Advanced Data Structures

Vasudevan T V

Course Contents

- Module 1 - Basic Data Structures, Set Data Structure, Hashing, Amortised Analysis
- Module 2 - Advanced Tree Structures
- Module 3 - Advanced Heap Structures
- Module 4 - Advanced Graph Structures
- Module 5 - Blockchain Data Structure
- For Syllabus, visit - www.ktu.edu.in and goto Academic $\rightarrow$ Regulations & Syllabus $\rightarrow$ MCA2020 $\rightarrow$ Syllabus

Module 1
Review of Basic Data Structures

Definition

A data structure is an organisation of data in a particular format that enables efficient access and modification

Classification of data structures

- *Linear:* Here data elements are stored sequentially - array, stack, queue, linked list
- *NonLinear:* Here data elements are stored in a non-sequential manner - tree, graph

Arrays

Definition

An array is a collection of elements, where each element is identified by at least one index or key.

```
int number[5];                          /*  Declaration  */

int number[5] = {1,2,3,4,5};    /* Initialisation */

number[0] = 1;
number[1] = 2;
number[2] = 3;     /* Individual element initialisation */
number[3] = 4;
number[4] = 5;
```

One Dimensional Arrays

- In a one dimensional array, each element is identified by only one index or key
- The starting index can be 0, 1 or n
- zero-based indexing is used in C and Java
- Example

Number

1	2	3	4	5
0	1	2	3	4

Multidimensional Arrays

- In a multidimensional array, each element is identified by more than one index or key
- An array which uses 2 indexes is a two dimensional array
- An array which uses 3 indexes is a three dimensional array
- An n-dimensional array uses n indexes

Two Dimensional Arrays

- A two dimensional array can be used for storing a matrix
- Example

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

- The elements of multidimensional arrays are also stored sequentially.
- The elements of a $m \times n$ array are stored in m.n sequential memory locations
- The elements of the above 3×3 matrix will be stored in 9 sequential memory locations

Applications of Arrays

- Used to store a list of values
- Used to perform matrix operations
- Used to implement other data structures such as stack, queue etc.
- Used to implement search algorithms
- Used to implement sort algorithms
- Used to implement CPU scheduling algorithms

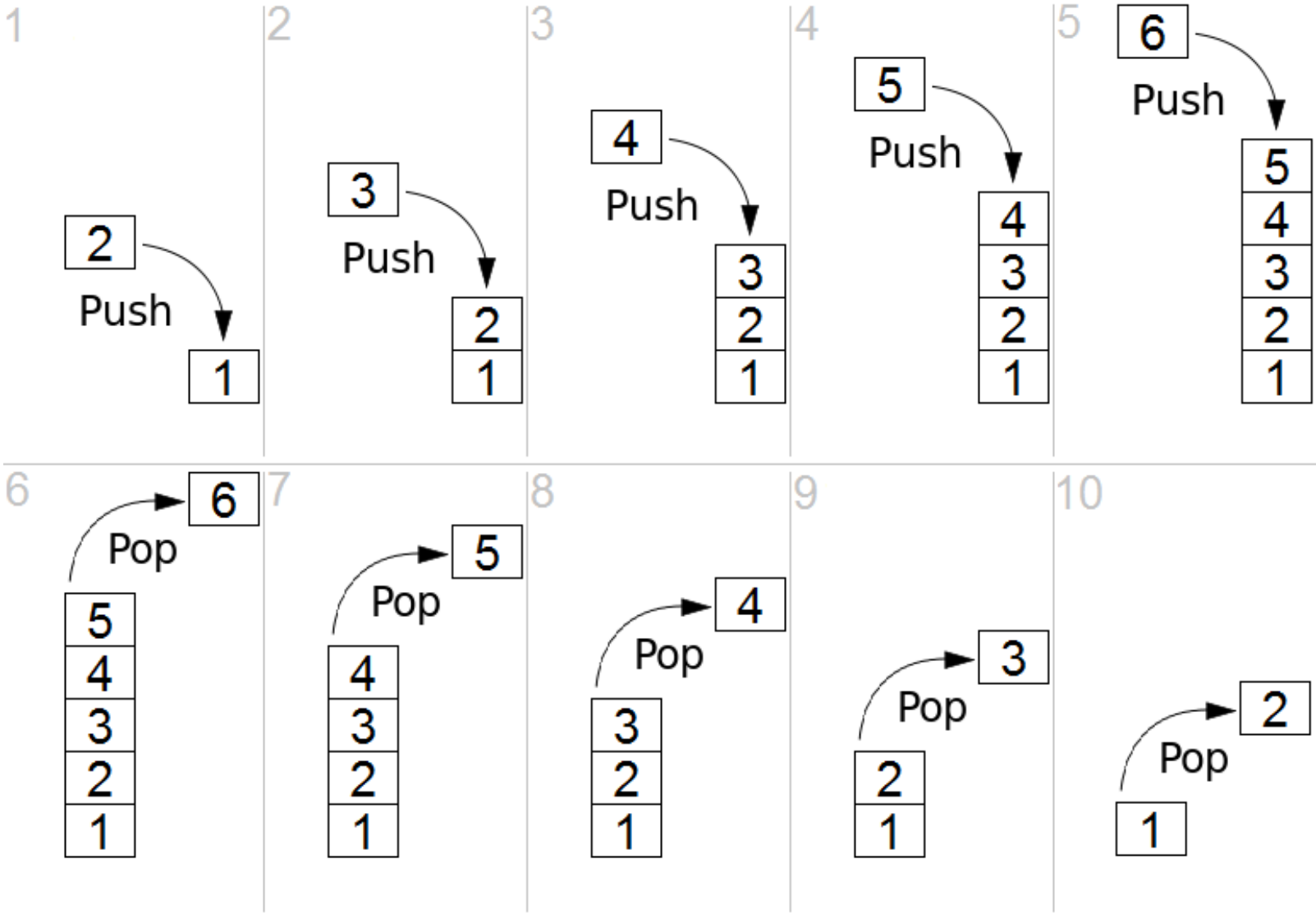
Stack

Definition

A stack is a collection of elements in which all insertions and deletions are made at one end called top.

- Inserting an element into a stack is called Push operation
- Deleting an element from a stack is called Pop operation

Stack - Example



- The last element to be inserted into the stack will be the first to be removed. Hence a stack is also called a Last In First Out(LIFO) list

Representation of a Stack

- The simplest way to represent a stack is using a one dimensional array, say STACK(1:n)
- Here n is the maximum number of allowable entries
- Associated with this array will be a variable, top, which points to the top element in the stack

Operations on Stacks

1. CREATE(S) which creates S as an empty stack
2. PUSH(S,i) which inserts the element i onto the stack S and returns the new stack
3. POP(S) which removes the top element of stack S and returns the new stack
4. TOP(S) which returns the top element of stack S
5. STACK_EMPTY(S) which returns *true* if S is empty else *false*

Applcations of Stacks

- Evaluation of Arithmetic Expressions
- Recursion

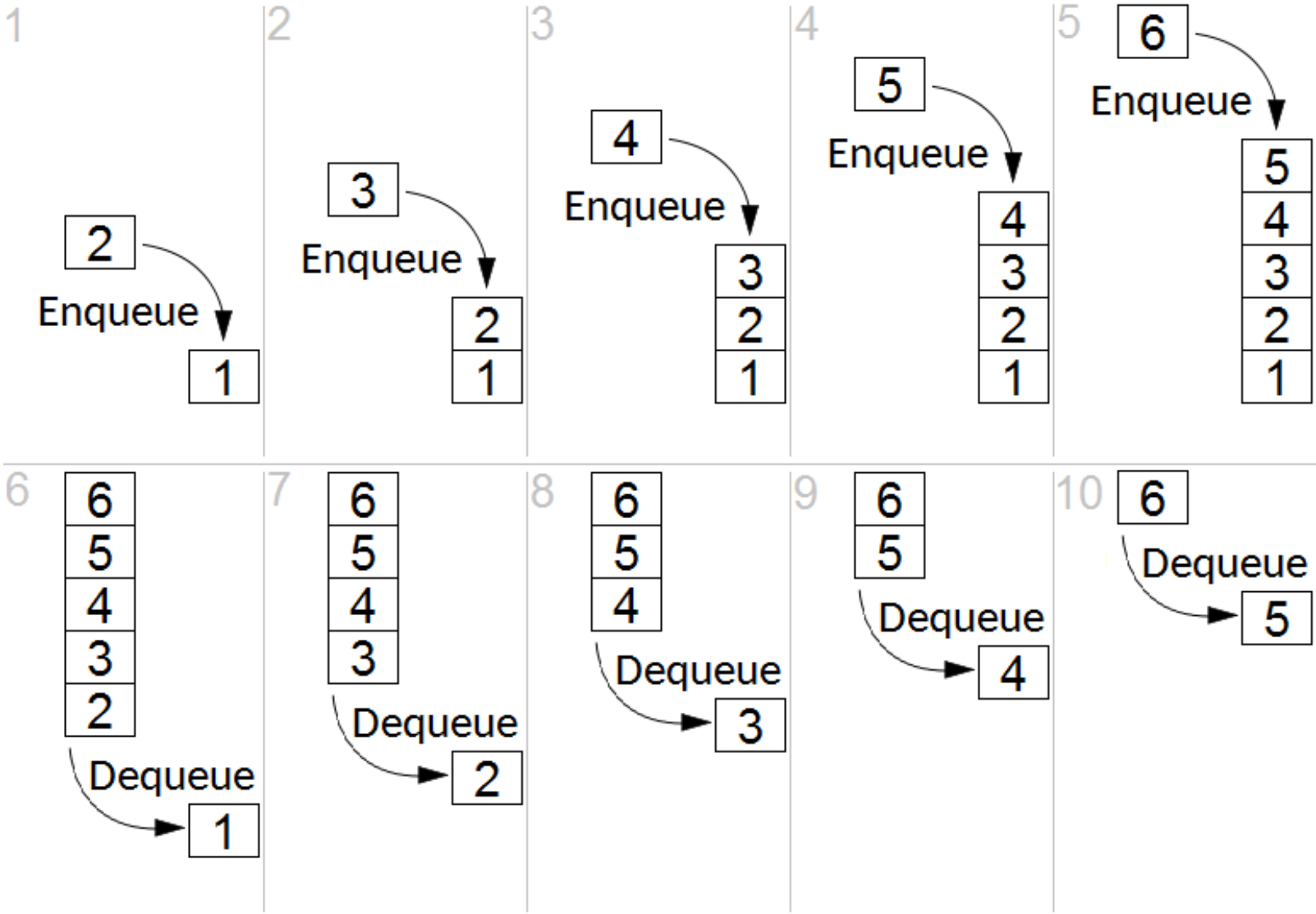
Queue

Definition

A queue is a collection of elements in which all insertions are made at one end,the rear and all deletions are made at other end, the front.

- Inserting an element into a queue is called Enqueue operation
- Deleting an element from a queue is called Dequeue operation

Queue - Example



- The first element to be inserted into the queue will be the first to be removed. Hence a queue is also called a First In First Out(FIFO) list

Representation of A Queue

- We can represent a queue using a one dimensional array, say QUEUE(1:n)
- Here n is the maximum number of allowable entries
- Associated with this array will be two variables, rear, which points to the last element in the queue, front, which points to the first element in the queue

Operations on Queues

1. CREATEQ(Q) which creates Q as an empty queue
2. ADDQ(Q,i) which inserts the element i onto the rear end of queue Q and returns the new queue
3. DELETEQ(Q) which removes the front element of queue Q and returns the new queue
4. FRONT(Q) which returns the front element of queue Q
5. QUEUE_EMPTY(Q) which returns *true* if Q is empty else *false*

Applications of Queues

- Simulation
  - It is the modelling of a real world problem using a computer program
  Example - Simulation of a Traffic Control System
- CPU Scheduling in a multiprogramming environment
- Scheduling of jobs to a printer

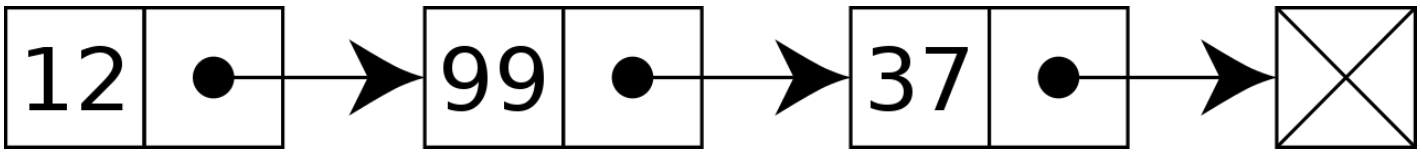
Linked Lists

- If we are storing data structures using arrays successive elements are stored sequentially in memory.ie.They are stored at a fixed distance apart
- Arrays have certain disadvantages
- Lot of data movement is needed during insertion and deletion
- Memory storage is wasted if we are storing only a few elements
- We cannot increase memory size dynamically

Linked Lists

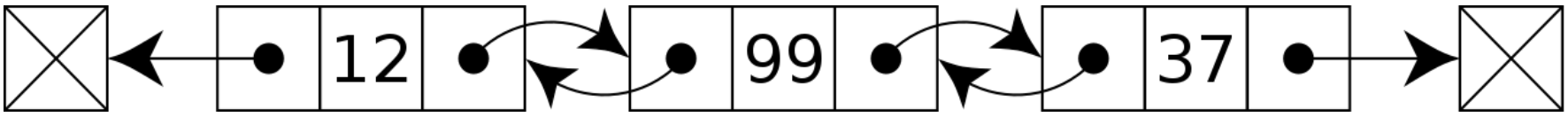
- It is better to store data structures using a linked list
- In a linked list, items are placed anywhere in memory
- They are not stored sequentially
- Here memory space is not wasted
- We can also increase the memory space dynamically

Singly Linked Lists



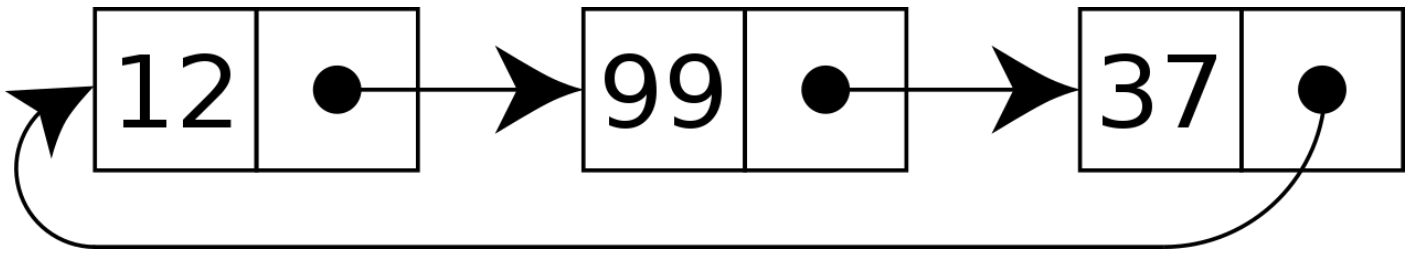
- A singly linked list whose nodes contain two fields: a value and a link to the next node
- The last node is linked to a terminator used to signify the end of the list

Doubly Linked Lists



- A doubly linked list whose nodes contain three fields
- a value
- the link forward to the next node
- and the link backward to the previous node

Circular Linked Lists



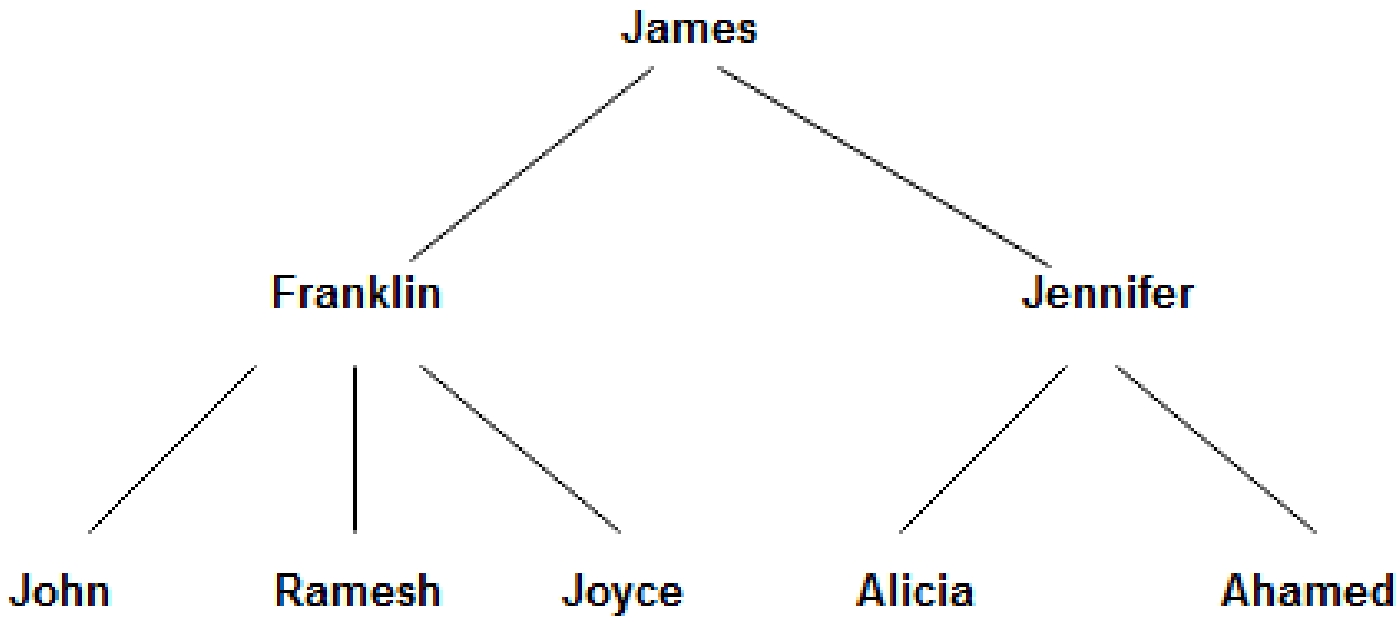
- Here the last node points to the first node
- In the case of circular doubly linked list, the first node points to the last node also

