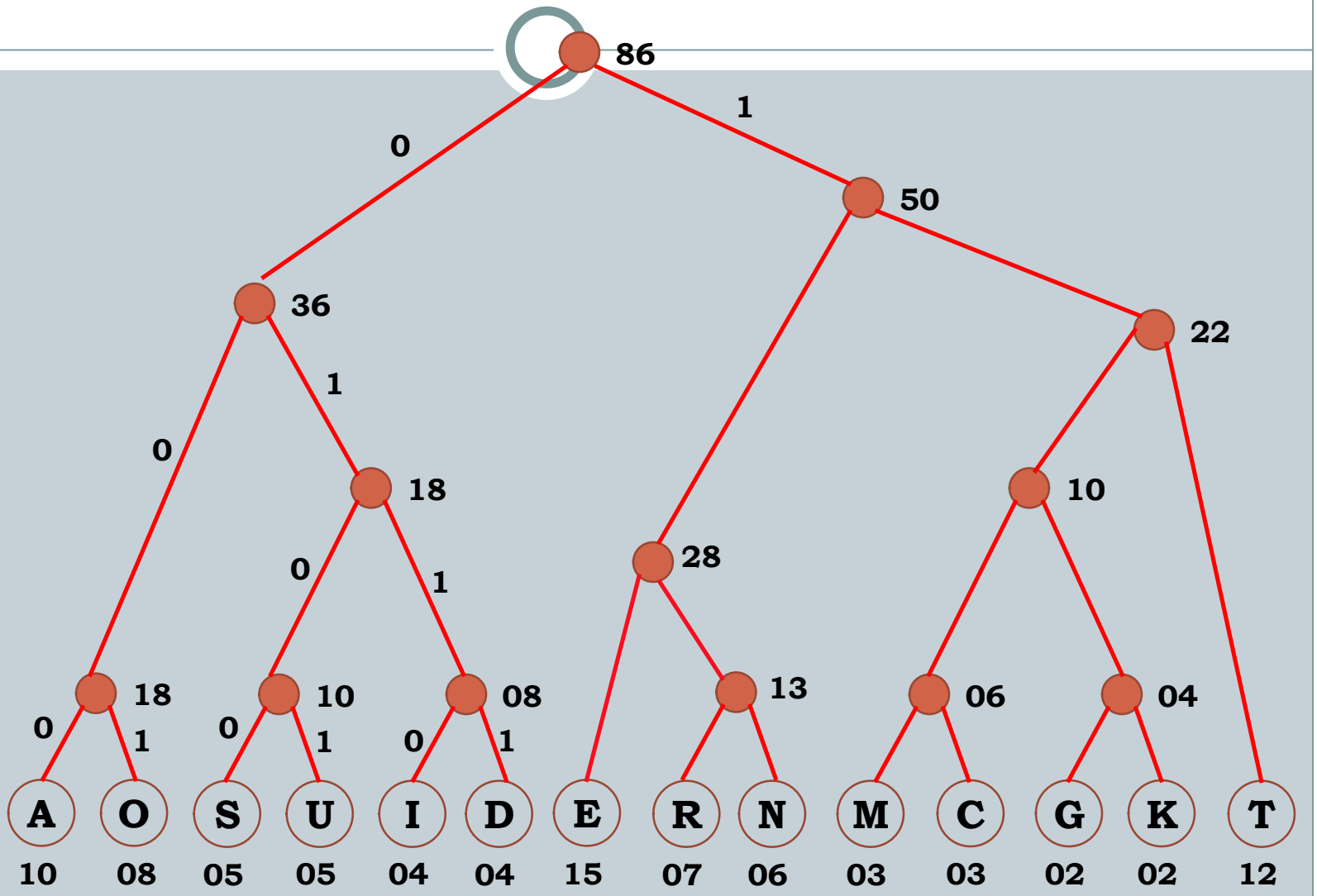
E=15	**T=12**	**A=10**	**O=8**	**R=7**	**N=6**	**S=5**
U=5	**I=4**	**D=4**	**M=3**	**C=3**	**G=2**	**K=2**

Algorithm

- Organize character set into row ordered according to frequency
- Find two nodes with smallest combined frequency and join them
- Weight of new node is combined weights of two nodes
- Sum of weights of the nodes chosen must be smaller than the combination of any other possible choices
- Repeat above steps till we get single tree
- If more number of leaves with same weight then choose nearest

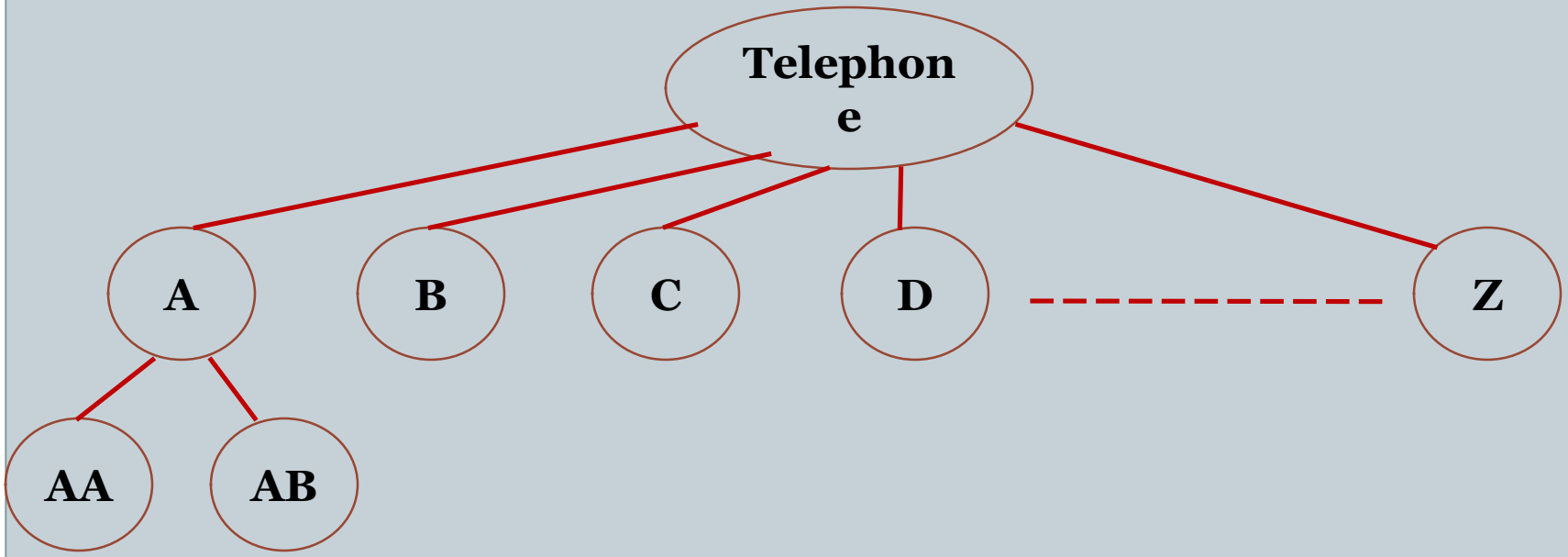

10
04
06
G
K
M
C
02
02
03
03
18
10
08
S
U
I
D
05
05
04
04
13
R
N
07
06
18
A
O
E
T
10
08
15
12

Huffman Code



General Tree

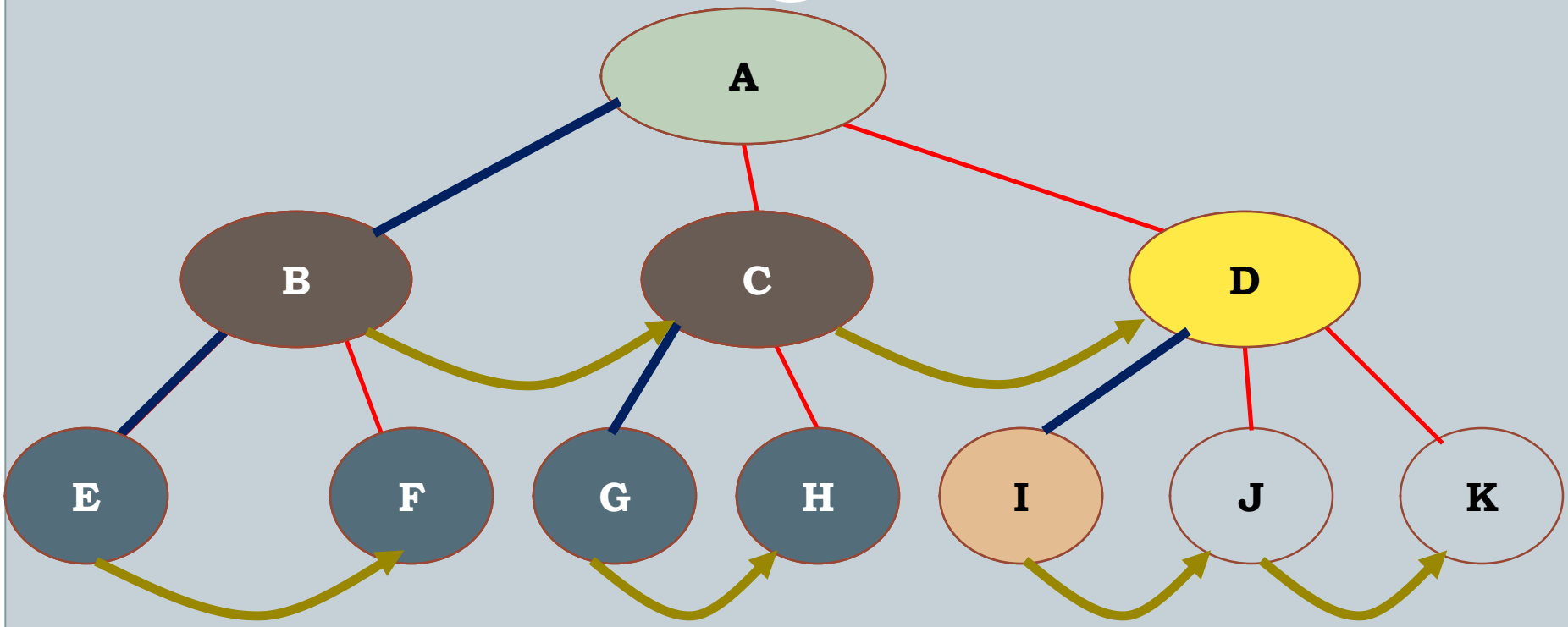
- A tree with unlimited out degree



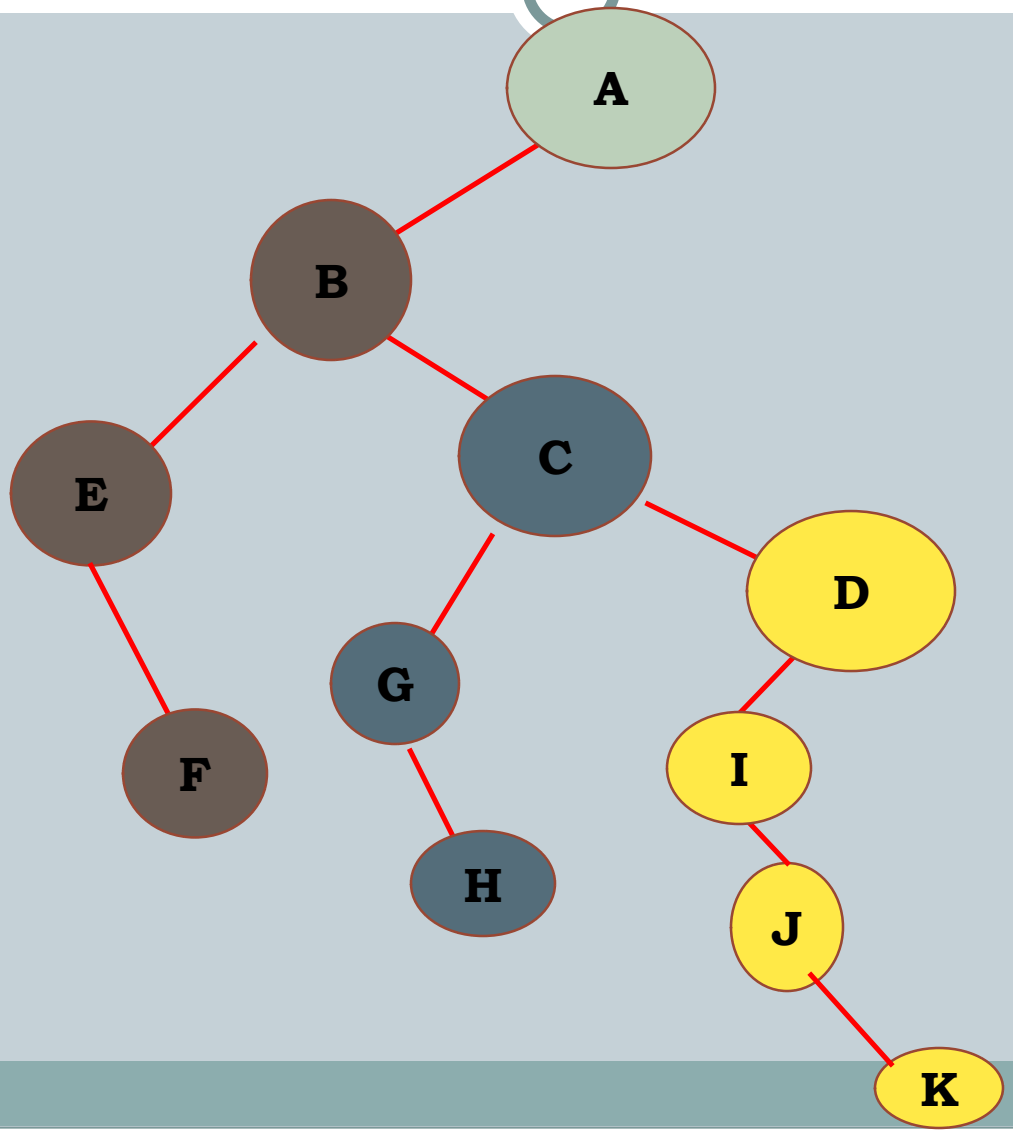
Algorithm to convert General Tree to Binary

