Sunbeam Institute of Information Technology
Pune and Karad

Data structures and Algorithms

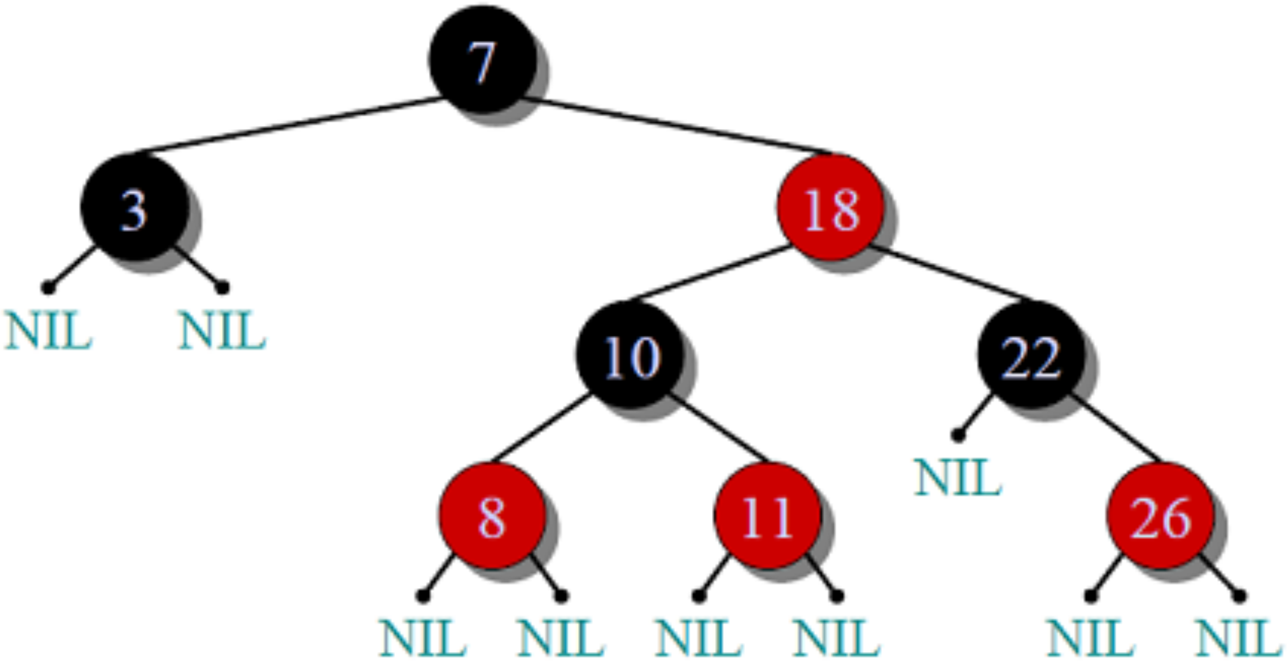
Trainer - Devendra Dhande

Email – devendra.dhande@sunbeaminfo.com

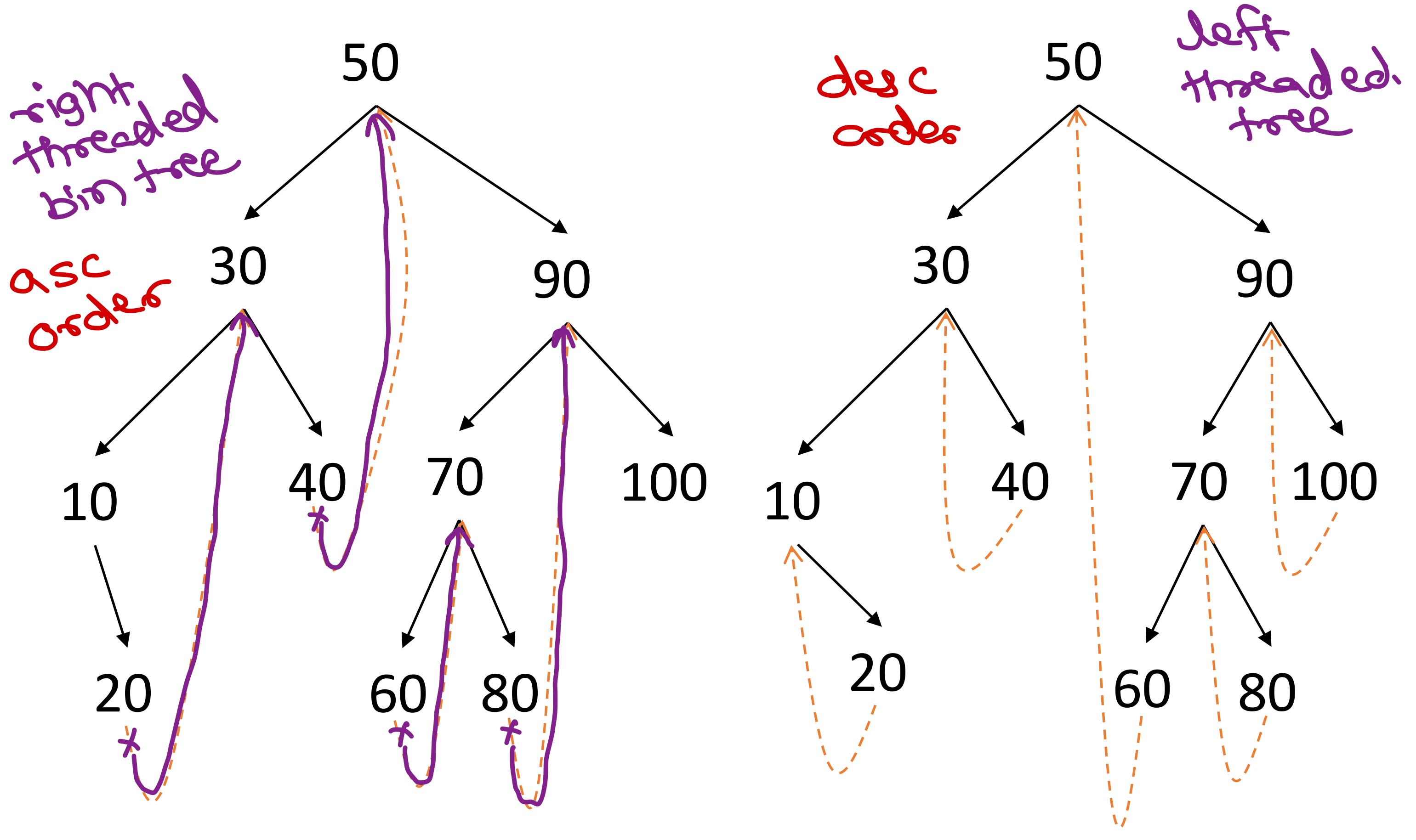
Red & Black tree

Java collection - TreeSet

- Red & Black tree is a self-balancing Binary Search Tree (BST).
- Each node follows some rules:
  - Every node has a color either red or black.
  - Root of tree is always black.
  - Two adjacent cannot be red nodes (Parent color should be different than child).