Applications of Linked Lists

- Polynomial Representation
- Linked Stacks and Linked Queues

Trees

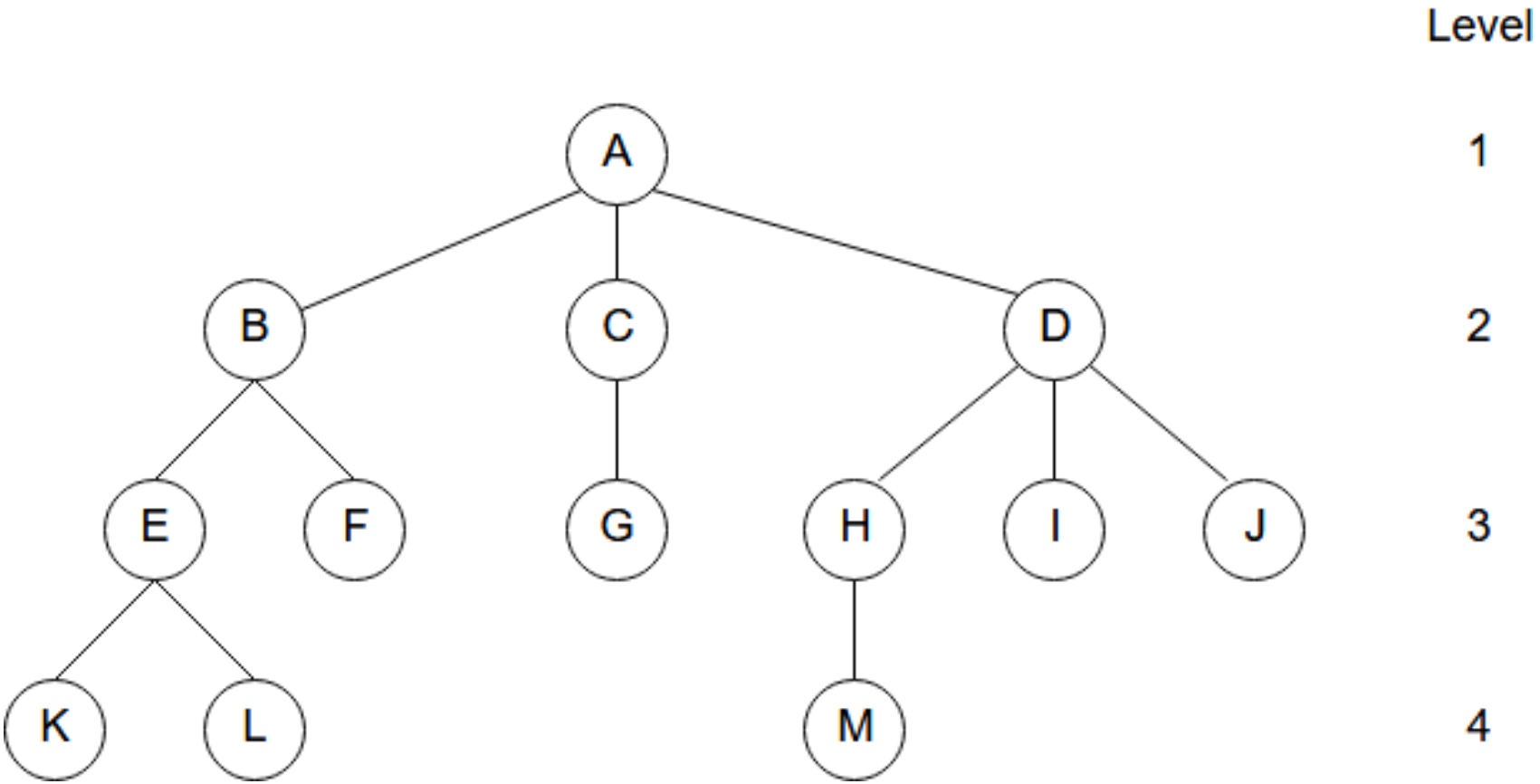
- A tree is used to represent data containing a hierarchical relationship between elements.
  Example
- Relation between family members
- Employee supervision in an organisation



Trees

Definition

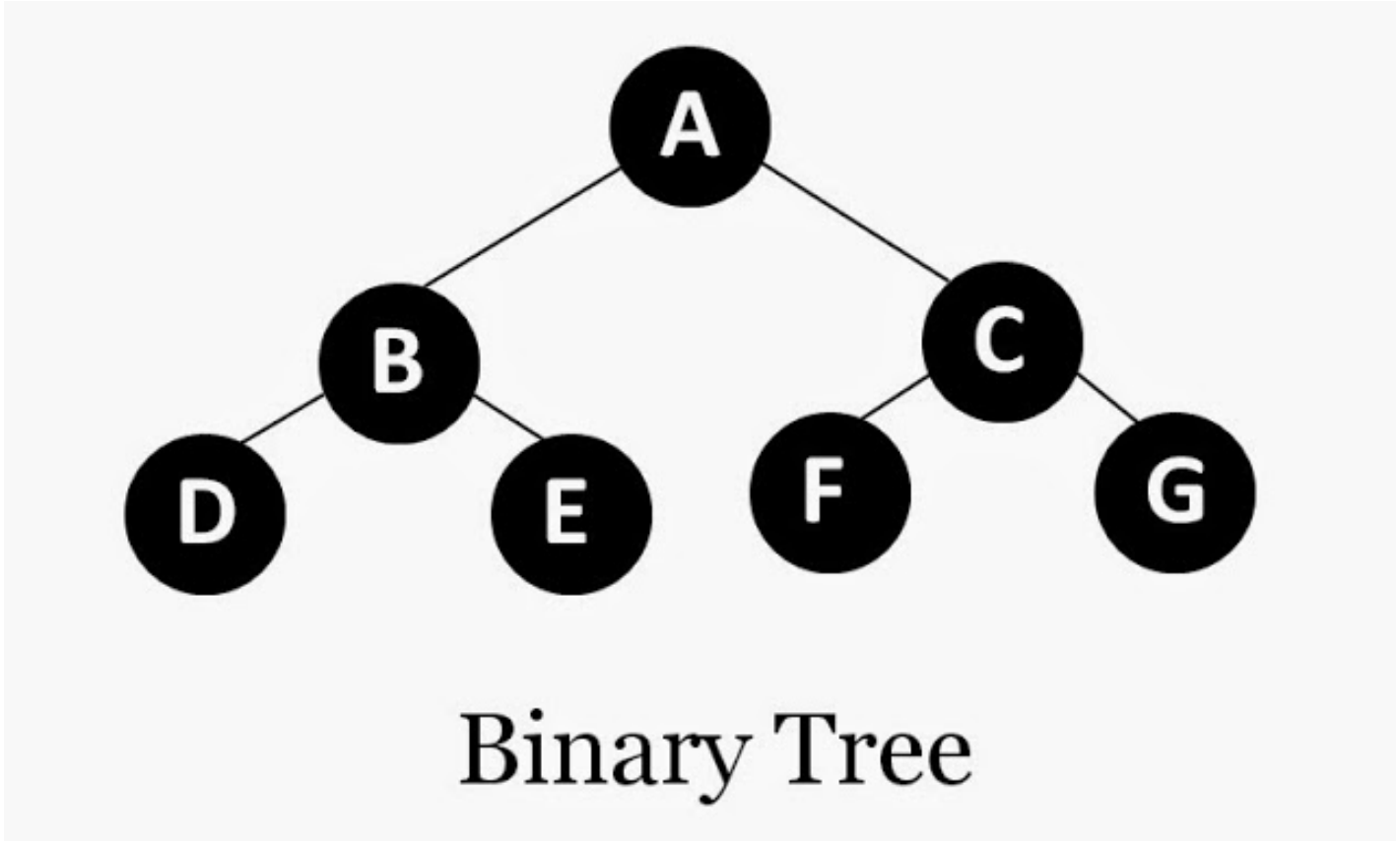
- A tree is a finite set of one or more nodes such that
  (i) there is a specially designated node called the root;
  (ii) the remaining nodes are partitioned into $n \geq 0$ disjoint sets $T_1, ..., T_n$ where each of these sets is a tree. $T_1, ..., T_n$ are called the subtrees of the root.



Binary Trees

Definition

- A binary tree is a finite set of nodes which is either empty or consists of a root and two disjoint binary trees called the left subtree and the right subtree.



Binary Tree

Set Data Structure

Definition

- A set is a collection of distinguishable objects called its members or elements

Examples

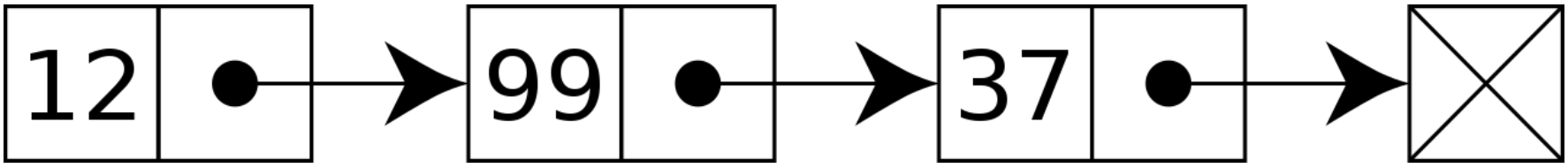
- {0,1,2,....} - Set of Natural Numbers
- {a,e,i,o,u} - Set of Vowels
- {red,green,blue} - Set of 3 colours

Representation

- In computers, a set can be represented using linked list, tree or array

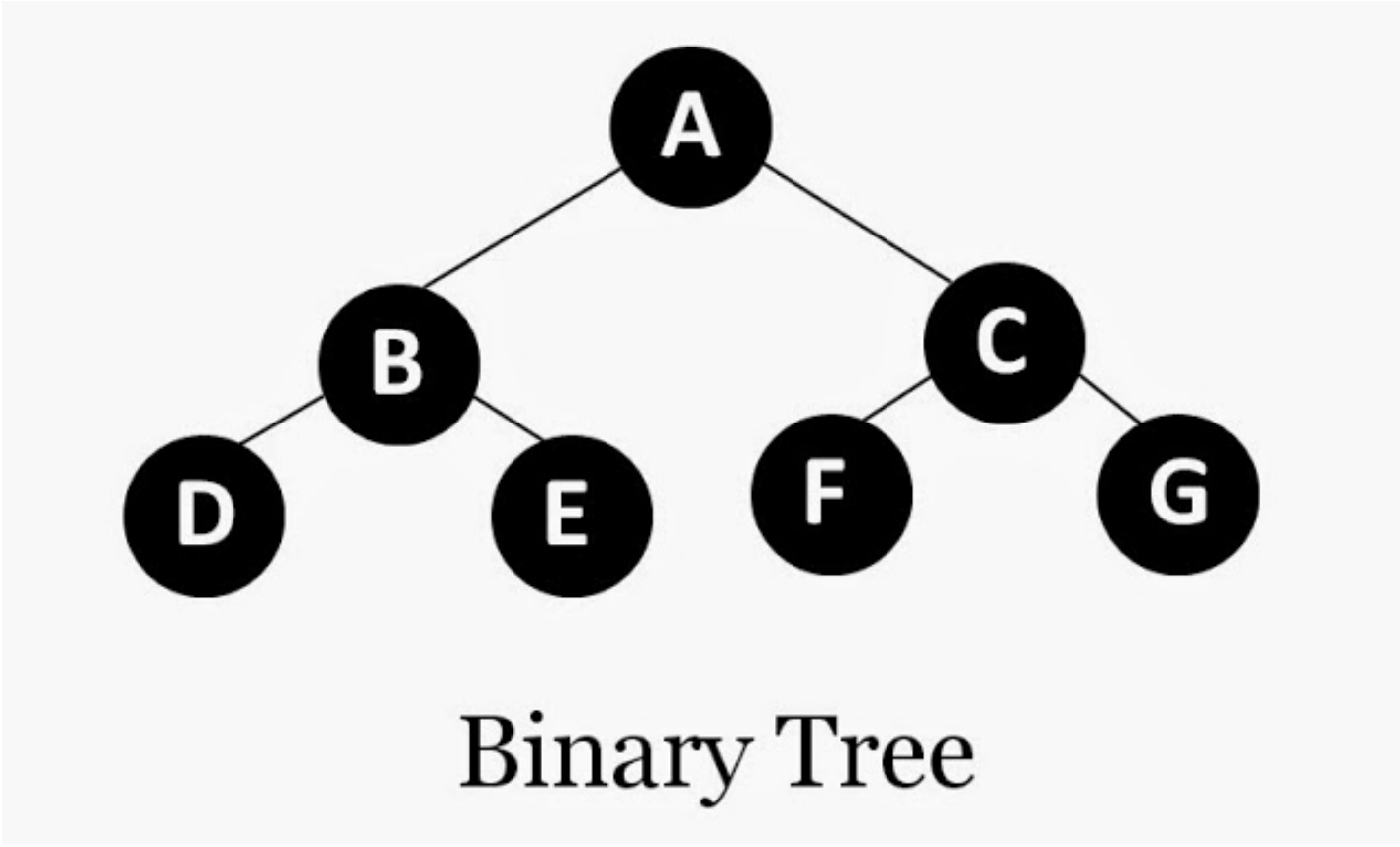
Representation of Set Using a Linked List

- $\{12, 99, 37\}$



Representation of Set Using a Tree

- {A, B, C, D, E, F, G, H }



Binary Tree

Implementation of a Set Using a Bit String

- A set can be implemented using an array of bits, called bit string(bit array)(bit vector)
- Every set we consider is a subset of a Universal Set denoted by U
- Set = {A,E,I,O,U}, U = {A,B,C,D,....X,Y,Z}
- Set = {Apr,Jun,Sep,Nov}, U = {Jan,Feb,Mar,....,Nov,Dec}
- Set = {1,3,5,7,9,....}, U = {1,2,3,4,5,6,7,8,9,....}

Implementation of a Set Using a Bit String

- Let U = {1,2,3,4,5,6,7,8,9,10}
- Let A = {1,3,5,7,9}
- This subset A of U can be represented using the bit string

1	0	1	0	1	0	1	0	1	0

- The size of the bit string will be the size of U
- If an element in A belongs to U, then the corresponding bit string value will be 1
- If an element in A does not belong to U, then the corresponding bit string value will be 0
- Let B = {2,4,6,8,10}
- The subset B of U can be represented using the bit string

0	1	0	1	0	1	0	1	0	1

Implementation of a Set Using a Bit String

- The advantage of such a representation is that we can perform the set operations Union and Intersection using bitwise OR and bitwise AND operations, respectively
- Let A = {1,2,3,4,5}

1	1	1	1	1	0	0	0	0	0

- Let B = {1,3,5,7,9}

1	0	1	0	1	0	1	0	1	0

- Now $A \cup B = \{1,2,3,4,5,7,9\}$ will be bit string(A) $\vee$ bit string(B)

1	1	1	1	1	0	1	0	1	0

- Now $A \cap B = \{1,3,5\}$ will be bit string(A) $\wedge$ bit string(B)

1	0	1	0	1	0	0	0	0	0

Implementation of a Set Using a Bit String

- We can perform the Set Difference operation in the following way
- Let A = {1,2,3,4,5}

1	1	1	1	1	0	0	0	0	0

- Let B = {4,5}

0	0	0	1	1	0	0	0	0	0

- Now $A - B = \{1,2,3\}$ will be bit string(A) $\wedge \sim$ (bit string(B))
- $\sim$ (bit string(B))

1	1	1	0	0	1	1	1	1	1

- bit string(A) $\wedge \sim$ (bit string(B))

1	1	1	0	0	0	0	0	0	0

Hashing

- Each identifier we use in a computer program is stored in a particular address in memory
- This address is calculated using an arithmetic function called a hash function
- The memory used for storing identifiers is referred to as a hash table
- The hash table is partitioned into b buckets, each bucket is capable of holding s slots
- The process of mapping an identifier into a location in hash table is called hashing

Hashing

- Hash Table (with 26 buckets, each bucket having 2 slots)

	SLOT 1	SLOT 2
1	A	A2
2	0	0
3	0	0
4	D	0
5	0	0
6	0	0
7	GA	G
⋮	⋮	⋮
26	0	0

Zeros indicate empty slots

- Here identifiers starting with letter A are stored in bucket 1
- Identifiers starting with letter B are stored in bucket 2, and so on

Hashing

- Associated with hashing two events can occur - overflow and collision
- An overflow occurs when a new identifier is mapped into a full bucket
- In the above example, if the new identifier is A3, an overflow occurs
- A collision occurs when two different identifiers map into the same bucket
- Collision occurs in the above example
- A good hashing function shall be easily computable and shall minimise the number of collisions

Simple Hash Functions

1. Mid-Square
   This function is computed by squaring the numerical value of the identifier (internal binary representation) and then using an appropriate number of bits from the middle of the square to obtain the hash address
2. Division
   The identifier (its numeric value) is divided by some number and the remainder is used as the hash address
3. Folding
   The identifier (its numeric value) is divided into several parts, all but the last being of the same length.These parts are then added together to obtain the hash address

Collision / Overflow Resolution Techniques

- When a new identifier gets hashed into a full bucket, it is necessary to find another bucket for this identifier.
- The following techniques are used for this.

1. Linear Probing (Linear Open Addressing)
   Here we will search the hash table to find the closest unfilled bucket to place the new identifier.
2. Quadratic Probing
3. Random Probing
4. Chaining

Collision / Overflow Resolution Techniques

1. Linear Probing (Linear Open Addressing)

   Hash Table - 26 buckets, 1 slot per bucket

1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0
10	0
11	0
12	0
...	...
26	0

