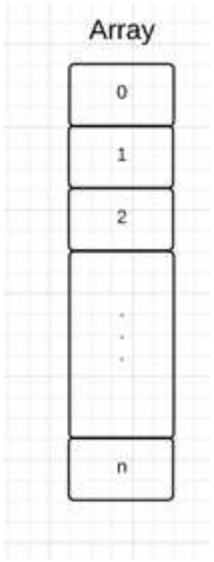
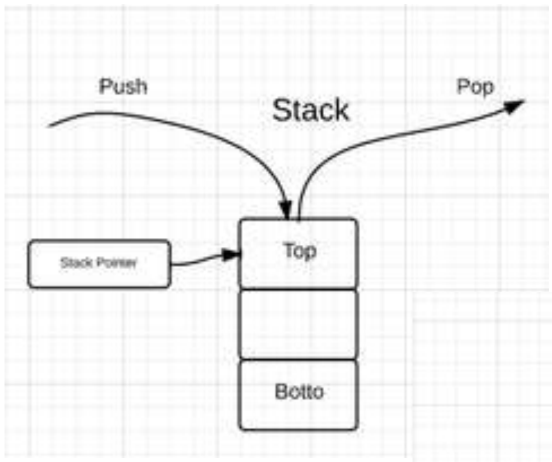
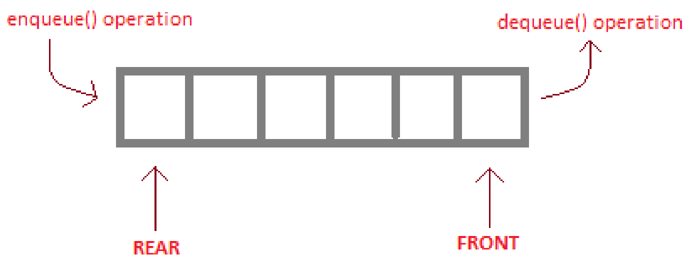
Tree data structure

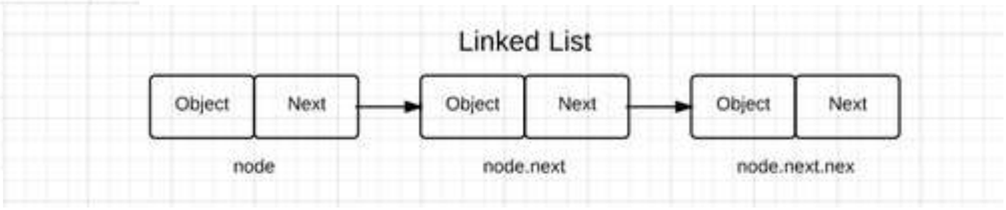
Binary tree, Binary search tree (BST)

- Linear data structures : data structures where the data elements are organised in some sequence is called linear data structure.
- Linear access time of linked lists is prohibitive
  – Does there exist any simple data structure for which the running time of most operations (search, insert, delete) is O(log N)?





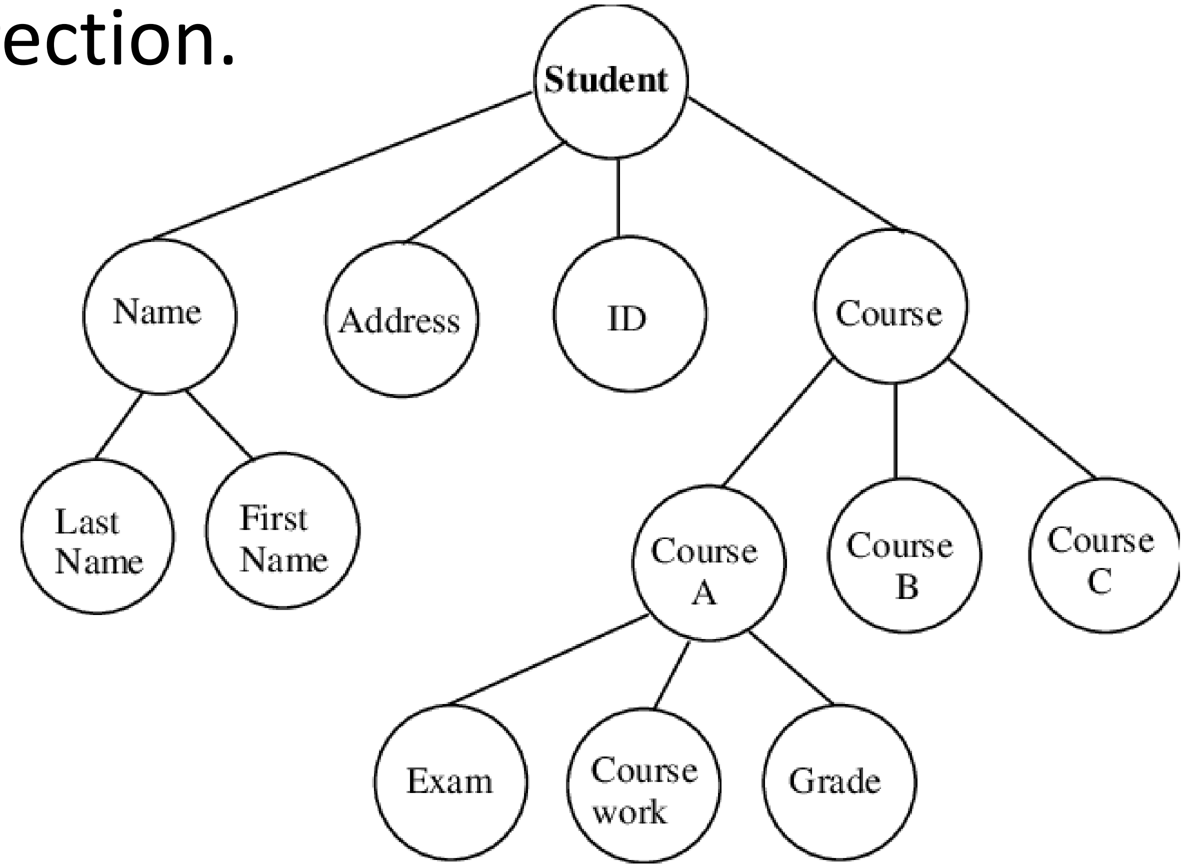




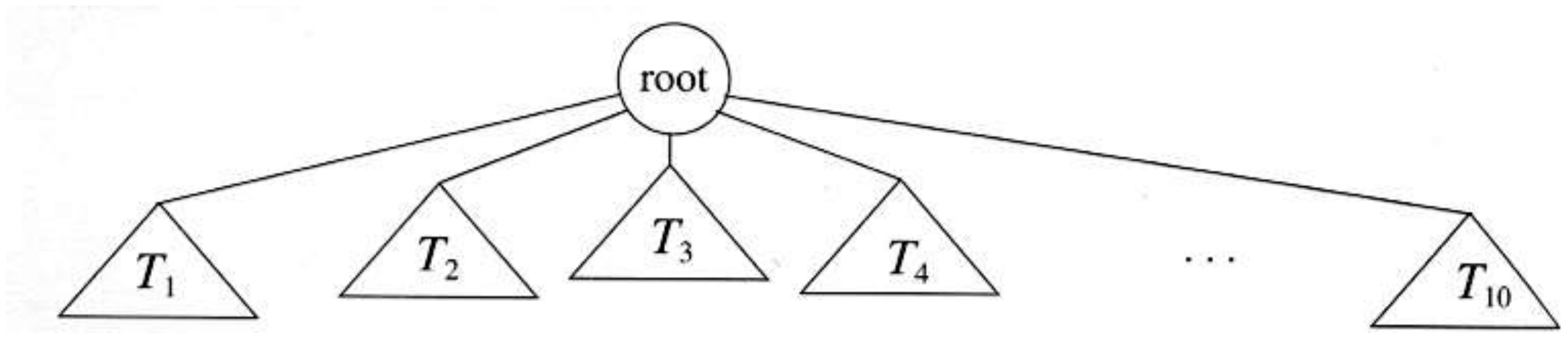
- Deciding on which data structure to use
  - What needs to be stored
  - Costs of operation
  - Memory usage
  - Easy to implement

Tree

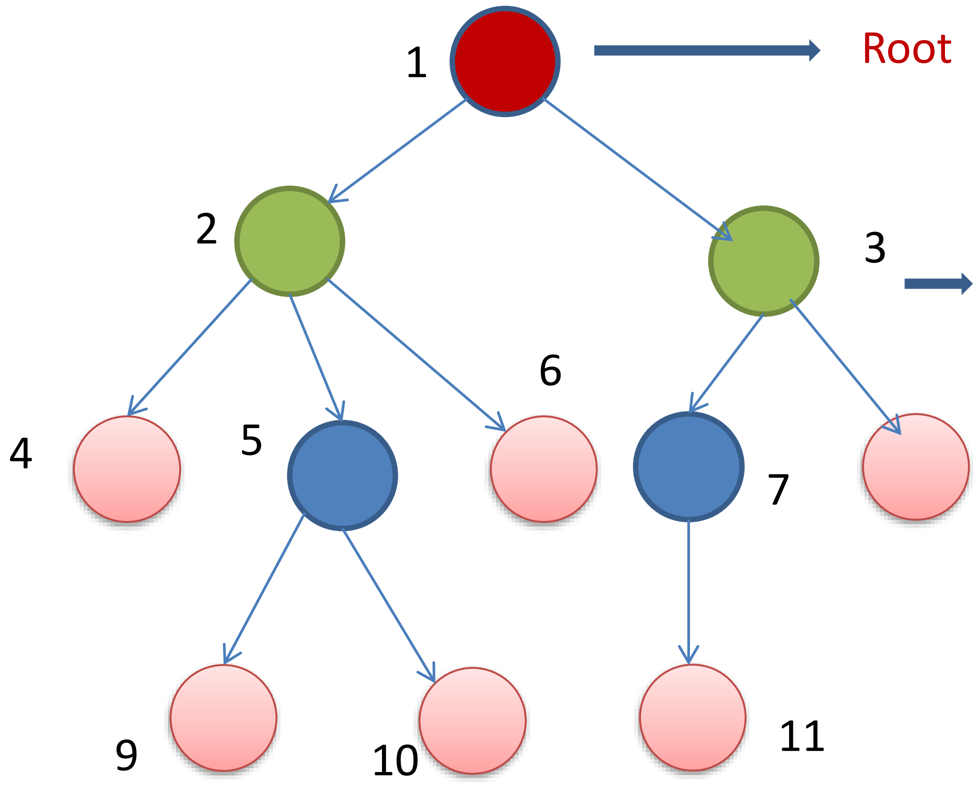
- Tree data structure is used to represent hierarchical data.
- Root is at the top and it braches out in downward direction.



- **Definition**
  - Tree data structure is a collection of entities called nodes linked together to simulate a hierarchy.
  - Tree is a nonlinear data structure (hierarchical)
  - The topmost node in the tree is called the root of the tree.
  - Each node can hold data and it makes a link to other nodes called children.
- (recursive definition) If not empty, a tree consists of a distinguished node r (the *root*), and zero or more nonempty *subtrees* T_1, T_2, ..., T_k, each of whose roots are connected by a directed *edge*



Basic terminology of tree



Root

2, 3 are child of 1
1 is parent of 2 and
3

4, 9, 10, 11, 8 are leaf nodes

If we can go from one node to another in a tree e.g from 1 to 10, then
Ancestor of 10 is 1, 2, 5
10 is the descendent of 1, 2, 5

Children of the same parent are called siblings
2,3 are siblings
4, 5, 6 are siblings
7, 8 are siblings
9, 10 are siblings

- Root: topmost node (1)
- Siblings: nodes share the same parent
- Internal node: node with at least one child (1, 2, 3 , 5, 7)
- External node (leaf): node without children (4, 8, 9, 10, 11)
- Ancestors of a node: parent, grandparent, grand-grandparent, etc.
- Descendant of a node: child, grandchild, grand-grandchild, etc.
- Depth of a node: number of ancestors
- Height of a tree: maximum depth of any node.
- Degree of a node: the number of its children
- Degree of a tree: the maximum number of its node.

1. Tree is a recursive data structure

2. In a valid tree if there are ‘n’ nodes, then there will be exactly n-1 edges.

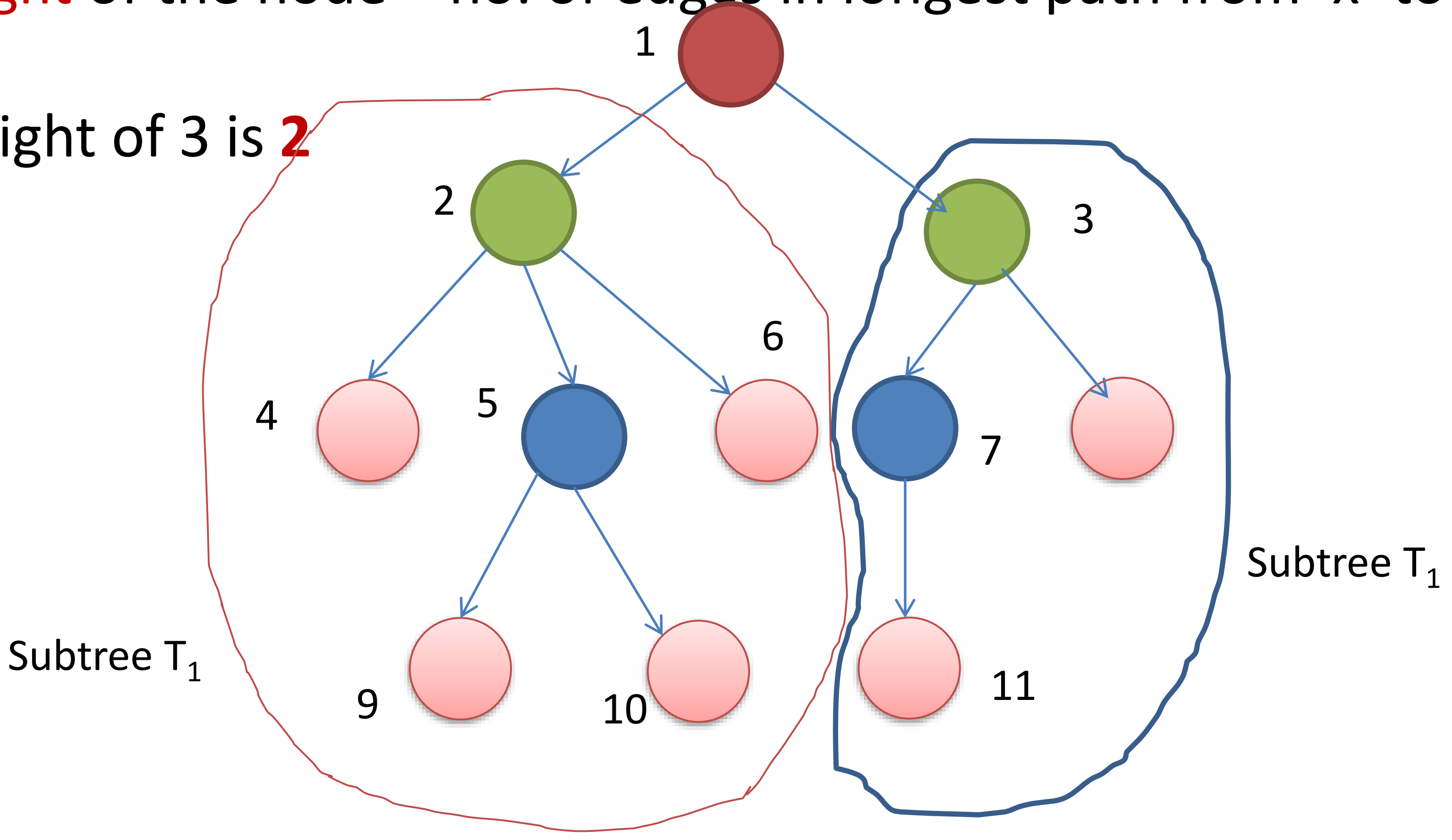
3. depth – length of the path from root to ‘x’

e.g depth of 5 is **2**

depth of root node is

4. Height of the node – no. of edges in longest path from ‘x’ to a leaf

e.g height of 3 is **2**

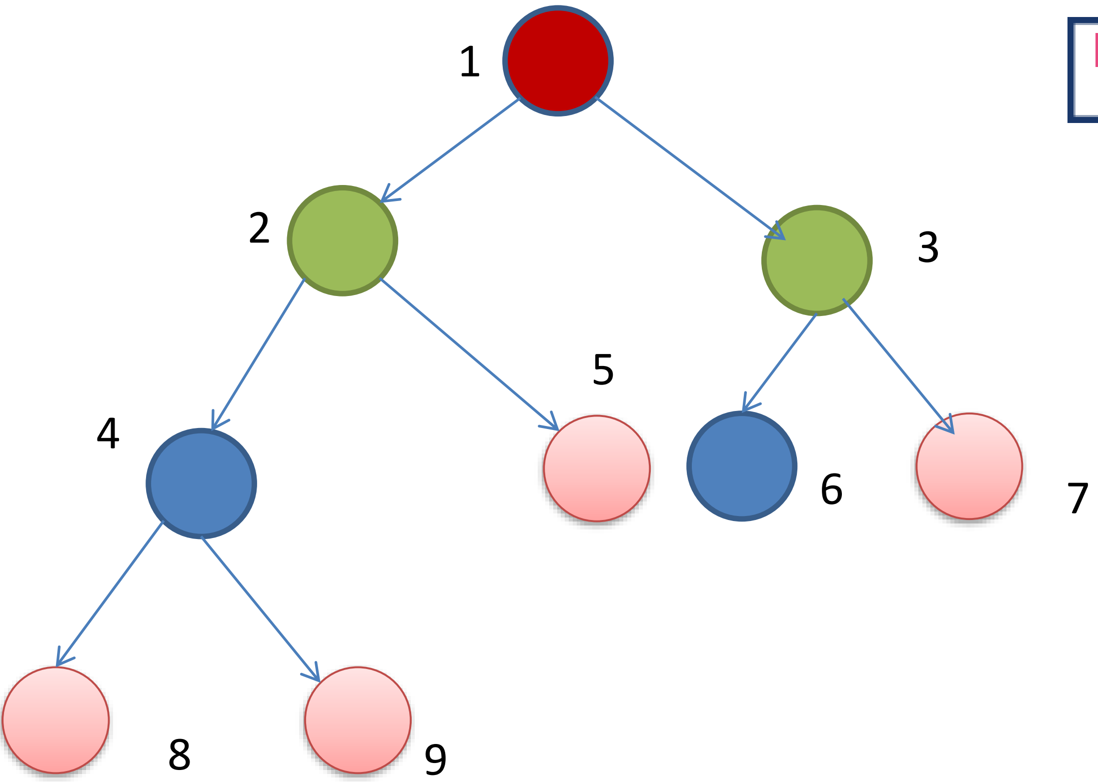


Applications of trees

- Storing hierarchical data (file system)
- Organize data for faster insertion deletion and search (binary search tree)
- Trie (dictionary used for dynamic spell check)
- Networking routing

Binary tree

- A tree in which each node can have atmost two children is called binary tree.