  - Every path from a node (including root) to any of its descendant NULL node has the equal number of black nodes.
- Most of BST operations are done in O(h) i.e. O(log n) time.
- For frequent insert/delete, RB tree is preferred over AVL tree.



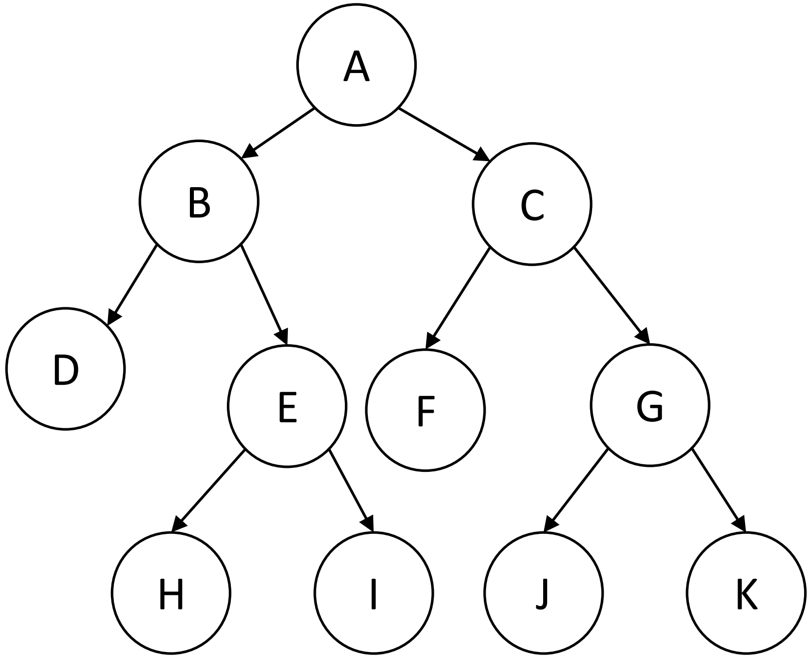
Threaded BST



- Typical BST in-order traversal involves recursion or stack. It slows execution and also need more space.
- Threaded BST keep address of in-order successor or predecessor addresses instead of NULL to speed up in-order traversal (using a loop).
- Left threaded BST
- Right threaded BST
- In-threaded BST