Collision / Overflow Resolution Techniques

1. Linear Probing (Linear Open Addressing)

   Identifiers - GA, D, A, G, L, A2, A1, A3, A4, Z, ZA, E

1	0
2	0
3	0
4	0
5	0
6	0
7	GA
8	0
9	0
10	0
11	0
12	0
...	...
26	0

Collision / Overflow Resolution Techniques

1. Linear Probing (Linear Open Addressing)

   Identifiers - GA, D, A, G, L, A2, A1, A3, A4, Z, ZA, E

1	0
2	0
3	0
4	D
5	0
6	0
7	GA
8	0
9	0
10	0
11	0
12	0
...	...
26	0

Collision / Overflow Resolution Techniques

1. Linear Probing (Linear Open Addressing)

   Identifiers - GA, D, A, G, L, A2, A1, A3, A4, Z, ZA, E

1	A
2	0
3	0
4	D
5	0
6	0
7	GA
8	0
9	0
10	0
11	0
12	0
...	...
26	0

Collision / Overflow Resolution Techniques

1. Linear Probing (Linear Open Addressing)

   Identifiers - GA, D, A, G, L, A2, A1, A3, A4, Z, ZA, E

1	A
2	0
3	0
4	D
5	0
6	0
7	GA
8	G
9	0
10	0
11	0
12	0
...	...
26	0

Collision / Overflow Resolution Techniques

1. Linear Probing (Linear Open Addressing)

   Identifiers - GA, D, A, G, L, A2, A1, A3, A4, Z, ZA, E

1	A
2	0
3	0
4	D
5	0
6	0
7	GA
8	G
9	0
10	0
11	0
12	L
...	...
26	0

Collision / Overflow Resolution Techniques

1. Linear Probing (Linear Open Addressing)

   Identifiers - GA, D, A, G, L, A2, A1, A3, A4, Z, ZA, E

1	A
2	A2
3	0
4	D
5	0
6	0
7	GA
8	G
9	0
10	0
11	0
12	L
...	...
26	0

Collision / Overflow Resolution Techniques

1. Linear Probing (Linear Open Addressing)

   Identifiers - GA, D, A, G, L, A2, A1, A3, A4, Z, ZA, E

1	A
2	A2
3	A1
4	D
5	0
6	0
7	GA
8	G
9	0
10	0
11	0
12	L
...	...
26	0

Collision / Overflow Resolution Techniques

1. Linear Probing (Linear Open Addressing)

   Identifiers - GA, D, A, G, L, A2, A1, A3, A4, Z, ZA, E

1	A
2	A2
3	A1
4	D
5	A3
6	0
7	GA
8	G
9	0
10	0
11	0
12	L
...	...
26	0

Collision / Overflow Resolution Techniques

1. Linear Probing (Linear Open Addressing)

   Identifiers - GA, D, A, G, L, A2, A1, A3, A4, Z, ZA, E

1	A
2	A2
3	A1
4	D
5	A3
6	A4
7	GA
8	G
9	0
10	0
11	0
12	L
...	...
26	0

Collision / Overflow Resolution Techniques

1. Linear Probing (Linear Open Addressing)

   Identifiers - GA, D, A, G, L, A2, A1, A3, A4, Z, ZA, E

1	A
2	A2
3	A1
4	D
5	A3
6	A4
7	GA
8	G
9	0
10	0
11	0
12	L
...	...
26	Z

Collision / Overflow Resolution Techniques

1. Linear Probing (Linear Open Addressing)

   Identifiers - GA, D, A, G, L, A2, A1, A3, A4, Z, ZA, E

1	A
2	A2
3	A1
4	D
5	A3
6	A4
7	GA
8	G
9	ZA
10	0
11	0
12	L
...	...
26	Z

Collision / Overflow Resolution Techniques

1. Linear Probing (Linear Open Addressing)

   Identifiers - GA, D, A, G, L, A2, A1, A3, A4, Z, ZA, E

1	A
2	A2
3	A1
4	D
5	A3
6	A4
7	GA
8	G
9	ZA
10	E
11	0
12	L
...	...
26	Z

Collision / Overflow Resolution Techniques

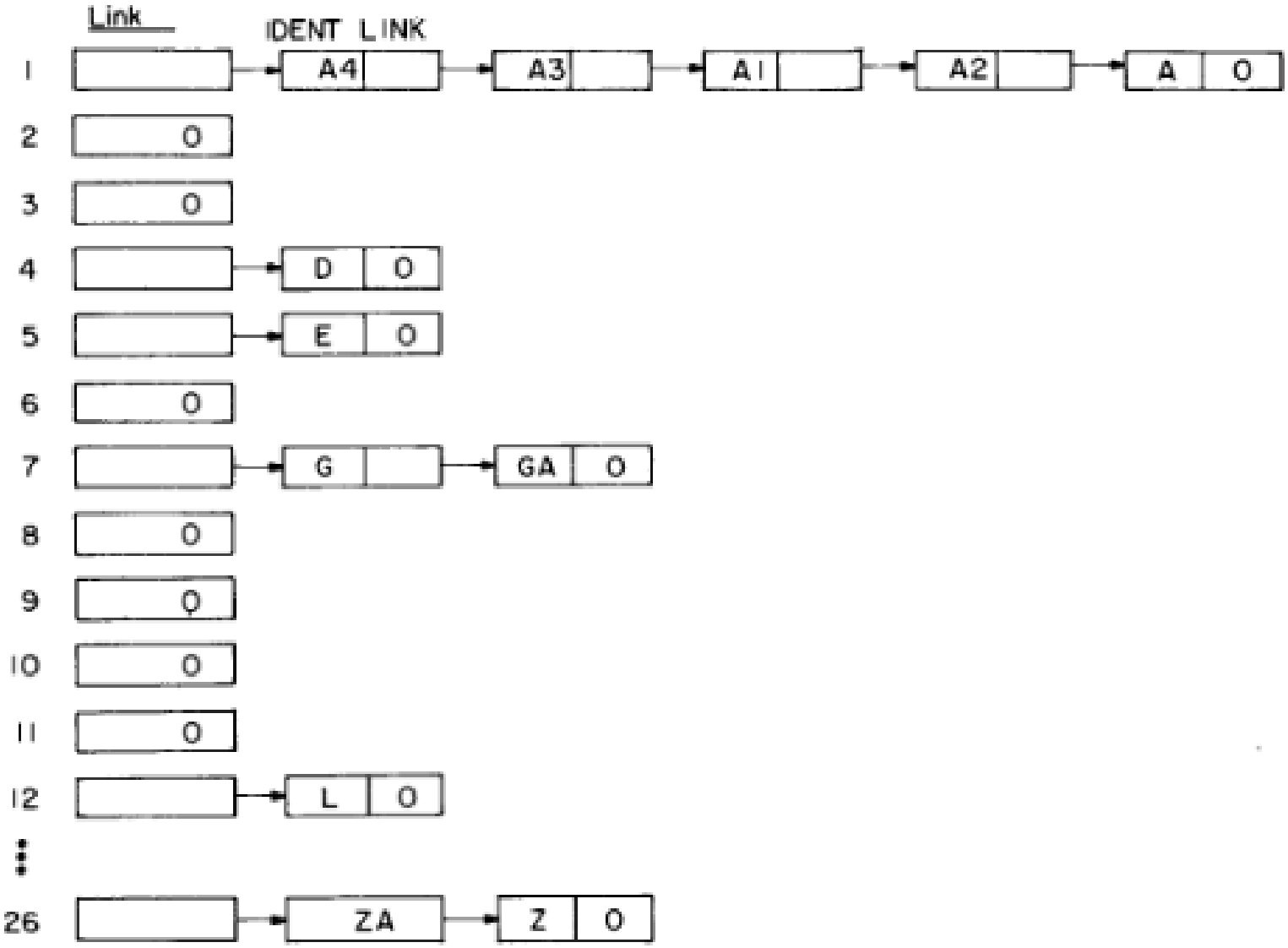
1. Linear Probing (Linear Open Addressing)
   - One problem with this is that it tends to create clusters of identifiers
   - This will increase the average number of probes needed for searching an identifier
   - The given below probing techniques will minimise the average number of probes needed
2. Quadratic Probing

   Here a quadratic function is used to calculate the hash address of the identifier
3. Random Probing

   In this method, the hash address of the identifier is calculated in a random manner
4. Chaining
   - Here we maintain linked lists of identifiers
   - For each bucket, one linked list is maintained
   - Each linked list will contain all the synonyms of the identifier
   - However, additional storage is needed for links

Collision / Overflow Resolution Techniques

- Chaining



Amortised Analysis

- Amortised analysis is a method for analysing the time complexity(time taken for its execution) of an algorithm
- This is used in algorithms when most operations are fast, while an occasional operation execute slowly
- By performing this analysis, we gain insight into a particular data structure, and this can help us in optimising the design
- Here we do not consider the probability of each operation involved
- The following are the techniques used in this analysis.

1. Aggregate Analysis
2. Accounting Method
3. Potential Method

Aggregate Analysis

- Here we find out the worst case computing time T(n) for the algorithm which contains a sequence of n operations.
- Then we find out the average computing time (average cost)(amortised cost) $= \frac{T(n)}{n}$
- Example 1 - MultiPop Stack
- Example 2 - Incrementing a Binary Counter

Aggregate Analysis

- Example 1 - MultiPop Stack
- In this stack, apart from the basic PUSH and POP operations there is a MULTIPOP(S,k) operation which removes the k top elements of it

```
MULTIPOP(S,k)
1 while not STACK_EMPTY(S) and k > 0
2   POP(S)
3   k = k - 1
```

- Here, the number of times POP can be called (including those within MULTIPOP) is at most the number of PUSH operations
- Hence, for any sequence of n PUSH,POP and MULTPOP operations, worst case computing time is of O(n)
- amortised cost $= \frac{O(n)}{n} =$ O(1)

Aggregate Analysis

- Example 2 - Incrementing a Binary Counter
- Here we implement a binary counter that counts upward from 0.
- For incrementing the binary counter, we flip the bits

Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Total cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

Aggregate Analysis

- Example 2 - Incrementing a Binary Counter

```
INCREMENT(A)
1    i = 0
2    while i < A.length and A[i] == 1
3        A[i] = 0
4        i = i + 1
5    if i < A.length
6        A[i] = 1
```

- Here, the cost of each INCREMENT operation is the number of bits flipped
- The total cost is always less than twice the total number of INCREMENT operations
- Hence, the worst case computing time for a sequence of n INCREMENT operations is O(n)
- amortised cost $= \frac{O(n)}{n} =$ O(1)

Accounting Method

- Here we assign different charges to different operations in the algorithm
- The charge assigned to an operation is called its amortised cost
- When an operation's amortised cost exceeds its actual cost, the amount we save is called credit.
- Credit can be used to pay for those operations whose actual cost exceeds their amortised cost.
- This method differs from aggregate analysis, in which all operations have the same amortised cost.
- We have to choose amortised costs in such a way that the total actual cost of a sequence of operations does not exceed the total amortised cost of the sequence.

Accounting Method

- Example 1 - MultiPop Stack

Operation	Actual Cost	Amortised Cost(Chosen)
PUSH	1	2
POP	1	0
MULTIPOP	min(k,s)	0

s - stack size, k - no of items popped

- For every PUSH operation there is a credit of 1
- This credit can be used to pay for POP and MULTIPOP operations
- The amortised cost of the individual operations is O(1),which is same as the actual cost
- For any sequence of n PUSH, POP and MULTIPOP operations, the total amortised cost is O(n), which is same as the total actual cost

Accounting Method

- Example 2 - Incrementing a Binary Counter

Operation	Actual Cost	Amortised Cost(Chosen)
Set bit to 1	1	2
Reset bit to 0	1	0

- Since there is a set operation before every reset operation, there will be enough credits to pay for the reset.
- The amortised cost of the individual operations is O(1),which is same as the actual cost
- Thus, for n INCREMENT operations, the total amortised cost is O(n), so is the total actual cost.

Potential Method

- Here there is a potential energy or potential associated with every data structure
- The value of the potential changes with every operation that takes place in the data structure
- A potential function Φ is used to calculate the potential
- Amortised cost of an operation $(\hat{c}_i)$ = actual cost of the operation (c_i) + change in potential due to the operation $(\Phi(D_i) - \Phi(D_{i-1}))$
- Here D_i and D_{i-1} are the states of the data structure after operations i and i-1 respectively

Potential Method

- Example 1 - MultiPop Stack

Operation	Actual Cost	Amortised Cost
PUSH	1	2 (1 + 1)
POP	1	0 (1 - 1)
MULTIPOP	min(k,s)	0 (min(k,s) - min(k,s))

s - stack size, k - no of items popped

- Chosen potential function for the multipop stack, $\Phi(S)$ is the number of elements in it
- The amortised cost of the individual operations is O(1),which is same as the actual cost
- For any sequence of n PUSH, POP and MULTIPOP operations, the total amortised cost is O(n), which is same as the total actual cost

Potential Method

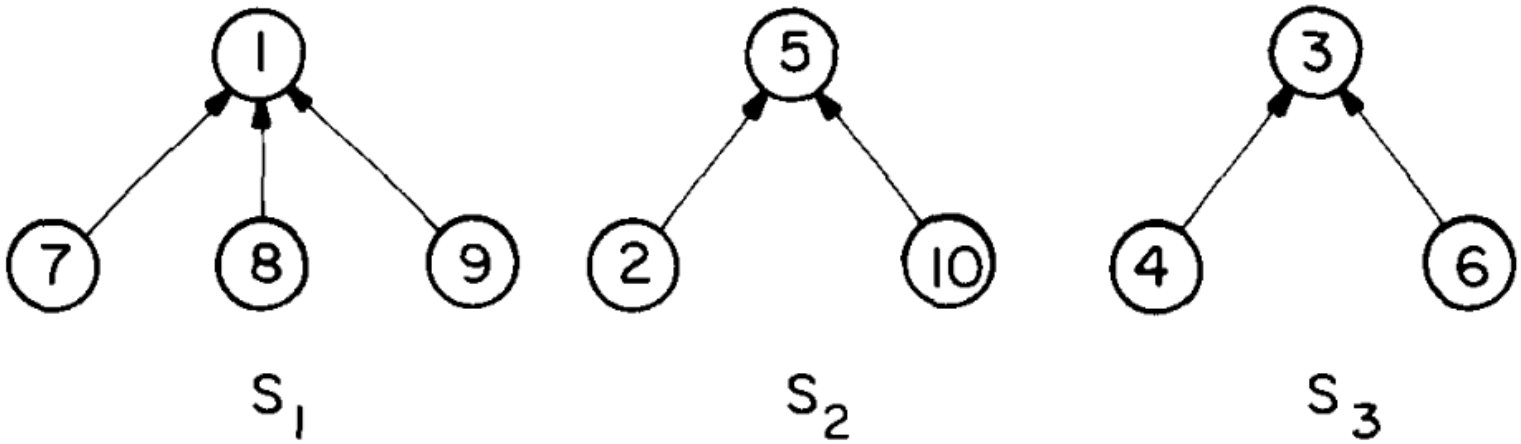
- Example 2 - Incrementing a Binary Counter

Operation	Actual Cost	Amortised Cost
Set bit to 1	1	2 (1 + 1)
Reset bit to 0	1	0 (1 - 1)

- Chosen potential function for the binary counter, $\Phi(S)$ is the number of 1's in it
- The amortised cost of the individual operations is O(1),which is same as the actual cost
- Thus, for n INCREMENT operations, the total amortised cost is O(n), which is same as the total actual cost.

Disjoint Sets - Representation

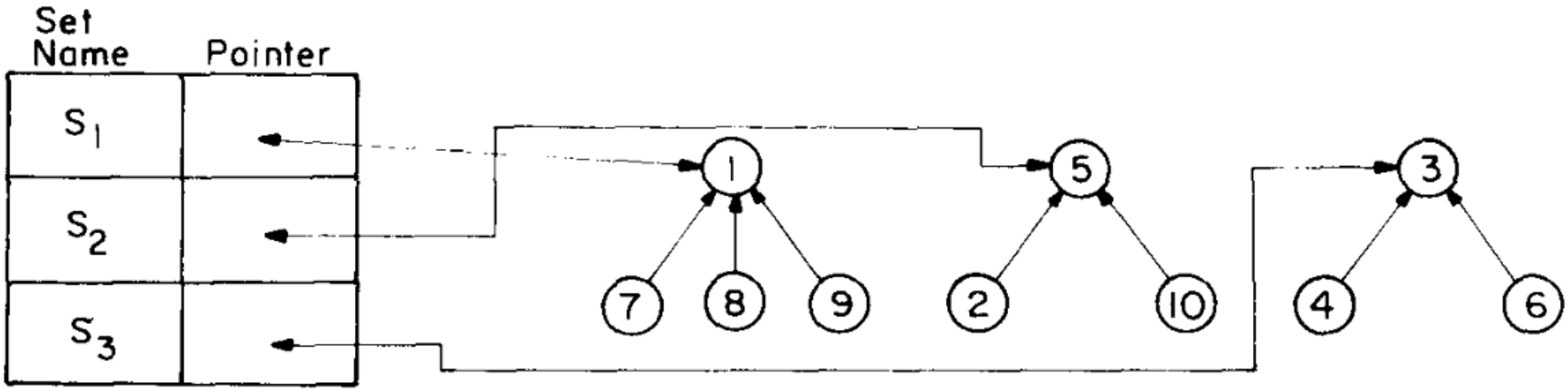
- Sets which do not have any element in common are called disjoint sets.
- Example
- $S_1 = \{1, 7, 8, 9\}$
- $S_2 = \{5, 2, 10\}$
- $S_3 = \{3, 4, 6\}$
- They can be represented using trees



- Here each node other than the root node are linked to its parent.

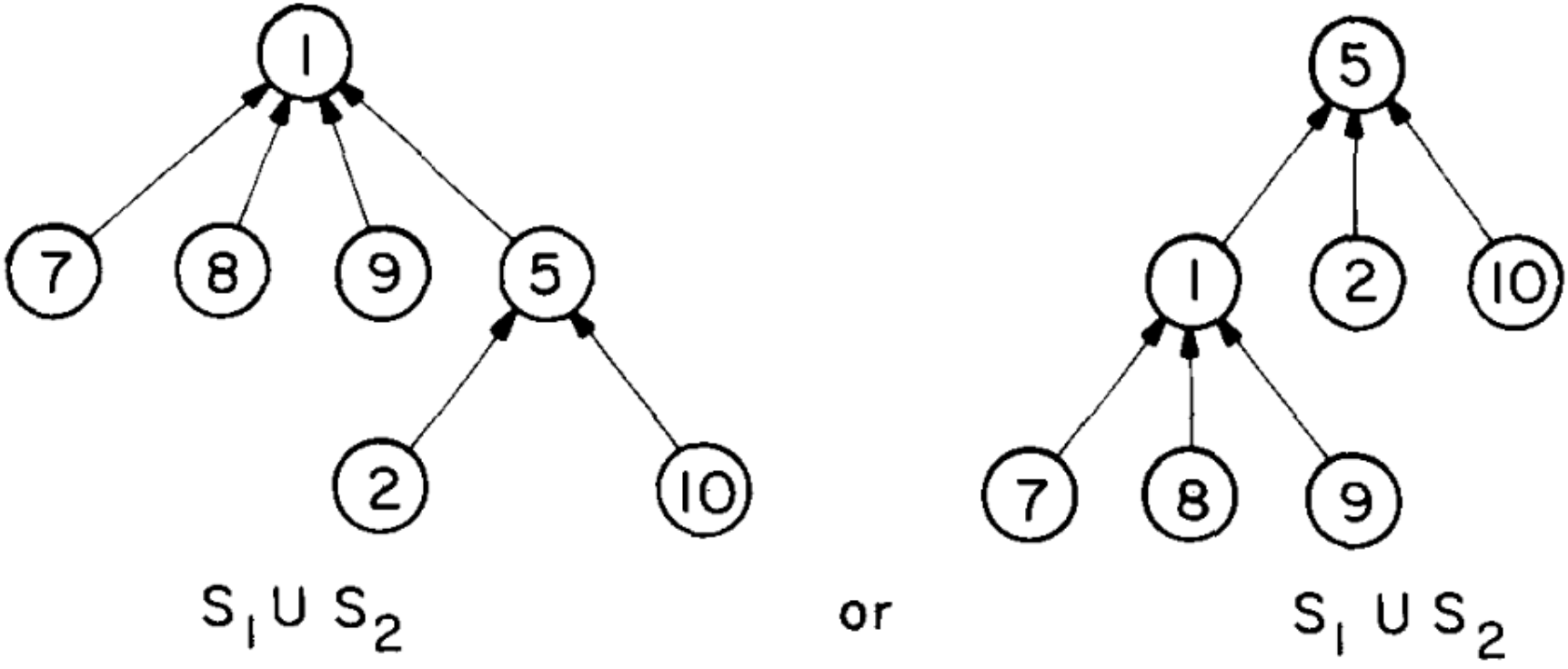
Disjoint Sets - Representation

- With each set name, we keep a pointer to the root of the tree representing that set.
- In addition, each root has a pointer to the set name.
- Then, the above representation becomes



Disjoint Sets - Union Algorithm

- To find the union of S_1 and S_2, we simply make one of the trees a subtree of the other



Disjoint Sets - Union Algorithm

- Here we assume that the node index corresponds to the element value.
- Thus, element 6 is represented by the node with index 6.
- Hence, each node needs only one field: the PARENT field to link to its parent.
- Root nodes have a PARENT field of zero.
- To find the union of two sets, we set the parent field of one of the roots to the other root.
- The performance of the UNION algorithm can be improved, if we use a weighting rule for UNION.

weighting rule

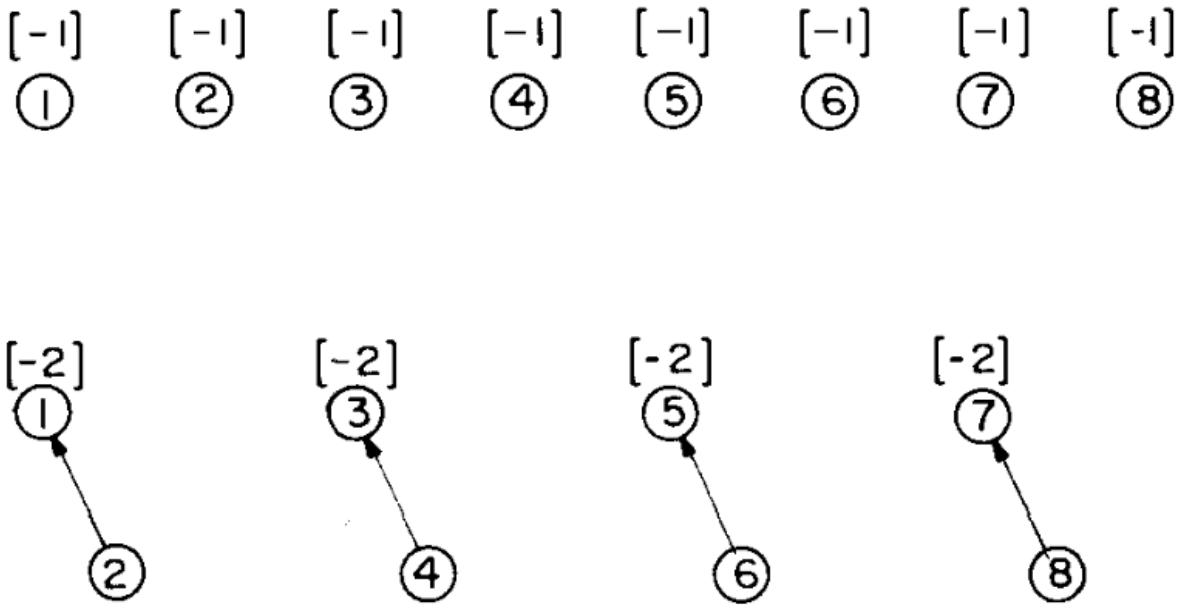
If the number of nodes in tree i is less than the number in tree j, then make j the parent of i, otherwise make i the parent of j.