- Identify branches from parent to left most child
- Connect remaining siblings from left most child using branches for each sibling to its right
- Delete all unneeded branches

Example Tree Conversion General to Binary



Example Tree Conversion General to Binary



Binary Search Trees

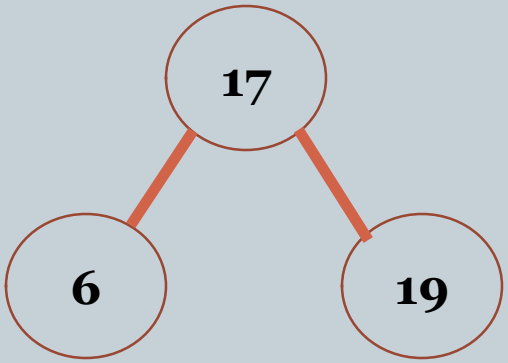
- Ordered data stored in Array will have efficient search algorithms but inefficient insertion and deletion

- Linked list storage will increase efficiency of insertion and deletion but search will always be sequential as you always have to start from ROOT node

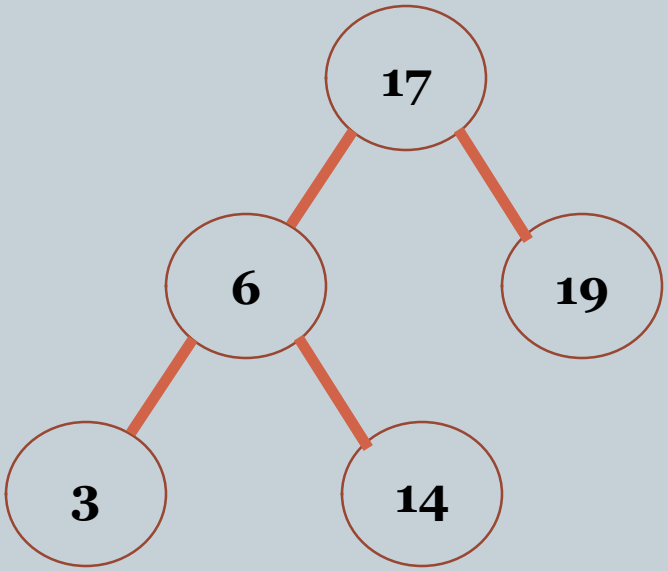
Properties of Binary Search Trees

- All items in left sub tree are less than root
- All items in right sub tree are greater than or equal to root
- Each sub tree is itself a binary search tree

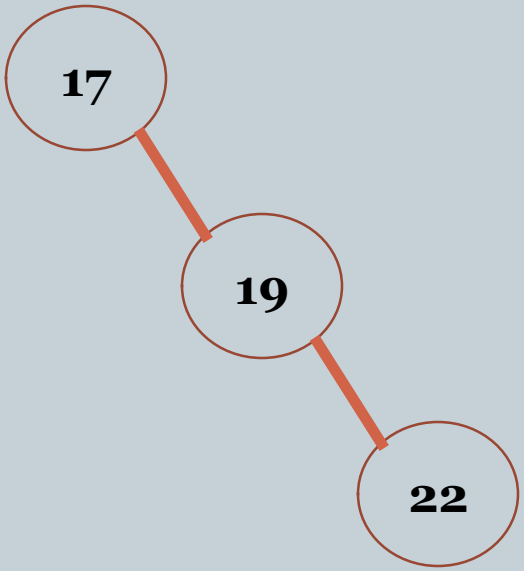
Is this Binary Search Tree



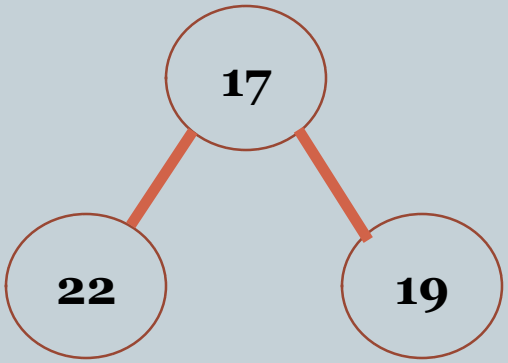
Is this Binary Search Tree



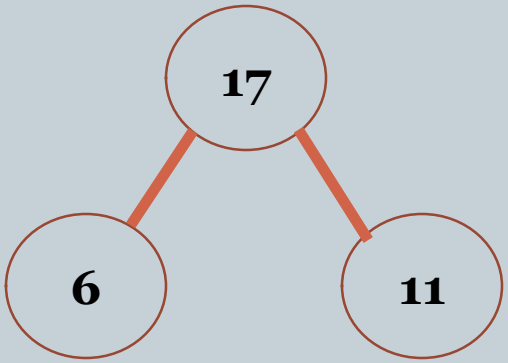
Is this Binary Search Tree



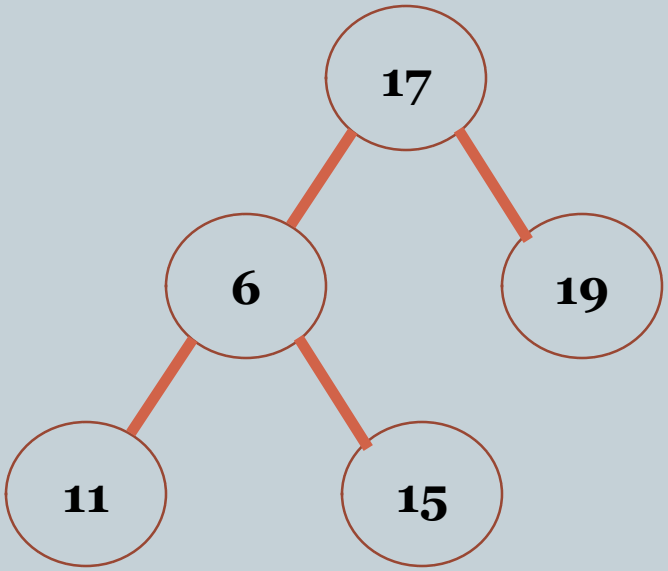
Is this Binary Search Tree



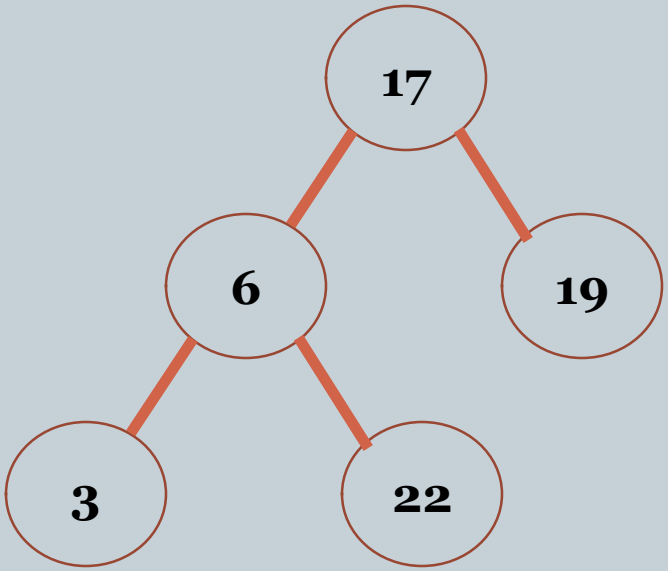
Is this Binary Search Tree



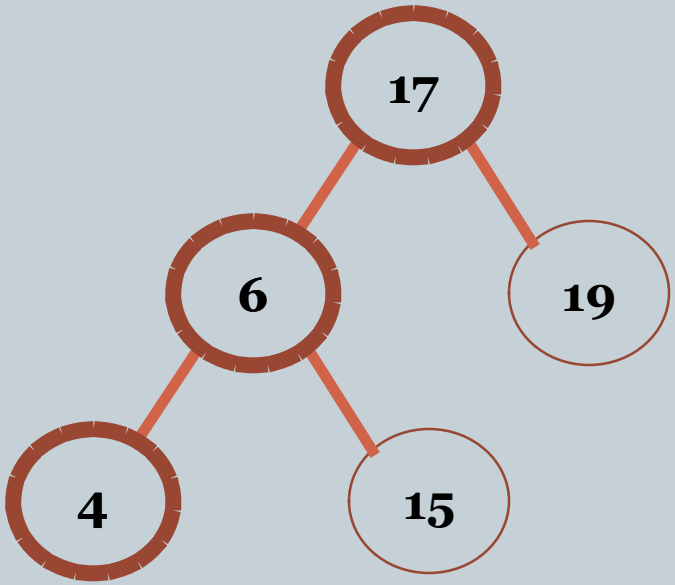
Is this Binary Search Tree



Is this Binary Search Tree

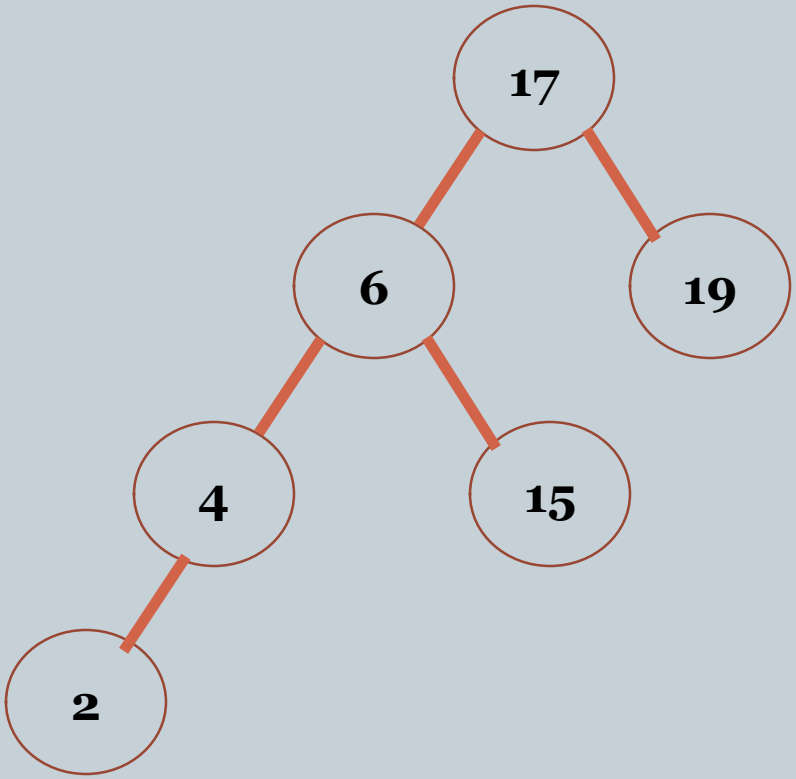


Insertion in BST

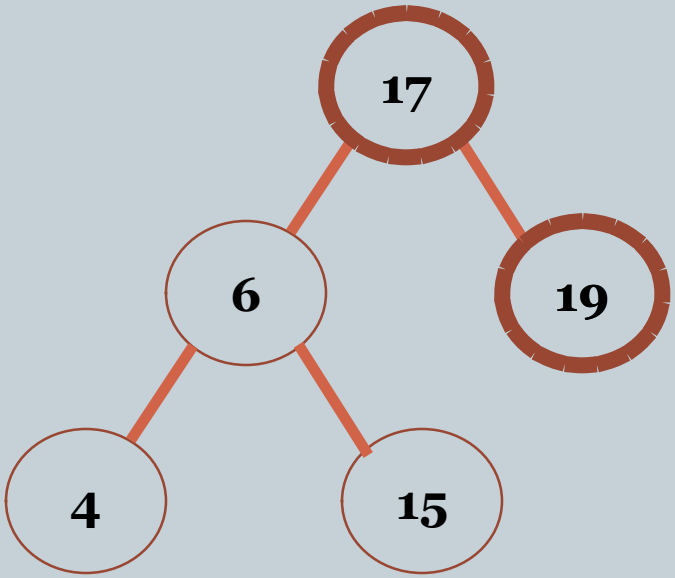


Add node - 2

Insertion in BST

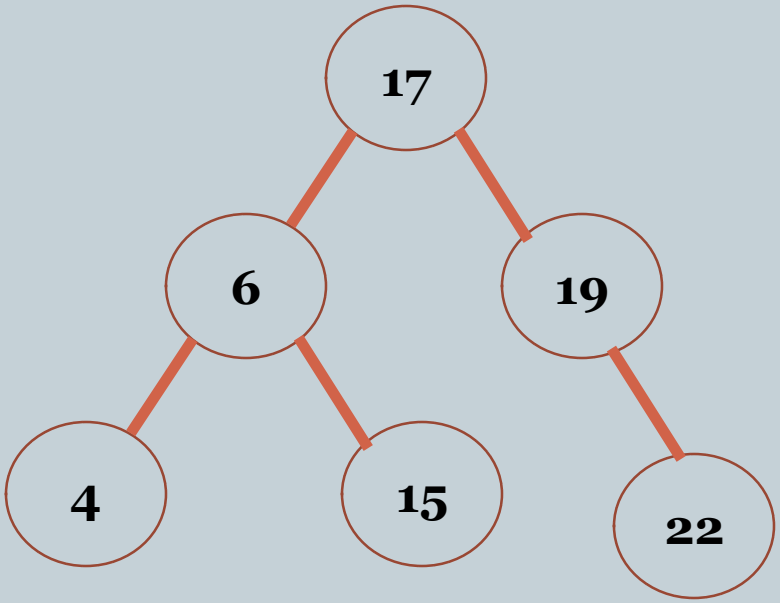


Insertion in BST



Add node - 22

Insertion in BST



Insertion

- Algorithm addBST (root, newnode)