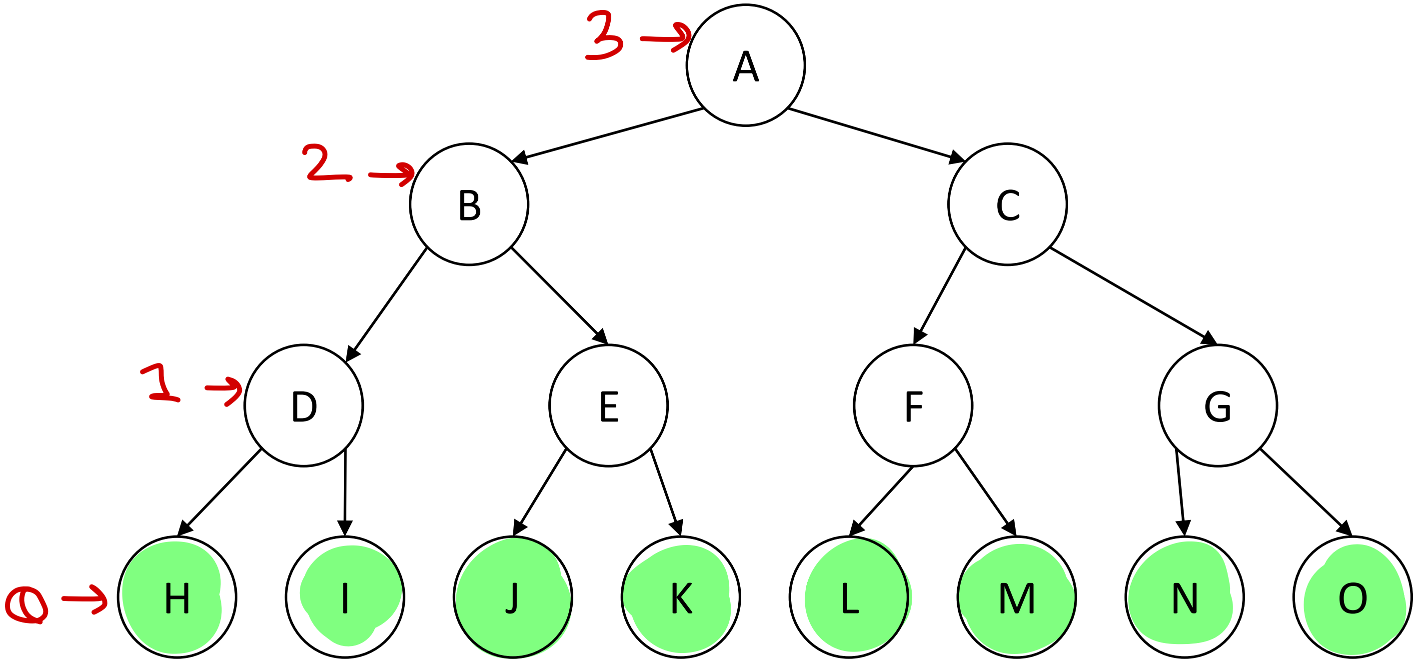
right threaded BST + left threaded BST = In-threaded BST

Strictly Binary Tree



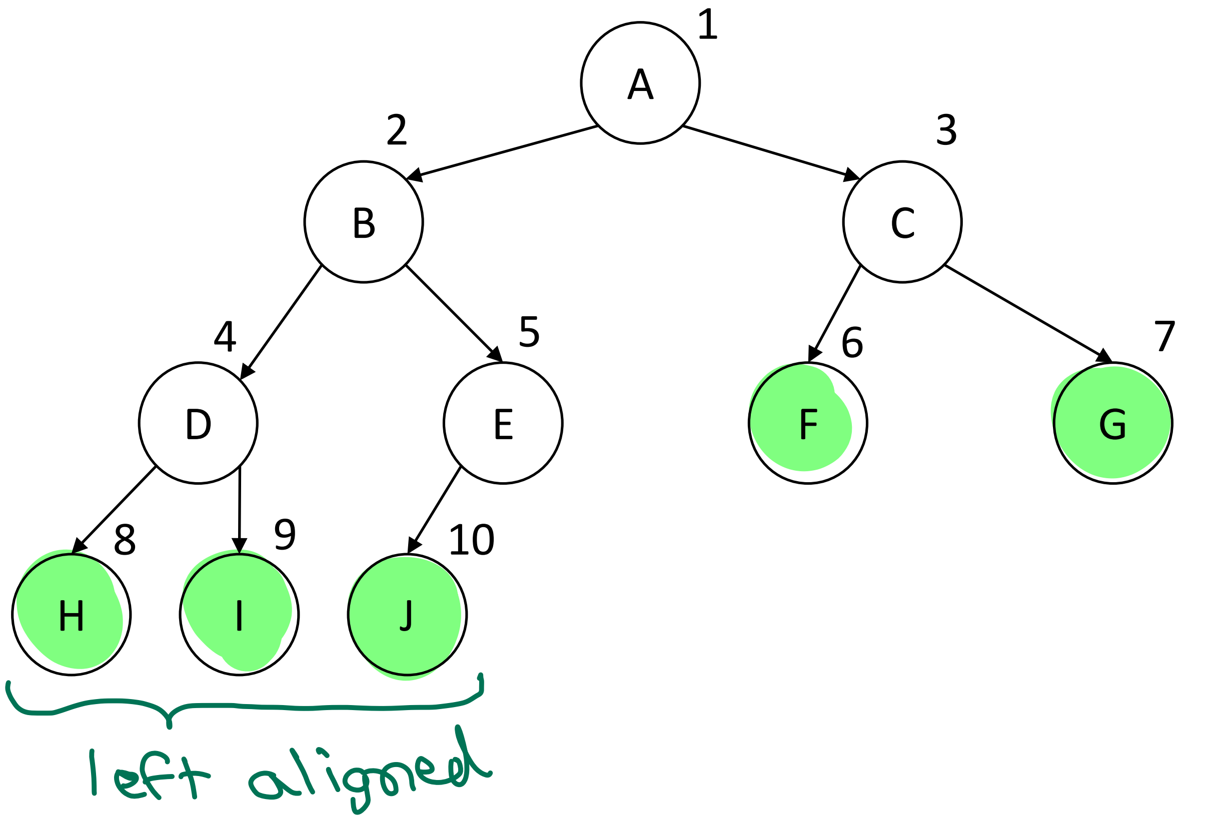
- Binary tree in which each non-leaf node has exactly two child nodes.
- Strictly binary tree with n leaves always has 2*n – 1 nodes.