Disjoint Sets - Union Algorithm

- The number of nodes in every tree is maintained in the root node.
- This count is maintained in the PARENT field as a negative number.
- This is done so, since for all other nodes the PARENT is positive.

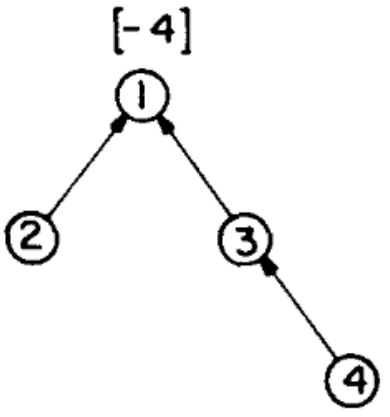
```
procedure UNION (i,j)
  //union sets with roots i and j, i ≠ j, using the weighting rule. PARENT
    (i) = -COUNT (i) and PARENT (j) = -COUNT (j)//
    x ← PARENT (i) + PARENT (j)
    if PARENT (i) > PARENT (j)
      then [PARENT (i) ← j          //i has fewer nodes//
            PARENT (j) ← x]
      else [PARENT (j) ← i          //j has fewer nodes//
            PARENT (i) ← x]

end UNION
```

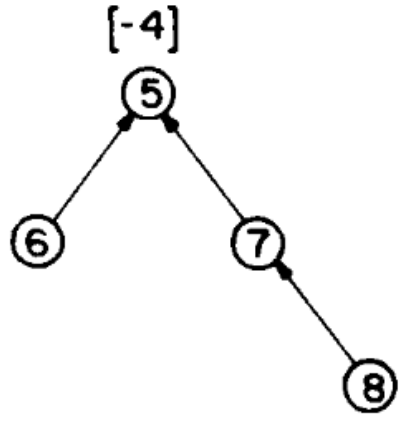
Disjoint Sets - Union Algorithm

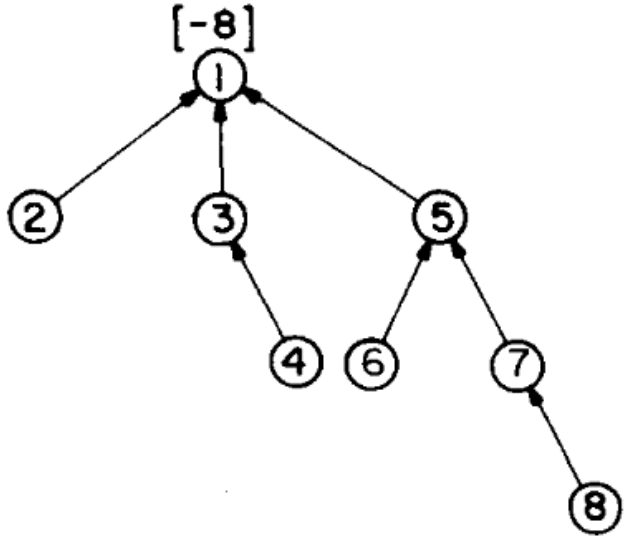


Level	Command
1	Initial
1	UNION (1,2)
	UNION (3,4)
2	UNION (5,6)
	UNION (7,8)

Disjoint Sets - Union Algorithm







LEVEL	COMMAND
1	UNION (1,3)
2	UNION (5,7)
1	
2	UNION (1,5)
3	
4	

Disjoint Sets - Find Algorithm

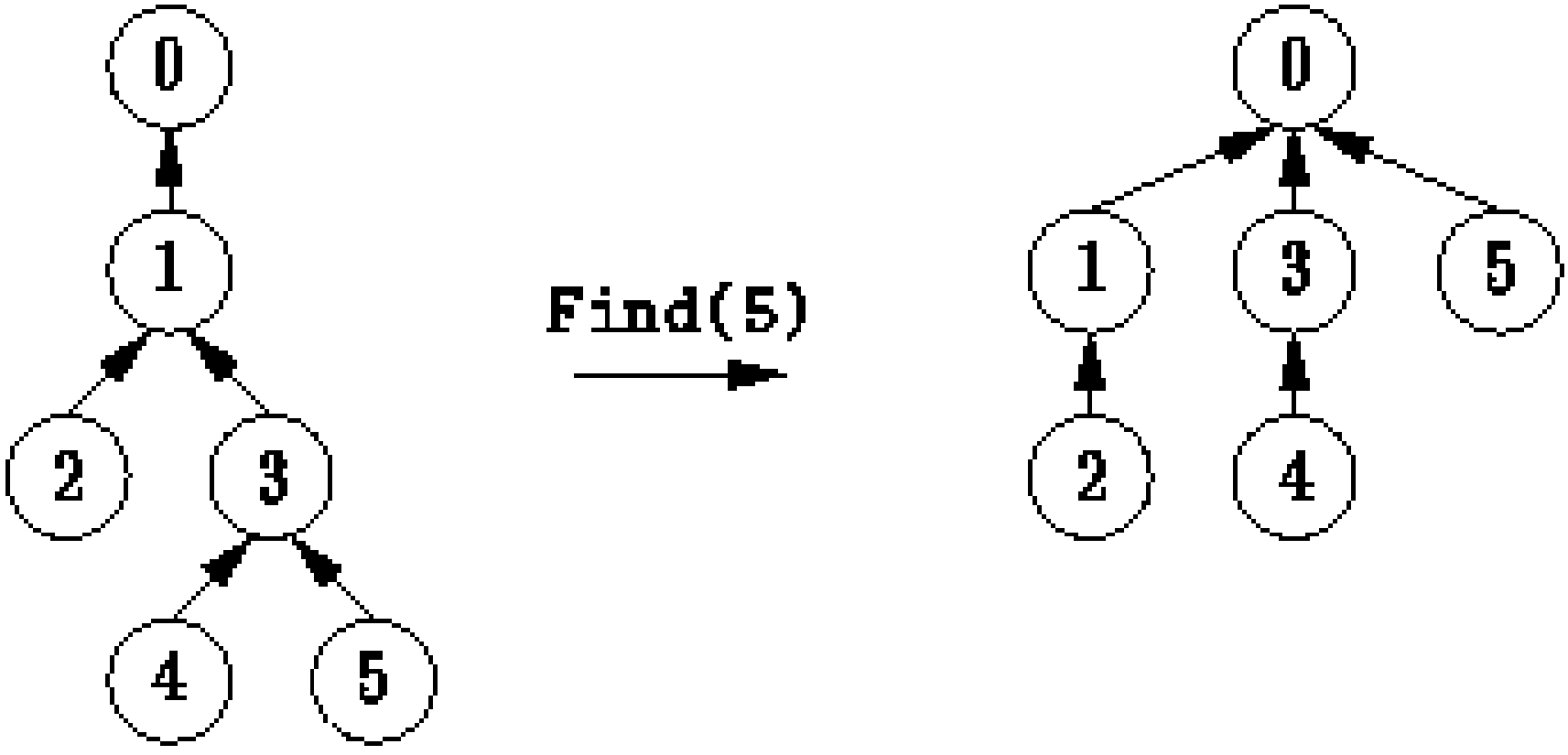
- This is used to find the root of the tree containing a particular element(say i).
- Here we will be using a rule called collapsing rule.

collapsing rule

If j is a node on the path from i to its root and PARENT(j) $\neq$ root(i), then set PARENT(j) $\leftarrow$ root(i)

- This modification roughly doubles the time for an individual find.
- However, it reduces the worst case time over a sequence of finds.

Disjoint Sets - Find Algorithm



Disjoint Sets - Find Algorithm

```
procedure FIND(i)
  //find the root of the tree containing element i. Use the collapsing
    rule to collapse all nodes from i to the root j//
    j ← i
    while PARENT(j) > 0 do          //find root//
      j ← PARENT(j)
    end
    k ← i
    while k ≠ j do          //collapse nodes from i to root j//
      t ← PARENT(k)
      PARENT(k) ← j
      k ← t
    end
    return (j)
end FIND
```

Module 2
Advanced Tree Structures

- Balanced Binary Search Trees
- Red-Black Tree
- B-Tree
- Splay Tree
- Suffix Tree

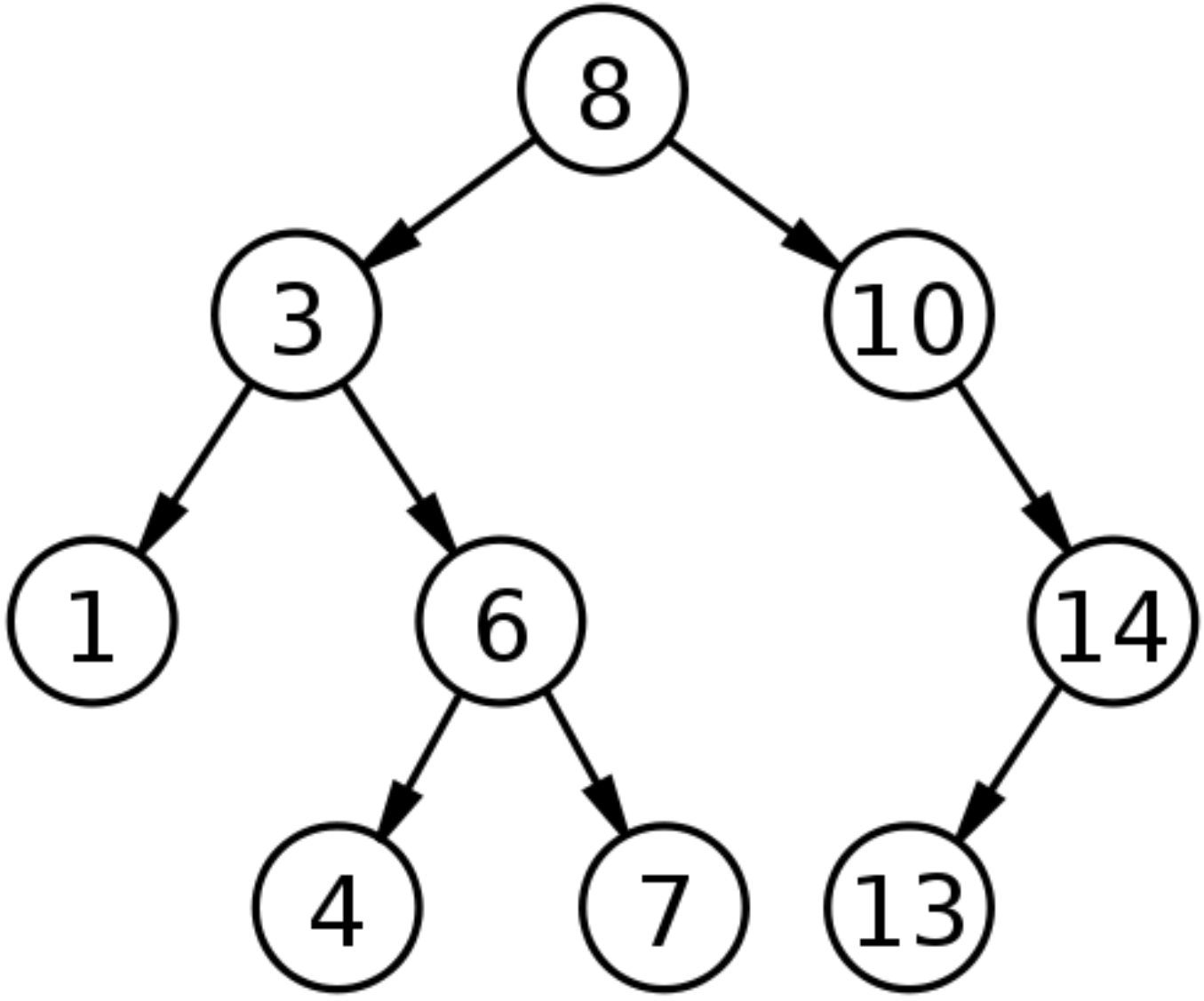
Binary Search Tree

Definition

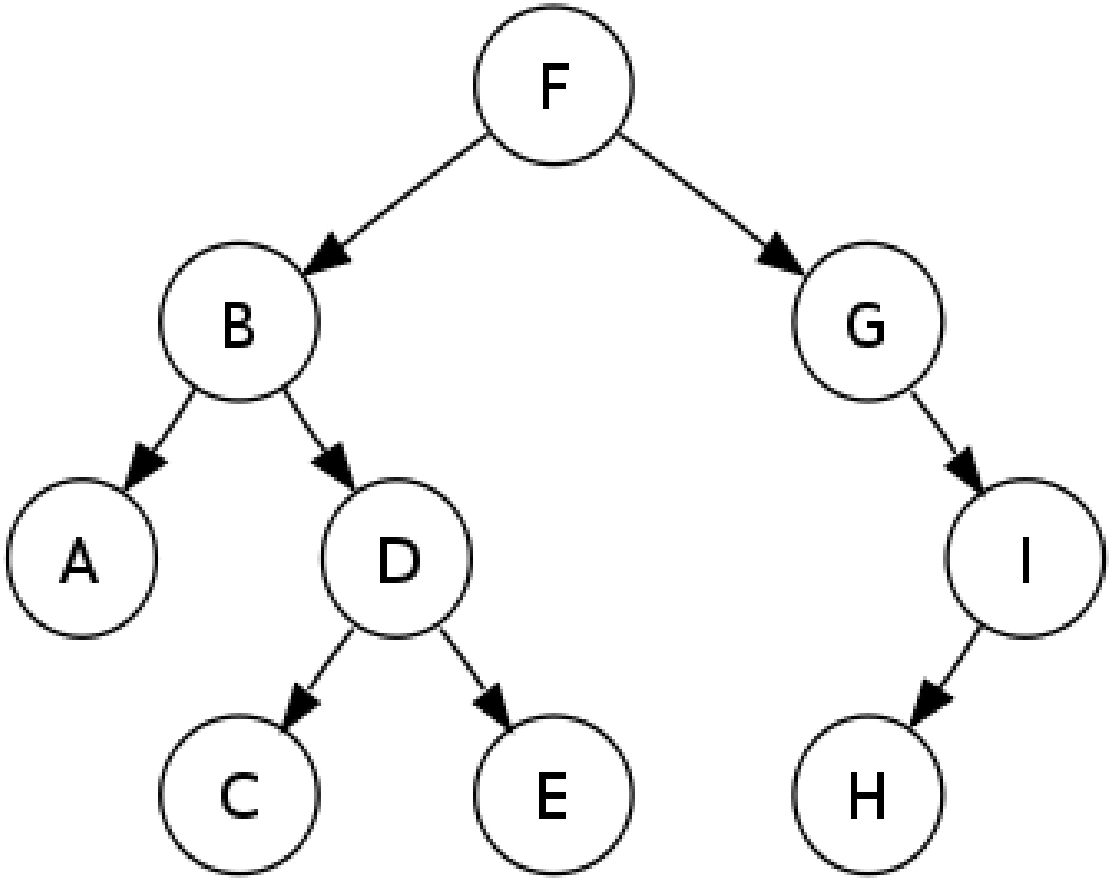
A binary search tree T is a binary tree; either it is empty or each node in the tree contains an identifier and

1. All identifiers in the left subtree of T are less(numerically or alphabetically) than the identifier in the root node T
2. All identifiers in the right subtree of T are greater than the identifier in the root node T
3. The left and right subtrees of T are also binary search trees

Binary Search Tree



Binary Search Tree



Balanced Binary Search Tree

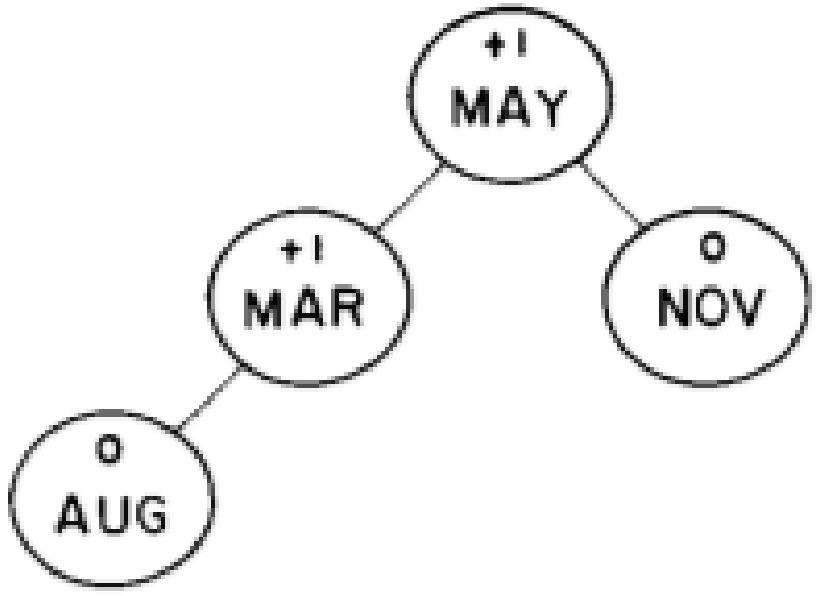
Definition

A binary search tree is said to be balanced(height balanced) if for every node in the tree, the heights of its left and right sub trees differ my utmost 1