Left Child Address	**Data**	**Right Child** Address

Tree as a linked list
Has three fields

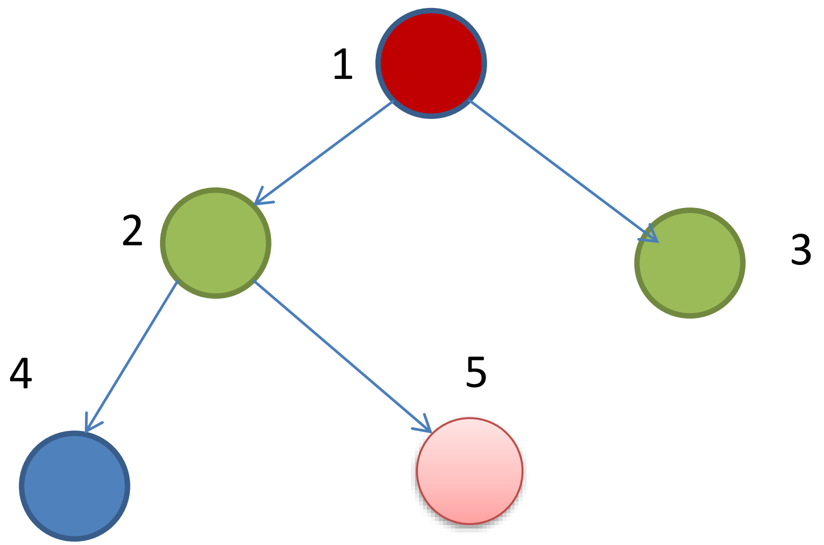
1. Data
2. Left child
3. Right child

```
Struct node
{
   int data;
   struct node* left;
   strcut node* right;
}
```

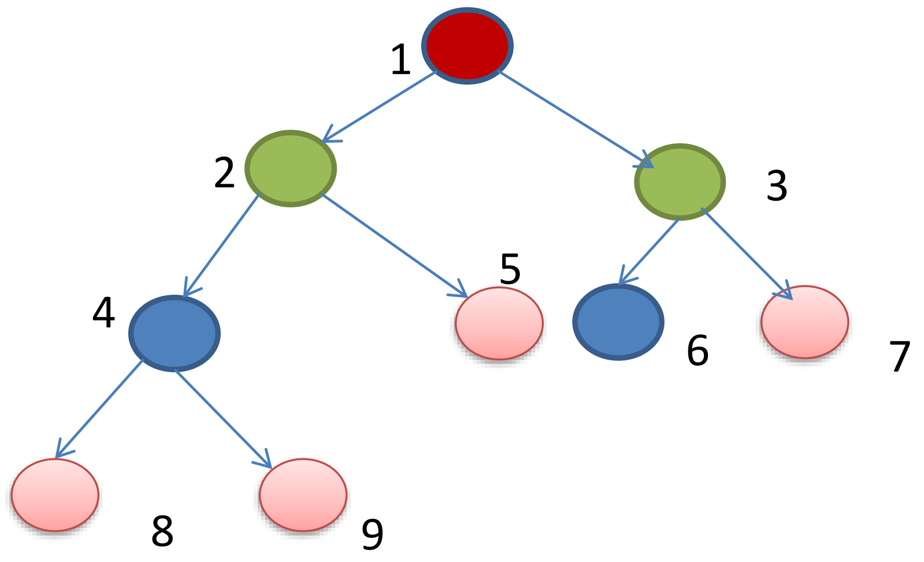
Property

- Each node can have atmost 2 children
- A node can have 1 child also.
- **Proper binary tree/strict binary tree** – each node can have either 2 or 0 children
- **Complete binary tree** – all levels except possibly the last level are completely filled and all nodes are as left as possible.
- Maximum number of nodes at level i = 2^i
- **Perfect binary tree - ** all levels are filled with child

Proper binary tree



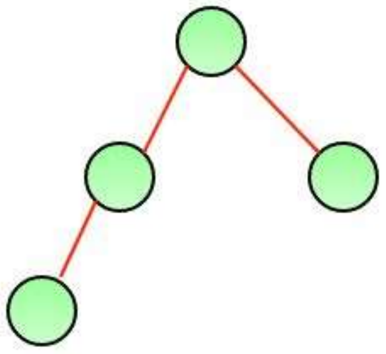
Complete binary tree

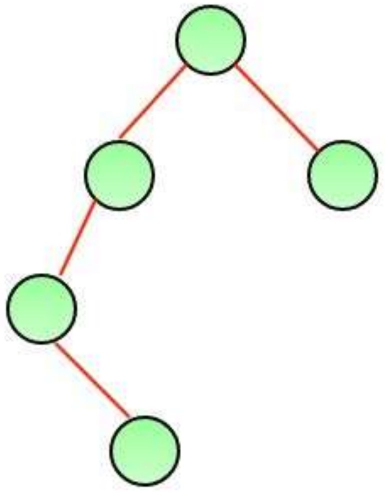


- Maximum no. of nodes in a binary tree with height h

  $= 2^{h+1} - 1$

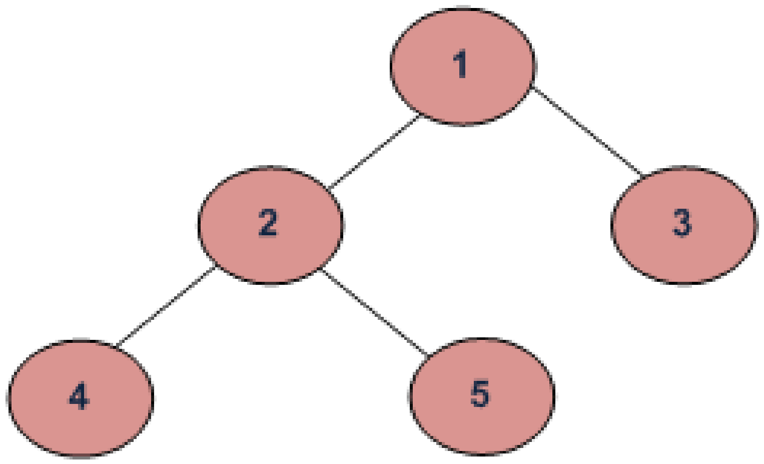
- Balanced binary tree – difference between height of the left and right subtree for every node is not more than 1.


A height balanced tree


Not a height balanced tree

- Binary Tree Traversals

- In a traversal of a binary tree, each element of the binary tree is visited exactly once.
  - Preorder
  - Inorder
  - Postorder



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

- While performing searching, inorder traversal gives the result in non decreasing order.
- Preorder traversals creates the copy of the tree.
- Postorder traversal can be used to delete a tree.

- **Preorder**

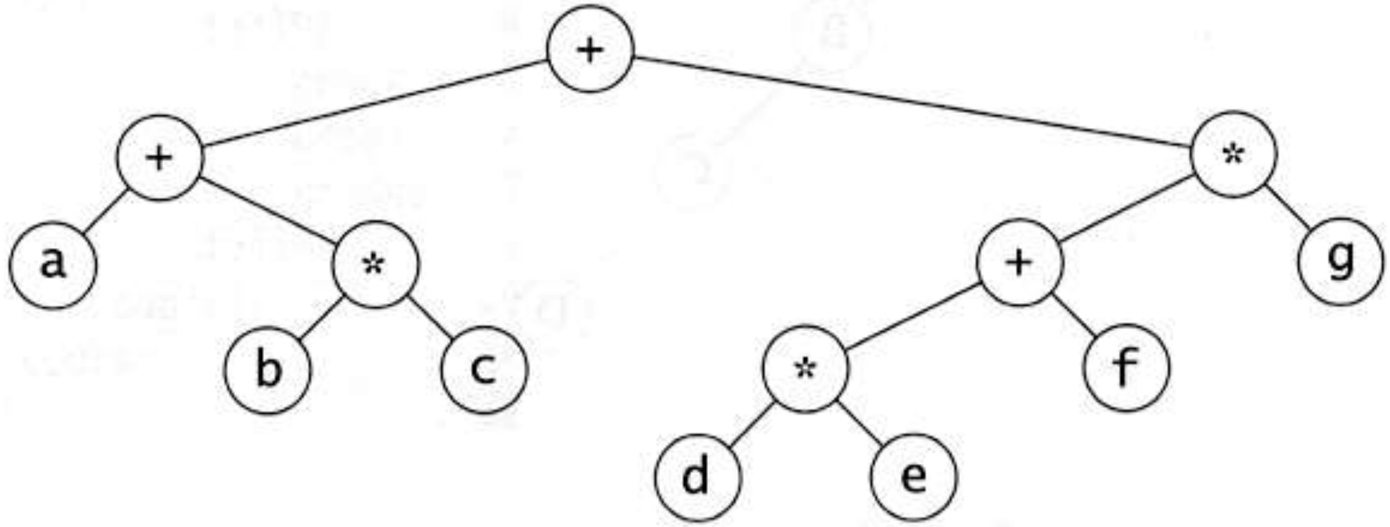
```
preorder(struct  node *root)
{
        if(root == NULL)
                return;
        printf(" %d",root->data);
        preorder(root->left);
        preorder(root->right);

}
```

- **Inorder**

```
inorder(struct node *root)
{
        if(root == NULL)
                return;
        inorder(root->left);
        printf(" %d",root->data);
        inorder(root->right);
}
```

```
postorder(struct  node *root)
{
        if(root == NULL)
                return;
        postorder(root->left);
        postorder(root->right);
        printf(" %d",root->data);
}
```

Example: Expression Trees



- Leaves are operands (constants or variables)
- The other nodes (internal nodes) contain operators
- Will not be a binary tree if some operators are not binary

General tree to binary tree

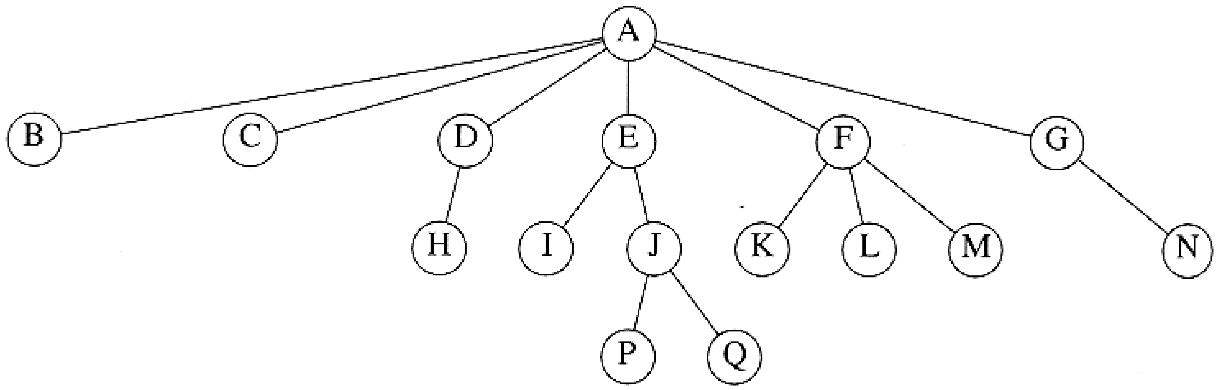


Figure 4.2 A tree

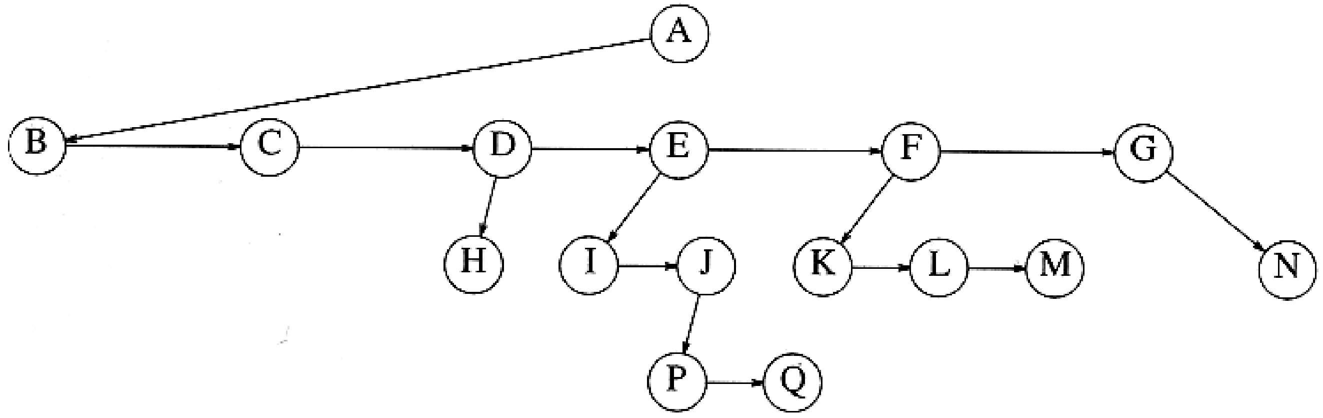


Figure 4.4 First child/next sibling representation of the tree shown in Figure 4.2

Procedure (from general to binary tree)

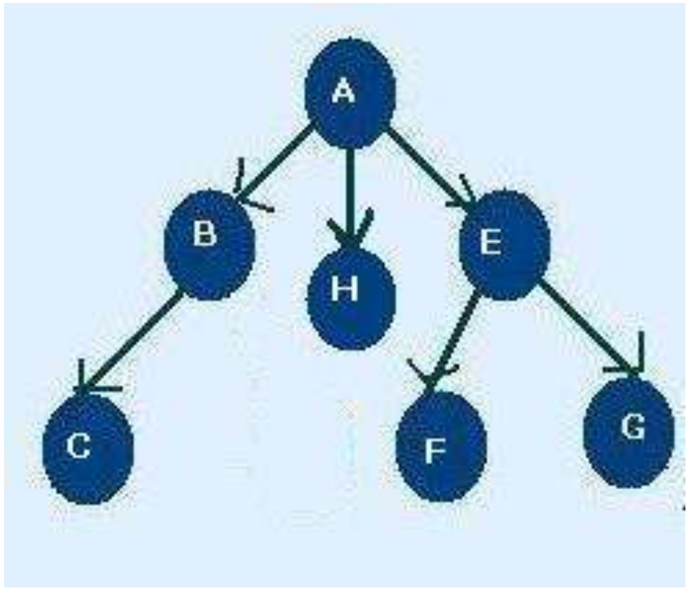
The process of converting general tree in to binary tree is given below:

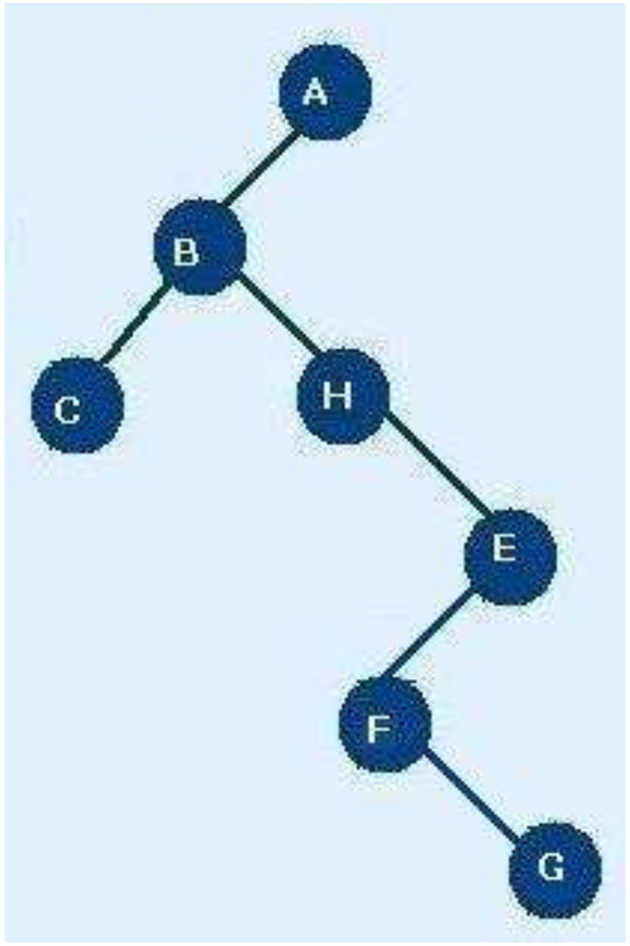
(1) Root node of general tree becomes root node of Binary Tree.