Complete Binary Tree



- Strictly binary tree of height h whose all leaves are at same level. Also called as full tree.
- Number of nodes = $2^{h+1} - 1$ $\quad 2^{3+1} - 1 = 15$
- Number of leaf nodes = 2^h $\quad 2^3 = 8$

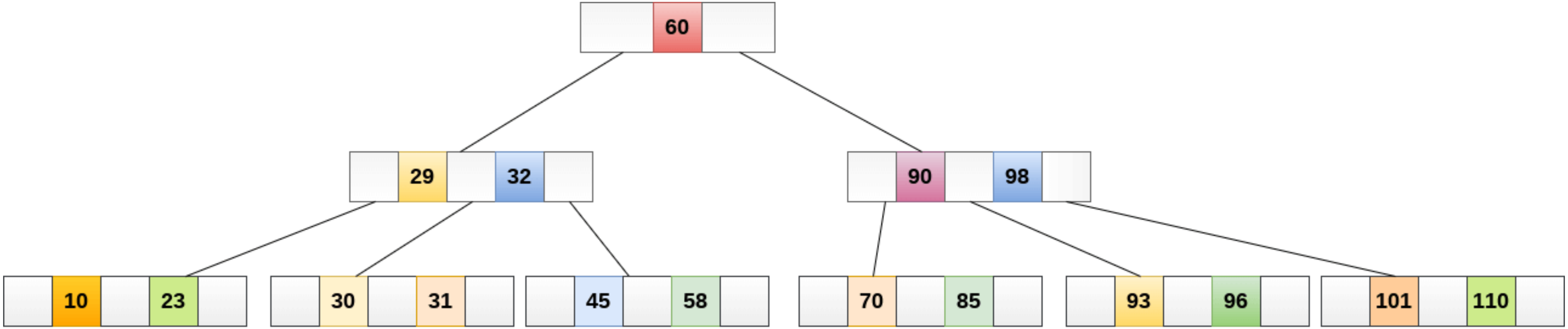
Almost Complete Binary Tree and Heap



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

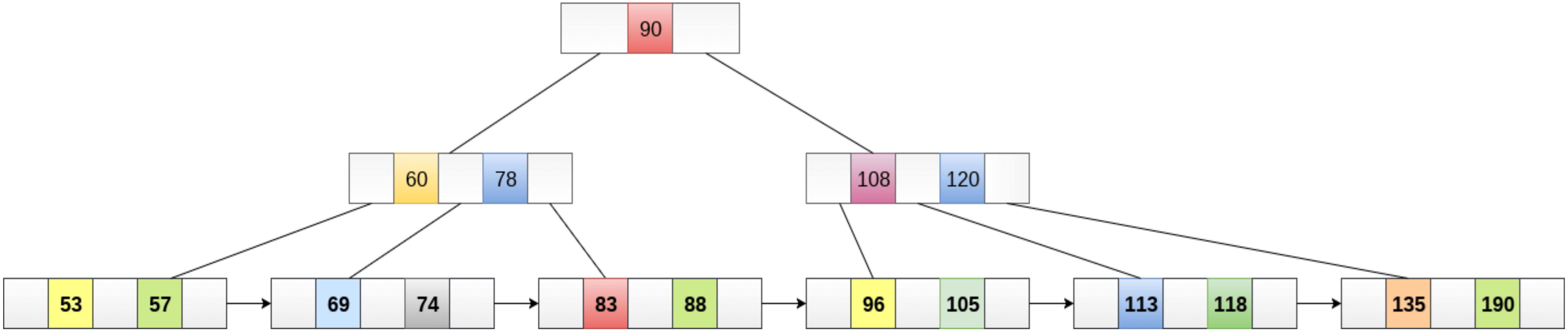
- Almost complete binary tree (of height d)
  - All leaf nodes must be at level d or d-1.
  - All leaf nodes at level d must aligned as left as possible.
- Heap is array implementation of almost complete binary tree.
- Parent child relation is maintained through index calculations
  - parent index = child index / 2
  - left child index = parent index * 2
  - right child index = parent index * 2 + 1