- This difference between the heights of left and right sub trees of a node is called its balance factor
- In a balanced binary tree, the balance factor of every node is -1,0 or 1
- The height of a tree is defined to be the maximum level of any node in the tree
- We can perform insert, delete and search operations efficiently in a balanced binary search tree

Balanced Binary Search Tree

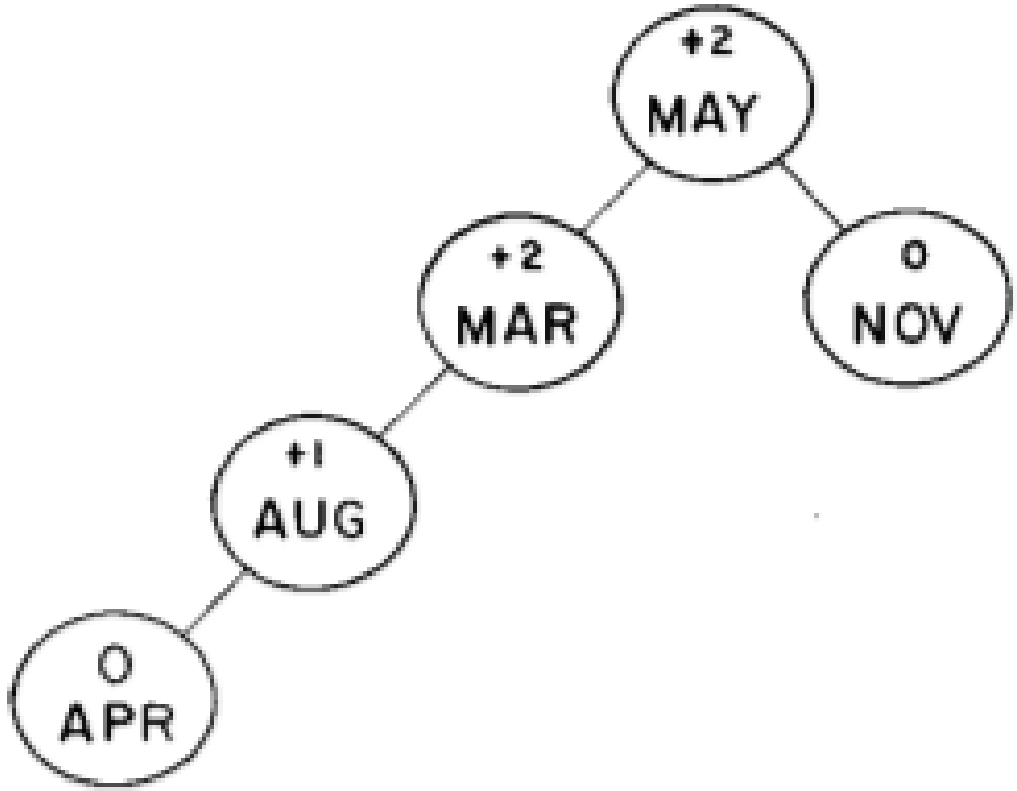
- Balanced Binary Search Tree - Example



- The balance factor of each node is represented on its top

Unbalanced Binary Search Tree

- Unbalanced Binary Search Tree - Example



Red Black Tree

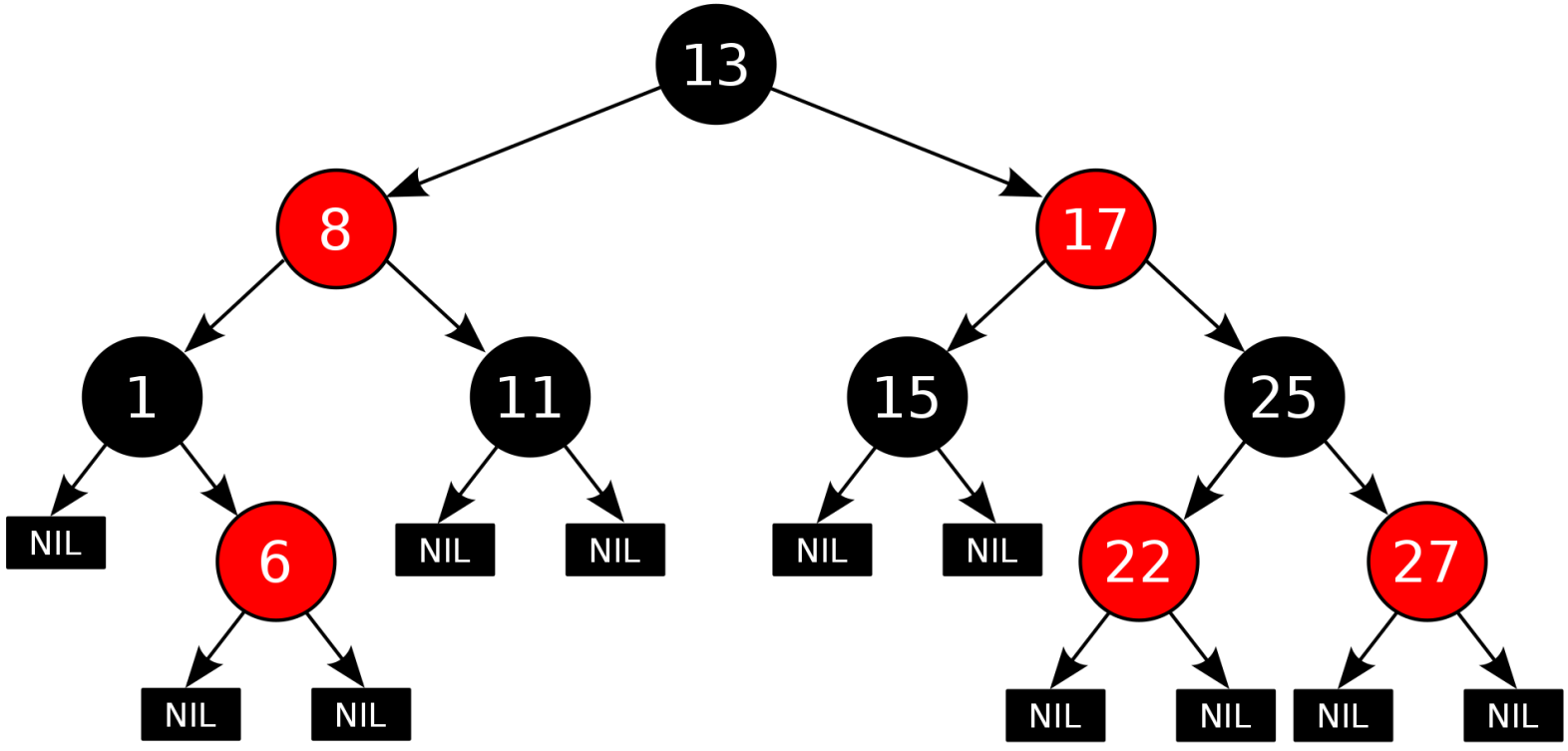
Definition

A red-black tree is a binary search tree that satisfies the following properties.

1. Every node is coloured either red or black.
2. The root node and all leaf nodes are black.
3. If a node is red, then both its children are black.
4. All paths from the root node to the leaves contain the same number of black nodes.

Red Black Tree

- A red-black tree is represented using an extended binary tree
- In an extended binary tree, special square nodes called external nodes are added at every place there is a null link

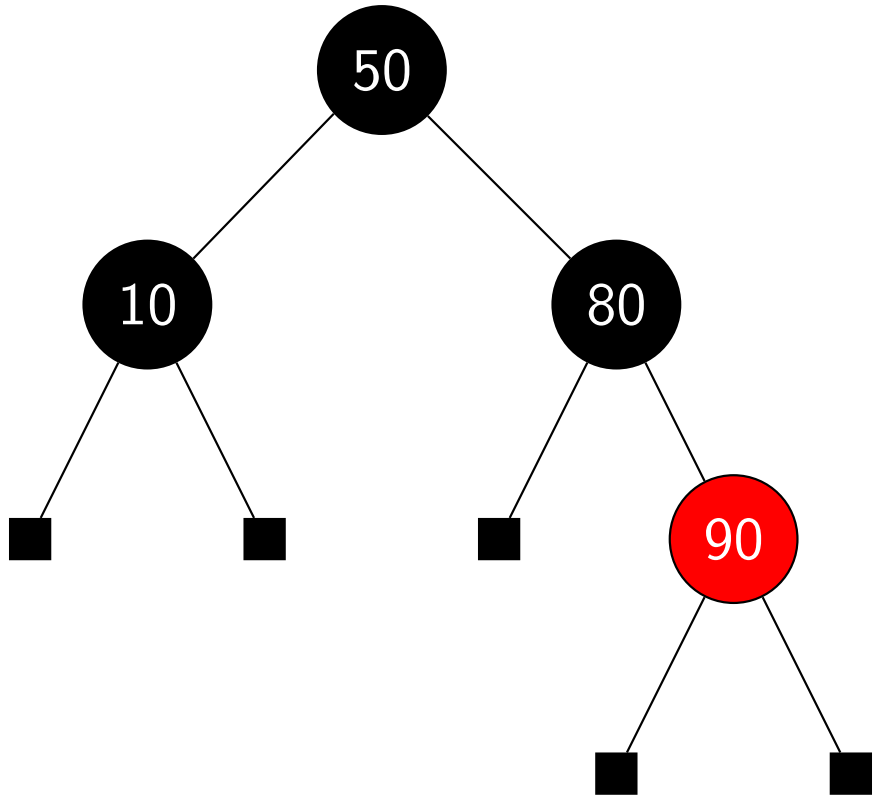


Insertion

1. Let N be the new node to be inserted, P its parent, G its grandparent and U its uncle
2. If the tree is empty, set N as its root and color it black
3. If the tree is not empty, insert N in its appropriate position in the tree and colour it red
4. If P is black, terminate.
5. Check U's color, if red, then mark P and U as black and G as red, go to G (i.e N=G) and repeat all the steps.
6. If P is at left
   - 6.1 If N is a right child, left rotate about P.
   - 6.2 Mark P as black and G as red, then right rotate about G and terminate.
7. If P is at right
   - 7.1 If N is a left child, right rotate about P.
   - 7.2 Mark P as black and G as red, then left rotate about G and terminate.