(2) Now consider T1, T2, T3 ... Tn are child nodes of the root node in general tree. The left most child (T1) of the root node in general tree becomes left most child of root node in the binary tree. Now Node T2 becomes right child of Node T1, Node T3 becomes right child of Node T2 and so on in binary tree.

(3) The same procedure of step 2 is repeated for each leftmost node in the general tree.

General tree binary tree

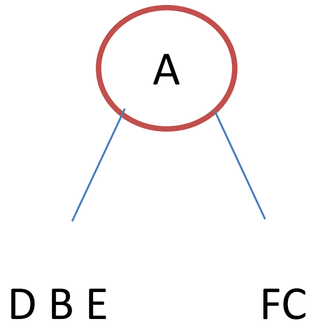


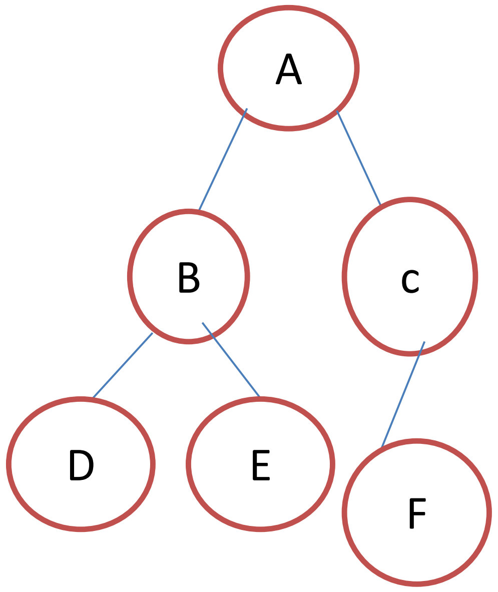


Conversion from tree traversal to binary tree

Inorder sequence: D B E A F C
Preorder sequence: A B D E C F

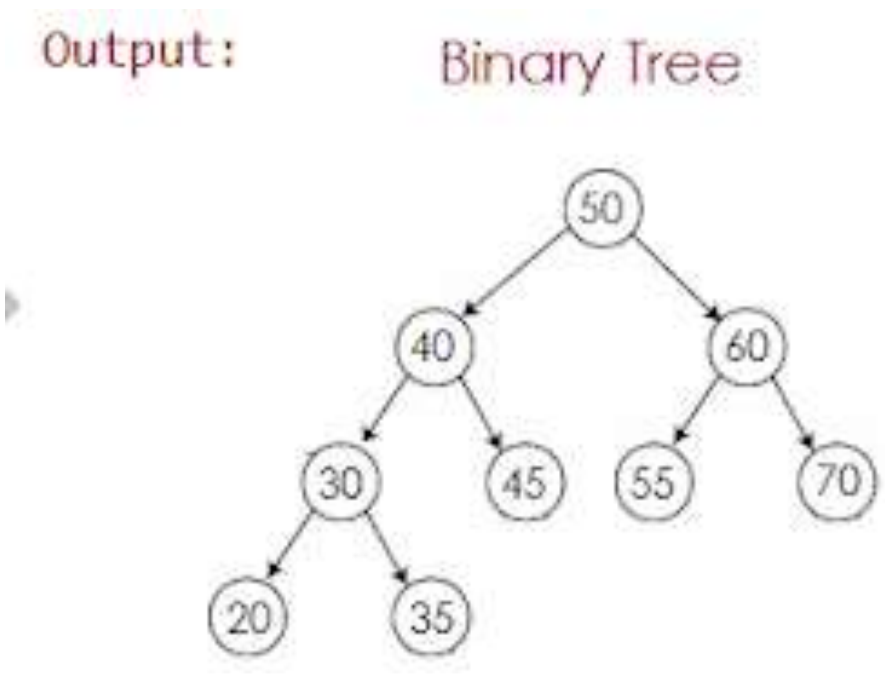
- In a Preorder sequence, leftmost element is the root of the tree.
- So we know 'A' is root for given sequences.
- By searching 'A' in Inorder sequence, we can find out all elements on left side of 'A' are in left subtree and elements on right are in right subtree.
- So we know below structure now.





Inorder = {20, 30, 35, 40, 45, 50, 55, 60, 70}
Postorder = {20, 35, 30, 45, 40, 55, 70, 60, 50};

Root element will be present at the last in Post-order traversal
For finding the Left and Right child's we will use In-order traversal



Practice questions

- Answer the following questions about the tree below.

(a) Which nodes are leaves?

(b) Which node is the root?

(c) Which is the parent of node C?

(d) Which nodes are children of C?

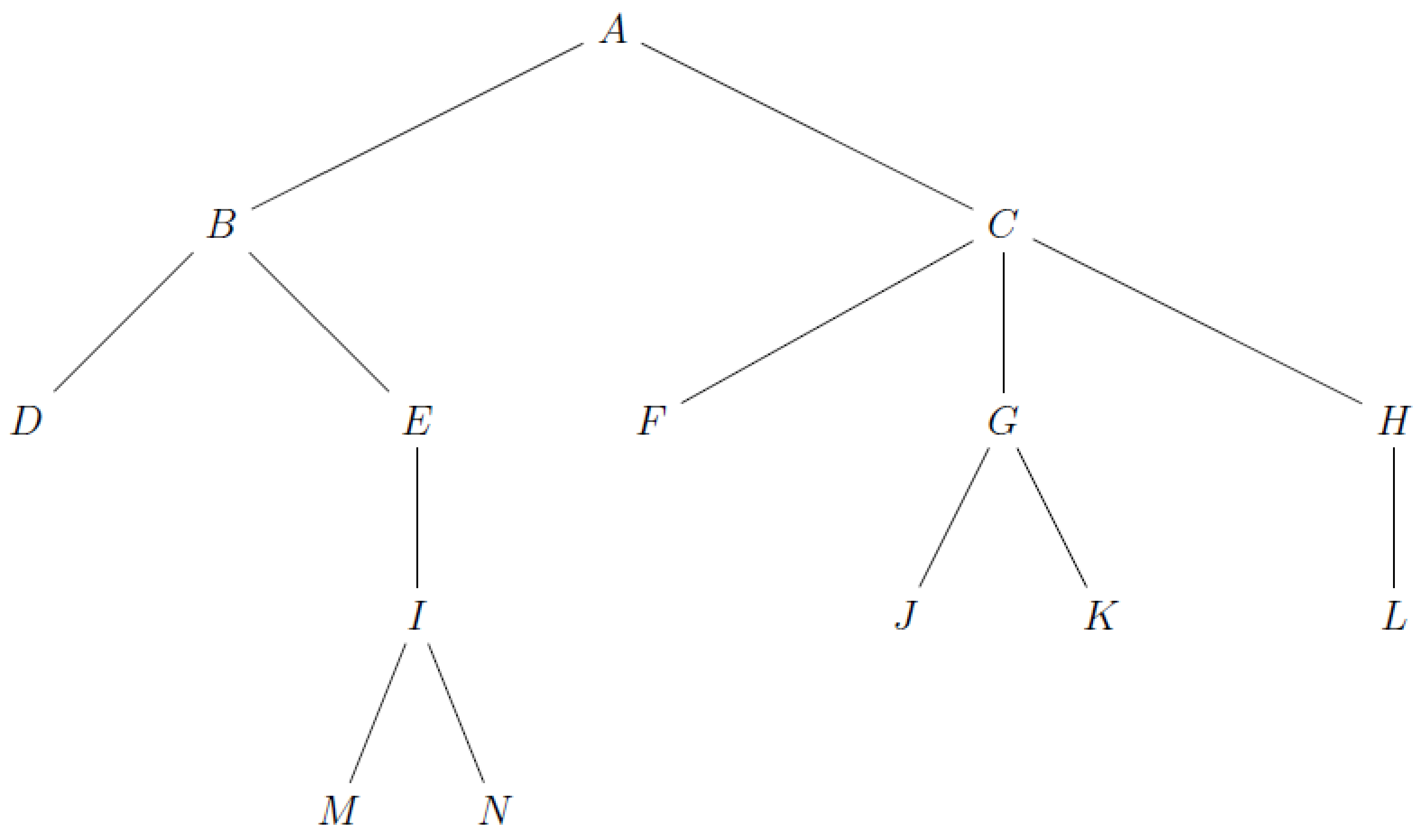
(e) Which nodes are ancestors of E?

(f) Which nodes are descendants of E?

(g) What is the depth (i.e. level) of node C?

(h) What is the height of node C? (The height of a node in a tree is dened to

be the length of a longest downward path from the node to a leaf.)


A
B
C
D
E
F
G
H
I
J
K
L
M
N

- List the nodes of the above tree in

(a) preorder,

(b) inorder, and

(c) postorder.

Binary search trees

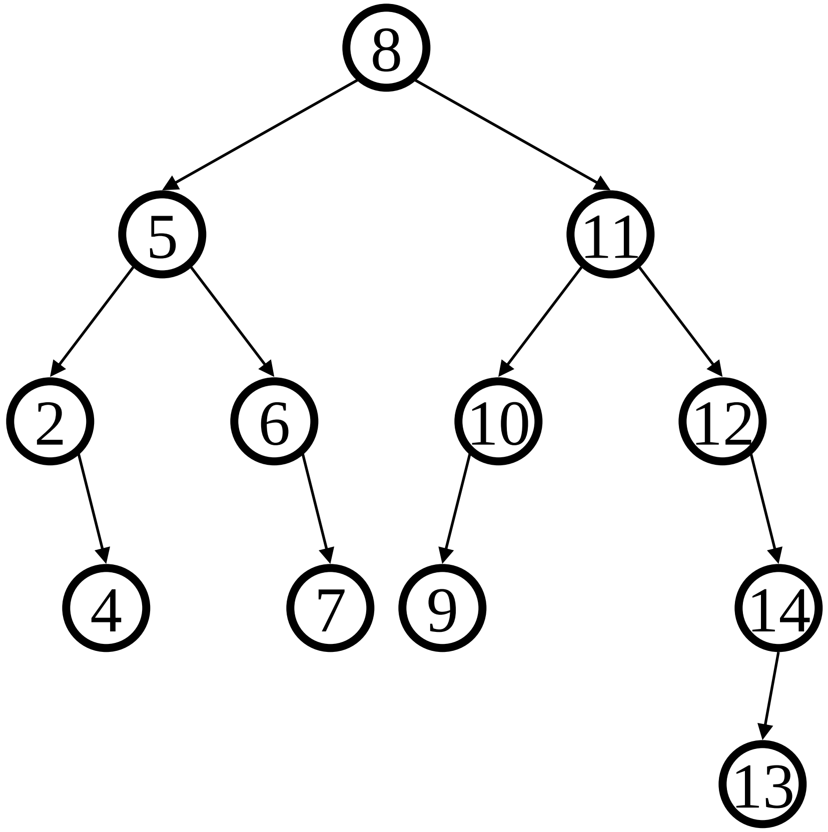
Binary Search Tree Dictionary Data Structure

Binary tree property

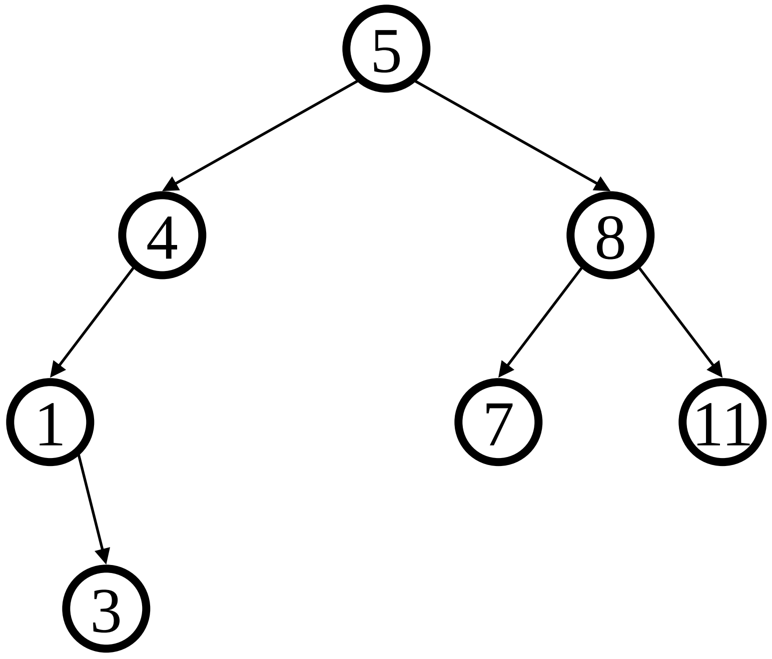
- each node has ≤ 2 children
- result:
  - storage is small
  - operations are simple
  - average depth is small

Search tree property

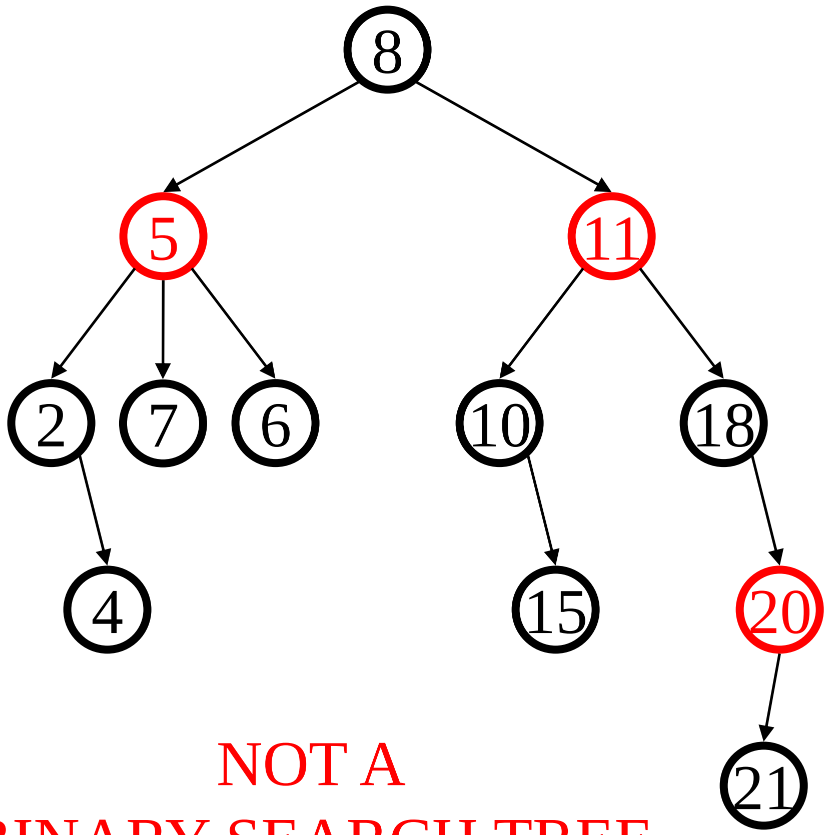
- all keys in left subtree smaller than root's key
- all keys in right subtree larger than root's key
- result:
  - easy to find any given key
  - Insert/delete by changing links



Example and Counter-Example



BINARY SEARCH TREE



NOT A
BINARY SEARCH TREE

Time compexity

	Array (unsorted)	Linked list	Array (sorted)	Binary search tree (balanced)
Search(x)	O(n)	O(n)	O(log n)	O(log n)
Insert(x)	O(1)	O(1)	O(n)	O(log n)
Remove(x)	O(n)	O(n)	O(n)	O(log n)

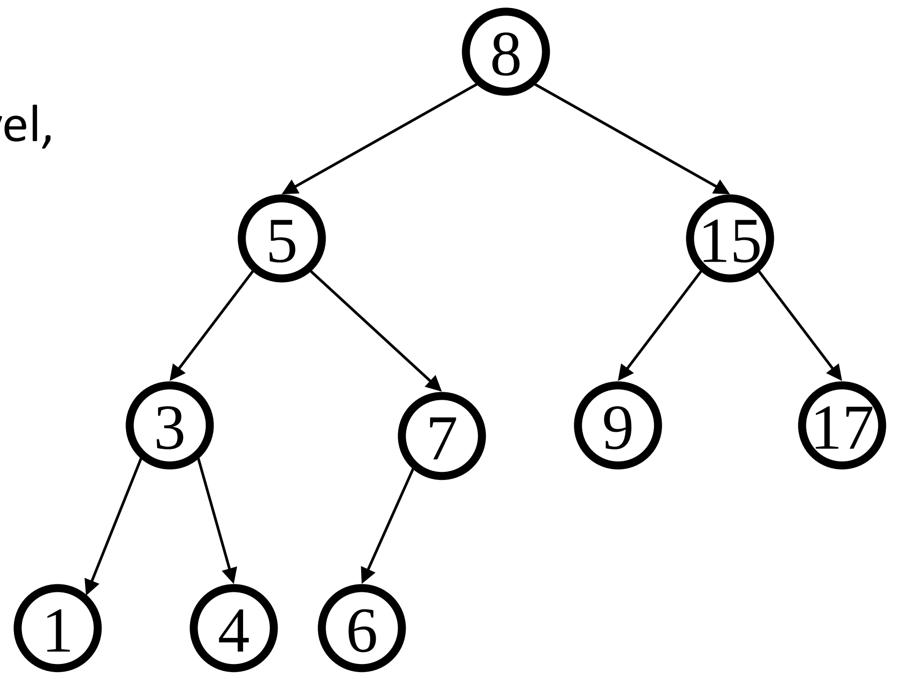
We can perform binary search in a sorted array in O(logn)

Complete Binary Search Tree

Complete binary search tree

(aka binary heap):

- Links are completely filled, except possibly bottom level, which is filled left-to-right.