B Tree



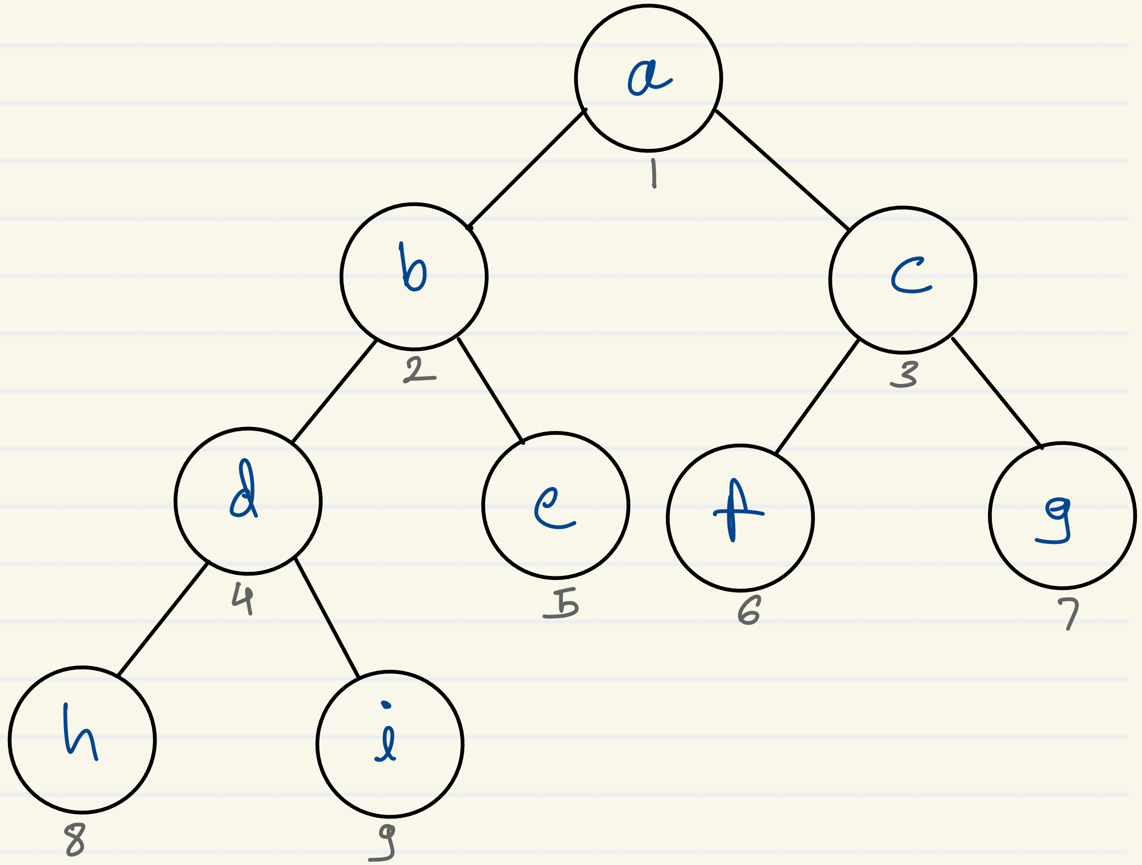
- A B-Tree of order m can have at most m-1 keys and m children.
- B tree store large number of keys in a single node. This allows storing number of values keeping height minimal.
- Note that in B-Tree all leaf nodes are at same level.
- B-Tree is commonly used for indexing into file systems and databases. It ensures quick data searching and speed up disk access.

B+ Tree



- Extension of B-Tree for efficient insert, delete and search operation.
- Data is stored in leaf nodes only and all leaf nodes are linked together for sequential access.
- Search keys may be redundant.
- Faster searching, simplified deletion (as only from leaf nodes).
- B+Tree is commonly used for indexing into file systems and databases. It ensures quick data searching and speed up disk access.