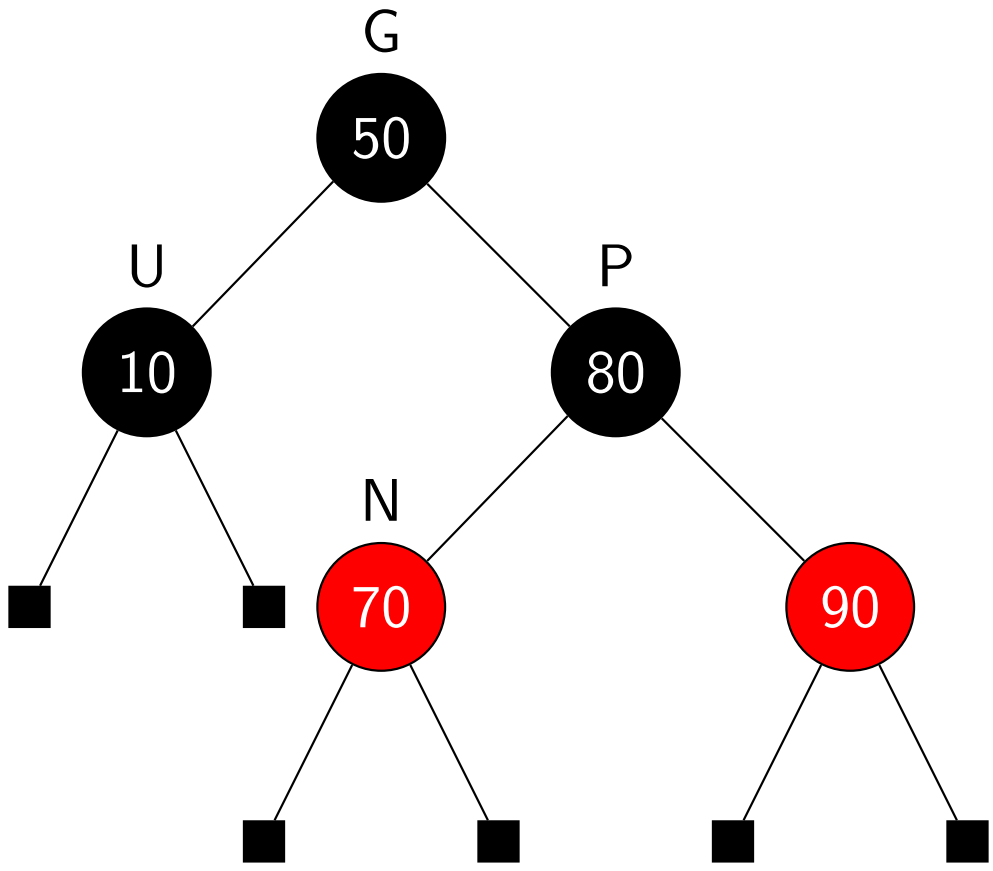
Insertion - Example

- Initial



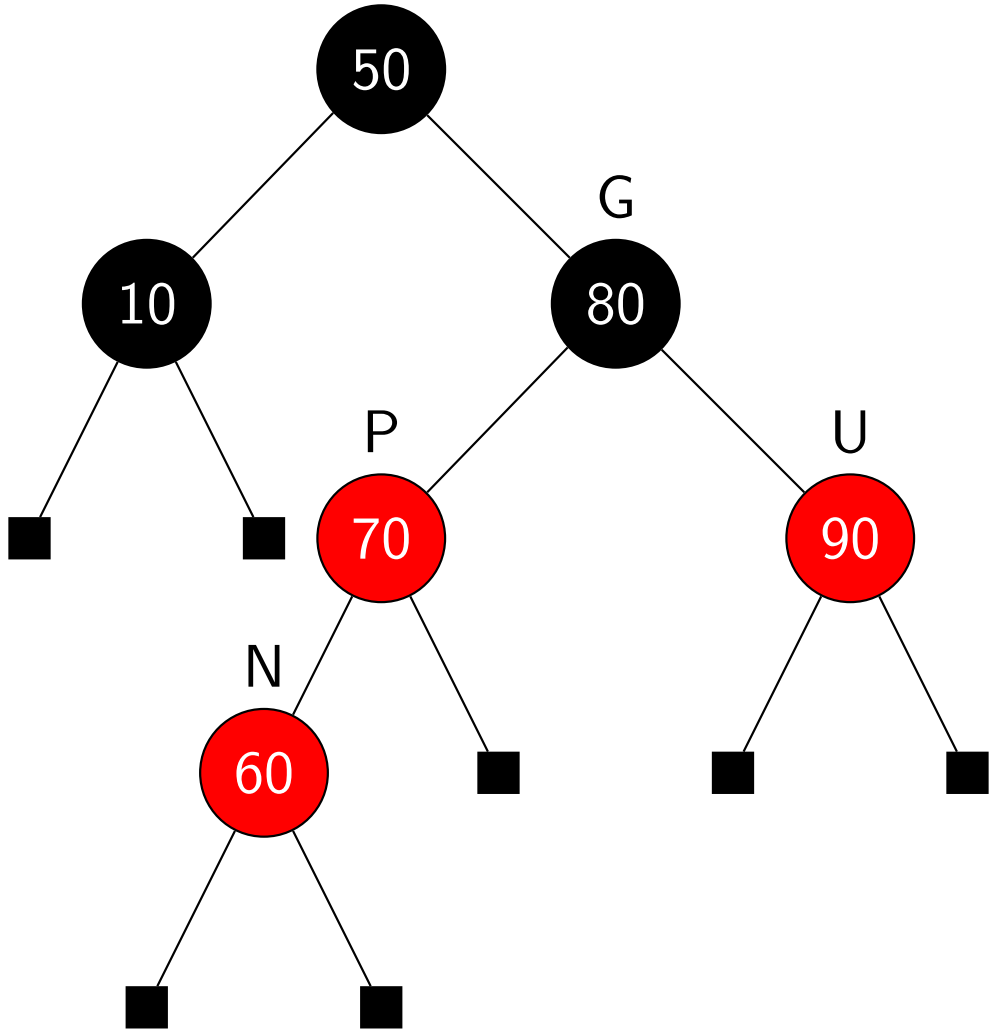
Insertion - Example

- Insert 70



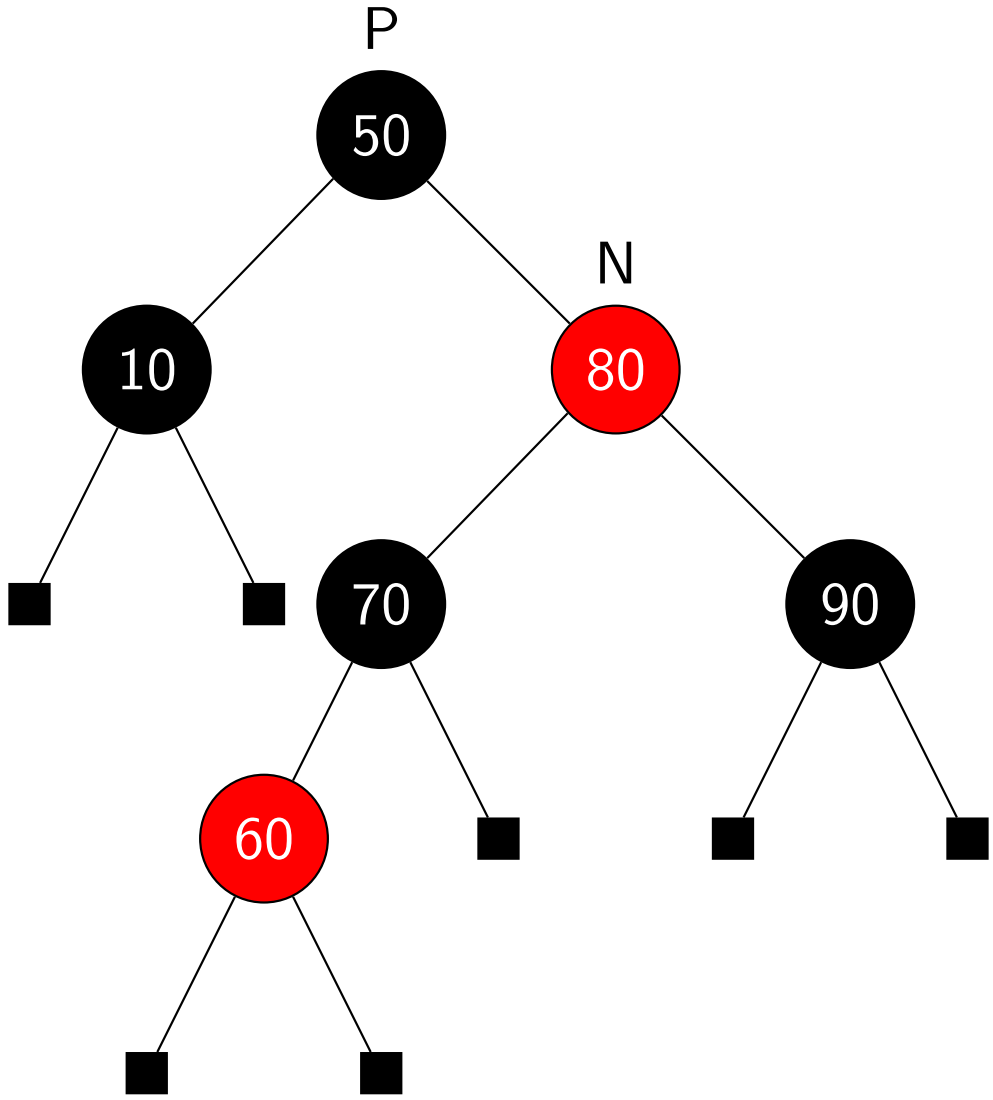
Insertion - Example

- Insert 60



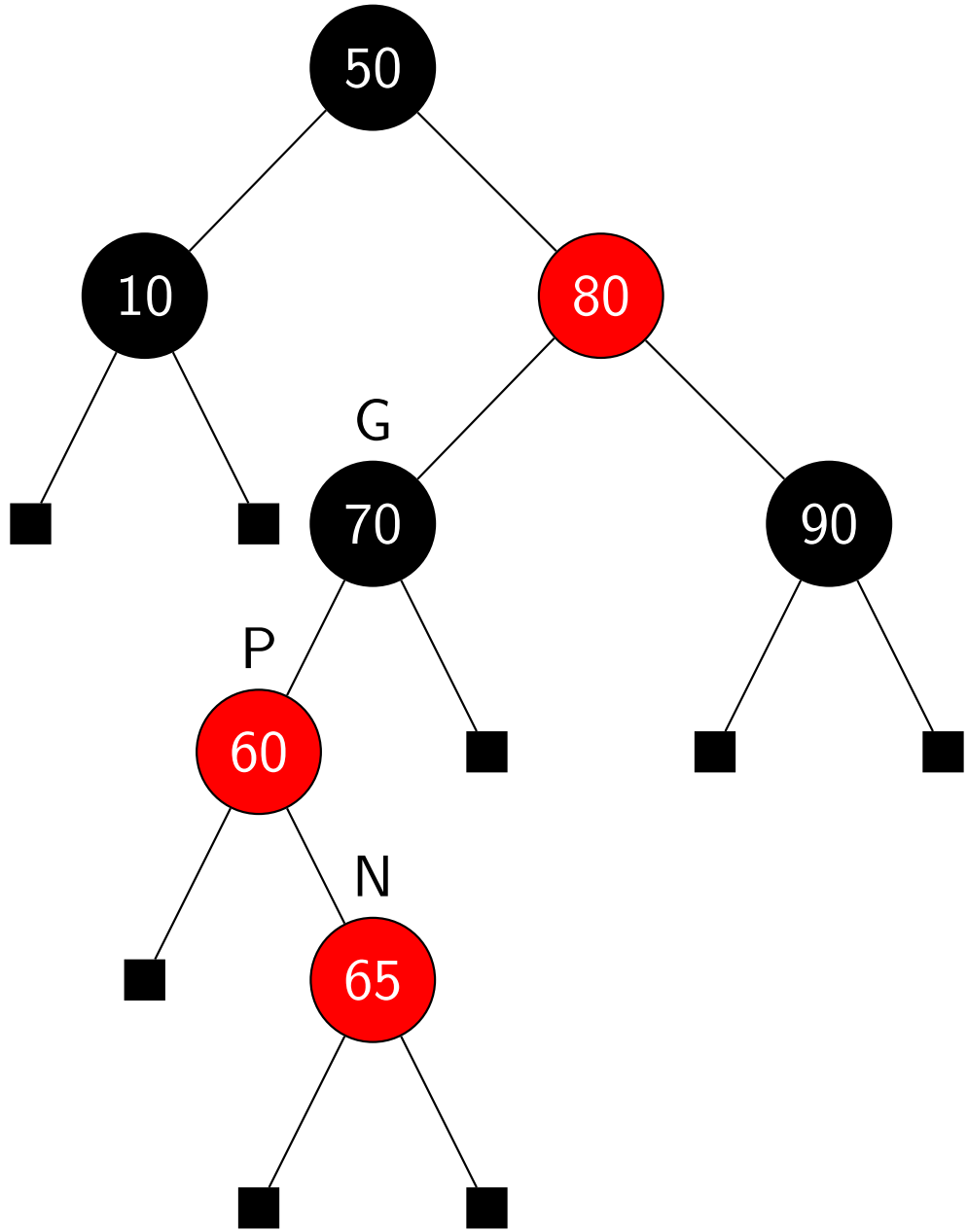
Insertion - Example

- Colour Change



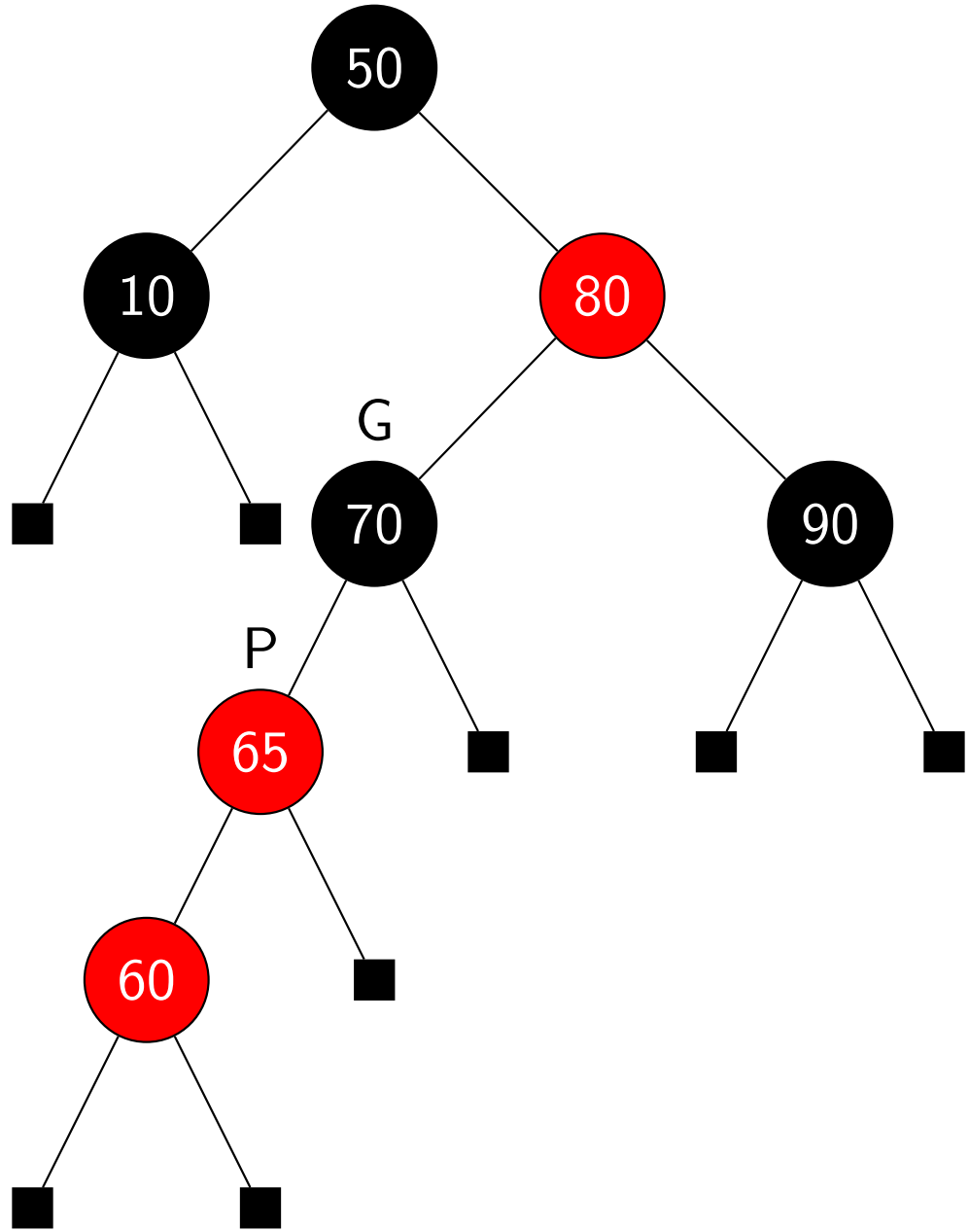
Insertion - Example

- Insert 65



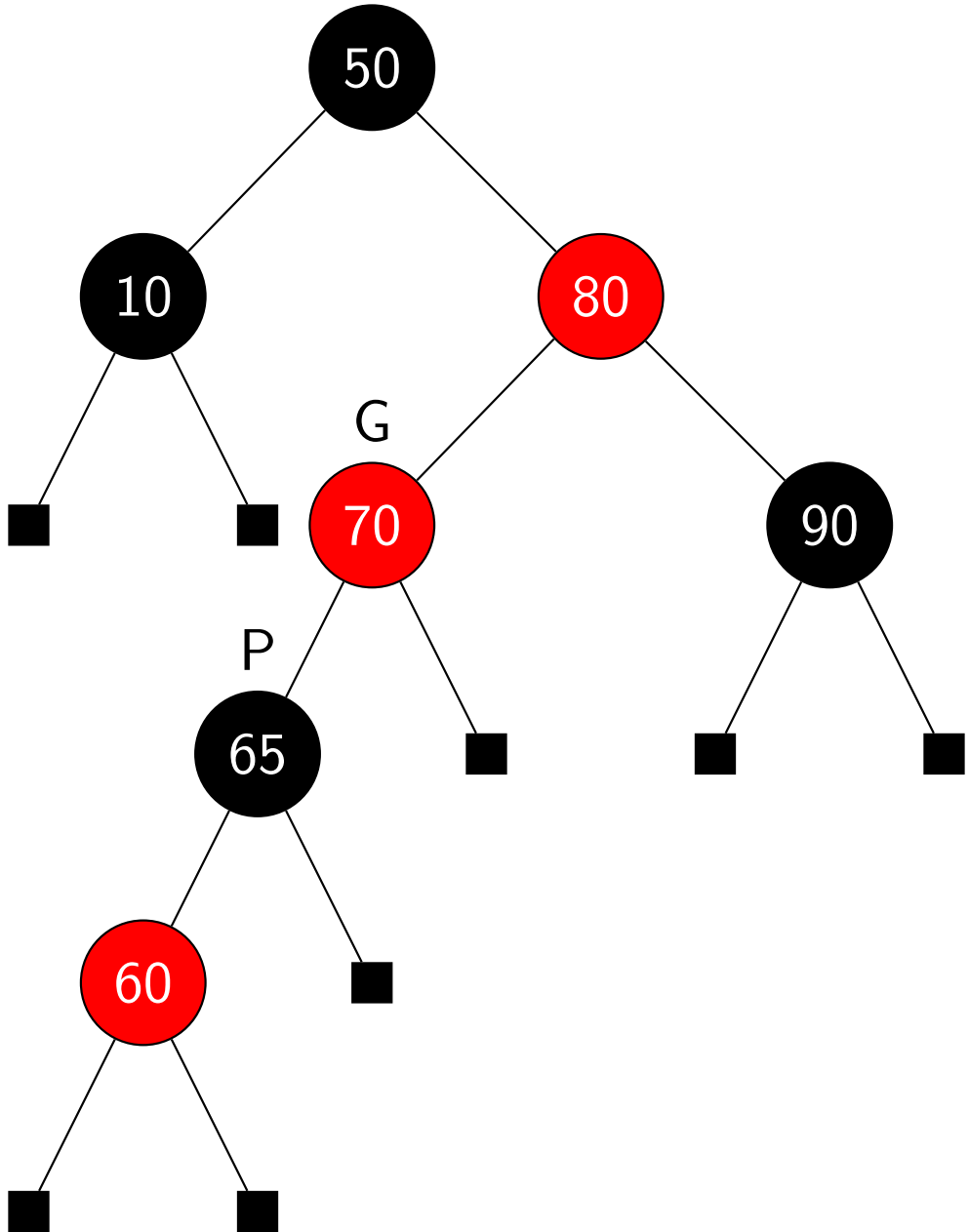
Insertion - Example

- Left Rotate about P



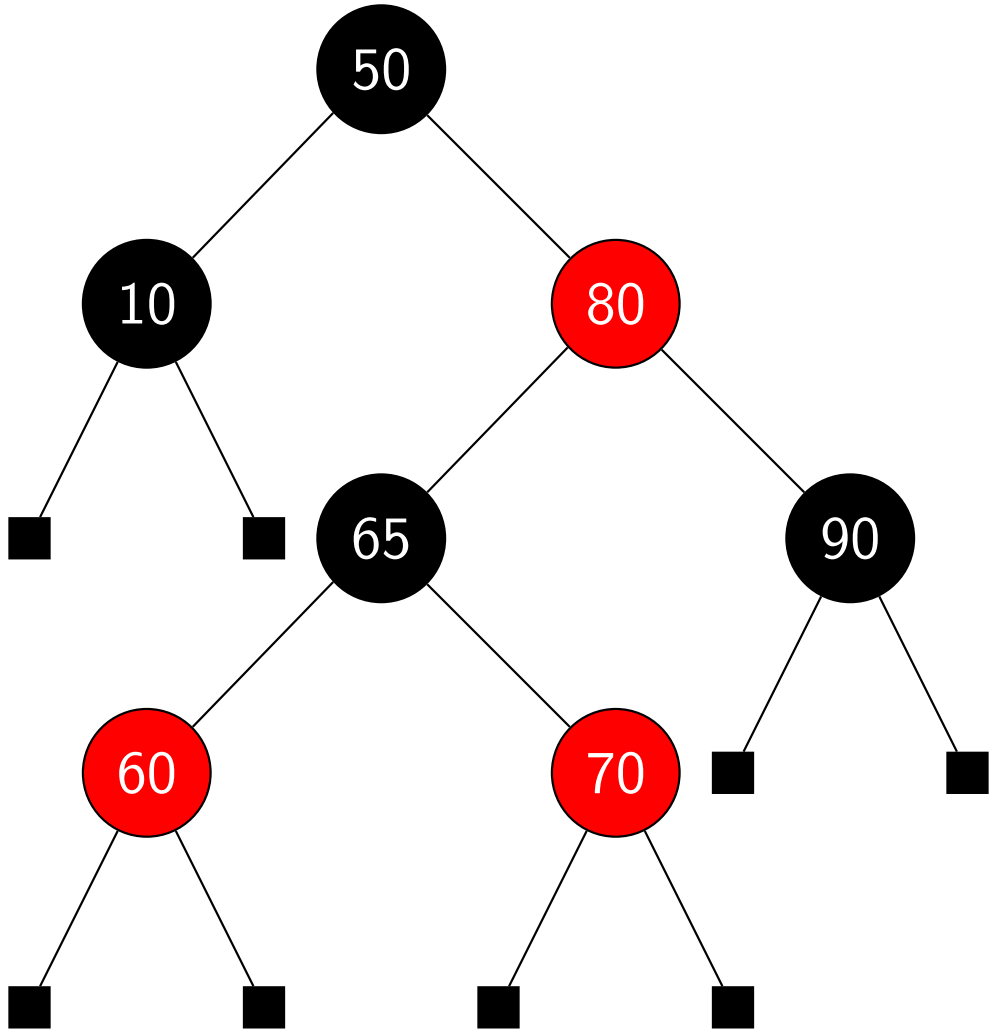
Insertion - Example

- Colour Change



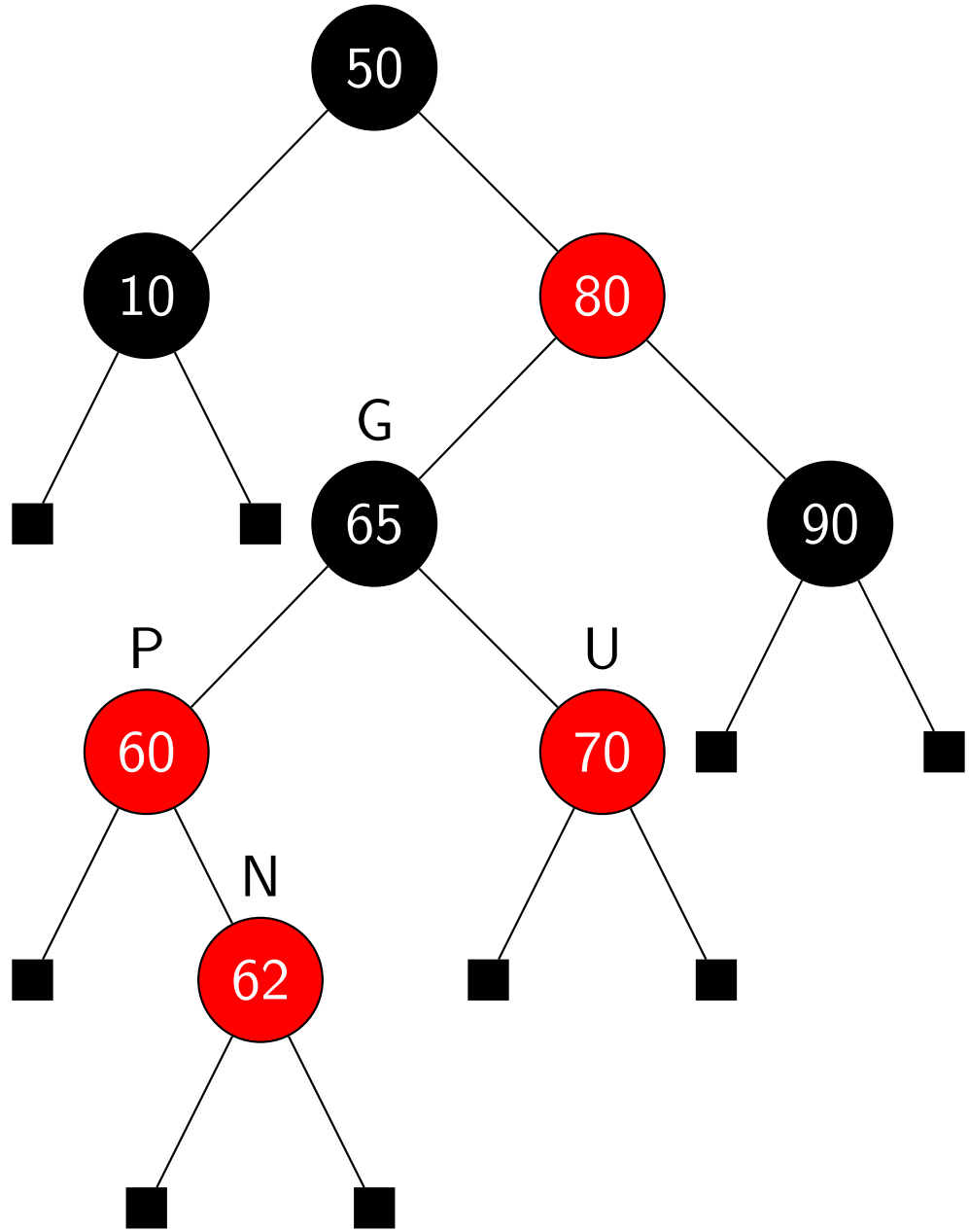
Insertion - Example

- Right Rotate about G



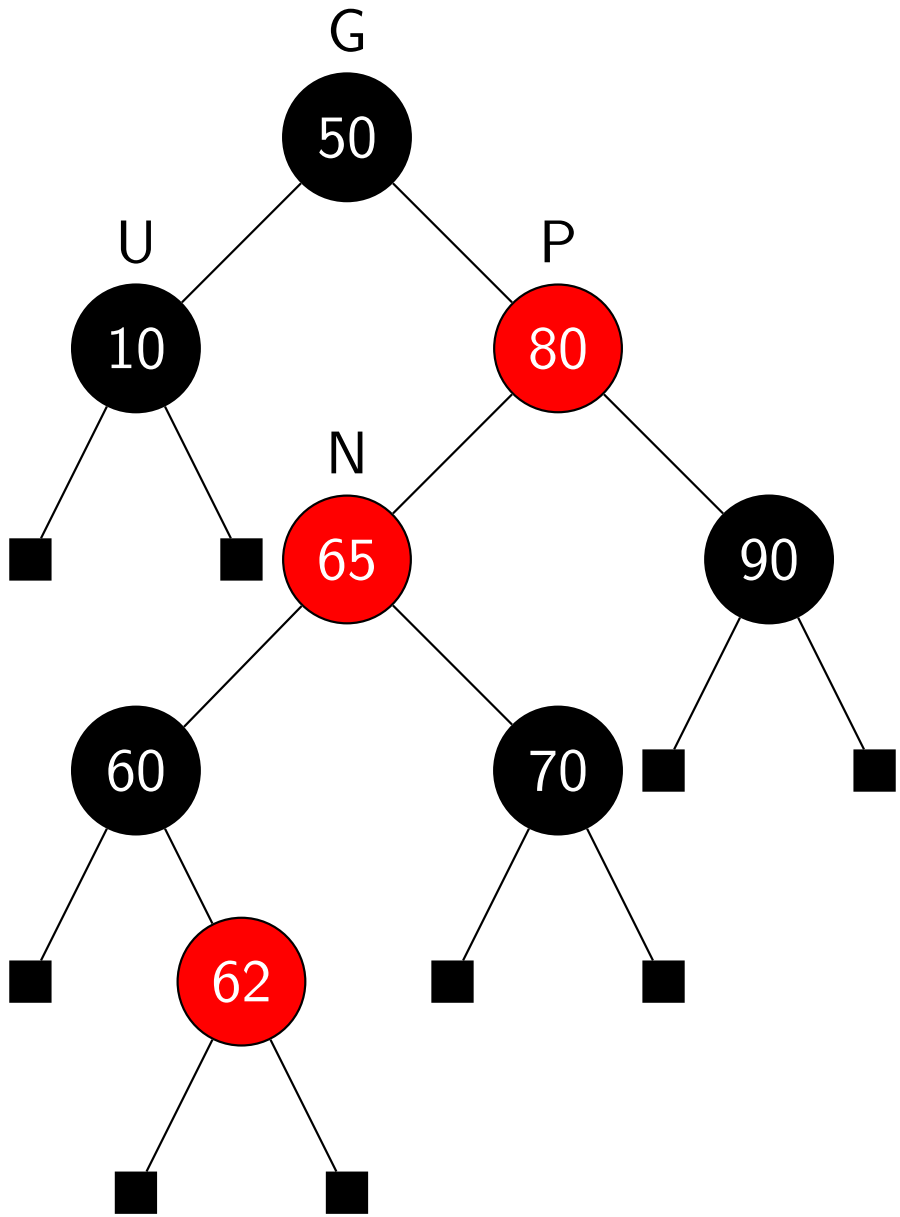
Insertion - Example

- Insert 62