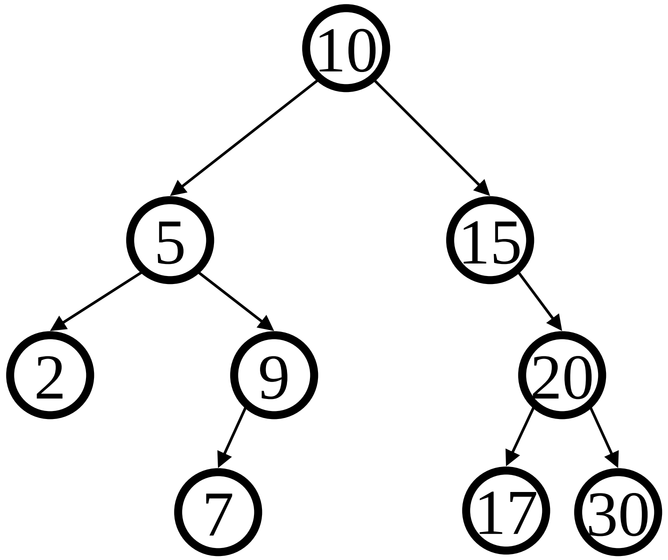


BST node creation

```
struct node
{
        int data;
        node* left;
        node* right
}
node* getnewnode(int data)
{
        node* temp = malloc(sizeof(struct node*));
        temp -> data = data;
        temp->left =temp->right = NULL;
        return temp;

}
```

Nodes will be created using malloc in c and new node in c++

Searching a key - Iterative Find



```
Node *
find(int key, Node * root)
{
  while (root != NULL && root->key !=
   key)
  {
    if (key < root->key)
      root = root->left;
    else
      root = root->right;
  }

  return root;
}
```

Insert

Concept:

- Proceed down tree as in Find
- If new key not found, then insert a new node at last spot traversed

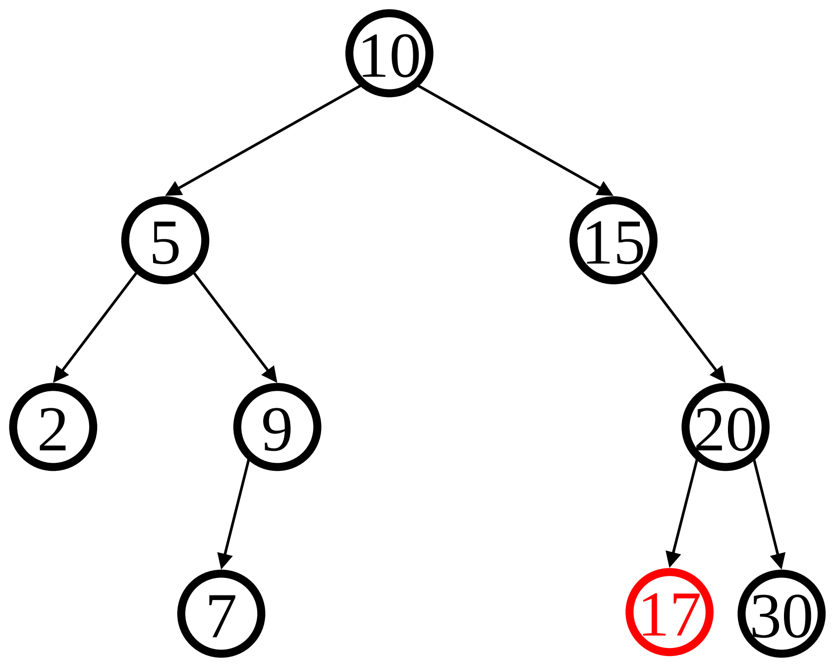
```
node* insert(node* root, int data)
{
     if ( root == NULL ) {
          root = getnewnode(data);

     else if (data < root->key)
      root->left = insert(root->left,data);

     else (data > root->key)
      root->right = insert(root->right,data);

    return root;
}
```

Deleting a node

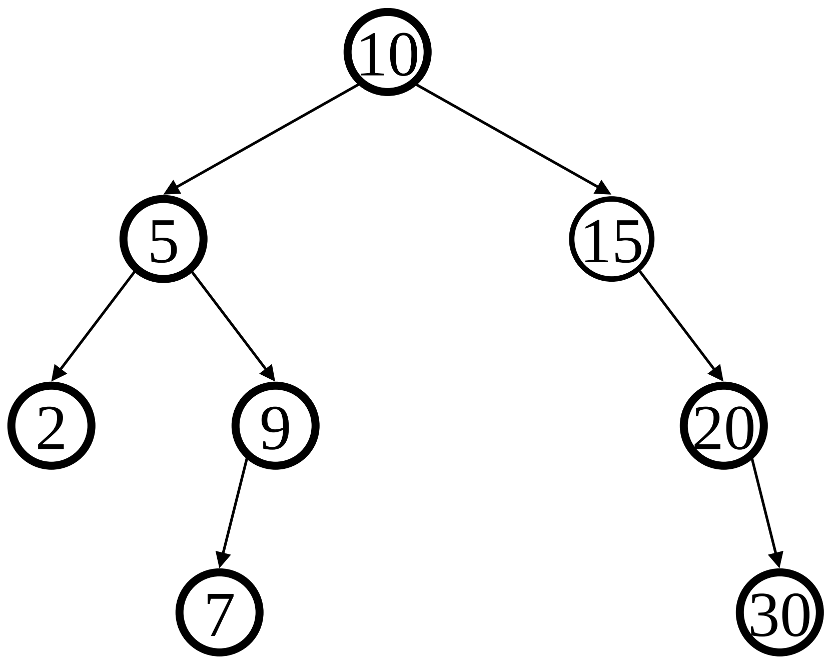
- 3 cases
  - Leaf node
  - One child
  - Two children

Deletion - Leaf Case

Delete(17)

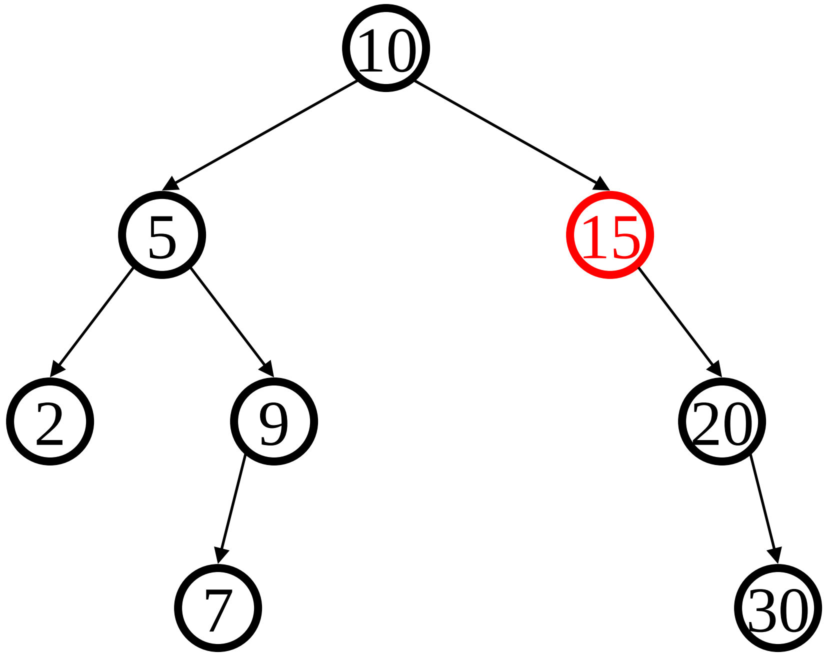


After deletion

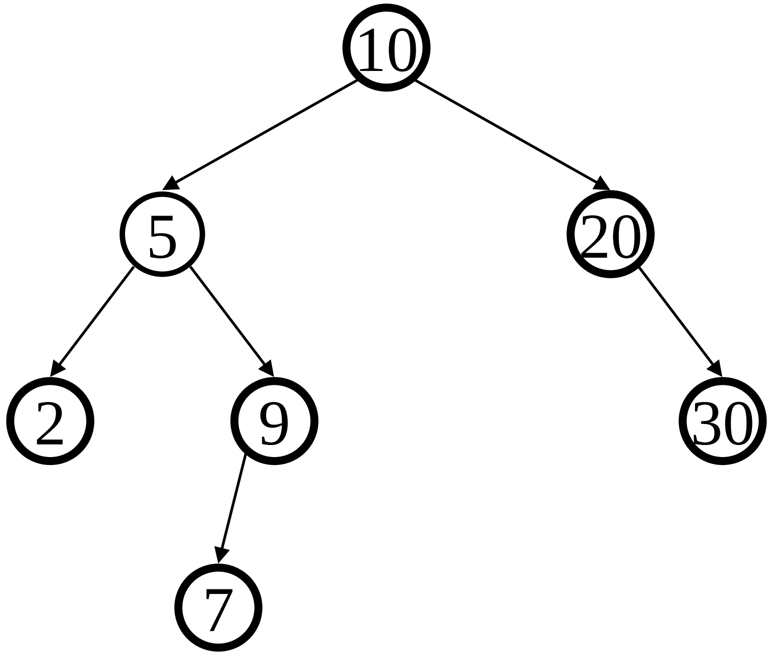


Deletion - One Child Case

Delete(15)

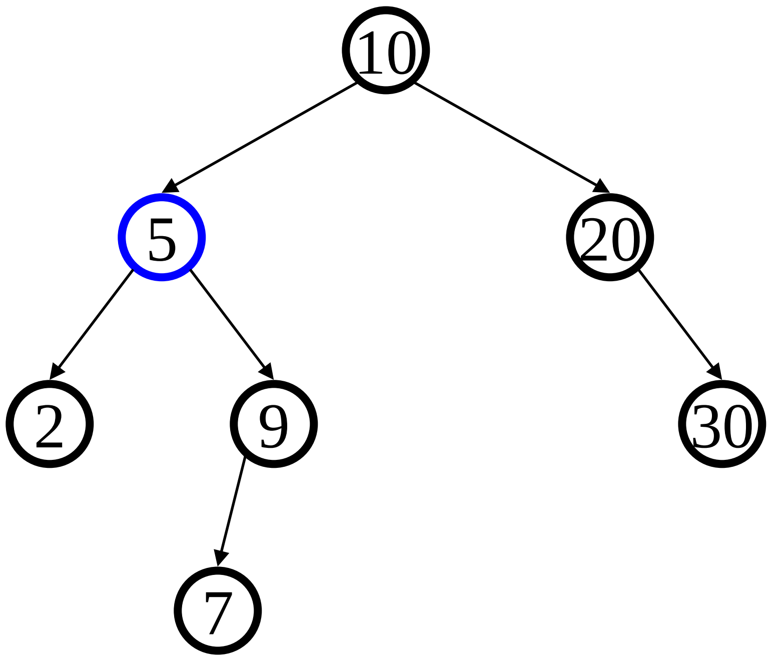


After deletion



Deletion - Two Child Case

Replace node with descendant whose value is guaranteed to be between left and right subtrees: the successor



Delete(5)

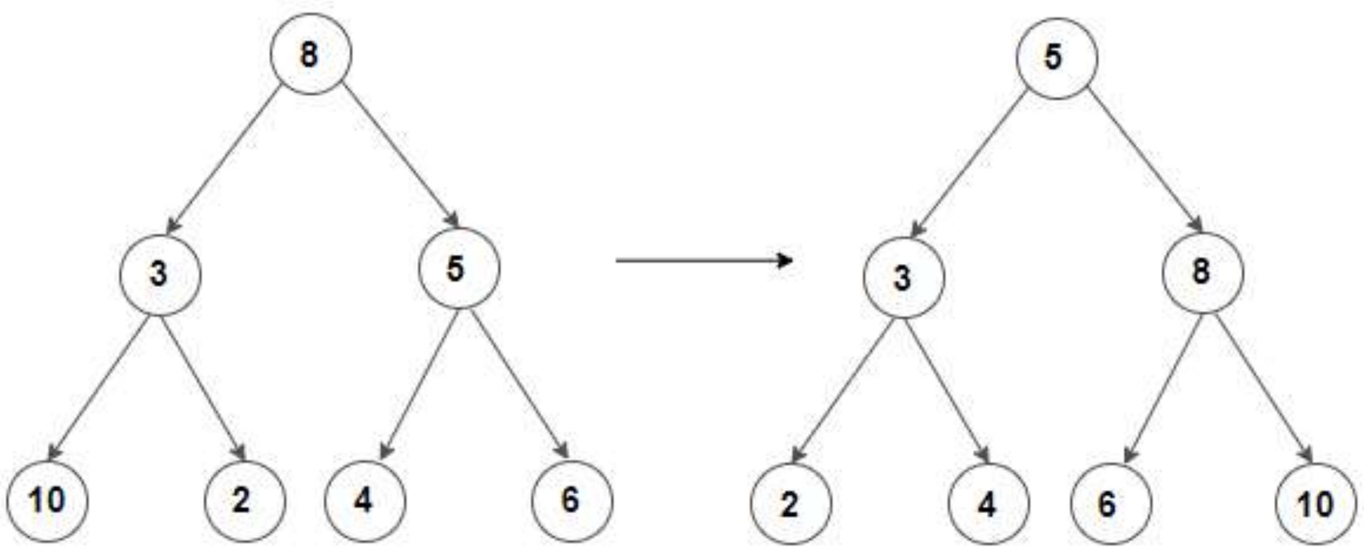
Could we have used predecessor instead?

```
Struct node* delete(struct node* root, int data)
{
   if(root === NULL)
   return root;
   else if(data < root-> data)
        root-> left = delete(root -> left, data)
   else if (data > root-> data)
        root-> right = delete(root->right, data)
   else
   {
        //case 1: leaf node
        if(root-> left == NULL && root->right == NULL)
        {
               free(root);
               root = NULL;
               return root;
        }
//case 2: one child
      else if(root-> left == NULL)
      {
            struct node* temp = root;
            root = root->right;
            free root;
            return root;
      }
      else if (root-> right ==NULL)
      {
            struct node* temp = root;
            root = root-> left;
            free(temp);
            return root;
      }
  }
```

```
else
{
        //Case 3: 2 children
        struct node* temp = findmin(root->right);
        root-> data = tmep -> data;
        root->right = free(root->right, temp-> data);
}
}
return root;
```

Conversion from binary tree to BST

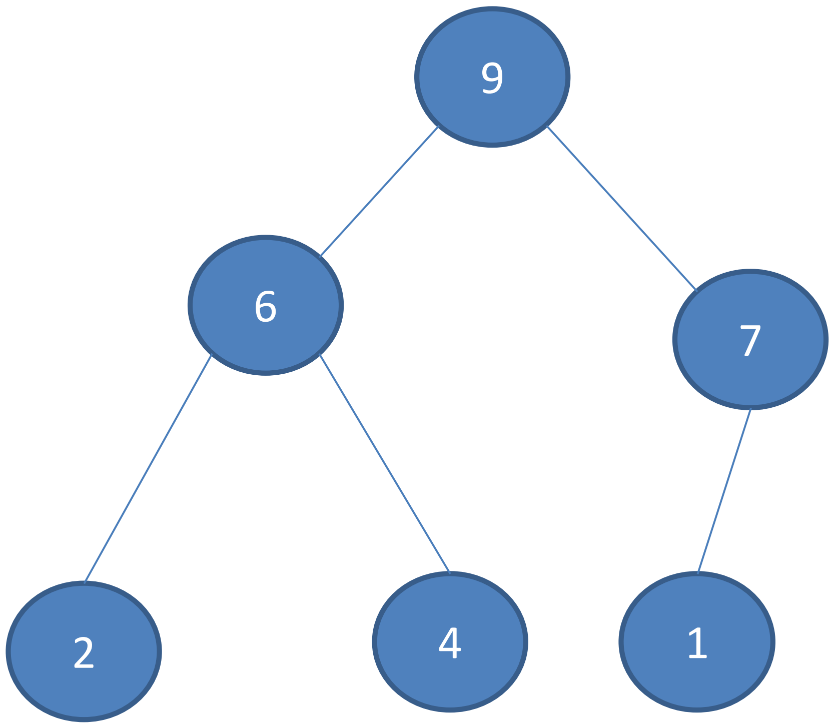
- Take the inorder of the binary tree
- Sort the inorder array
- STORE IN position based on INORDER



Heap

Introduction to heap

- A heap data structure is a binary tree with the following property:
  - It is a complete binary tree: each level of the tree is completely filled, except the bottom level. At the bottom level it is filled from left to right.
  - A Binary Heap is either Min Heap or Max Heap.
    - In a **Min Binary Heap**, the key at root must be minimum among all keys present in Binary Heap. The same property must be recursively true for all nodes in Binary Tree
    - **Max Binary Heap** – key at root is maximum.