If (empty)

Set root to newnode

Return newnode

End if

If (newnode < root)

Return addBST (left sub tree, newnode)

Else

Return addBST (right sub tree, newnode)

End if

End addBST

Deleting from BST

- If the node doesn’t have children's then just delete the reference for this node from its parent and recycle the memory

- If the node has **only right sub tree** then

  Simply attach right subtree to delete's nodes parent

- If the node has **only left sub tree** then

  Simply attach left subtree to delete's nodes parent

Deleting from BST

- If the node has **left and right** subtree then
  - Find **largest node from left subtree** and replace that with node we have to delete

OR

  - Find **smallest node from right subtree** and replace that with node we have to delete

BST Operations: Removal

- removes a specified item from the BST and adjusts the tree uses a binary search to locate the target item: starting at the root it probes down the tree till it finds the target or reaches a leaf node (target not in the tree)
- removal of a node must not leave a ‘gap’ in the tree,

Removal in BST - Pseudocode

method remove (key)

I if the tree is empty return false

II Attempt to locate the node containing the target using the binary search algorithm

if the target is not found return false

else the target is found, so remove its node:

Case 1: if the node has 2 empty subtrees

replace the link in the parent with null

Case 2: if the node has a left and a right subtree

- replace the node's value with the max value in the left subtree
- delete the max node in the left subtree

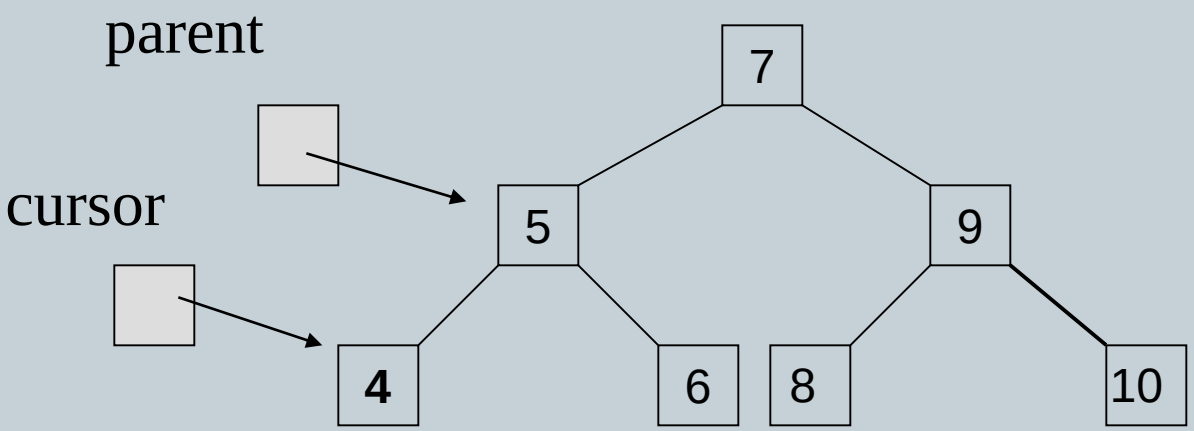
Removal in BST - Pseudocode

Case 3: if the node has no left child
- link the parent of the node
- to the right (non-empty) subtree

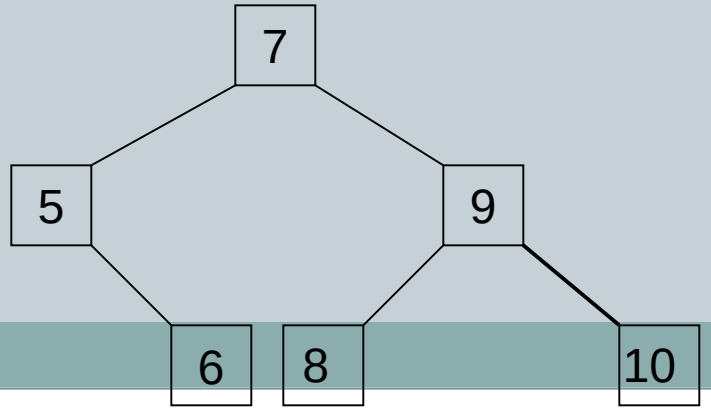
Case 4: if the node has no right child
- link the parent of the target
- to the left (non-empty) subtree

Removal in BST: Example

Case 1: removing a node with 2 EMPTY SUBTREES



Removing 4replace the link in the parent with null



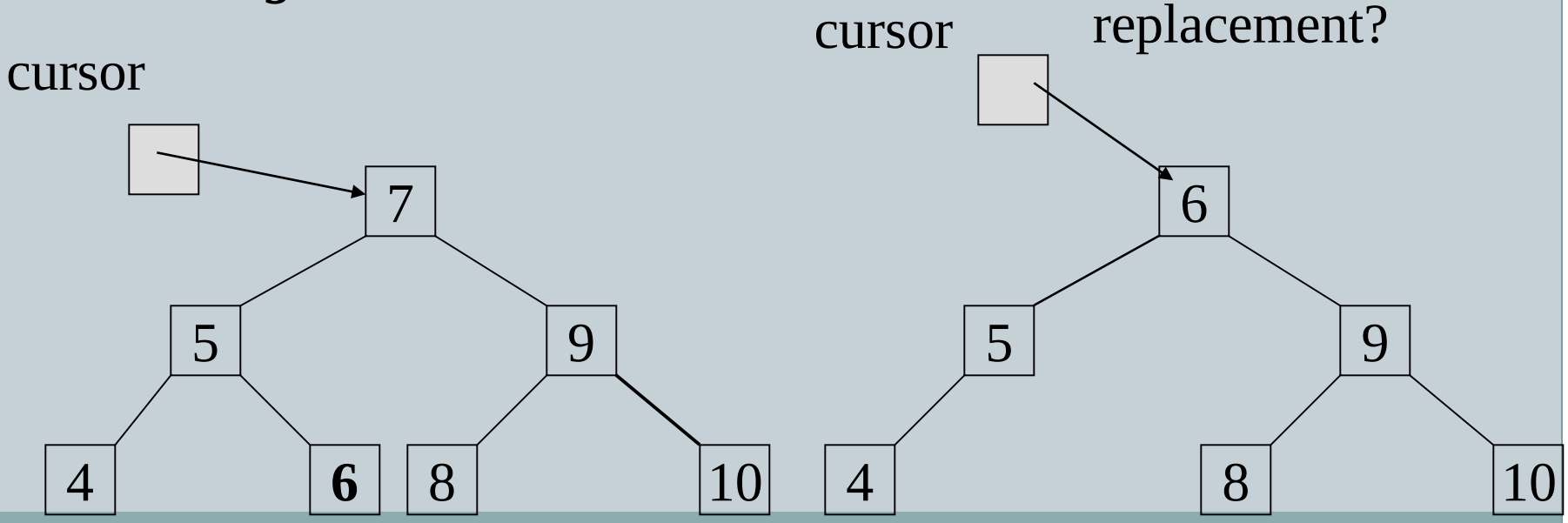
Removal in BST: Example

Case 2: removing a node with 2 SUBTREES

- replace the node's value with the max value in the left subtree
- delete the max node in the left subtree

What other element can be used as replacement?

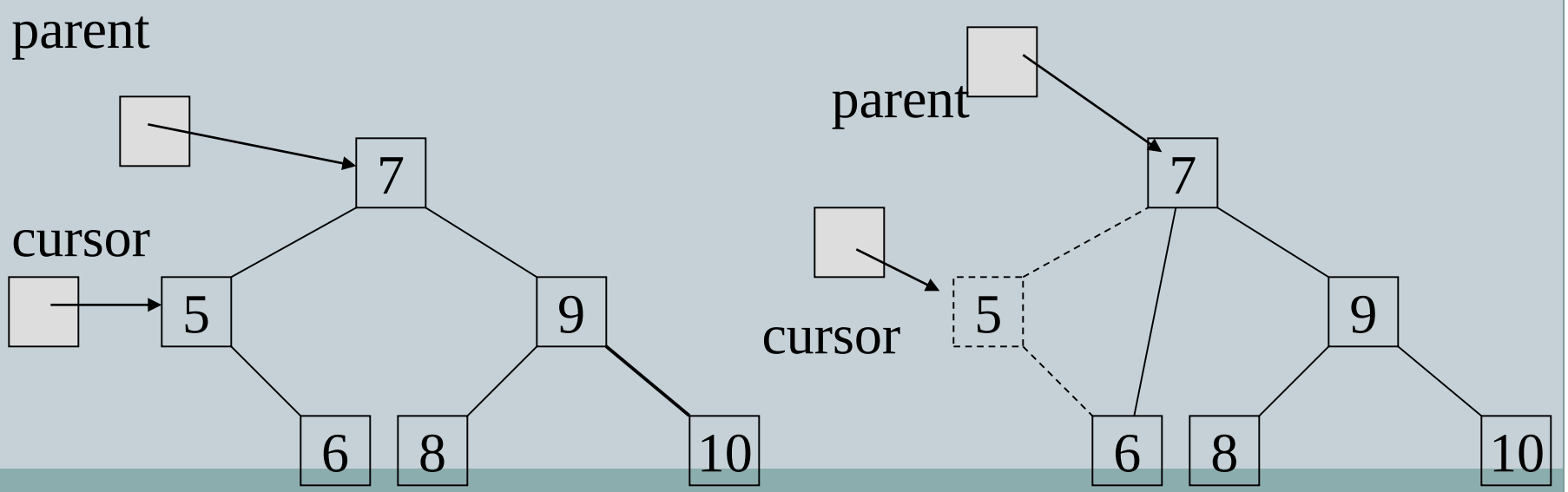
Removing 7



Removal in BST: Example

Case 3: removing a node with 1 EMPTY SUBTREE

the node has no left child:
link the parent of the node to the right (non-empty) subtree

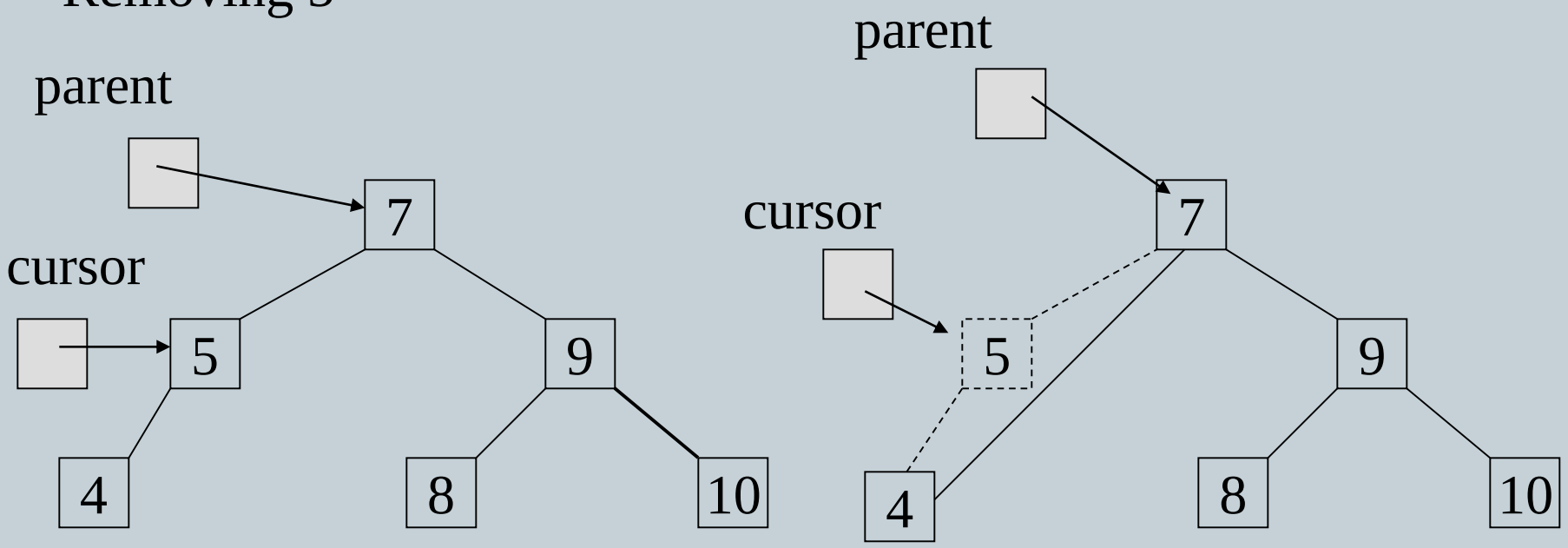


Removal in BST: Example

Case 4 : removing a node with 1 EMPTY SUBTREE

the node has no right child:
link the parent of the node to the left (non-empty) subtree

Removing 5



Threaded Binary Tree

- A Threaded Binary Tree is a binary tree in which every node that does not have a right child has a THREAD (in actual sense, a link) to its INORDER successor.
- Left thread to its INORDER Predecessor
- By doing this threading we avoid the recursive method of traversing a Tree, which makes use of stacks and consumes a lot of memory and time. The node structure for a threaded binary tree varies a bit and its like this -

- Binary trees have a lot of wasted space: the leaf nodes each have 2 null pointers
- We can use these pointers to help us in **inorder** traversals
- We have the pointers reference to the next node in an inorder traversal; called *threads*
- We need to know if a pointer is an actual link or a thread, so we keep a boolean for each pointer