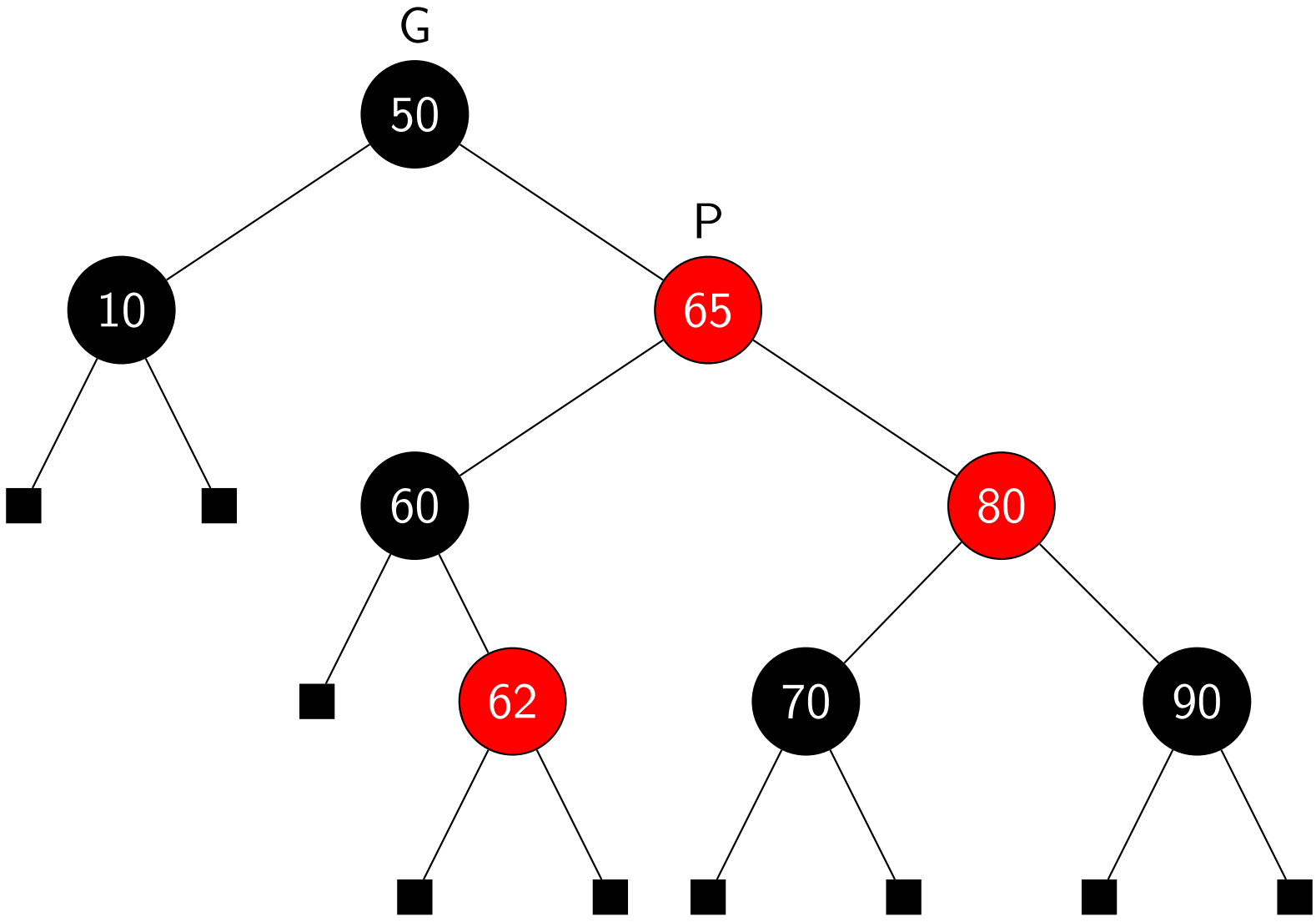
Insertion - Example

- Colour Change



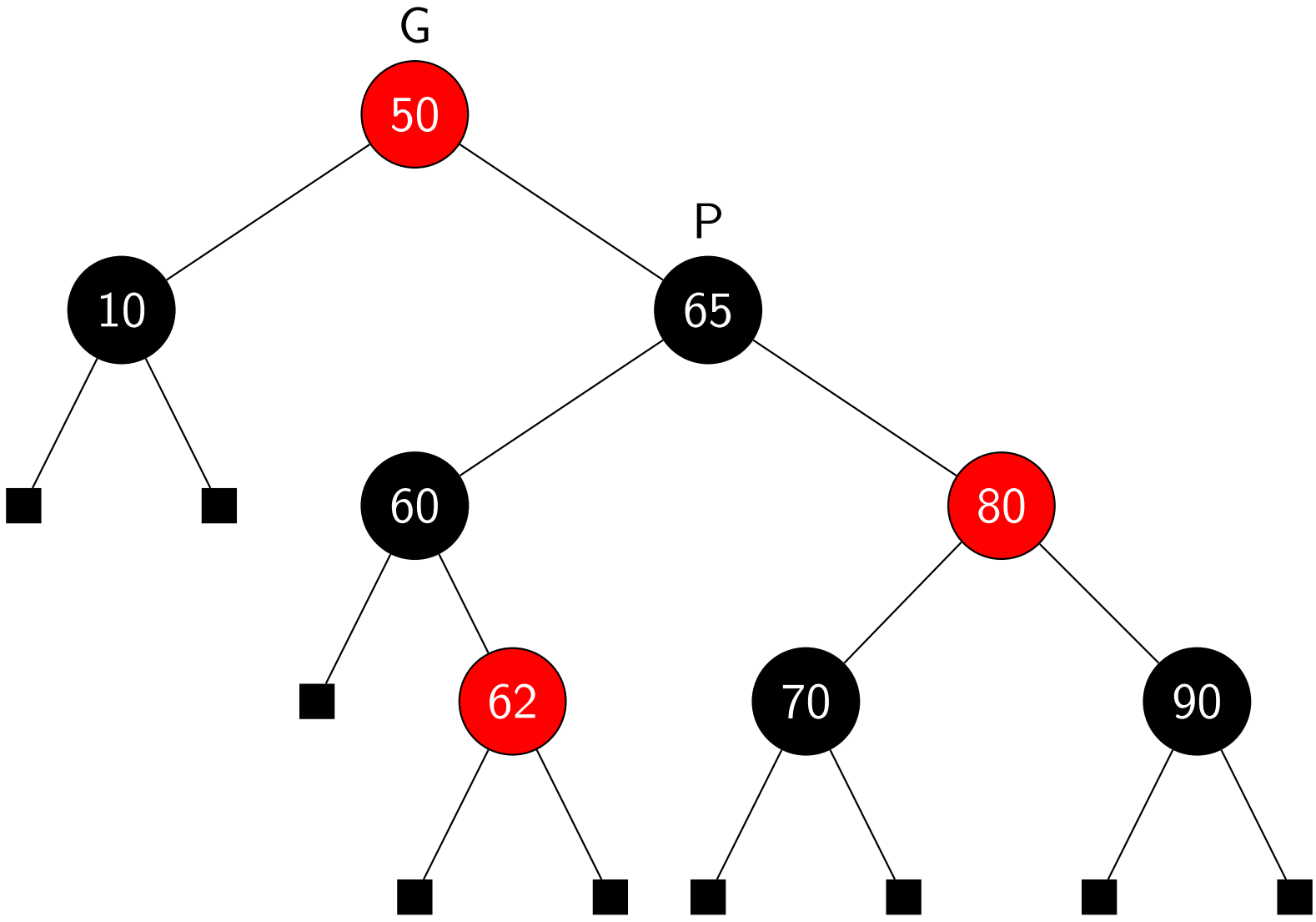
Insertion - Example

- Right Rotate about P



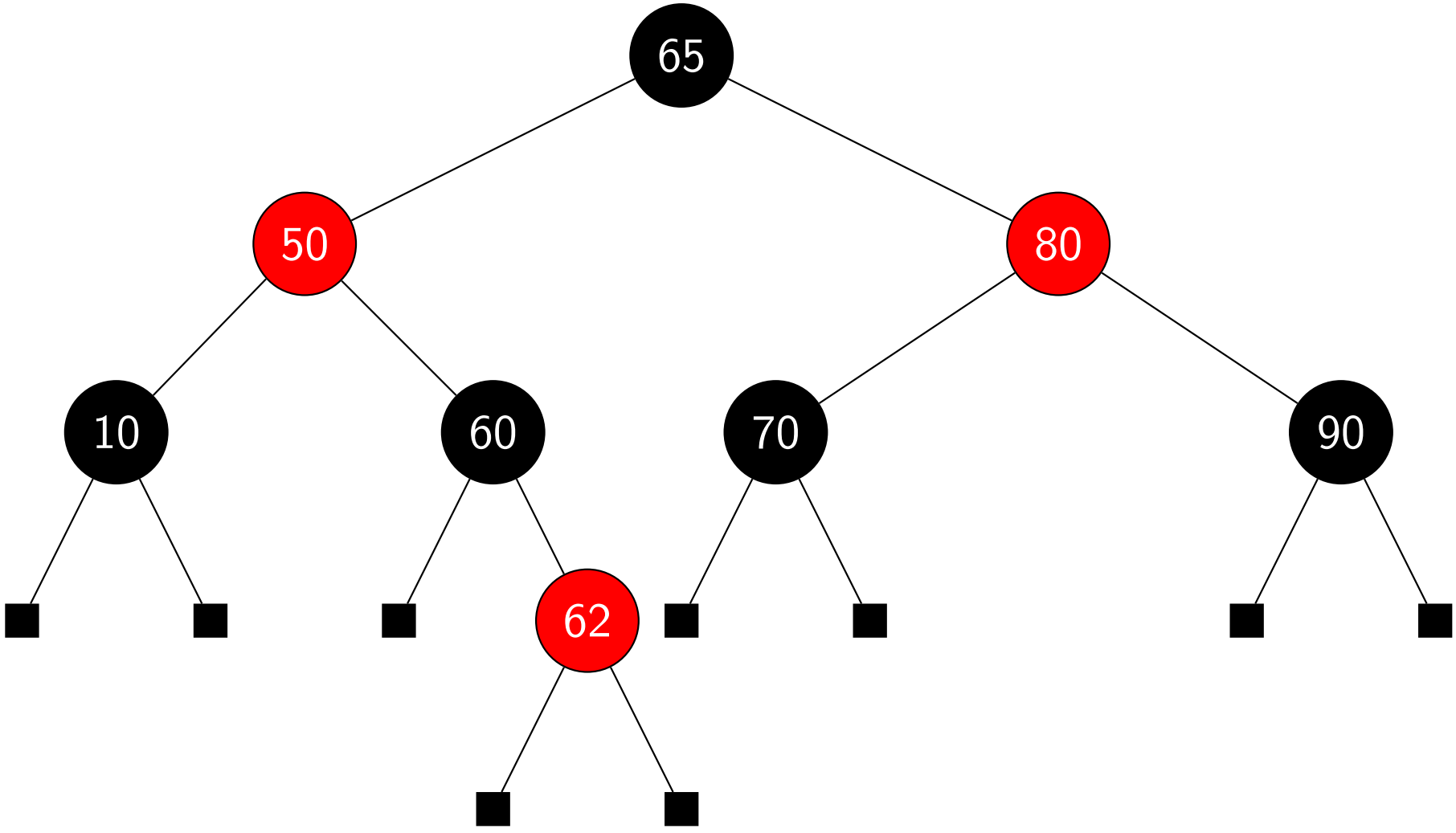
Insertion - Example

- Colour Change



Insertion - Example

- Left Rotate about G

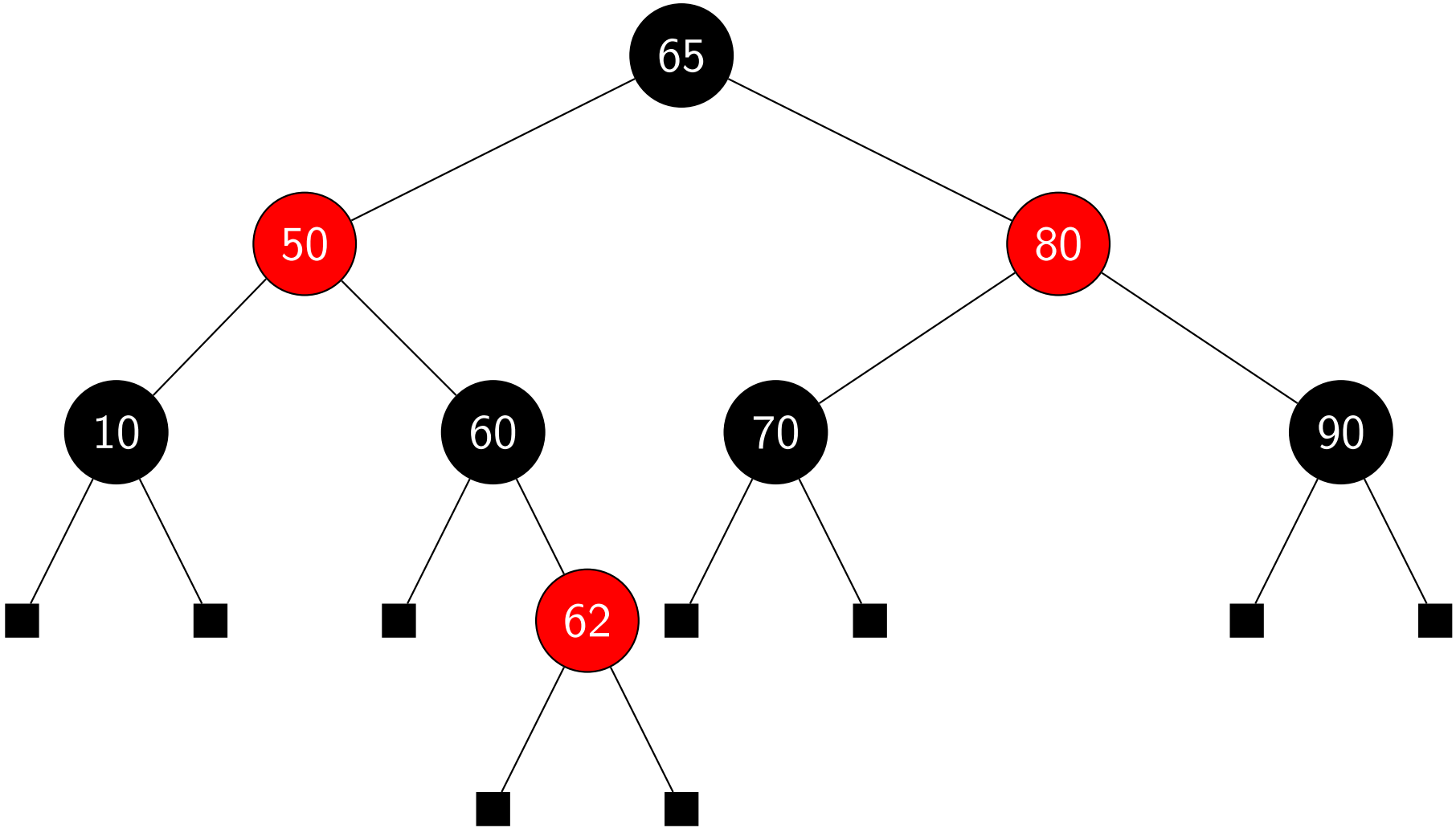


Deletion

1. Let D be the node to be deleted and let M be its inorder predecessor/successor node
2. Then deletion of D is done by moving the contents of M to D and then deleting M
3. If M is a red node, then its deletion causes no problem
4. If M is a black node, then property 4 of a red-black tree is violated.(We say that the tree has become imbalanced)
5. This is corrected by colour change/rotation