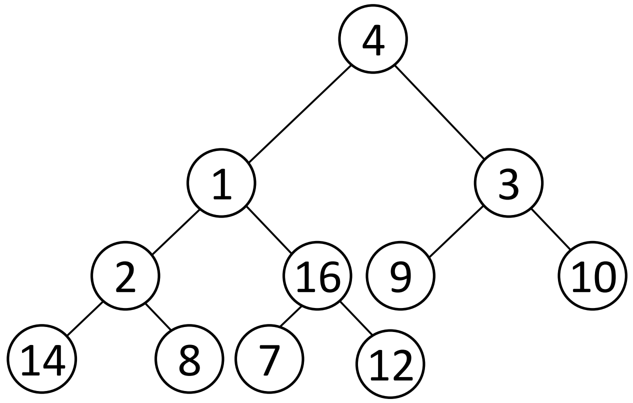


Binary Heap Example

- It is a complete binary tree
- Satisfies the heap property

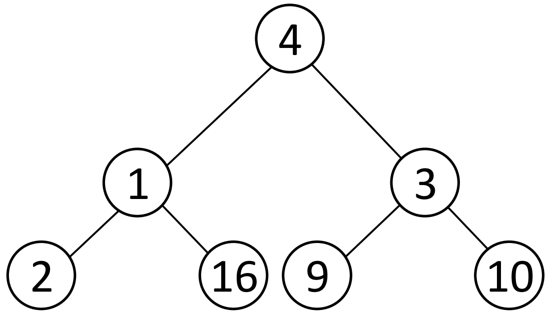
Special Types of Trees

- *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 in which all leaves are on the same level and all internal nodes have degree 2.



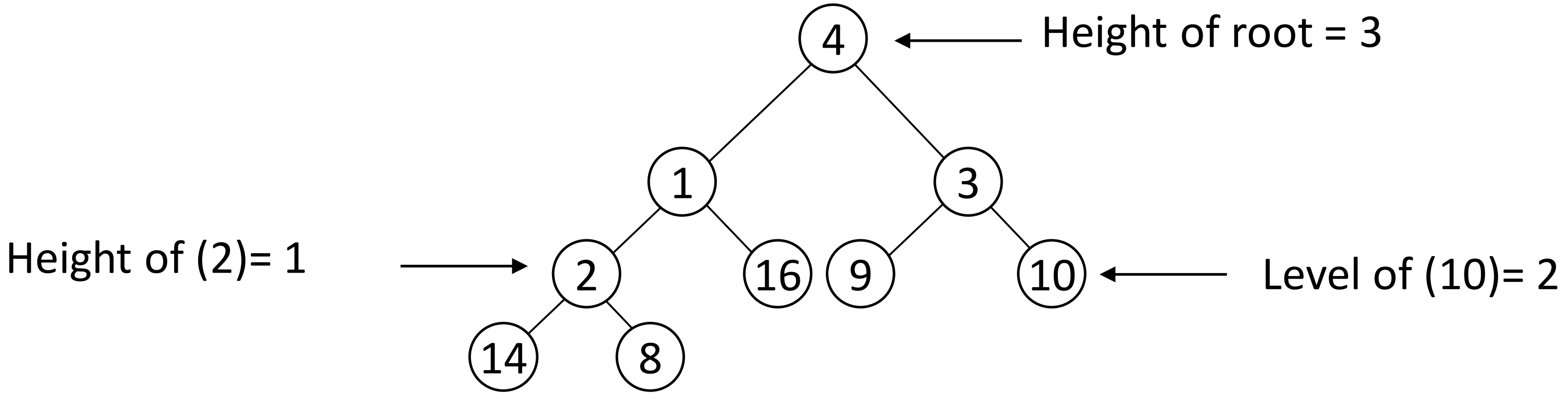
Complete binary tree

- There are **at most** 2^l nodes at level (or depth) l of a binary tree

- A binary tree with depth d has **at most** $2^{d+1} - 1$ nodes

- A binary tree with n nodes has depth **at least** $\lfloor lgn \rfloor$

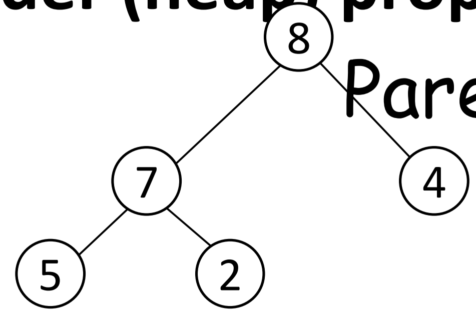
- **Height** of a node = the number of edges on the longest simple path from the node down to a leaf
- **Level** of a node = the length of a path from the root to the node
- **Height** of tree = height of root node



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) ≥ 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

Heap Types

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

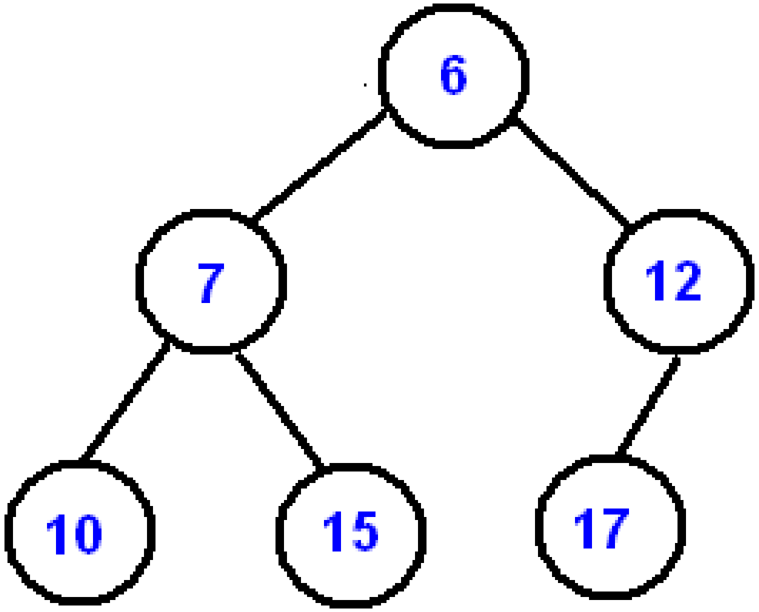
$$A[PARENT(i)] \geq A[i]$$

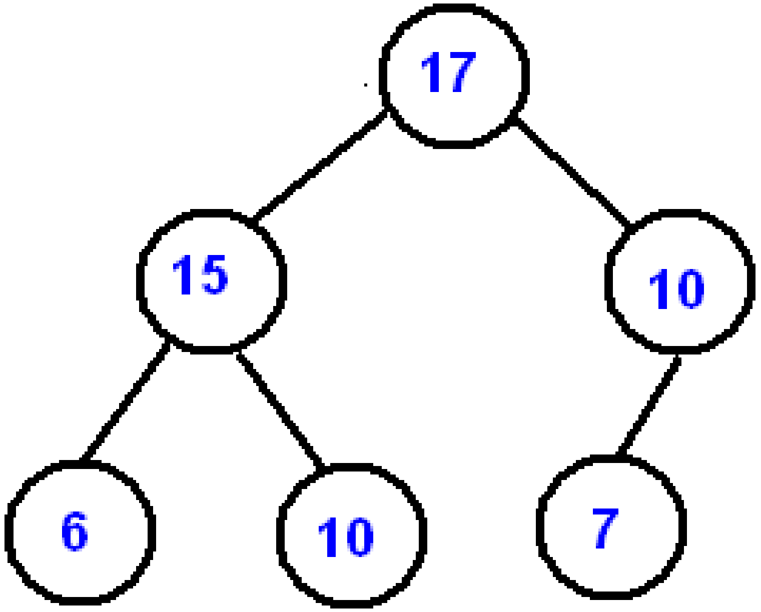
- **Min-heaps** (smallest element at root), have the *min-heap property:*
  - for all nodes i, excluding the root:

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

Min heap

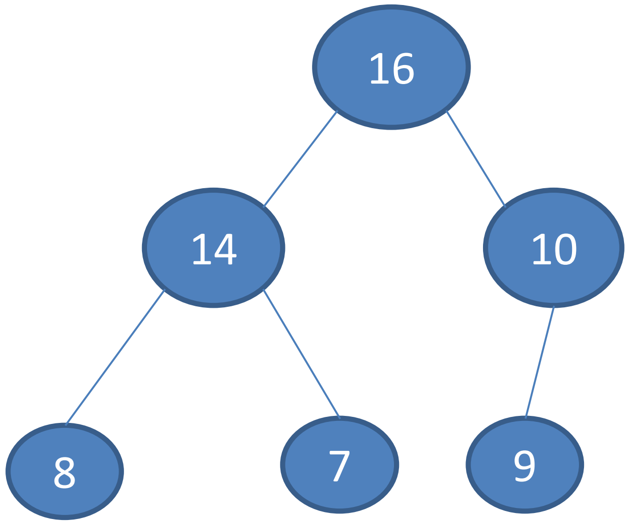
Max heap



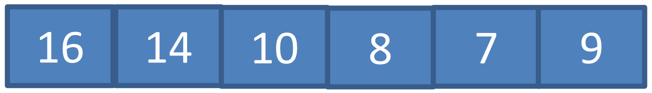


Array representation

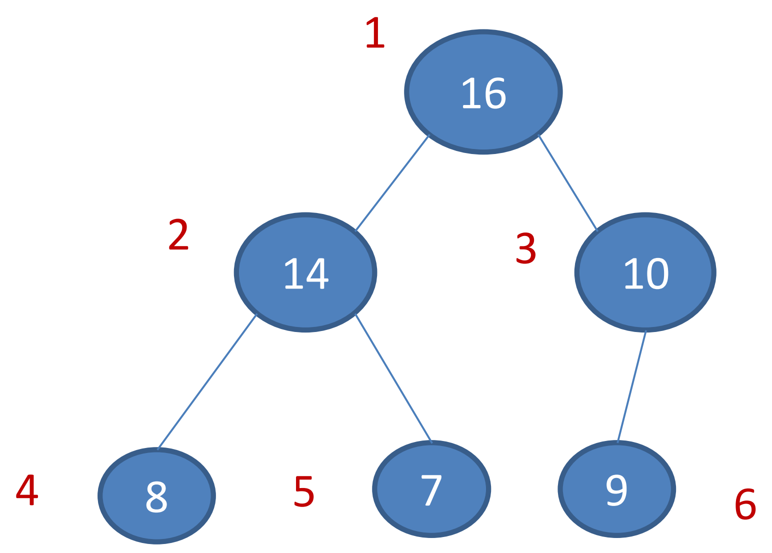
- Array representation has two attributes
  - Length(array) – number of elements in the array
  - heap_size(array) – the number of elements in the heap stored in the array.



Array representation of heap



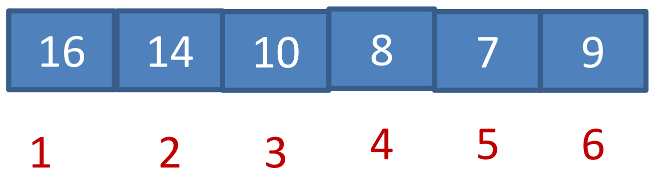
- For an array
  - *length(array) = heap_size(array)*
  - For a node *i*, the parent node is *i/2*
  - For a node *i*, left child is *2*i* and right child is *2*i + 1*



Example

For node i=2, parent is i/2= 2/2 = 1
Left child is 2*I = 2*2 = 4
Right child is 2*i + 1 = 5

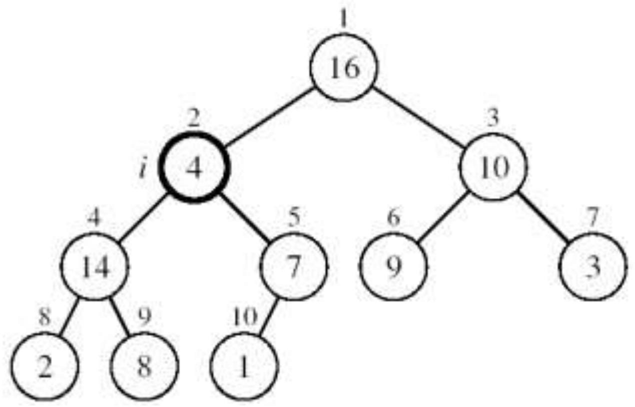
Array representation of heap



16	14	10	8	7	9
1	2	3	4	5	6

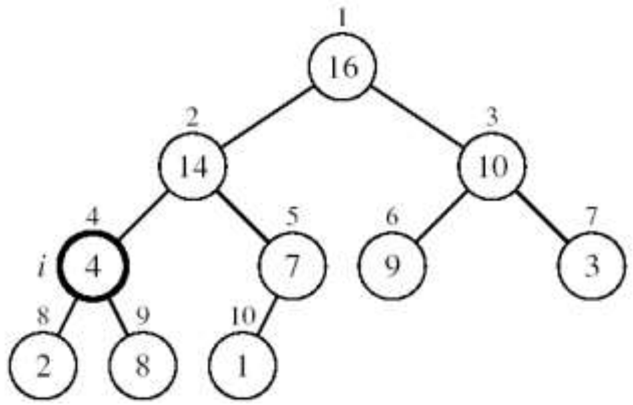
Example

<u>MAX-HEAPIFY(A, 2, 10)</u>



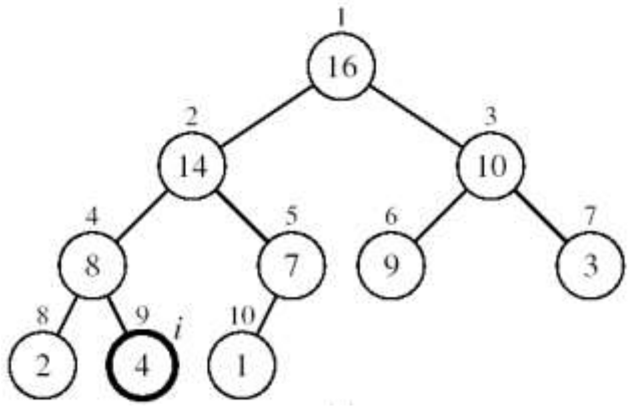
A[2] violates the heap property

A[2] ↔ A[4]



A[4] violates the heap property

A[4] ↔ A[9]



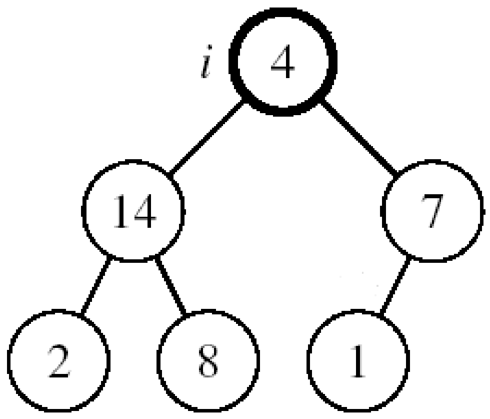
Heap property restored

Maintaining the Heap Property

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

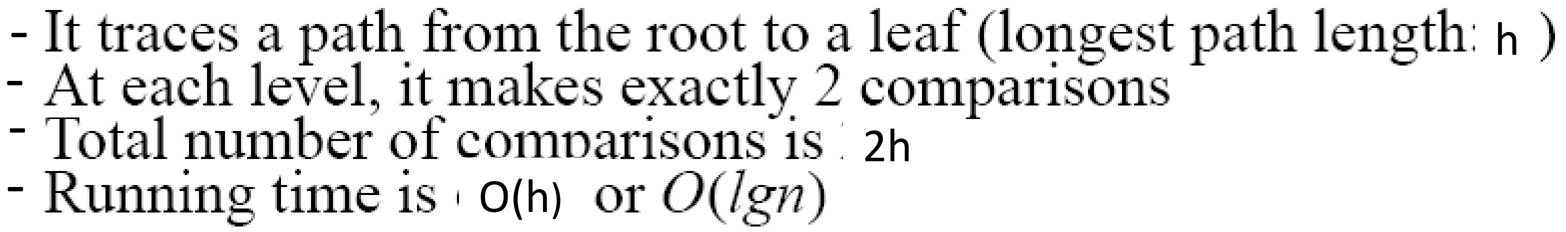
Alg: MAX-HEAPIFY(*A*, i, n)

```
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)
```



MAX-HEAPIFY Running Time

- Intuitively:



- It traces a path from the root to a leaf (longest path length: h)
- At each level, it makes exactly 2 comparisons
- Total number of comparisons is 2h
- Running time is O(h) or $O(lgn)$

- Running time of MAX-HEAPIFY is $O(lgn)$
- Can be written in terms of the height of the heap, as being $O(h)$
  - Since the height of the heap is $\lfloor lgn \rfloor$