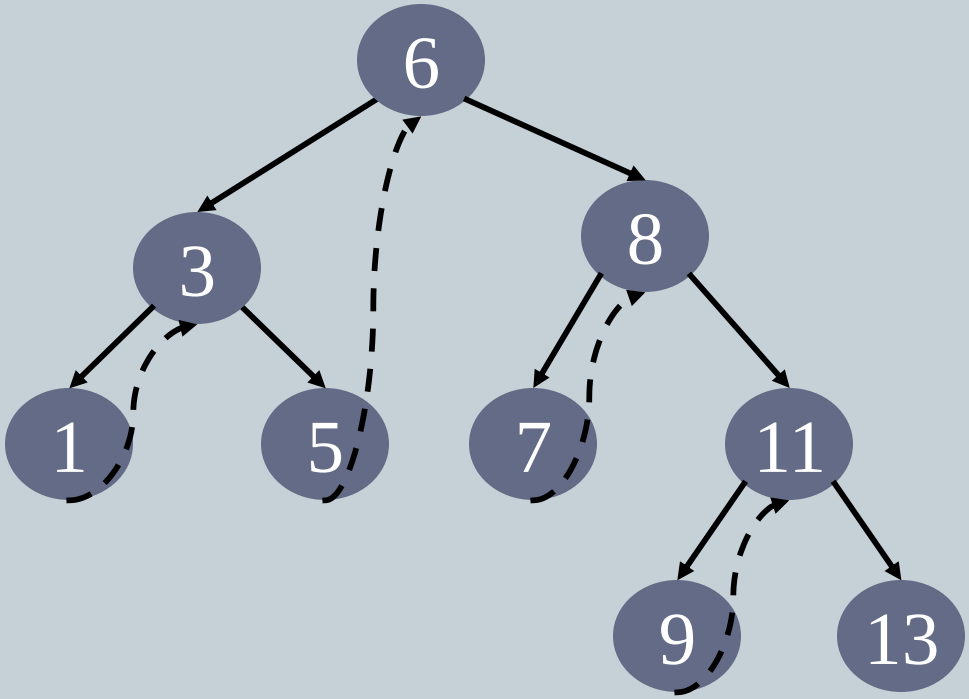
Threaded Binary Tree

- A binary tree is *threaded* by making all **right child pointers** that would normally be null, point to the **inorder successor** of the node

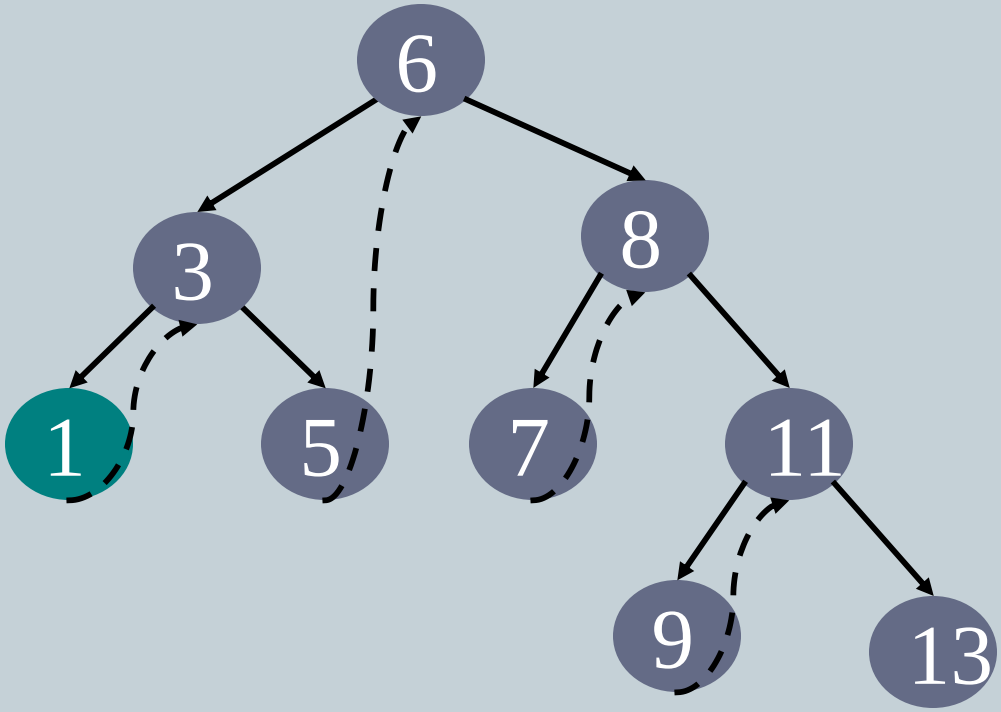
- Example code:

```
class Node
{
    int value;
    Node left, right;
    boolean leftThread, rightThread;
}
```