Almost Complete Binary Tree or Heap



a	b	c	d	e	f	g	h	i
1	2	3	4	5	6	7	8	9

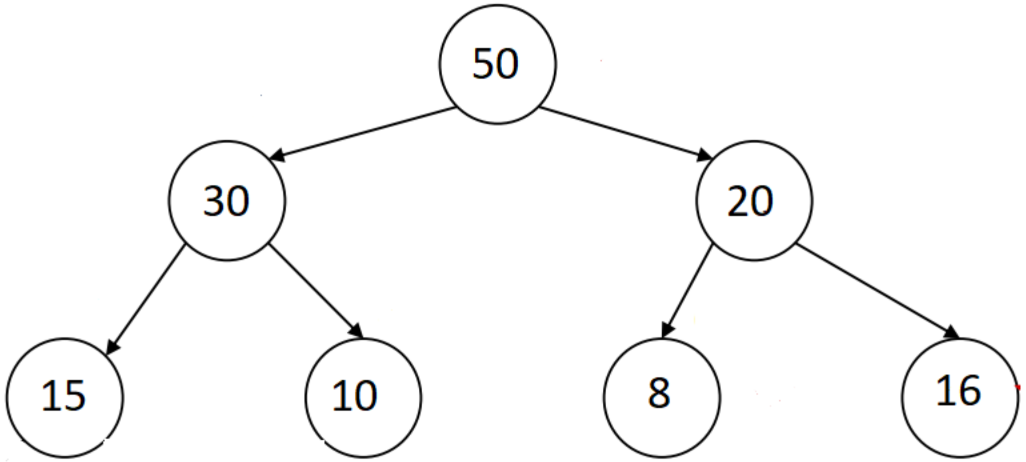
- Almost Complete Binary Tree (height = h)
- All leaf nodes must be at level h or h-1
- All leaf nodes at level h must aligned as left as possible

- Array implementation of Almost Complete Binary Tree is called as heap

Node → ith index
parent → i/2 index
left child → i*2 index
rigut child → i*2+1 index