Heap sort

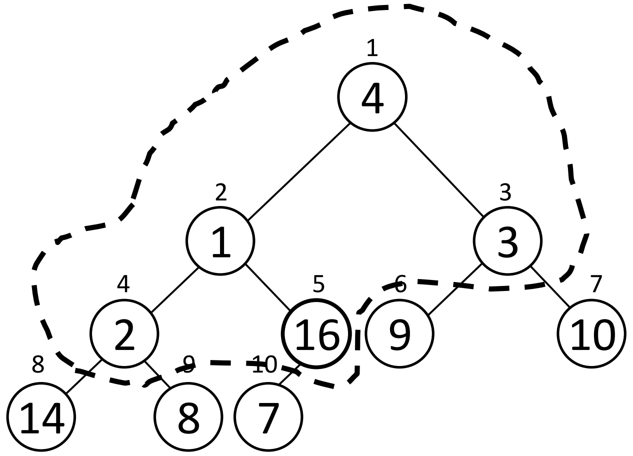
1. Build the complete binary tree using given elements
2. Create a max heap to sort in ascending order.
3. Once the heap is created, swap the last node with the root node and delete the last node from the heap.
4. Repeat 2 and 3 until the heap is empty

Building a Heap

- Convert an array $A[1 \dots n]$ 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:

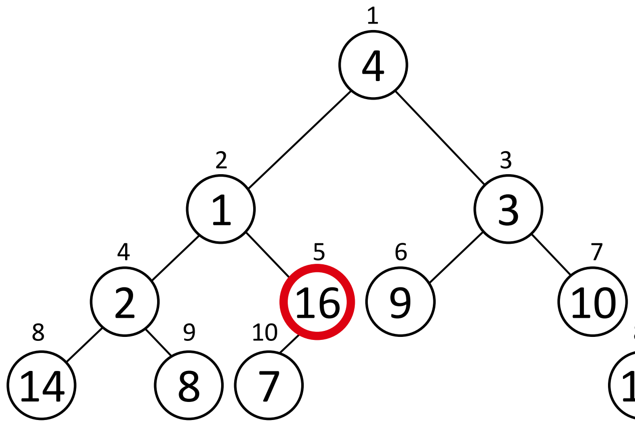
4	1	3	2	16	9	10	14	8	7

Example:

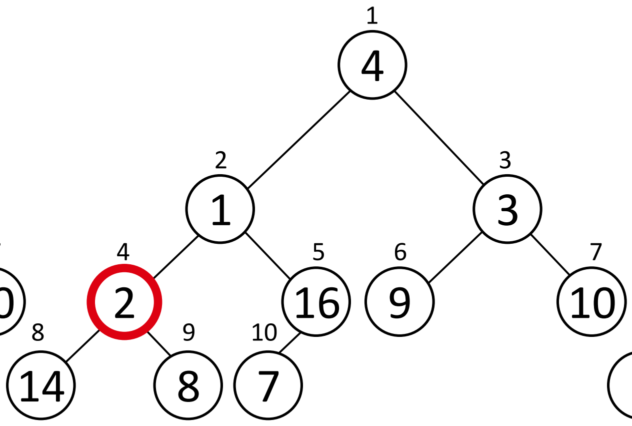
A

4	1	3	2	16	9	10	14	8	7

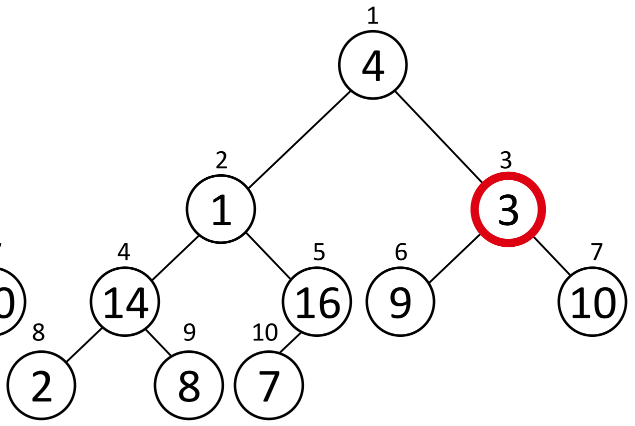
$i = 5$



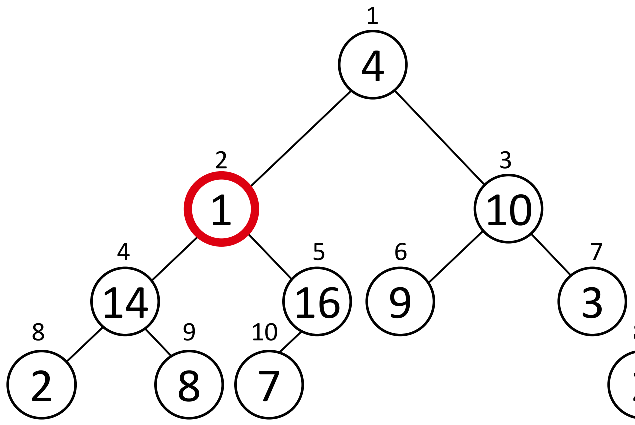
$i = 4$



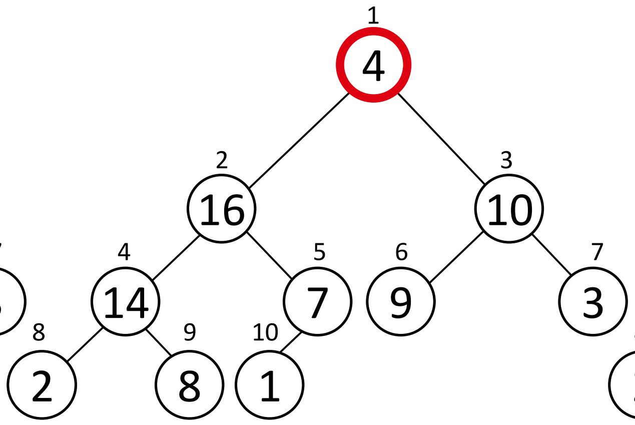
$i = 3$

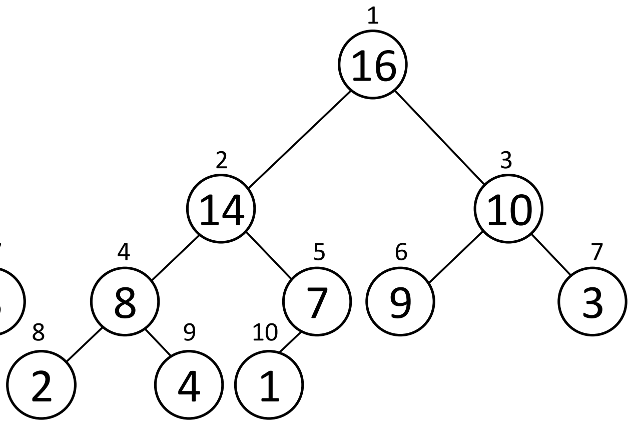


$i = 2$



$i = 1$





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)
```

$$\left.\begin{array}{r} \\ O(\lg n) \end{array}\right\} O(n)$$

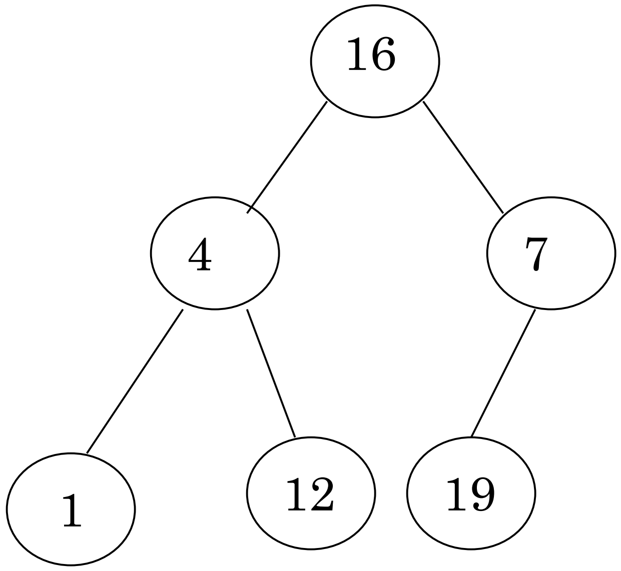
$\Rightarrow$ Running time: $O(n\lg n)$

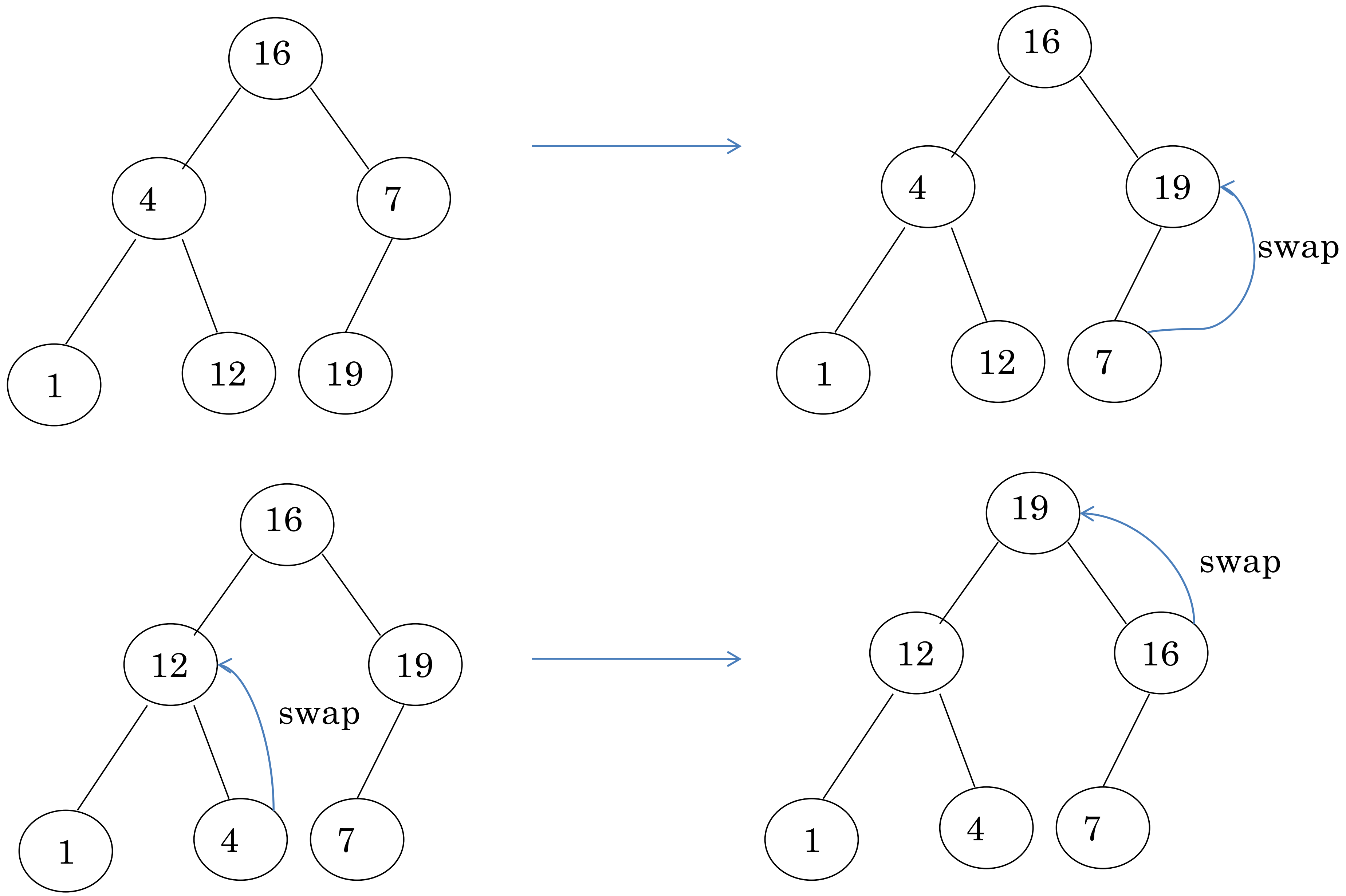
- This is not an asymptotically tight upper bound

Example: Convert the following array to a heap and sort

16	4	7	1	12	19

Picture **the array as a complete binary tree:**

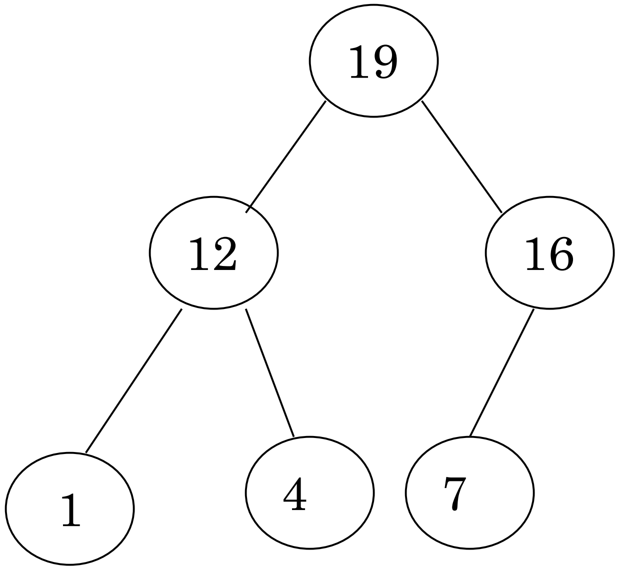



16
4
7
1
12
19
16
4
19
1
12
7
swap
16
12
19
1
4
7
swap
19
12
16
1
4
7
swap

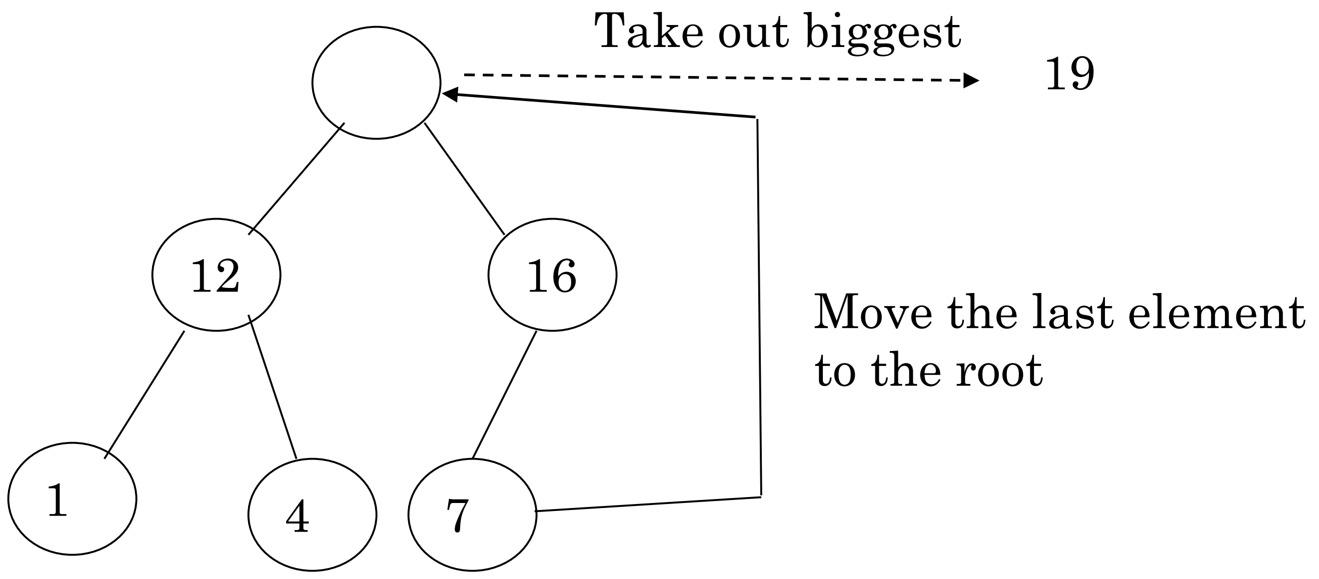
Heap Sort

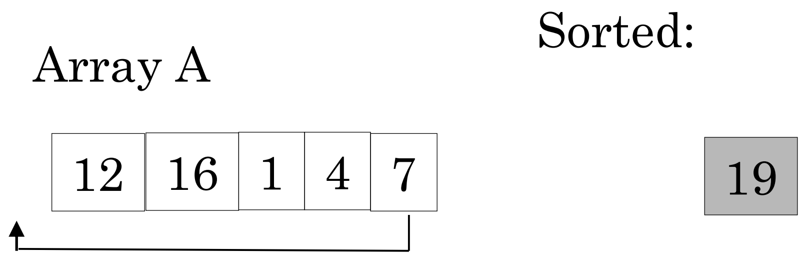
- The heapsort algorithm consists of two phases:
  - build a heap from an arbitrary array
  - use the heap to sort the data

- To sort the elements in the decreasing order, use a min heap
- To sort the elements in the increasing order, use a max heap



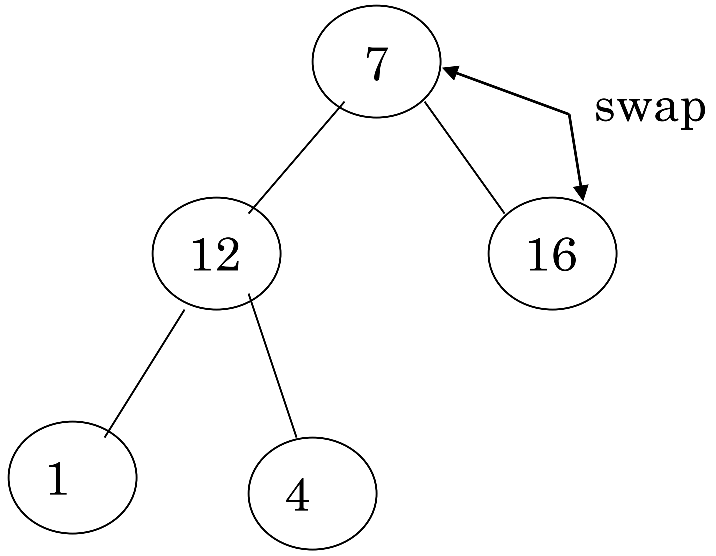
Example of Heap Sort





12	16	1	4	7

HEAPIFY()

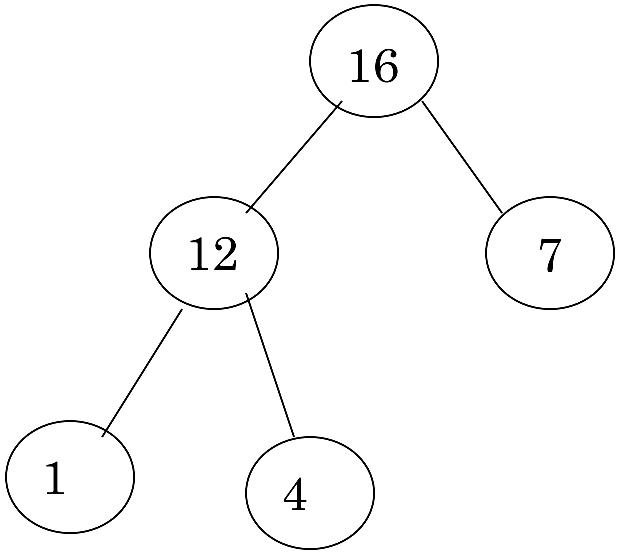


Array A

7	12	16	1	4

Sorted:

19

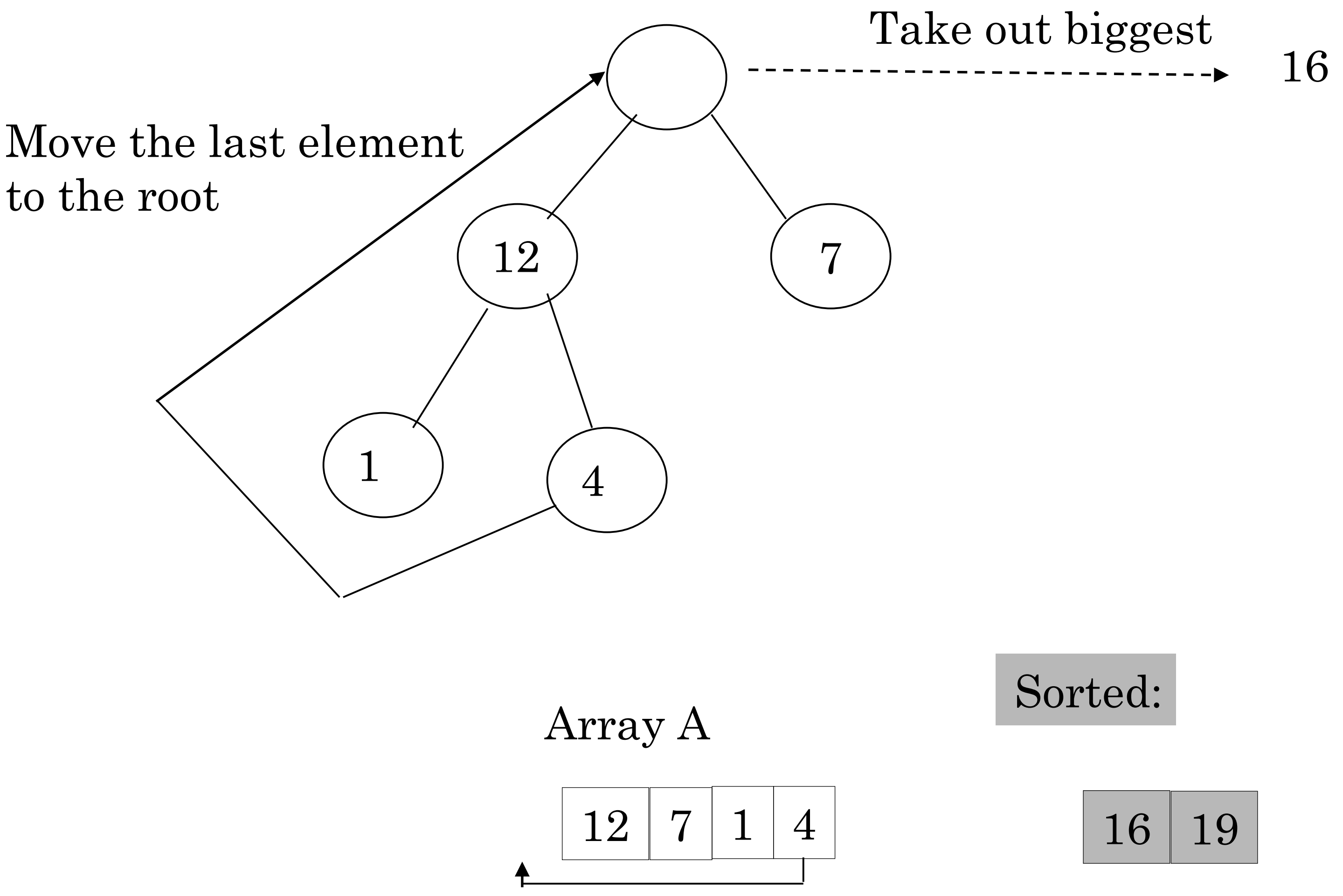


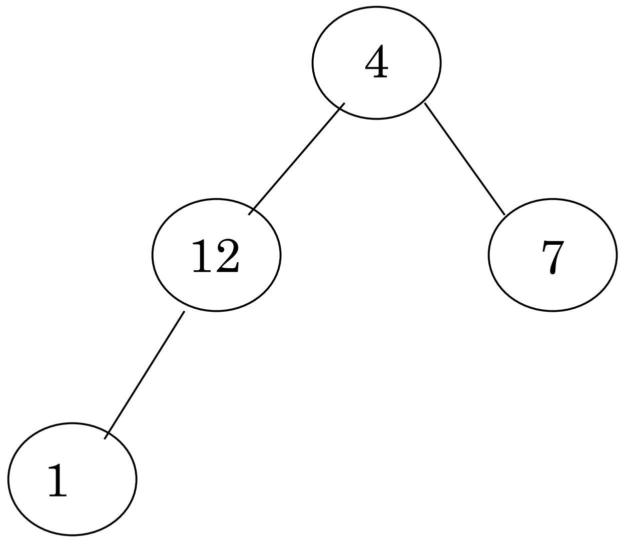
Array A

16	12	7	1	4

Sorted:

19


Take out biggest
16
Move the last element
to the root
12
7
1
4
Array A
12 7 1 4
Sorted:
16 19



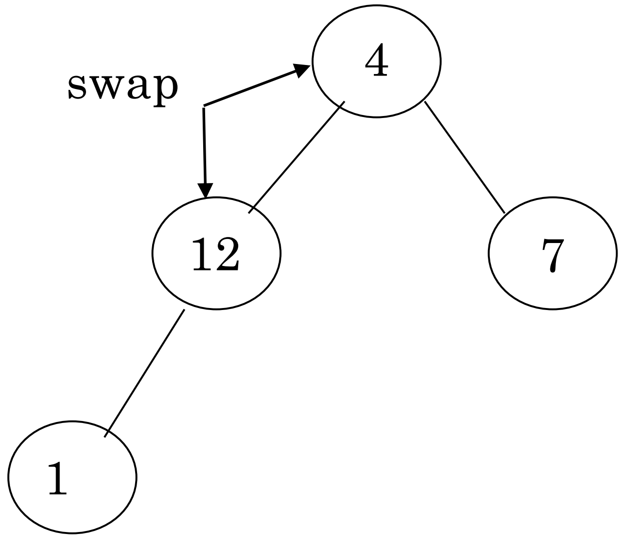
Sorted:

Array A

4	12	7	1

16	19

HEAPIFY()

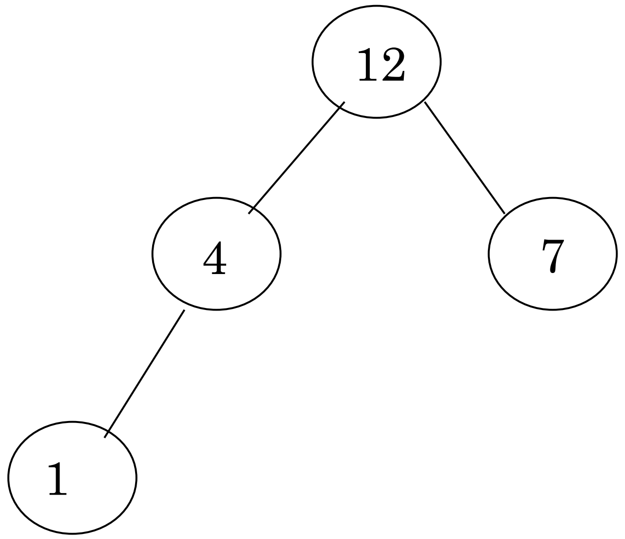


Array A

4	12	7	1

Sorted:

16	19

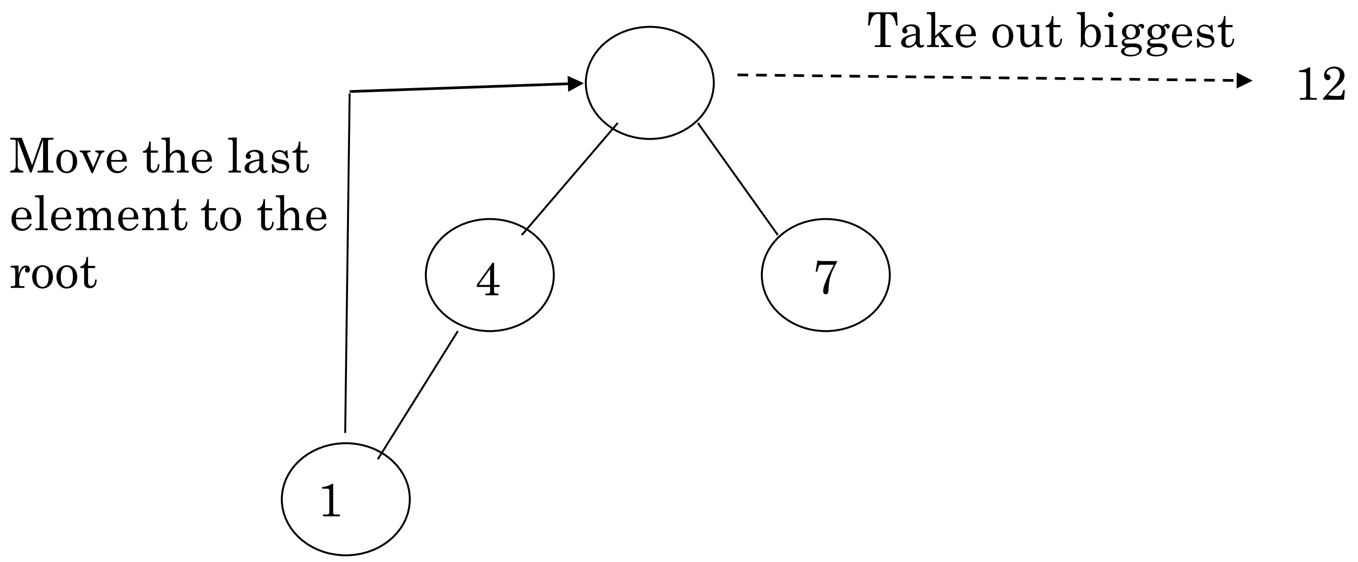


Array A

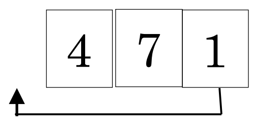
12	4	7	1

Sorted:

16	19

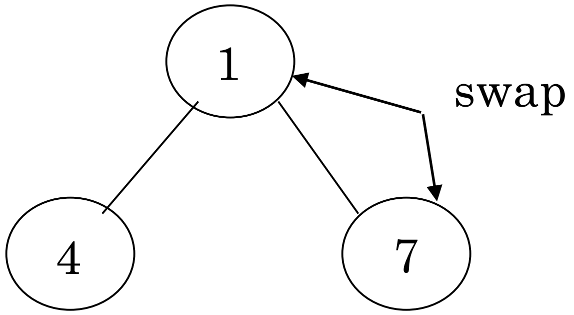


Array A



Sorted:

12	16	19

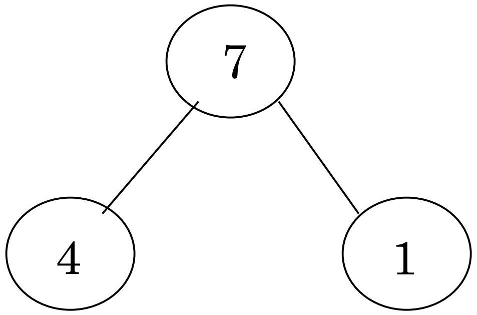


Array A

1	4	7

Sorted:

12	16	19

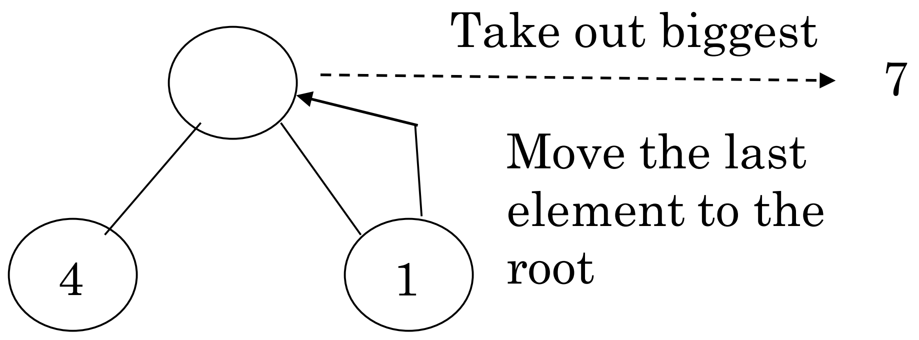


Array A

7	4	1

Sorted:

12	16	19



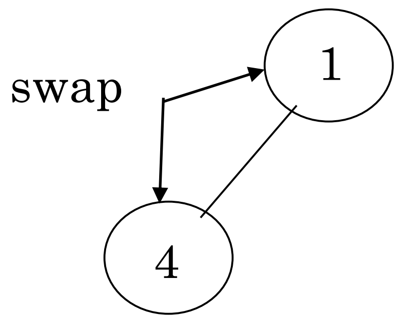
Array A

1	4

Sorted:

7	12	16	19

HEAPIFY()



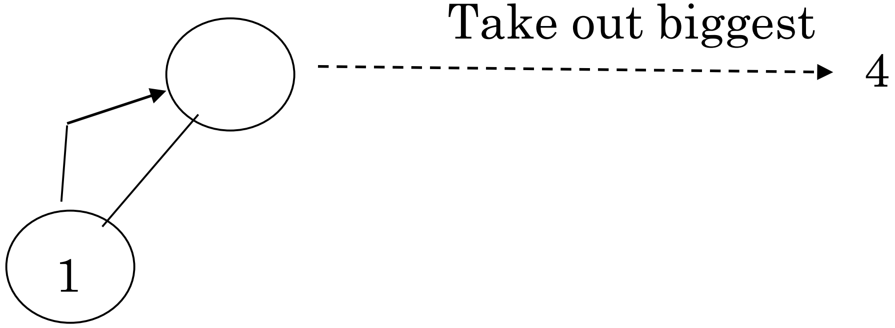
Array A

4	1

Sorted:

7	12	16	19

Move the last element to the root

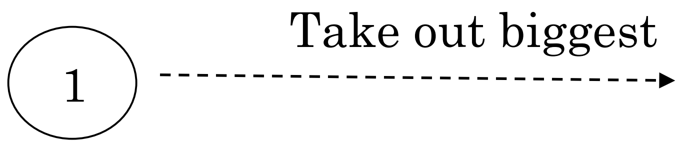


Sorted:

Array A

1

4	7	12	16	19



Sorted:

Array A

1	4	7	12	16	19

Sorted:

1	4	7	12	16	19

Analysis – summary

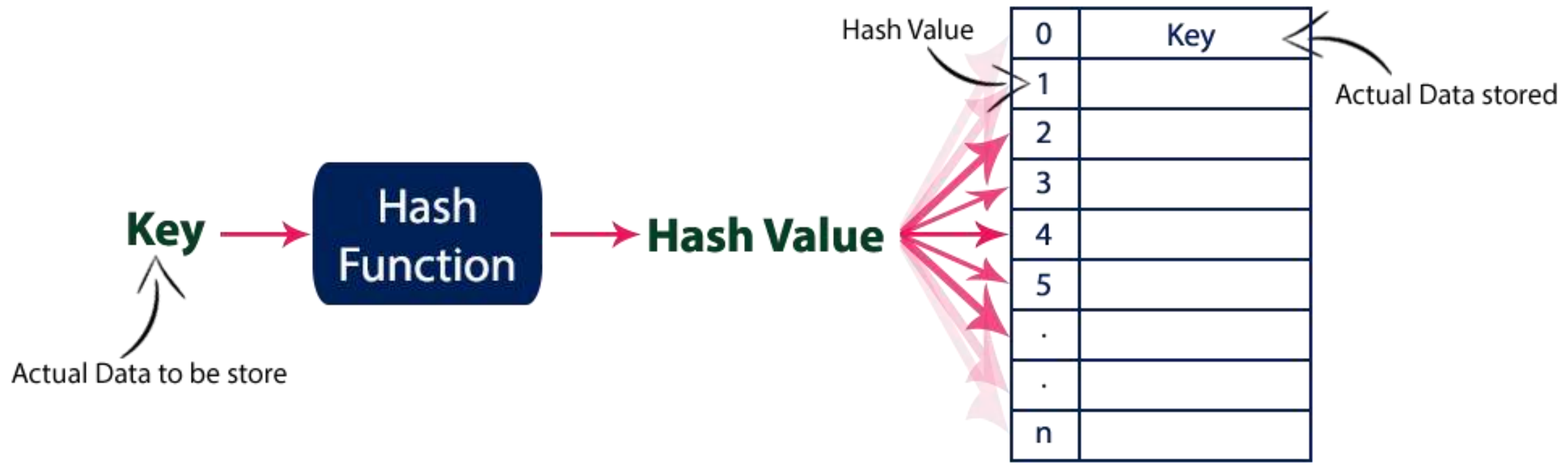
- Running time
  - time to build max-heap is O(N)
  - time for N DeleteMax operations is N O(log N)
  - total time is **O(N log N)**
- Can also show that running time is Ω(N log N) for some inputs,
  - so *worst case* is **Θ(N log N)**
  - *Average case* running time is also O(N log N)
- Heapsort is in-place but not stable (why?)

Hashing

- Sequential search requires, on the average O(n) comparisons to locate an element. This is not desirable in large databases.
- Binary search requires much fewer comparisons on the average O(log n) but there is an additional requirement that the data should be sorted. Sorting of elements require O(nlog n) comparisons.
- An widely used technque called hashing is used ot store the data.
  - its best case running time is O(1)
  - and in worst case it is O(n).

What is hashing?

- In hashing, the record for a key value 'key', is directly referred by calculating the address from the key value.
- Address or location of an element or record x, is obtained using arithmetic function (hash function) f
- f(key) gives the address of x in the table



- Records may be stored in unlimited space. It can be stored in a hard disk (open hashing/external hashing)
- Records is stored in a fixed space (closed hashing/internal hashing)

Properties of good hash function

- Must return number 0, …, tablesize
- Should be efficiently computable – O(1) time
- Should not waste space unnecessarily
  - For every index, there is at least one key that hashes to it
  - Load factor lambda λ = (number of keys / TableSize)
- Should minimize collisions

  = different keys hashing to same index

General idea

- The ideal hash table structure is merely an array of some fixed size, containing the items.
- A stored item needs to have a data member, called ***key***, that will be used in computing the index value for the item.
  - Key could be an *integer*, a *string*, etc
  - e.g. a name or Id that is a part of a large employee structure
- The size of the array is *TableSize*.
- The items that are stored in the hash table are indexed by values from *0* to *TableSize – 1*.
- Each key is mapped into some number in the range 0 to *TableSize – 1.*
- The mapping is called a *hash function.*

Hash functions

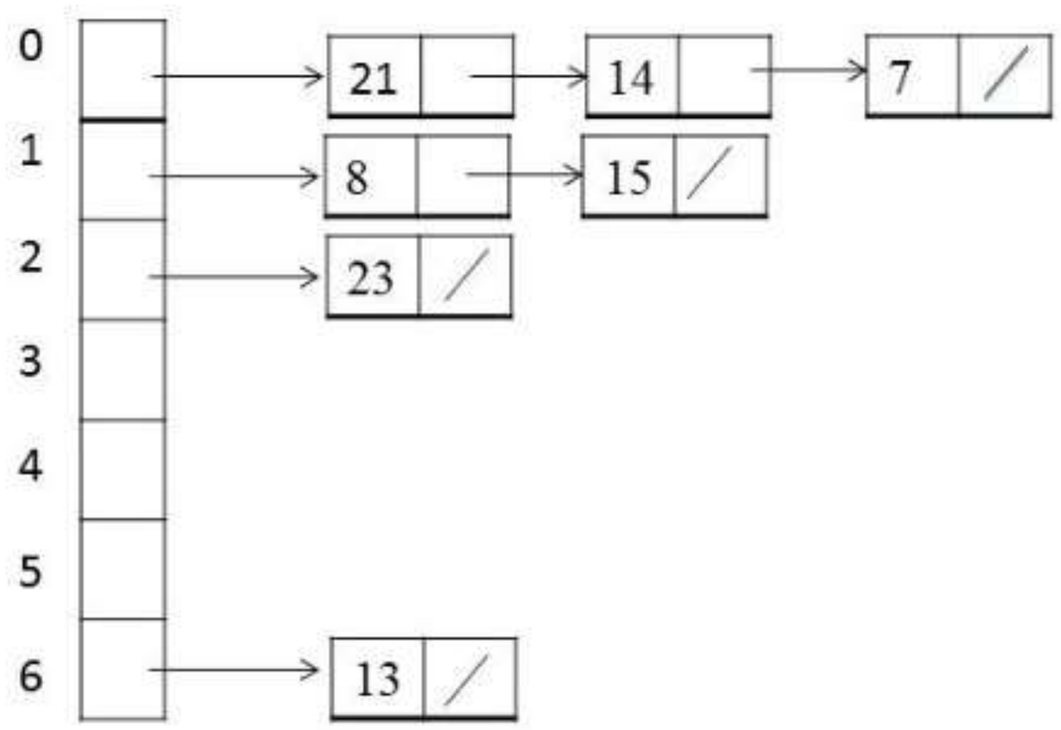
- If the input keys are integers then simply *Key* mod *TableSize* is a general strategy.
  - Unless key happens to have some undesirable properties. (e.g. all keys end in 0 and we use mod 10)
- If the keys are strings, hash function needs more care.
  - First convert it into a numeric value.

Collisions and their Resolution

- A collision occurs when two different keys hash to the same value
  - E.g. For *TableSize* = 17, the keys 18 and 35 hash to the same value
  - 18 mod 17 = 1 and 35 mod 17 = 1
- Cannot store both data records in the same slot in array!
- Two different methods for collision resolution:
  - **Separate Chaining:** Use a dictionary data structure (such as a linked list) to store multiple items that hash to the same slot
  - **Open addressing:** All elements are stored in the database itself.

Separate chaining

- In this strategy, a separate list (linked list) of all elements that is mapped to the same value is maintained

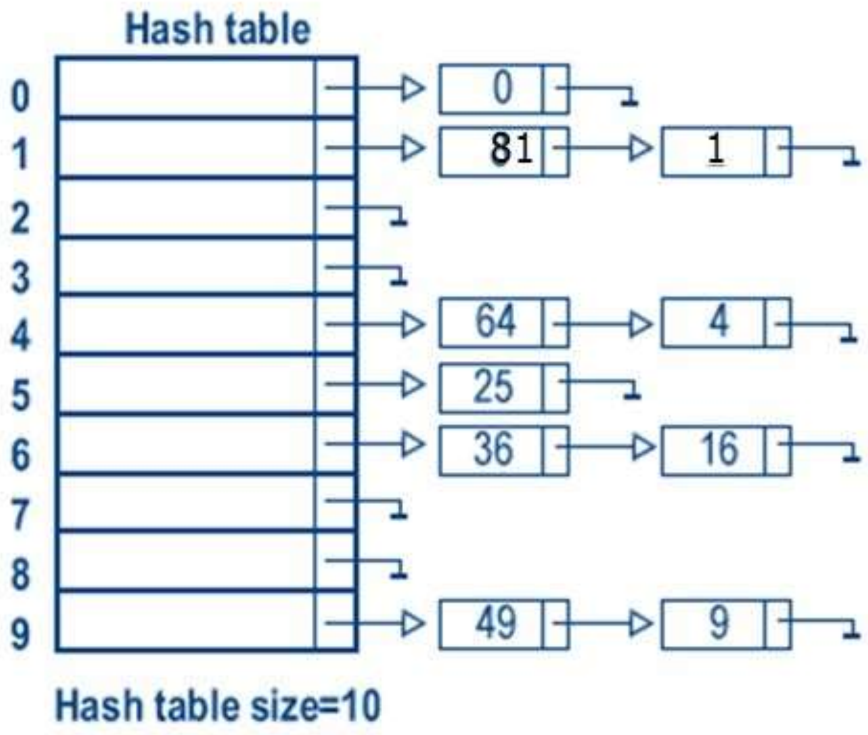


Collision Resolution: Separate Chaining

- Keep a list of all elements that hash to the same location
- For example: Suppose we use *hash*(*x*) = *x* mod 10 as the hash function.

  0, 81, 1, 64, 4, 25, 36, 6, 49, 9

  hash the above keys of items to a hash table of size 10.



- **Advantages:**
  - Simple to implement.
  - Hash table never fills up, we can always add more elements to chain.
  - Less sensitive to the hash function or load factors.
  - It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted
- **Disadvantages:**
  - Cache performance of chaining is not good as keys are stored using linked list. Open addressing provides better cache performance as everything is stored in same table.
  - Wastage of Space (Some Parts of hash table are never used)
  - If the chain becomes long, then search time can become O(n) in worst case.
  - Uses extra space for links.

Performance of chaining

m = Number of slots in hash table
n = Number of keys to be inserted in hash table

Load factor, $\alpha = n/m$

Expected time to search = $O(1 + \alpha)$

Expected time to insert/delete = $O(1 + \alpha)$

Time complexity of search insert and delete is $O(1)$ if α is $O(1)$

Open Addressing

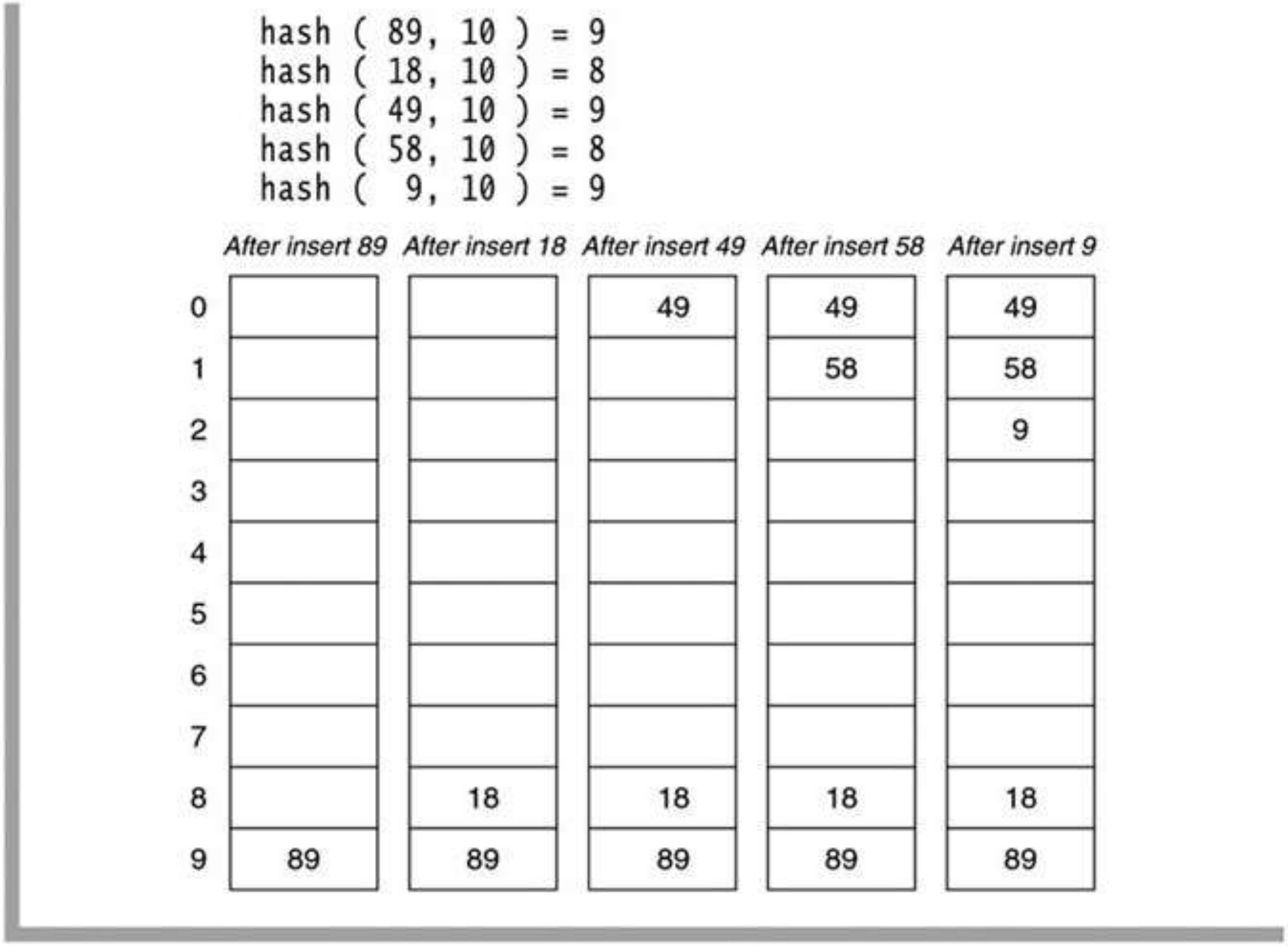
- In Open Addressing, all elements are stored in the hash table itself. So at any point, size of the table must be greater than or equal to the total number of keys.
- If collision occurs alternate cells are tried until an empty cell is found.
  - Linear probing
  - Quadratic probing
  - Double hashing

Linear probing

- In linear probing, whenever there is collision, cells are searched sequentially for an empty cell
- The result of inserting keys {89, 18, 49, 58, 9}

Linear probing: example

h(k,n) = k % n