Threaded Binary Tree Example



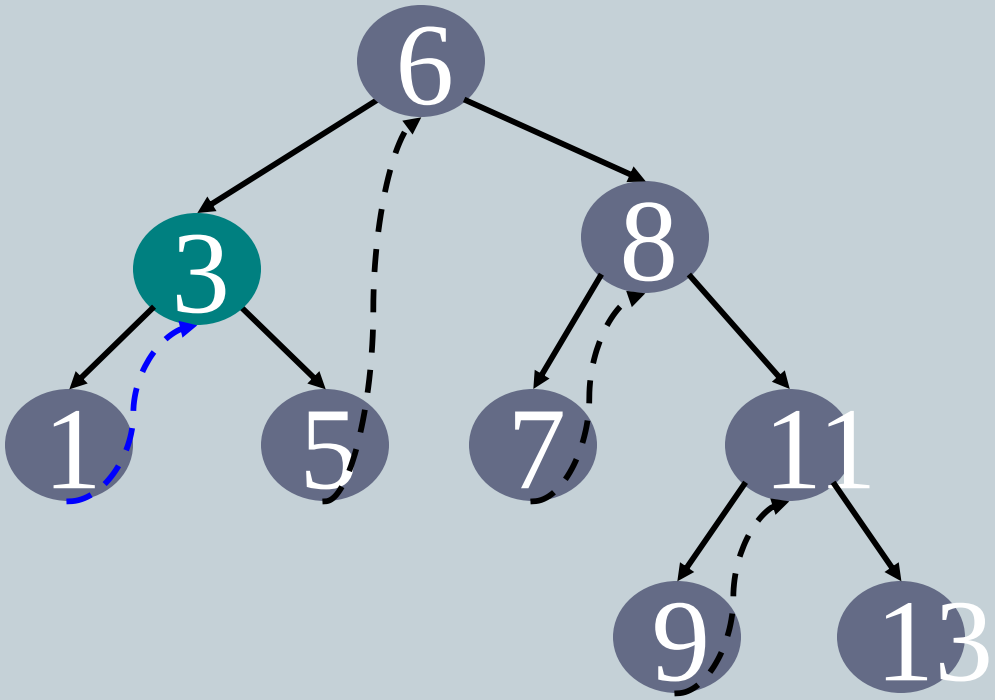
Threaded Tree Traversal



Output
1

Start at leftmost node, print it

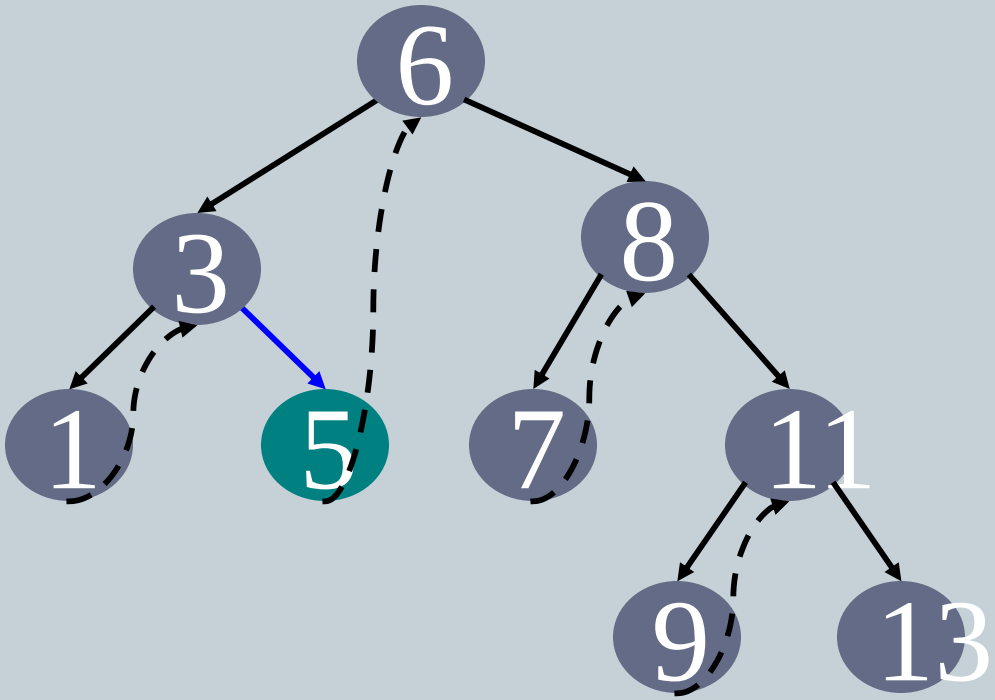
Threaded Tree Traversal



Output
1
3

Follow thread to right, print node

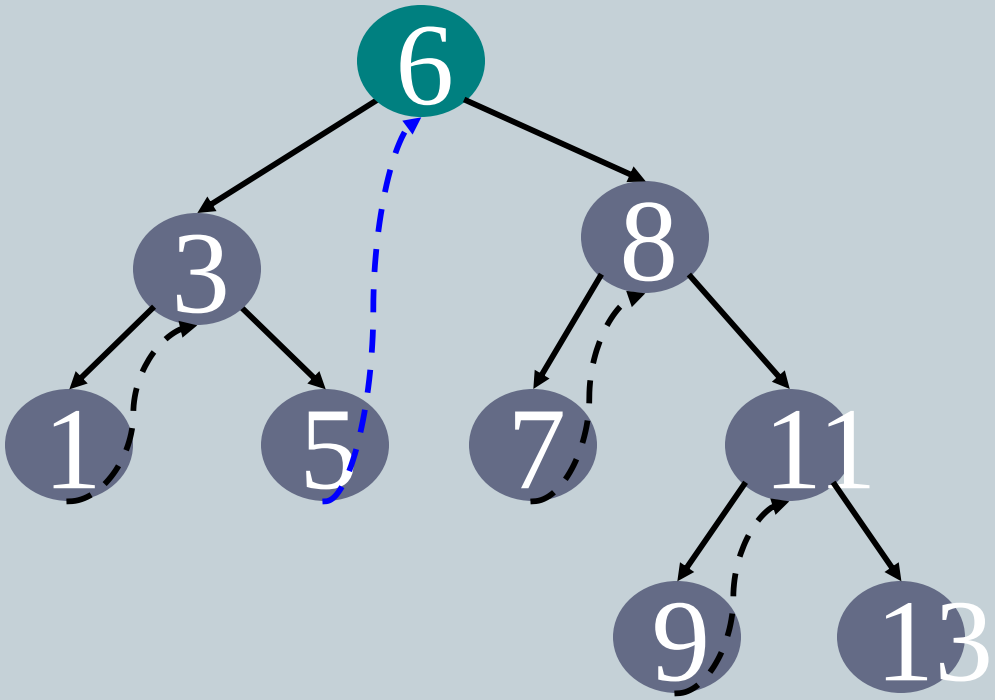
Threaded Tree Traversal



Output
1
3
5

Follow link to right, go to leftmost node and print

Threaded Tree Traversal



Output

1

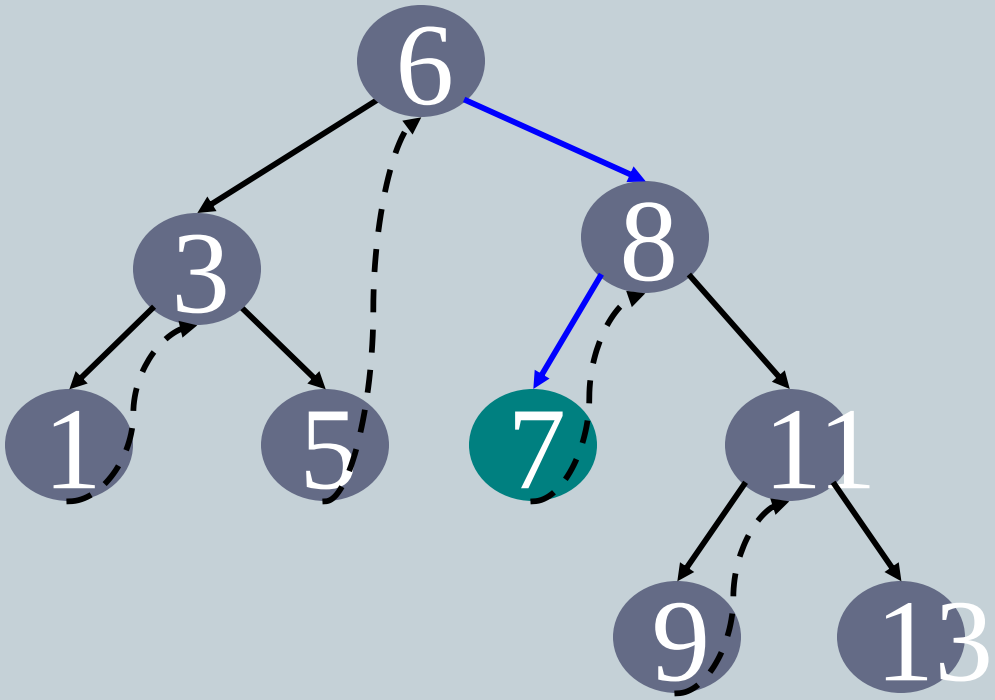
3

5

6

Follow thread to right, print node

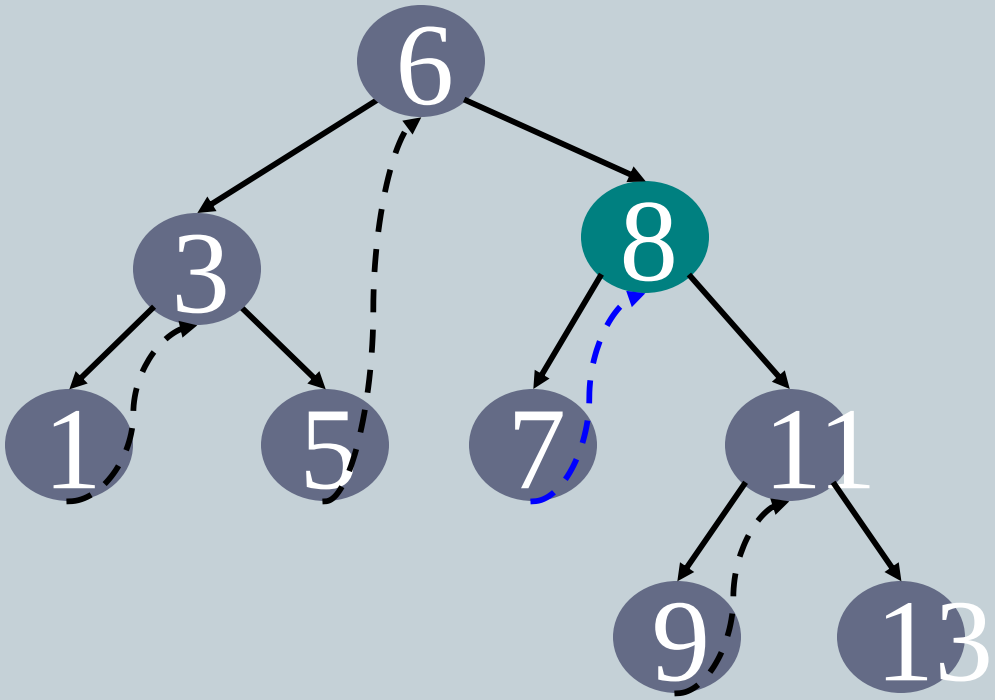
Threaded Tree Traversal



Output
1
3
5
6
7

Follow link to right, go to leftmost node and print

Threaded Tree Traversal

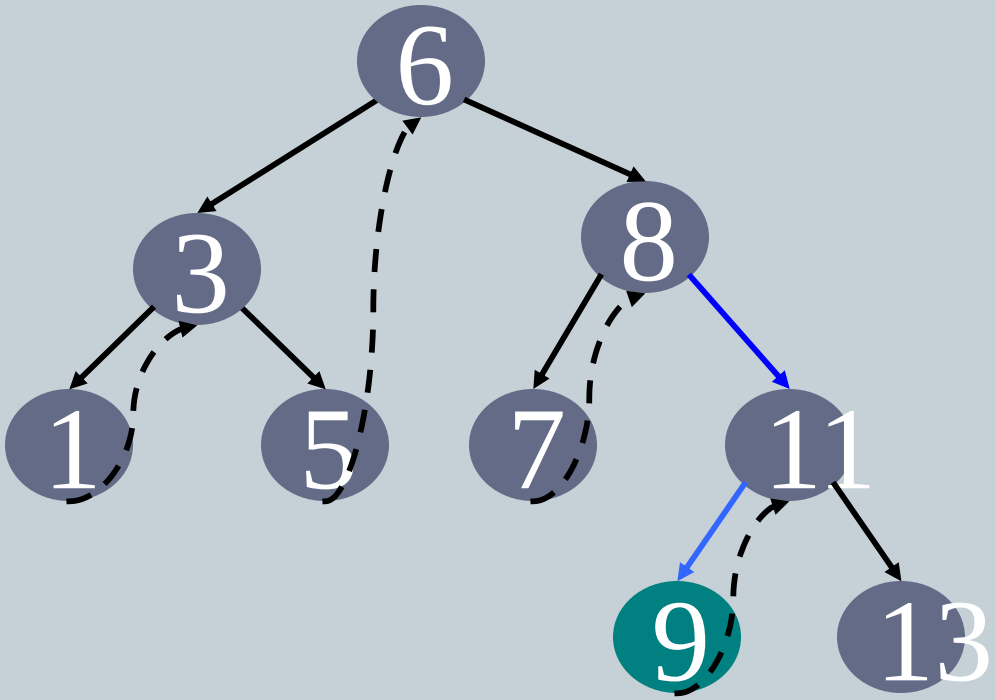


Output

1
3
5
6
7
8

Follow thread to right, print node

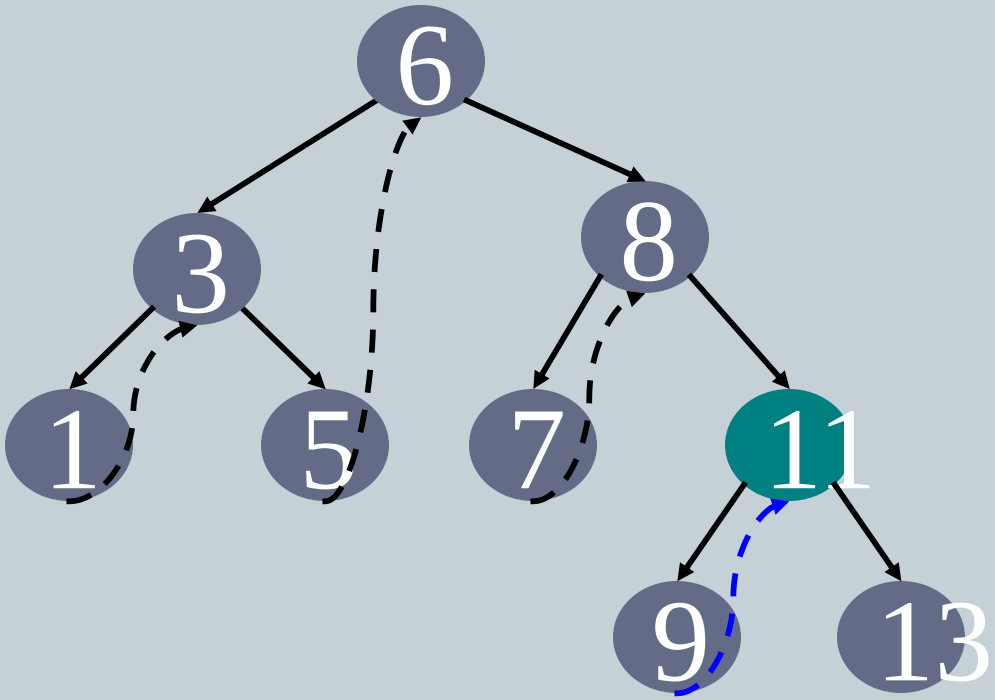
Threaded Tree Traversal



Output
1
3
5
6
7
8
9

Follow link to right, go to leftmost node and print

Threaded Tree Traversal

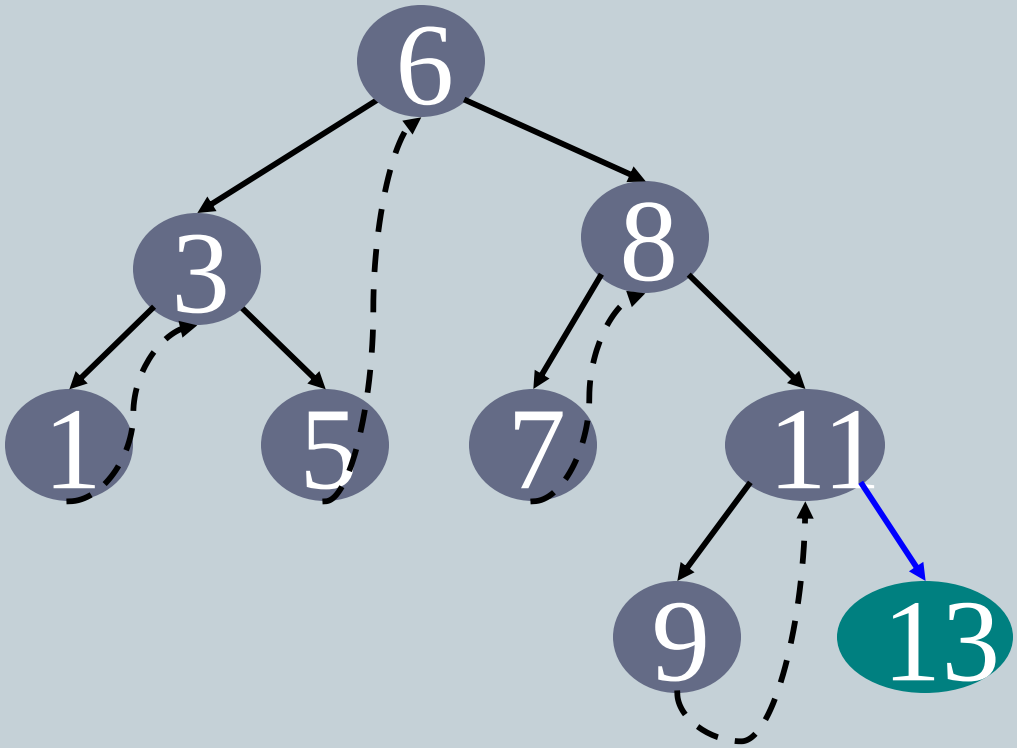


Output

1
3
5
6
7
8
9
11

Follow thread to right, print node

Threaded Tree Traversal



Output
1
3
5
6
7
8
9
11
13

Follow link to right, go to leftmost node and print

Streams and File I/O

Objectives

- become familiar with the concept of an I/O stream
- understand the difference between binary files and text files
- learn how to save data in a file
- learn how to read data from a file
- learn how use the classes `ObjectOutputStream` and `ObjectInputStream` to read and write class objects with binary files

Outline

- Overview of Streams and File I/O
- Text-File I/O
- Using the `File` Class
- Basic Binary-File I/O
- Object I/O with Object Streams

I/O Overview

- *I/O* = Input/Output
- In this context it is input to and output from programs
- Input can be from keyboard or a file
- Output can be to display (screen) or a file
- Advantages of file I/O
  - permanent copy (Preserves copy of your data)
- Significance of File I/O:
  - output from one program can be input to another
  - input can be automated (rather than entered manually)

Streams

- ***Stream***: an object that either delivers data to its destination (screen, file, etc.) or that takes data from a source (keyboard, file, etc.)
  - it acts as a buffer between the data source and destination
  - INPUT (Source) -→ WRAP INTO STREAM OBJECT-→ OUTPUT (Destination)
- ***Input stream***: a stream that provides input to a program
  - `System.in` is an input stream (Ex. scanf, cin)
  - Object- Converts Human readable code to machine code
- ***Output stream***: a stream that accepts output from a program
  - `System.out` is an output stream (Ex. Printf, cout)
- A stream connects a program to an I/O object (Program –HLL and I/O Object- Machine Level Language)
  - `System.out` connects a program to the screen (Ex. System.out.println)
  - `System.in` connects a program to the keyboard

Binary Versus Text Files

- *All* data and programs are ultimately just zeros and ones
  - each digit can have one of two values, hence *binary*
  - *bit* is one binary digit
  - *byte* is a group of eight bits
- *Text files*: the bits represent printable characters
  - one byte per character for ASCII, the most common code
  - for example, Java source files are text files
  - so is any file created with a "text editor"
- *Binary files*: the bits represent other types of encoded information, such as executable instructions or numeric data
  - these files are easily read by the computer but not humans
  - they are *not* "printable" files
    - actually, you *can* print them, but they will be unintelligible
    - "printable" means "easily readable by humans when printed"

Java: Text Versus Binary Files

- Text files are more readable by humans
- Binary files are more efficient
  - computers read and write binary files more easily than text
  - Binary files created by C, C++ are not portable(.obj are readable on specific architecture)
- Java binary files are portable (.class has byte codes and it works on any platform)
  - they can be used by Java on different machines
  - Reading and writing binary files is normally done by a program
  - text files are used only to communicate with humans

Java Text Files

- Source files
- Occasionally input files
- Occasionally output files

Java Binary Files

- Executable files (created by compiling source files)
- Usually input files
- Usually output files

Text Files vs. Binary Files

- Memory Requirement-
- Number: 127 (decimal)
  - Text file
    - Three bytes(character): “1”, “2”, “7”
    - ASCII (decimal): 49, 50, 55
    - ASCII (octal): 61, 62, 67
    - ASCII (binary): 00110001, 00110010, 00110111
  - Binary file:
    - One byte (`byte`): 01111110
    - Two bytes (`short`): 00000000 01111110
    - Four bytes (`int`): 00000000 00000000 00000000 01111110

Text Vs. Binary Files

- Text files are bulky as compared to binary files.
- Ascii values of space, enter key also get stored in text files.
- EOF when get detected that is explicitly get stored in text files but not in binary file.
- Binary files are more efficient than Text files.