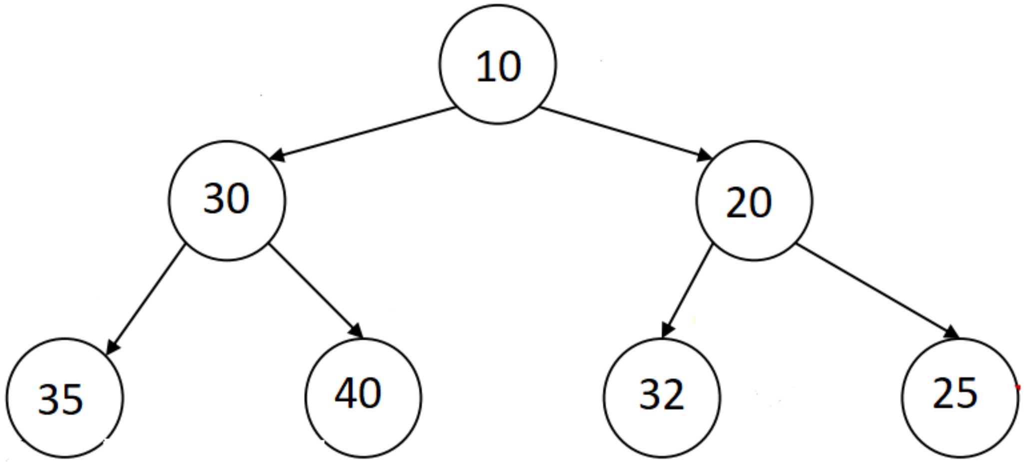
Heap Types – Max and Min

Max Heap



50	30	20	15	10	8	16
1	2	3	4	5	6	7

Min Heap

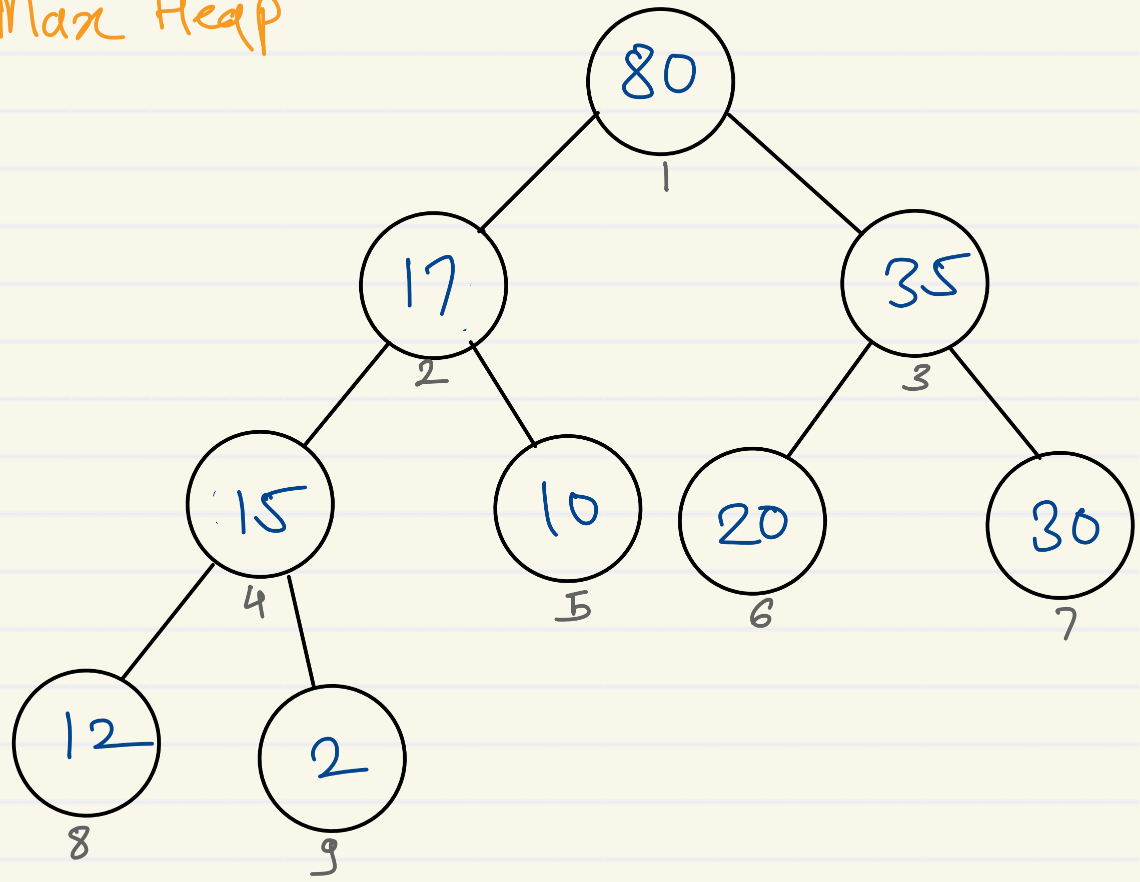


10	30	20	35	40	32	25
1	2	3	4	5	6	7

- Max heap is a heap data structure in which each node is greater than both of its child nodes.
- Min heap is a heap data structure in which each node is smaller than both of its child nodes.

Heap - Add heap

Max Heap



80	17	35	15	10	20	30	12	2
1	2	3	4	5	6	7	8	9

Keys : 20, 12, 35, 15, 10, 80, 30, 17, 2

Algorithm :

1. add new value at first empty index from left side
2. adjust position of the newly added value by comparing with all its ancestors one by one.

- to add value into heap, need to traverse from leaf to root position.