```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9
```

	After insert 89	*After insert 18*	*After insert 49*	*After insert 58*	*After insert 9*
0			49	49	49
1				58	58
2					9
3					
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89

- Linear probing is easy to implement but it suffers from primary clustering.
- When many keys are mapped to the same location (clustering) linear probing will not distribute these keys evenly in the hash table.
- These keys will be stored in neighbourhood of the location where they are mapped.
- This will lead to clustering of keys around the point of collision.

Quadratic probing

- Quadratic probing eliminates primary clusters.
- The probe sequences are then given by:

$$h_i(key) = [h(key) + i^2] \% tableSize$$

$$\textit{for } i = 0, 1, \ldots, tableSize - 1$$

- Quadratic probing does not ensure that all cells in the table will be examined to find an empty cell.
- Thus, it may be possible that key will not be inserted even if there is an empty cell in the table.

- Load the keys *23, 13, 21, 14, 7, 8* in this order, in a hash table of size *7* using quadratic probing with $c(i) = i^2$ and the hash function: $h(key) = key \% 7$

$$h_i(key) = (h(key) + i^2) \% 7$$

where i = 0, 1, 2, 3

Hash table

23

0	
1	
2	23
3	
4	
5	
6	

h_0 (23) = (23 % 7) = 2

13

0	
1	
2	23
3	
4	
5	
6	13

h_0 (13) = (13 % 7) = 6

21

0	21
1	
2	23
3	
4	
5	
6	13

h_0 (21) = (21 % 7) = 0

14

0	21
1	14
2	23
3	
4	
5	
6	13

h_0 (14) = (14 % 7) = 0

collision

h_1(14) = (0 + 1 ^{2}) % 7 = 1

7

0	21
1	14
2	23
3	
4	7
5	
6	13

h_0 (7) = (7 % 7) = 0

collision

h_1 (7) = (0 + 1 ^{2}) % 7) = 1

collision

h_2 (7) = (0 + 2 ^{2}) % 7) = 4

8

0	21
1	14
2	23
3	
4	7
5	8
6	13

h_0 (8) = (8 % 7) = 1

collision

h_1 (8) = (1 + 1 ^{2}) % 7) = 2

collision

h_2 (8) = (1 + 2 ^{2}) % 7) = 5

Double hashing

- This method requires two hashing function $h_1(key)$ and $h_2(key)$.
- The problem of clustering is handled using double hashing.
- $h_1(key)$ - is called primary hash function.
- In case the address evaluated by $h_1(key)$ is already occupied, then $h_2(key)$ will be evaluated.

$$(h_1(key) + i * h_2(key)) \% TABLE\_SIZE$$

- First hash function is typically

  $h_1(key) = key \% TABLE\_SIZE$

- A popular second hash function is :

$h_2(key) = PRIME - (key \% PRIME)$ where PRIME is a prime smaller than the TABLE_SIZE.

- Insert keys 89, 18, 49, 58, 69 using double hashing (assume table size to be 10)