Text File I/O

- Important classes for text file **output** (to the file)
  - **`PrintWriter (Note-First letter should be capital otherwise it gives error)`**
  - **`FileOutputStream`** [or **`FileWriter`**]
- Important classes for text file **input** (from the file):
  - **`BufferedReader`**
  - **`FileReader`**
- **`FileOutputStream`** and **`FileReader`** take file names as arguments.
- **`PrintWriter`** and **`BufferedReader`** provide useful methods for easier writing and reading.
- To use these classes your program needs a line like the following:

```
import java.io.*;
```

Buffering

- Not buffered: each byte is read/written from/to disk as soon as possible
  - "little" delay for each byte
  - A disk operation per byte---higher overhead
- Buffered: reading/writing in "chunks"
  - Some delay for some bytes
    - Assume 16-byte buffers
    - Reading: access the first 4 bytes, need to wait for all 16 bytes are read from disk to memory
    - Writing: save the first 4 bytes, need to wait for all 16 bytes before writing from memory to disk
  - A disk operation per a buffer of bytes---lower overhead

Every Text File Has Two Essential Things

1. the stream name used by Java
   - `outputStream` (Used for writing in a file) in the example
2. the file name used by the operating system
   - `out.txt (Name of file in which we want to write)` in the example

Text File Output

- To open a text file for output: connect a text file to a stream for writing

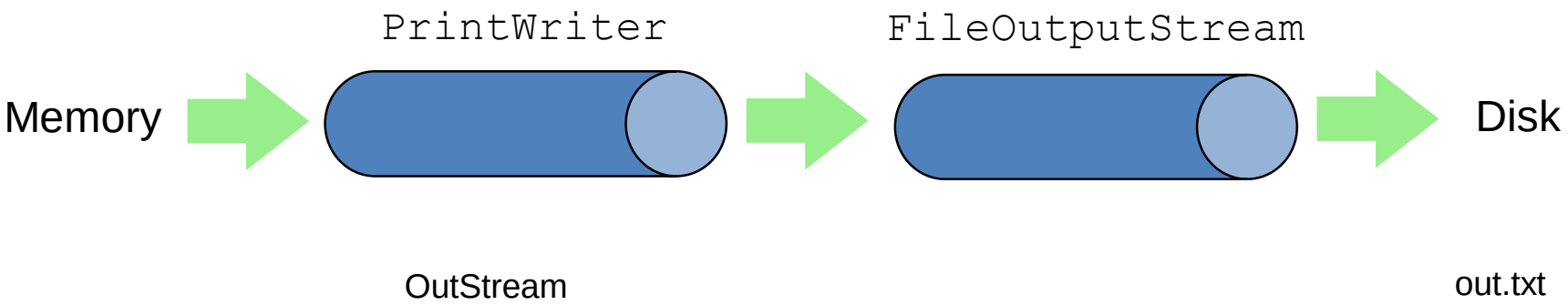
```
PrintWriter outputStream =
   new PrintWriter(new FileOutputStream("out.txt"));
```

- Similar to the long way: (Works from Right to Left Ex. C=5;)

```
FileOutputStream s = new FileOutputStream("out.txt");
PrintWriter outputStream = new PrintWriter(s);
```

- Goal: create a `PrintWriter` object
  - which uses `FileOutputStream` to open a text file
- `FileOutputStream` "connects" `PrintWriter` to a text file.

Output File Streams



PrintWriter smileyOutStream = new PrintWriter(new FileOutputStream(“out.txt”));

Methods for `PrintWriter`

- Similar to methods for `System.out`
- `println`

```
outputStream.println(count + " " + line);
```

- `print`
- `format`
- `flush`: write buffered output to disk
- `close`: close the `PrintWriter` stream (and file)

TextFileOutputDemo

Part 1

```
public static void main(String[] args)
{
    PrintWriter outputStream = null;
    try
    {
        outputStream =
              new PrintWriter(new FileOutputStream("out.txt"));
    }
    catch(FileNotFoundException e)
    {
        System.out.println("Error opening the file out.txt. "
                           + e.getMessage());
        System.exit(0);
    }
```

A `try`-block is a block: `outputStream` would not be accessible to the rest of the method if it were declared inside the `try`-block

Opening the file

Creating a file can cause the `FileNotFound-Exception` if the new file cannot be made.

TextFileOutputDemo

Part 2

```
System.out.println("Enter three lines of text:");
String line = null;
int count;
    for (count = 1; count <= 3; count++)
    {
        line = keyboard.nextLine();
        outputStream.println(count + "
    }
    outputStream.close();
    System.out.println("...                xt.");
}
```

Writing to the file

Closing the file

The `println` method is used with two different streams: `outputStream` and `System.out`

Java Tip: Appending to a Text File

- To add/append to a file instead of replacing it, use a different constructor for **`FileOutputStream`**:

```
outputStream =
   new PrintWriter(new FileOutputStream("out.txt", true));
```

- Second parameter: append to the end of the file if it exists.
- Sample code for letting user tell whether to replace or append:

```
System.out.println("A for append or N for new file:");
char ans = keyboard.next().charAt(0);
boolean append = (ans == 'A' || ans == 'a');
outputStream = new PrintWriter(
      new FileOutputStream("out.txt", append));
```

true if user enters 'A'

Closing a File

- An output file should be closed when you are done writing to it (and an input file should be closed when you are done reading from it).

- Use the `close` method of the class `PrintWriter` (`BufferedReader` also has a `close` method).

- For example, to close the file opened in the previous example:

```
outputStream.close();
```

Text File Input

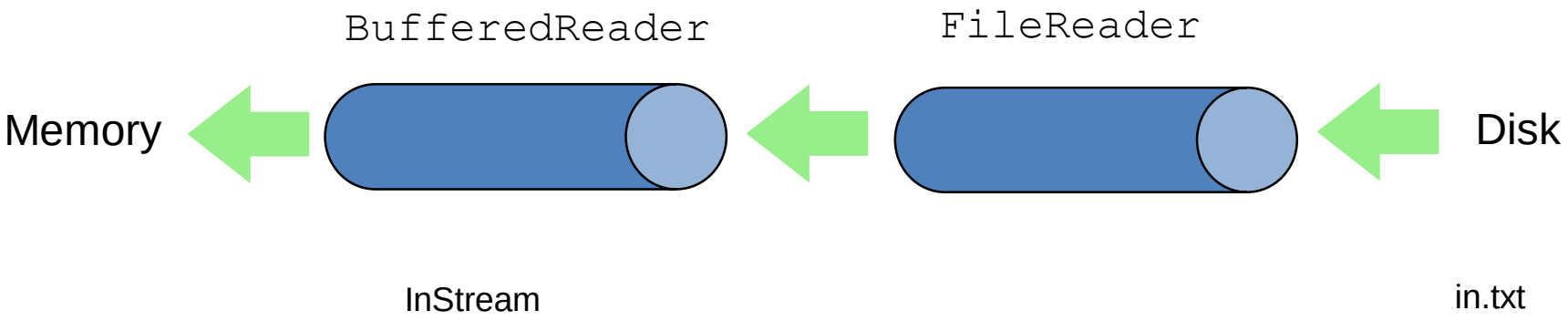
- To open a text file for input: connect a text file to a stream for reading
  - Goal: a `BufferedReader` object,
    - which uses `FileReader` to open a text file
  - `FileReader` "connects" `BufferedReader` to the text file
- For example:

```
BufferedReader smileyInStream =
  new BufferedReader(new FileReader("in.txt"));
```

- Similarly, the long way :

```
FileReader s = new FileReader("in.txt");
BufferedReader InStream = new BufferedReader(s);
```

Input File Streams




```
BufferedReader smileyInStream = new BufferedReader( new FileReader(“in.txt”) );
```

Methods for `BufferedReader`

- `readLine`: read a line into a `String`
- no methods to read numbers directly, so read numbers as `String`s and then convert them (`StringTokenizer` later)
- `read`: read a `char` at a time
- `close`: close `BufferedReader` stream

Exception Handling with File I/O

It handles abrupt termination of your program so it is necessary.

Catching IOExceptions

- `IOException` is a predefined class
- File I/O might throw an `IOException`
- catch the exception in a catch block that at least prints an error message and ends the program
- `FileNotFoundException` is derived from `IOException`
  - therefor any catch block that catches `IOException`s also catches `FileNotFoundException`s
  - put the more specific one first (the derived one) so it catches specifically file-not-found exceptions
  - then you will know that an I/O error is something other than file-not-found

Example: Reading a File Name from the Keyboard

reading a file name from the keyboard

using the file name read from the keyboard

reading data from the file

closing the file

```
public static void main(String[] args)
  {
    String fileName = null;  // outside try block, can be used in catch
    try
    { Scanner keyboard = new Scanner(System.in);
      System.out.println("Enter file name:");
      fileName = keyboard.next();
      BufferedReader inputStream =
        new BufferedReader(new FileReader(fileName));
      String line = null;
      line = inputStream.readLine();
      System.out.println("The first line in " + filename + " is:");
      System.out.println(line);
      // . . . code for reading second line not shown here . . .
      inputStream.close();
    }
    catch(FileNotFoundException e)
    {
      System.out.println("File " + filename + " not found.");
    }
    catch(IOException e)
    {
      System.out.println("Error reading from file " + fileName);
    }
  }
```

Exception.getMessage()

```
try
{
    …
}
catch (FileNotFoundException e)
{
    System.out.println(filename + " not found");
    System.out.println("Exception: " +
                        e.getMessage());
    System.exit(-1);
}
```

Reading Words in a String: Using `StringTokenizer` Class

- There are `BufferedReader` methods to read a line and a character, but not just a single word

- `StringTokenizer` can be used to parse a line into words
  - import `java.util.*`
  - some of its useful methods are shown in the text
    - e.g. test if there are more tokens
  - you can specify *delimiters* (the character or characters that separate words)
    - the default delimiters are "white space" (space, tab, and newline)

Example: **StringTokenizer**

- Display the words separated by any of the following characters: space, new line (\n), period (.) or comma (,).

```
String inputLine = keyboard.nextLine();
StringTokenizer wordFinder =
            new StringTokenizer(inputLine, " \n.,");
//the second argument is a string of the 4 delimiters
while(wordFinder.hasMoreTokens())
{
   System.out.println(wordFinder.nextToken());
}
```

Entering "`Question,2b.or !tooBee.`" gives this output:

```
Question
2b
or
!tooBee
```

Testing for End of File in a Text File

- When `readLine` tries to read beyond the end of a text file it returns the special value *`null`*
  - so you can test for `null` to stop processing a text file
- `read` returns -1 when it tries to read beyond the end of a text file
- Neither of these two methods (`read` and `readLine`) will throw an `EOFException`.

Example: Using Null to Test for End-of-File in a Text File

When using **`readLine`** test for `null`

Excerpt from `TextEOFDemo`

```
int count = 0;
String line = inputStream.readLine();
while (line != null)
{
   count++;
   outputStream.println(count + " " + line);
   line = inputStream.readLine();
}
```

When using **`read`** test for -1

Using Path Names

- ***Path name***—gives name of file and tells which directory the file is in
- ***Relative path name***—gives the path starting with the directory that the program is in
- Typical UNIX path name:

```
/user/smith/home.work/java/FileClassDemo.java
```

- Typical Windows path name:

```
D:\Work\Java\Programs\FileClassDemo.java
```

- When a backslash is used in a quoted string it must be written as two backslashes since backslash is the escape character:

```
"D:\\Work\\Java\\Programs\\FileClassDemo.java"
```

- Java will accept path names in UNIX or Windows format, regardless of which operating system it is actually running on.

`File` Class `[java.io]`

- Acts like a wrapper class for file names
- A file name like "`numbers.txt`" has only `String` properties
- `File` has some very useful methods
  - `exists`: tests if a file already exists
  - `canRead`: tests if the OS will let you read a file
  - `canWrite`: tests if the OS will let you write to a file
  - `delete`: deletes the file, returns true if successful
  - `length`: returns the number of bytes in the file
  - `getName`: returns file name, excluding the preceding path
  - `getPath`: returns the path name—the full name

```
File numFile = new File("numbers.txt");
if (numFile.exists())
   System.out.println(numfile.length());
```

`File` Objects and Filenames

- `FileInputStream` and `FileOutputStream` have constructors that take a `File` argument as well as constructors that take a `String` argument

```
PrintWriter smileyOutStream = new PrintWriter(new
   FileOutputStream("smiley.txt"));

File smileyFile = new File("smiley.txt");
if (smileyFile.canWrite())
   PrintWriter smileyOutStream = new PrintWriter(new
   FileOutputStream(smileyFile));
```

Alternative with Scanner

- Instead of `BufferedReader` with `FileReader`, then `StringTokenizer`
- Use `Scanner` with `File`:

```
Scanner inFile =
    new Scanner(new File("in.txt"));
```

- Similar to `Scanner` with `System.in`:

```
Scanner keyboard =
    new Scanner(System.in);
```

Reading in `int`'s

```
Scanner inFile = new Scanner(new File("in.txt"));
int number;
while (inFile.hasInt())
  {
     number = inFile.nextInt();
     // …
  }
```

Reading in lines of characters

```
Scanner inFile = new Scanner(new File(“in.txt"));
String line;
while (inFile.hasNextLine())
  {
     line = inFile.nextLine();
     // …
  }
```

Multiple types on one line

```
// Name, id, balance
Scanner inFile = new Scanner(new File("in.txt"));
while (inFile.hasNext())
  {
    name = inFile.next();
    id = inFile.nextInt();
    balance = inFile.nextFloat();
    // …  new Account(name, id, balance);
  }
--------------------
String line;
while (inFile.hasNextLine())
  {
    line = inFile.nextLine();
    Scanner parseLine = new Scanner(line) // Scanner again!
    name = parseLine.next();
    id = parseLine.nextInt();
    balance = parseLine.nextFloat();
    // …  new Account(name, id, balance);
  }
```

Multiple types on one line

```
// Name, id, balance
Scanner inFile = new Scanner(new File(“in.txt"));
String line;
while (inFile.hasNextLine())
  {
    line = inFile.nextLine();
    Account account = new Account(line);
  }
--------------------
public Account(String line) // constructor
{
  Scanner accountLine = new Scanner(line);
  _name = accountLine.next();
  _id = accountLine.nextInt();
  _balance = accountLine.nextFloat();
}
```

`BufferedReader` vs `Scanner` (parsing primitive types)

- `Scanner`
  - `nextInt(), nextFloat(),` … for parsing types
- `BufferedReader`
  - `read(), readLine(),` … none for parsing types
  - needs `StringTokenizer` then wrapper class methods like `Integer.parseInt(token)`

`BufferedReader` vs `Scanner` (Checking End of File/Stream (EOF))