Time $\propto$ height

$\boxed{T(n) = O(\log n)}$

90	80	35	17	10	20	30	12	15
1	2	3	4	5	6	7	8	9

value = 90

ci	pi	
9	4	✓
4	2	✓
2	1	✓
1	0	✗

not valid index

80	17	35	16	10	20	30	12	15
1	2	3	4	5	6	7	8	9

value = 16

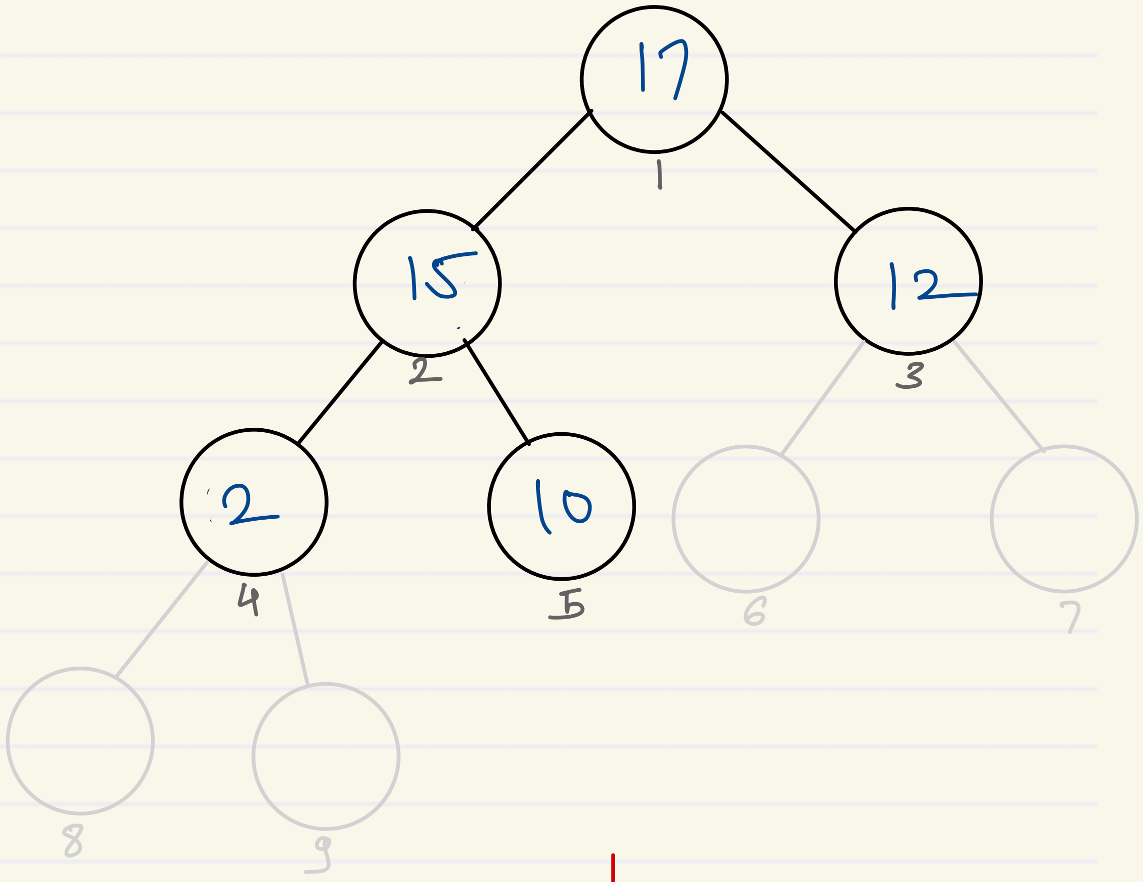
ci	pi	
9	4	✓
4	2	✗

correct place for newly addd value
(further parent is is greater)

ci = index of newly added value

```
pi = ci/2;
while (pi > 0) {
    if (parent is greater)
        break;
    swap;
    ci = pi;
    pi = ci/2
}
```

Heap - Delete heap



17	15	12	2	10				
1	2	3	4	5	6	7	8	9

property : can delete only root element

Max heap : always higher element is deleted
Min heap : always lower element is deleted

Max : 80, 35, 30, 20

Algorithm :

1. After deleting root, promote last element of heap to root place.
2. Adjust position of promoted element by comparing with all its descendants one by one.

- to delete element from heap, need to traverse from root to leaf.

Time $\propto$ height

$\boxed{T(n) = O(\log n)}$

left child = pi * 2   right child = pi * 2 + 1
= leftchild + 1

35	17	30	15	10	20	2	12	
1	2	3	4	5	6	7	8	9

Max = 80

pi	ci	
1	2, 3	✓
3	6, 7	✓
7	14	✗

invalid child index

```
pi = 1;
ci = pi * 2;
while( ci <= size) {
    - find index of maximum child
    - if ( parent is greater than max child)
          break;
    - swap parent & max child;
    pi = ci;
    ci = pi * 2;
}
```

Priority Queue

- Queue where high priority data is always peeked or deleted.
- Priority can be implemented using array, linked list and heap data structures.
- An array is used to store a value and its associated priority. In some simpler implementations, the value itself might represent the priority (e.g., lower value means higher priority).
- Array and linked list implementation of priority queue often leads to less efficient performance compared to heap-based implementations for insertion and deletion operations.

- **Ordered vs. Unordered Array**
  - **Ordered Array** / linked list
    - Elements are kept sorted by priority. Insertion requires shifting existing elements to maintain order, leading to O(n) time complexity for insertion.
    - Deletion of the highest priority element is O(1) as it's typically at the beginning or end of the array.
  - **Unordered Array:** / linked list
    - Elements are inserted without regard to order, making insertion O(1).
    - Deletion of the highest priority element requires searching the entire array to find it, resulting in O(n) time complexity for deletion.

Priority Queue Implementation

- Priority : number associated with value
- Priority range is defined by programmer

  e.g. Priority range : 1 to 10

Priority Queue

Higher value Higher priority (Max priority Queue)	Lower value Higher priority (Min priority Queue)

- Every element of priority queue will have two parts :

value : any data type
priority : integer

```
class item {
    int value;
    int priority;
};
```

```
class priorityQueue {
    class item arr[5];
    int capacity;      ( maxSize)
    int size;
};
```

Operations :

1) Enqueue
2) Dequeue
3) Peek
4) isEmpty → size == 0
5) isFull → size == capacity

Priority Queue Implementation

Ordered array:

capacity: 5 size: 4

arr

index	0	1	2	3	4
value	10	40	20	30	
priority	2	3	5	7	
address	400	408	416	424	432

insert - O(n) delete - O(1) peek - O(1)

for shifting - O(n)

Unordered array:

capacity: 5 size: 4

arr

index	0	1	2	3	4
value	10	20	30	40	
priority	2	5	7	3	
address	400	408	416	424	432

insert - O(1) delete - O(n) peek - O(n)

for shifting - O(n)