- `BufferedReader`
  - `readLine()` returns `null`
  - `read()` returns -1
- `Scanner`
  - `nextLine()` throws exception
  - needs `hasNextLine()` to check first
  - `nextInt(), hasNextInt(), ...`

```
BufferedReader inFile = …
line = inFile.readline();
while (line != null)
{
  // …
  line = inFile.readline();
}
```

```
Scanner inFile = …
while (inFile.hasNextLine())
{
  line = infile.nextLine();
  // …
}
```

```
BufferedReader inFile = …
line = inFile.readline();
while (line != null)
{
  // …
  line = inFile.readline();
}

--------------------

BufferedReader inFile = …
while ((line = inFile.readline()) != null)
{
  // …
}
```

suggestion

- Use `Scanner` with `File`
  - `new Scanner(new File("in.txt"))`
- Use `hasNext…()` to check for EOF
  - `while (inFile.hasNext…())`
- Use `next…()` to read
  - `inFile.next…()`
- Simpler and you are familiar with methods for `Scanner`

suggestion cont…

- File input
  - `Scanner inFile =`
    `new Scanner(new File("in.txt"));`
- File output
  - `PrintWriter outFile =`
    `new PrintWriter(new File("out.txt"));`
  - `outFile.print(), println(), format(), flush(), close(), …`
- http://www.cs.fit.edu/~pkc/classes/cse1001/FileIO/FileIONew.java

Basic Binary File I/O

- Important classes for binary file **output** (to the file)
  - **`ObjectOutputStream`**
  - **`FileOutputStream`**
- Important classes for binary file **input** (from the file):
  - **`ObjectInputStream`**
  - **`FileInputStream`**
- Note that **`FileOutputStream`** and **`FileInputStream`** are used only for their constructors, which can take file names as arguments.
  - **`ObjectOutputStream`** and **`ObjectInputStream`** cannot take file names as arguments for their constructors.
- To use these classes your program needs a line like the following:

```
import java.io.*;
```

Java File I/O: Stream Classes

- `ObjectInputStream` and `ObjectOutputStream`:
  - have methods to either read or write data one byte at a time
  - automatically convert numbers and characters into binary
    - binary-encoded numeric files (files with numbers) are not readable by a text editor, but store data more efficiently
- Remember:
  - *input* means data into a program, not the file
  - similarly, *output* means data out of a program, not the file

When Using **`ObjectOutputStream`** to Output Data to Files:

- The output files are binary and can store any of the primitive data types (`int`, `char`, `double`, etc.) and the `String` type
- The files created can be read by other Java programs but are not printable
- The Java I/O library must be imported by including the line: `import java.io.*;`
  - it contains `ObjectOutputStream` and other useful class definitions
- An `IOException` might be thrown

Handling **IOException**

- `IOException` cannot be ignored
  - either handle it with a catch block
  - or defer it with a `throws`-clause

We will put code to open the file and write to it in a `try`-block and write a `catch`-block for this exception :

```
catch(IOException e)
{
  System.out.println("Problem with output...";
}
```

Opening a New Output File

- The file name is given as a `String`
  - file name rules are determined by your operating system

- Opening an output file takes two steps
  1. Create a `FileOutputStream` object associated with the file name `String`
  2. Connect the `FileOutputStream` to an `ObjectOutputStream` object

  This can be done in one line of code

Example: Opening an Output File

To open a file named `numbers.dat`:

```
ObjectOutputStream outputStream =
  new ObjectOutputStream(
  new FileOutputStream("numbers.dat"));
```

- The constructor for `ObjectOutputStream` requires a `FileOutputStream` argument
- The constructor for `FileOutputStream` requires a `String` argument
  - the `String` argument is the output file name
- The following two statements are equivalent to the single statement above:

```
FileOutputStream middleman =
  new FileOutputStream("numbers.dat");
ObjectOutputStream outputStream =
  new ObjectOutputSteam(middleman);
```

Some `ObjectOutputStream` Methods

- You can write data to an output file after it is connected to a stream class
  - Use methods defined in `ObjectOutputStream`
    - `writeInt(int n)`
    - `writeDouble(double x)`
    - `writeBoolean(boolean b)`
    - etc.
    - See the text for more

- Note that each write method throws `IOException`
  - eventually we will have to write a catch block for it

- Also note that each write method includes the modifier `final`
  - `final` methods cannot be redefined in derived classes

Closing a File

- An Output file should be closed when you are done writing to it
- Use the `close` method of the class `ObjectOutputStream`
- For example, to close the file opened in the previous example:

```
outputStream.close();
```

- If a program ends normally it will close any files that are open

Writing a Character to a File: an Unexpected Little Complexity

- The method `writeChar` has an annoying property:
  - it takes an `int`, not a `char`, argument
- But it is easy to fix:
  - just cast the character to an int
- For example, to write the character 'A' to the file opened previously:

```
outputStream.writeChar((int) 'A');
```

- Or, just use the automatic conversion from `char` to `int`

Writing a **`boolean`** Value to a File

- `boolean` values can be either of two values, `true` or `false`

- `true` and `false` are not just names for the values, they actually are of type `boolean`

- For example, to write the `boolean` value `false` to the output file:

```
outputStream.writeBoolean(false);
```

Writing Strings to a File: Another Little Unexpected Complexity

- Use the `writeUTF` method to output a value of type `String`
  - there is no `writeString` method
- UTF stands for Unicode Text Format
  - a special version of Unicode
- Unicode: a text (printable) code that uses 2 bytes per character
  - designed to accommodate languages with a different alphabet or no alphabet (such as Chinese and Japanese)
- ASCII: also a text (printable) code, but it uses just 1 byte per character
  - the most common code for English and languages with a similar alphabet
- UTF is a modification of Unicode that uses just one byte for ASCII characters
  - allows other languages without sacrificing efficiency for ASCII files

When Using `ObjectInputStream` to Read Data from Files:

- Input files are binary and contain any of the primitive data types (`int`, `char`, `double`, etc.) and the `String` type
- The files can be read by Java programs but are not printable
- The Java I/O library must be imported including the line:
  `import java.io.*;`
  - it contains `ObjectInputStream` and other useful class definitions
- An `IOException` might be thrown

Opening a New Input File

- Similar to opening an output file, but replace "output" with "input"
- The file name is given as a `String`
  - file name rules are determined by your operating system
- Opening a file takes two steps
  1. Creating a `FileInputStream` object associated with the file name `String`
  2. Connecting the `FileInputStream` to an `ObjectInputStream` object
- This can be done in one line of code

Example: Opening an Input File

To open a file named `numbers.dat`:

```
ObjectInputStream inStream =
  new ObjectInputStream (new
  FileInputStream("numbers.dat"));
```

- The constructor for `ObjectInputStream` requires a `FileInputStream` argument
- The constructor for `FileInputStream` requires a `String` argument
  - the `String` argument is the input file name
- The following two statements are equivalent to the statement at the top of this slide:

```
FileInputStream middleman =
  new FileInputStream("numbers.dat");
ObjectInputStream inputStream =
  new ObjectInputStream (middleman);
```

Some **`ObjectInputStream`** Methods

- For every output file method there is a corresponding input file method
- You can read data from an input file after it is connected to a stream class
  - Use methods defined in `ObjectInputStream`
    - `readInt()`
    - `readDouble()`
    - `readBoolean()`
    - etc.
    - See the text for more
- Note that each write method throws `IOException`
- Also note that each write method includes the modifier `final`

Input File Exceptions

- A `FileNotFoundException` is thrown if the file is not found when an attempt is made to open a file
- Each read method throws `IOException`
  - we still have to write a catch block for it
- If a read goes beyond the end of the file an `EOFException` is thrown

Avoiding Common **`ObjectInputStream`** File Errors

There is no error message (or exception) if you read the wrong data type!

- Input files can contain a mix of data types
  - it is up to the programmer to know their order and use the correct read method
- `ObjectInputStream` works with binary, not text files
- As with an output file, close the input file when you are done with it

Common Methods to Test for the End of an Input File

- A common programming situation is to read data from an input file but not know how much data the file contains

- In these situations you need to check for the end of the file

- There are three common ways to test for the end of a file:
  1. Put a sentinel value at the end of the file and test for it.
  2. Throw and catch an end-of-file exception.
  3. Test for a special character that signals the end of the file (text files often have such a character).

The `EOFException` Class

- Many (but not all) methods that read from a file throw an end-of-file exception (`EOFException`) when they try to read beyond the file
  - all the `ObjectInputStream` methods in Display 9.3 do throw it
- The end-of-file exception can be used in an "infinite" (`while(true)`) loop that reads and processes data from the file
  - the loop terminates when an `EOFException` is thrown
- The program is written to continue normally after the `EOFException` has been caught

Using **EOFException**

`main` method from `EOFExceptionDemo`

```
try
{
   ObjectInputStream inputStream =
     new ObjectInputStream(new FileInputStream("numbers.dat"));
   int n;

   System.out.println("Reading ALL the integers");
   System.out.println("in the file numbers.dat.");
   try
   {
      while (true)
      {
         n = inputStream.readInt();
         System.out.println(n);
      }
   }
   catch(EOFException e)
   {
      System.out.println("End of reading from file.");
   }
   inputStream.close();
}
catch(FileNotFoundException e)
{
   System.out.println("Cannot find file numbers.dat.");
}
catch(IOException e)
{
   System.out.println("Problem with input from file numbers.dat.");
}
```

Intentional "infinite" loop to process data from input file

Loop exits when end-of-file exception is thrown

Processing continues after `EOFException`: the input file is closed

Note order of catch blocks: the most specific is first and the most general last

Character and Byte Streams In Java

- In Java, a **byte** is not the same thing as a **char** .
- Therefore a byte stream is different from a character stream.
- So, Java defines two types of streams: **Byte Streams** and **Character Streams** .

Character Streams

- A **character stream** will read a file character by character
- Character Stream is a higher level concept than **Byte Stream** .
- A Character Stream is, effectively, a Byte Stream that has been wrapped with logic that allows it to output characters from a specific **encoding** .
- That means, a character stream needs to be given the file's encoding in order to work properly as everything is stored as a character or string.
- Java **Character** streams are used to perform input and output for 16-bit Unicode
- Character stream can support all types of character sets ASCII, Unicode, UTF-8, UTF-16 etc.
- All character stream classes are descended from **Reader** and **Writer** .
- **When to use:**
- To read character streams either from Socket or File of characters

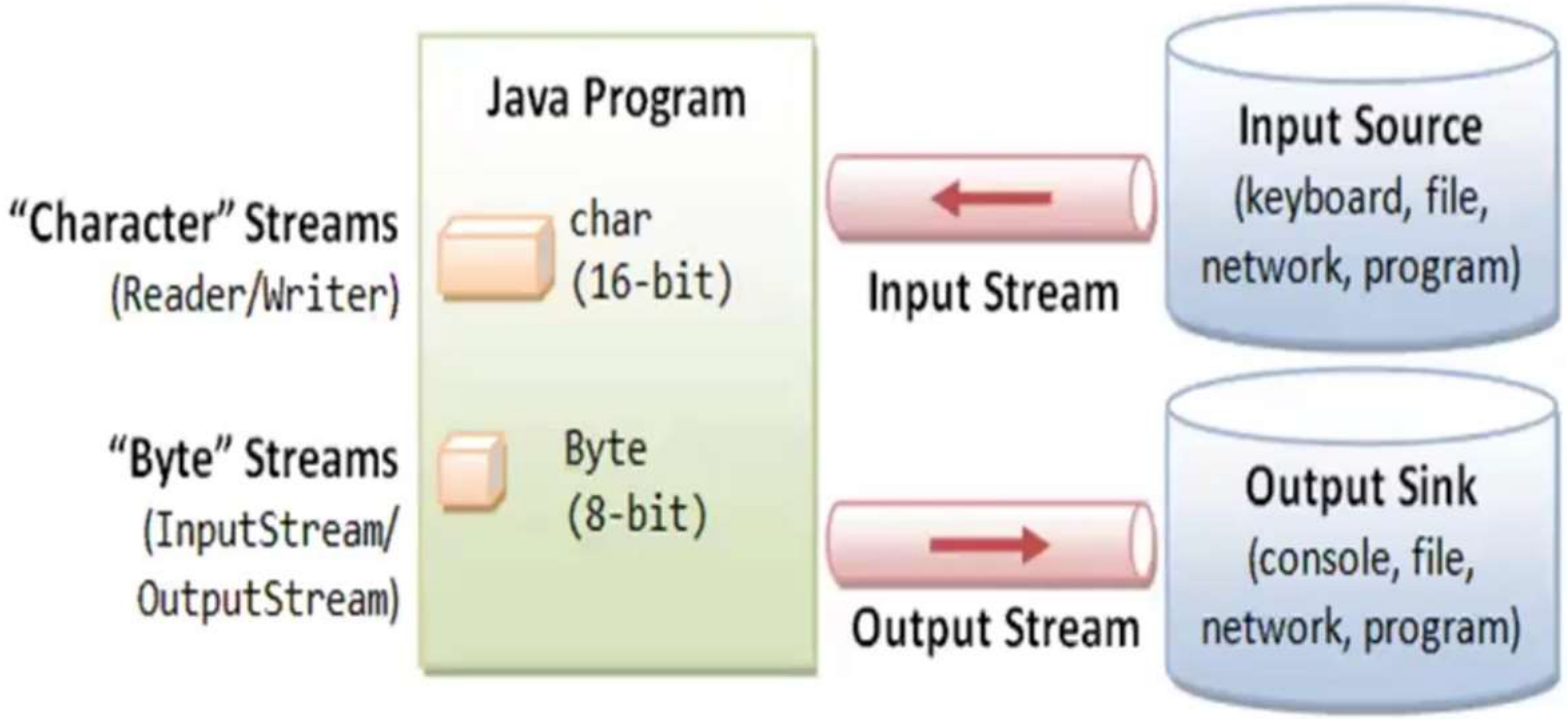
Byte Streams

- A **byte** stream access the file byte by byte.
- Java programs use byte streams to perform input and output of **8-bit/** bytes.
- It is suitable for any kind of file, however not quite appropriate for text files.
- For example, if the file is using a **unicode encoding** and a character is represented with two bytes, the byte stream will treat these separately and you will need to do the conversion yourself.
- Byte oriented streams do not use any **encoding scheme** while Character oriented streams use character encoding scheme(UNICODE).
- All byte stream classes are descended from **InputStream** and **OutputStream** .

Byte Streams:

- **When to use:**
  - Byte streams should only be used for the most primitive I/O
- **When not to use:**
  - You should not use Byte stream to read Character streams
  - e.g. To read a text file

Character and Byte Streams



Summary

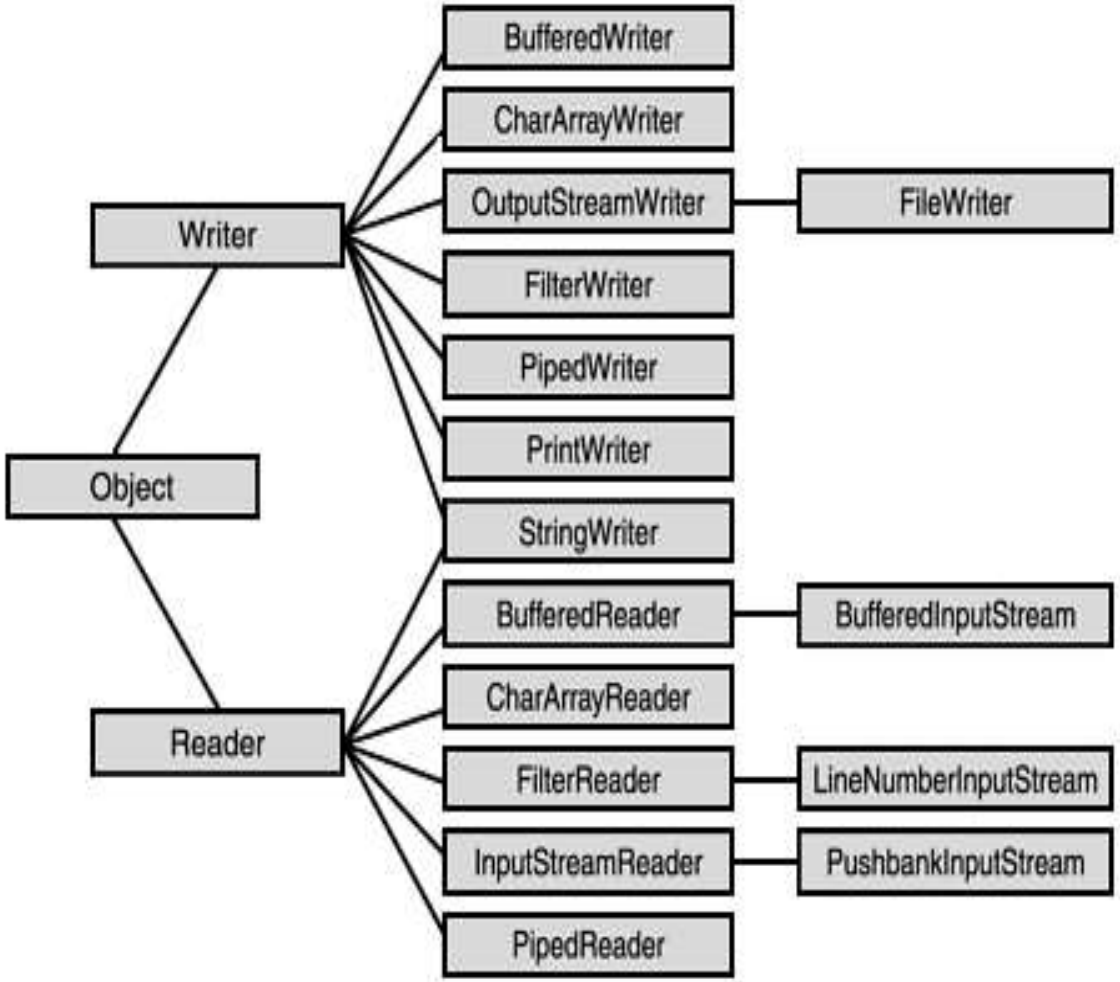
- Character oriented are tied to datatype. Only string type or character type can be read through it while byte oriented are not tied to any datatype, data of any datatype can be read(except string) just you have to specify it.

- Character oriented reads character by character while byte oriented reads byte by byte.

- Character oriented streams use character encoding scheme(UNICODE) while byte oriented do not use any encoding scheme.

- Character oriented streams are also known as reader and writer streams. Byte oriented streams are known as data streams-Data input stream and Data output stream.

Character Streams



File Writer

Constructors of FileWriter class

Constructor	Description
FileWriter(String file)	Creates a new file. It gets file name in string.
FileWriter(File file)	Creates a new file. It gets file name in File object.

Methods of FileWriter class

Method	Description
void write(String text)	It is used to write the string into FileWriter.
void write(char c)	It is used to write the char into FileWriter.
void write(char[] c)	It is used to write char array into FileWriter.
void flush()	It is used to flushes the data of FileWriter.
void close()	It is used to close the FileWriter.

PrintWriter

- used to print the formatted representation of objects to the text-output stream.

Method	Description
void println(boolean x)	It is used to print the boolean value.
void println(char[] x)	It is used to print an array of characters.
void println(int x)	It is used to print an integer.
PrintWriter append(char c)	It is used to append the specified character to the writer.
PrintWriter append(CharSequence ch)	It is used to append the specified character sequence to the writer.
PrintWriter append(CharSequence ch, int start, int end)	It is used to append a subsequence of specified character to the writer.
boolean checkError()	It is used to flushes the stream and check its error state.
protected void setError()	It is used to indicate that an error occurs.
protected void clearError()	It is used to clear the error state of a stream.
PrintWriter format(String format, Object... args)	It is used to write a formatted string to the writer using specified arguments and format string.
void print(Object obj)	It is used to print an object.
void flush()	It is used to flushes the stream.

Byte Stream:

Some important Byte stream classes.

Stream class	Description
BufferedInputStream	Used for Buffered Input Stream.
BufferedOutputStream	Used for Buffered Output Stream.
DataInputStream	Contains method for reading java standard datatype
DataOutputStream	An output stream that contain method for writing java standard data type
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that write to a file.
InputStream	Abstract class that describe stream input.
OutputStream	Abstract class that describe stream output.
PrintStream	Output Stream that contain `print()` and `println()` method

These classes define several key methods. Two most important are

1. `read()` : reads byte of data.
2. `write()` : Writes byte of data

Sample Codes: 1. FileReader

- ```
  import java.io.FileReader;
  import java.io.IOException;
  /**
  * This program demonstrates how to read characters from a text file.
  */
  public class TextFileReadingExample1 {
          public static void main(String[] args) {
                  try {
                          FileReader reader = new FileReader("MyFile.txt");
                          int character;
  ```

- ```
  /* int read(): It is used to return a character in ASCII form. It returns -1 at the end of file. */

                          while ((character = reader.read()) != -1) {
                                  System.out.print((char) character);
                          }
                          reader.close();
                  } catch (IOException e) {
                          e.printStackTrace();
          }
  }
  ```

Sample Codes: 2. FileWriter

- ```
  import java.io.FileWriter;
  import java.io.IOException;
  /**
  * This program demonstrates how to write characters to a text file. */
  public class TextFileWritingExample1 {
          public static void main(String[] args) {
               try {
                        FileWriter writer = new FileWriter("MyFile.txt", true);
                        writer.write("Hello World");
                        writer.write("\r\n"); // write new line
                        writer.write("Good Bye!");
                        writer.close();
                   } catch (IOException e) {
                        e.printStackTrace();
  }
  ```
  ```

Read Single Character:

- /*FileInputStream*/
  import java.io.FileInputStream;

```
public class DataStreamExample {
    public static void main(String args[]){
      try{
              FileInputStream fin=new
              FileInputStream("D:\\testout.txt");
                 int i=fin.read();
                 System.out.print((char)i); // Typecasting for converting
    byte to char
                 fin.close();
          }catch(Exception e){System.out.println(e);}
      }
    }
```

Reading from a text file using *BufferedReader* object

```
import java.io.*;
public class ReadTextFile {
        public static void main (String [] args) throws IOException        {
            File inFile = new File ("input.txt");
            FileReader fReader = new FileReader (inFile);
            BufferedReader bReader = new BufferedReader (fReader);
                try {
                                        String line = bReader.readLine();
                                         while (line != null)
                                        {
                                              System.out.println (line);
                                              line = bReader.readLine();
                                         }
                        }
                 catch (Exception e) {
                                System.out.println (e.getMessage());
                                e.printStackTrace();
                                System.exit(0);
                                   }
                    finally
                    {
                                 fReader.close();
                     }
                }
             }
```

A text file can also be read using a Scanner object.

```
import java.util.Scanner;
import java.io.File;
    public class ReadTextFile {
        public static void main (String [] args) throws IOException    {
           File inFile = new File ("input.txt");
           Scanner sc = new Scanner (inFile);
           while (sc.hasNextLine())
            {
                String line = sc.nextLine();
                System.out.println (line);
            }
            sc.close();
        }
    }
```

Writing to a Binary File

```
import java.io.*;
public class WriteBinaryFile {
    public static void main (String [] args) throws IOException        {
   File outFile = new File ("output.bin");
   FileOutputStream outStream = new FileOutputStream (outFile);
   DataOutputStream output = new DataOutputStream (outStream);
   String name = "John Doe";
   long ssNum = 123456789;
   double gpa = 3.85;
   try {
          output.writeUTF (name);
          output.writeLong (ssNum);
          output.writeDouble (gpa);
         } catch (Exception e)
           { System.out.println (e.getMessage()); e.printStackTrace(); System.exit(0); }
  finally { outStream.close(); } } }
```

Reading from a Binary File

```
import java.io.*;
public class ReadBinaryFile {
public static void main (String [] args) throws IOException
{
    File inFile = new File ("input.bin");
    FileInputStream inStream = new FileInputStream (inFile);
    DataInputStream input = new DataInputStream (inStream);
    String name;
    long ssNum;
    double gpa;
        try {
            while (true) {
                name = input.readUTF();
                ssNum = input.readLong();
                gpa = input.readDouble();
                System.out.println (name + " " + ssNum + " " + gpa); }
        } catch (EOFException e)
{ // Do nothing if it is the end of file. } catch (Exception e) { System.out.println (e.getMessage());
    e.printStackTrace(); System.exit(0); }
        finally { inStream.close(); }
        }
     }
```

The CharArrayWriter class can be used to write common data to multiple files. This class inherits Writer class. Its buffer automatically grows when data is written in this stream. Calling the close() method on this object has no effect.

```
import java.io.CharArrayWriter;
import java.io.FileWriter;
public class CharArrayWriterExample {
public static void main(String args[])throws Exception{
          CharArrayWriter out=new CharArrayWriter();
          out.write("Welcome to VIIT");
          FileWriter f1=new FileWriter("D:\\a.txt");
          FileWriter f2=new FileWriter("D:\\b.txt");
          FileWriter f3=new FileWriter("D:\\c.txt");
          FileWriter f4=new FileWriter("D:\\d.txt");
          out.writeTo(f1);
          out.writeTo(f2);
          out.writeTo(f3);
          out.writeTo(f4);
          f1.close();
          f2.close();
          f3.close();
          f4.close();
          System.out.println("Success...");
         }
        }
```

```
import java.io.*;
public class BufferedInputStreamExample{
 public static void main(String args[]){
  try{
    FileInputStream fin=new FileInputStream("D:\\testout.txt");
    BufferedInputStream bin=new BufferedInputStream(fin);
    int i;
    while((i=bin.read())!=-1){
     System.out.print((char)i);
    }
    bin.close();
    fin.close();
  }catch(Exception e){System.out.println(e);}
 }
}
```

```
import java.io.FileReader;
import java.io.IOException;

/**
 * This program demonstrates how to read characters from a text file.
 */
public class TextFileReadingExample1 {

    public static void main(String[] args) {
        try {
            FileReader reader = new FileReader("MyFile.txt");
            int character;

            while ((character = reader.read()) != -1) {
                System.out.print((char) character);
            }
            reader.close();

        } catch (IOException e) {
            e.printStackTrace();
        }
    }

}
```

```
import java.io.FileWriter;
import java.io.IOException;

/**
 * This program demonstrates how to write characters to a text file.
 *
 *
 */
public class TextFileWritingExample1 {

    public static void main(String[] args) {
        try {
            FileWriter writer = new FileWriter("MyFile.txt", true);
            writer.write("Hello World");
            writer.write("\r\n");   // write new line
            writer.write("Good Bye!");
            writer.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
```

```
import java.io.*;

public class ObjectInputStreamExample {

    public static class Person implements Serializable {
        public String name = null;
        public int    age  =   0;
    }


    public static void main(String[] args) throws IOException,
ClassNotFoundException {

        ObjectOutputStream objectOutputStream =
            new ObjectOutputStream(new
FileOutputStream("data/person.bin"));

        Person person = new Person();
        person.name = "AVC";
        person.age  = 4;

        objectOutputStream.writeObject(person);
        objectOutputStream.close();


        ObjectInputStream objectInputStream =
            new ObjectInputStream(new
FileInputStream("data/person.bin"));

        Person personRead = (Person) objectInputStream.readObject();

        objectInputStream.close();

        System.out.println(personRead.name);
        System.out.println(personRead.age);
    }
}
```

```
/*FileInputStream*/
    import java.io.FileInputStream;
    public class DataStreamExample {
         public static void main(String args[]){
              try{
                FileInputStream fin=new
FileInputStream("D:\\testout.txt");
                int i=fin.read();
                System.out.print((char)i); // Typecasting for converting
byte to char

                fin.close();
               }catch(Exception e){System.out.println(e);}
             }
            }

/* Try for reading all characters*/
```