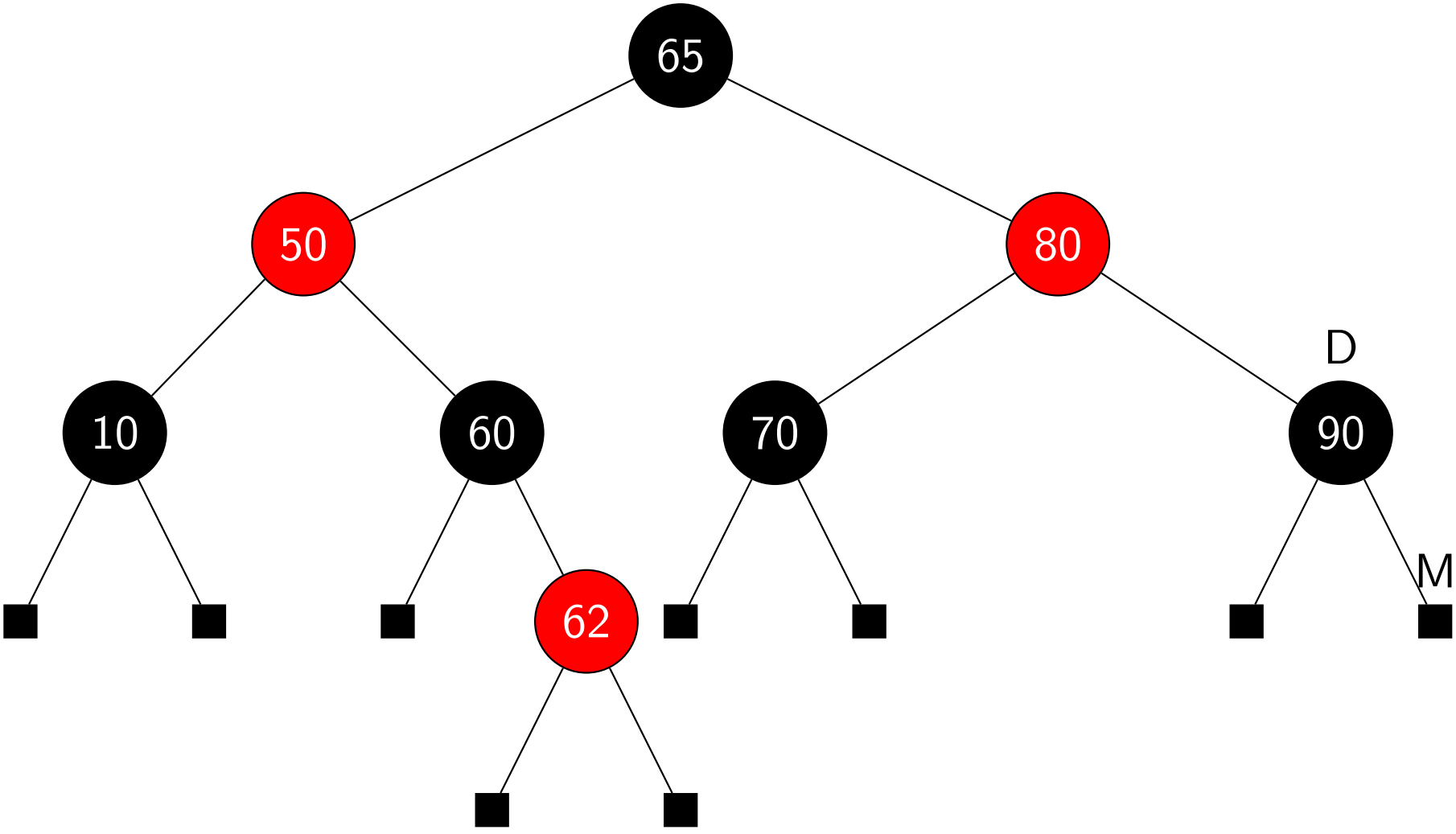
Deletion - Example

- Initial



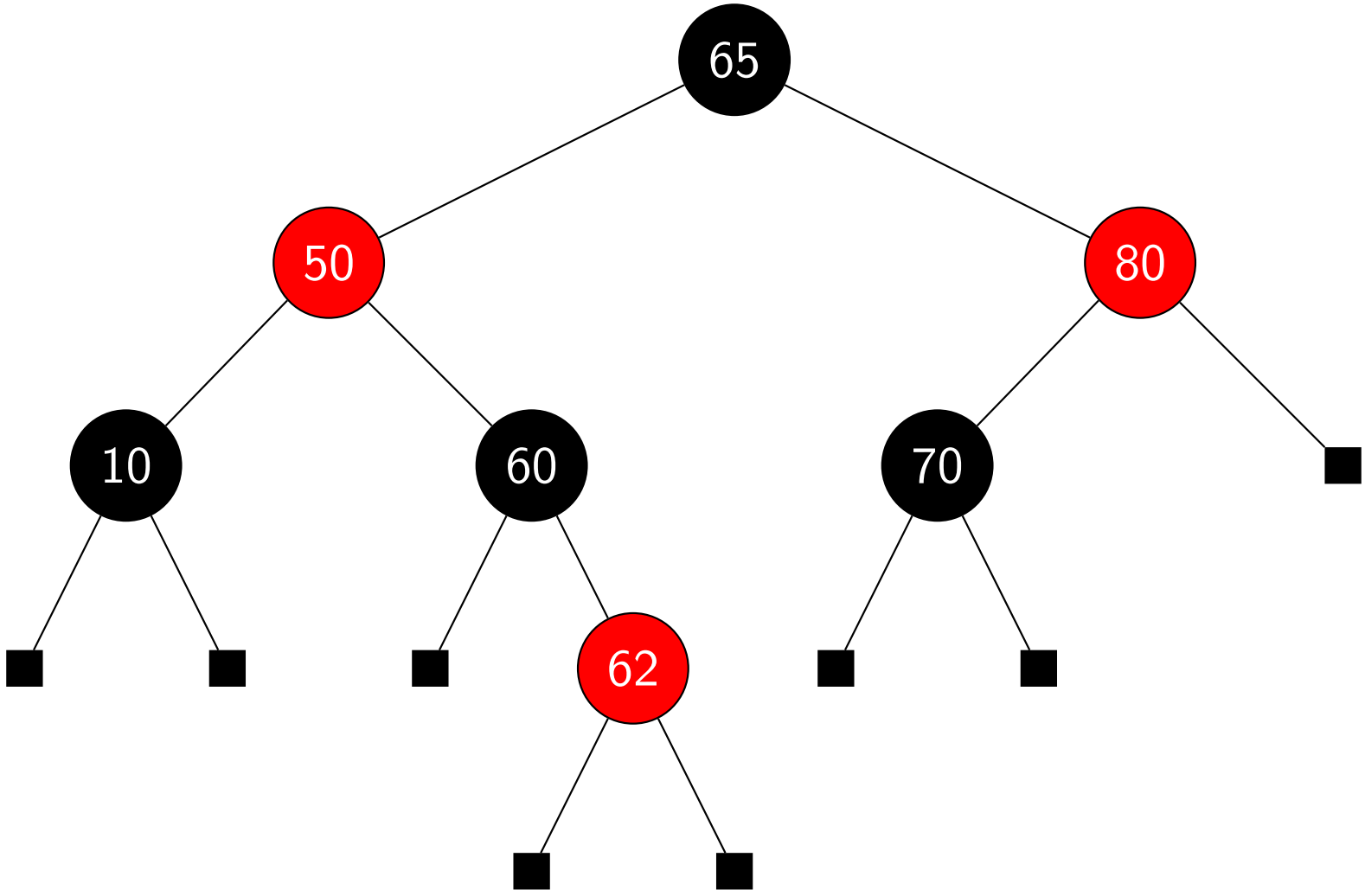
Deletion - Example

- Delete 90



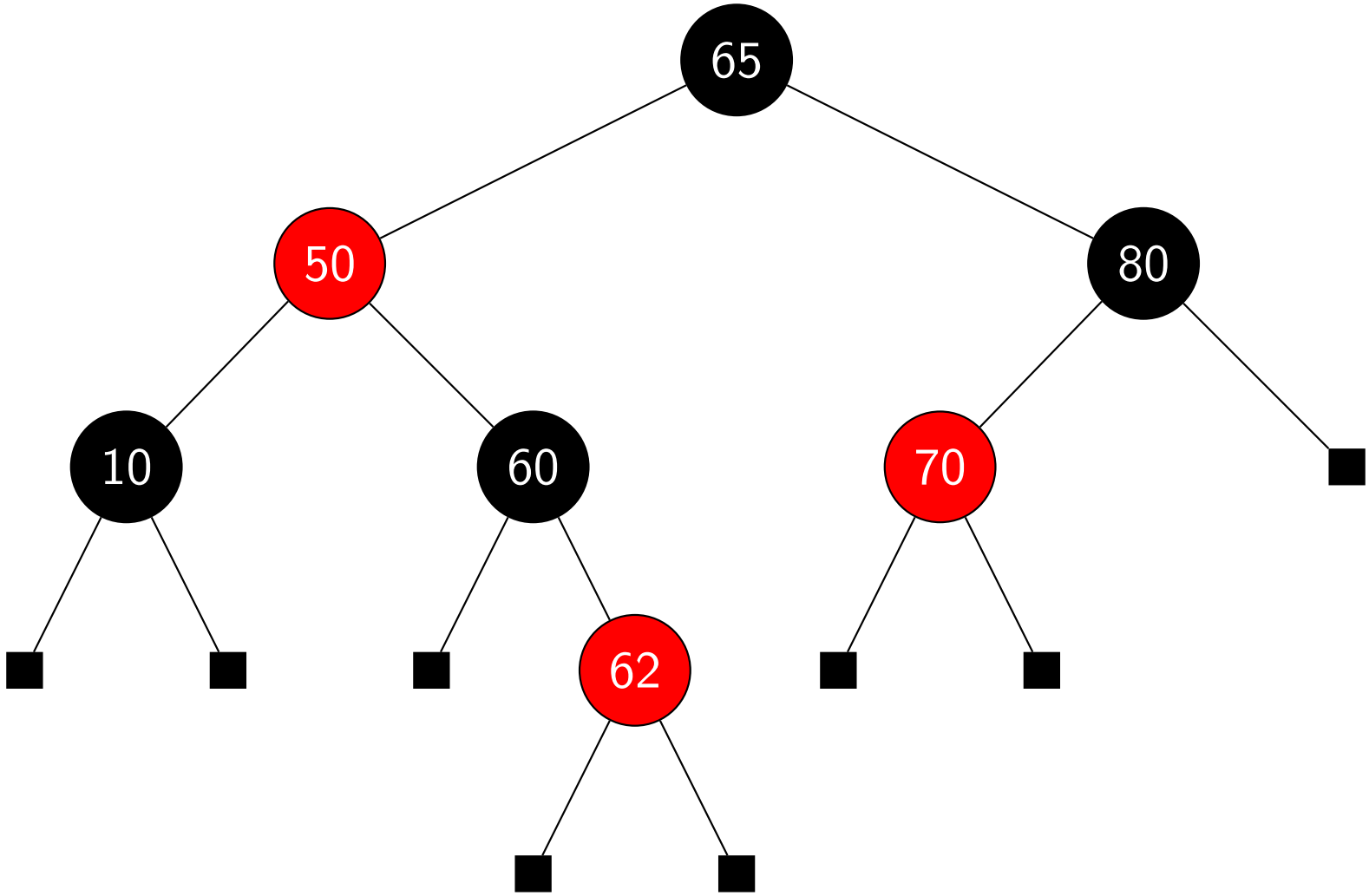
Deletion - Example

- 90 Deleted - Imbalanced Tree



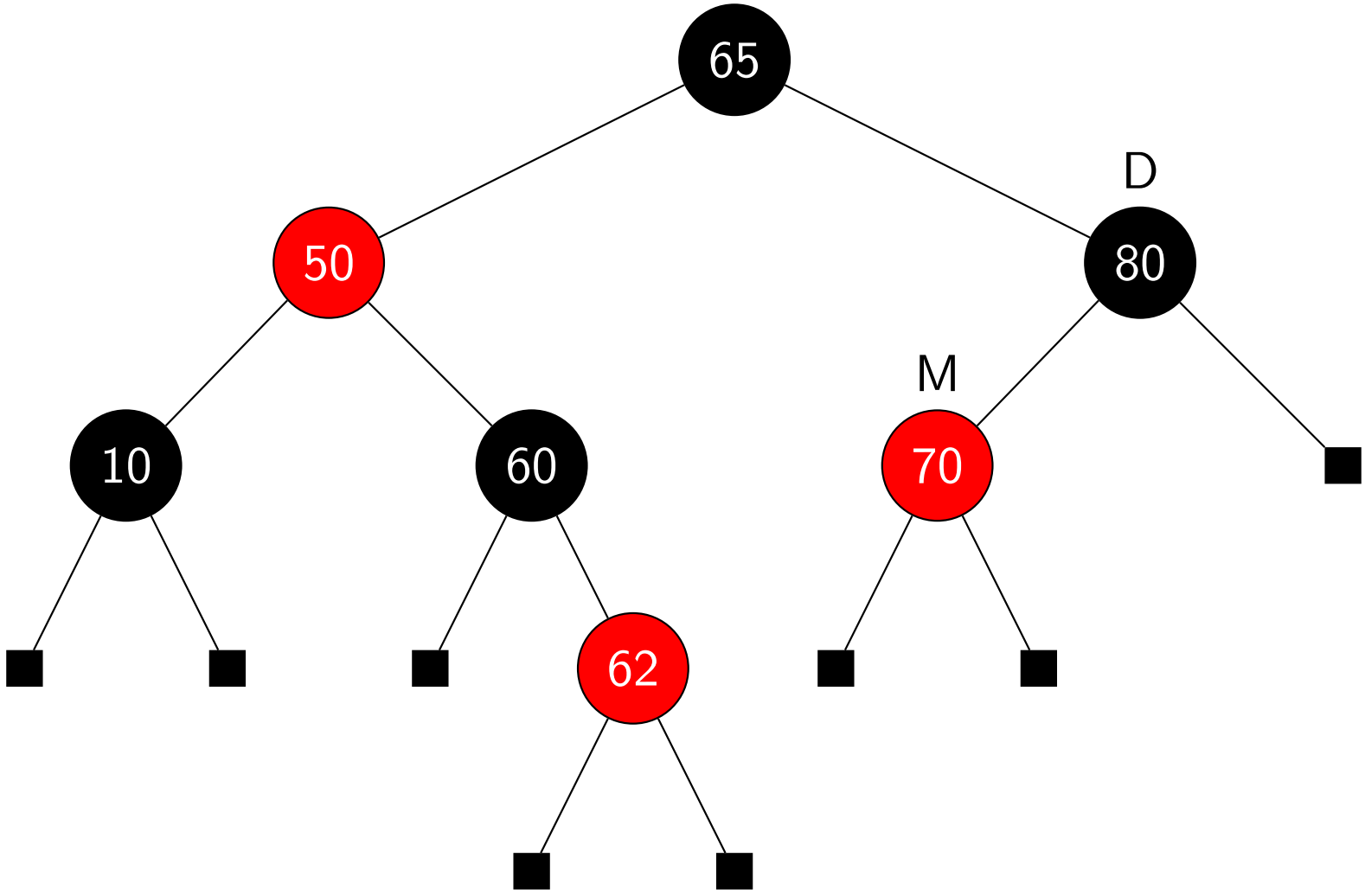
Deletion - Example

- Colour Change for Correcting Imbalance



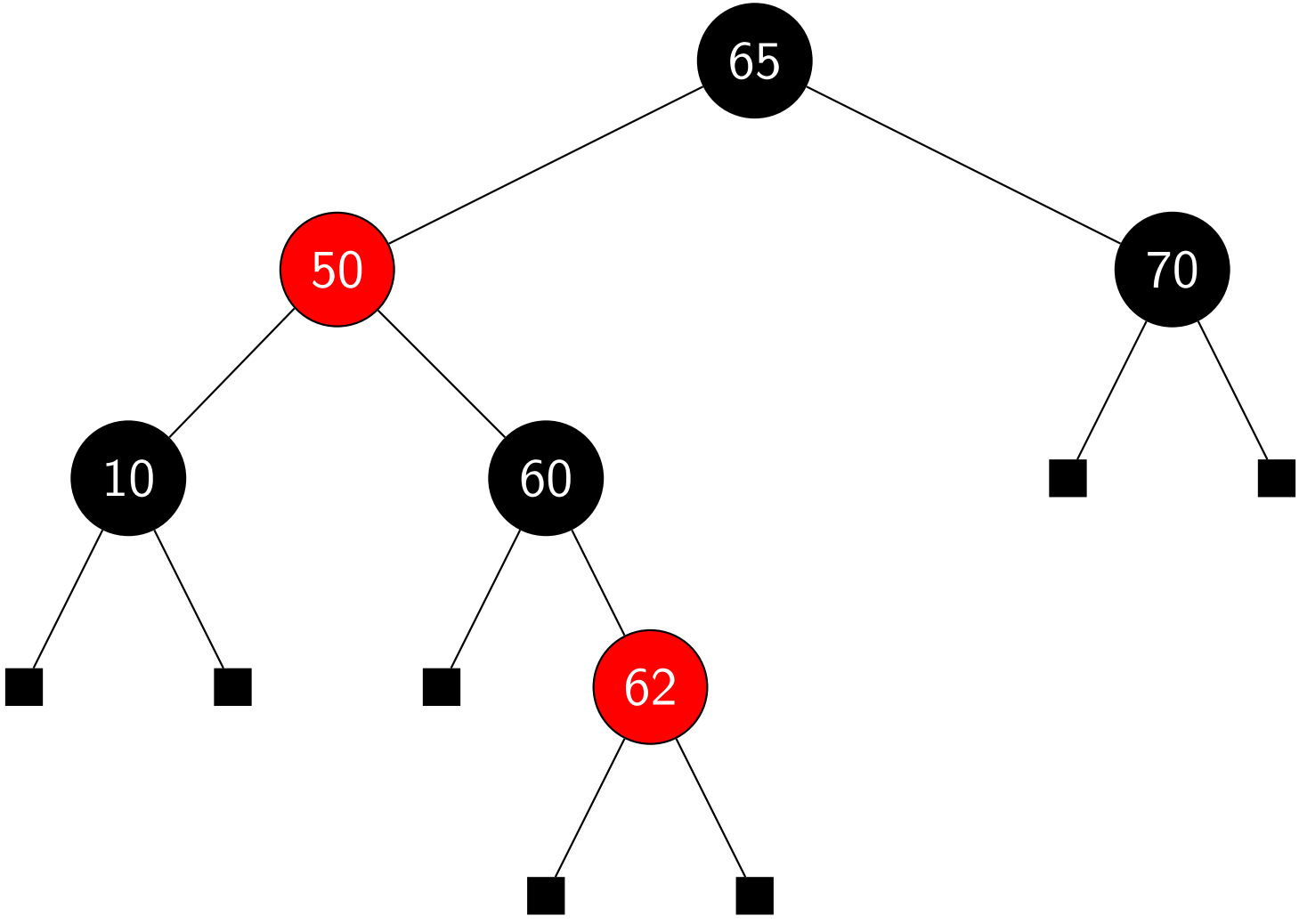
Deletion - Example

- Delete 80



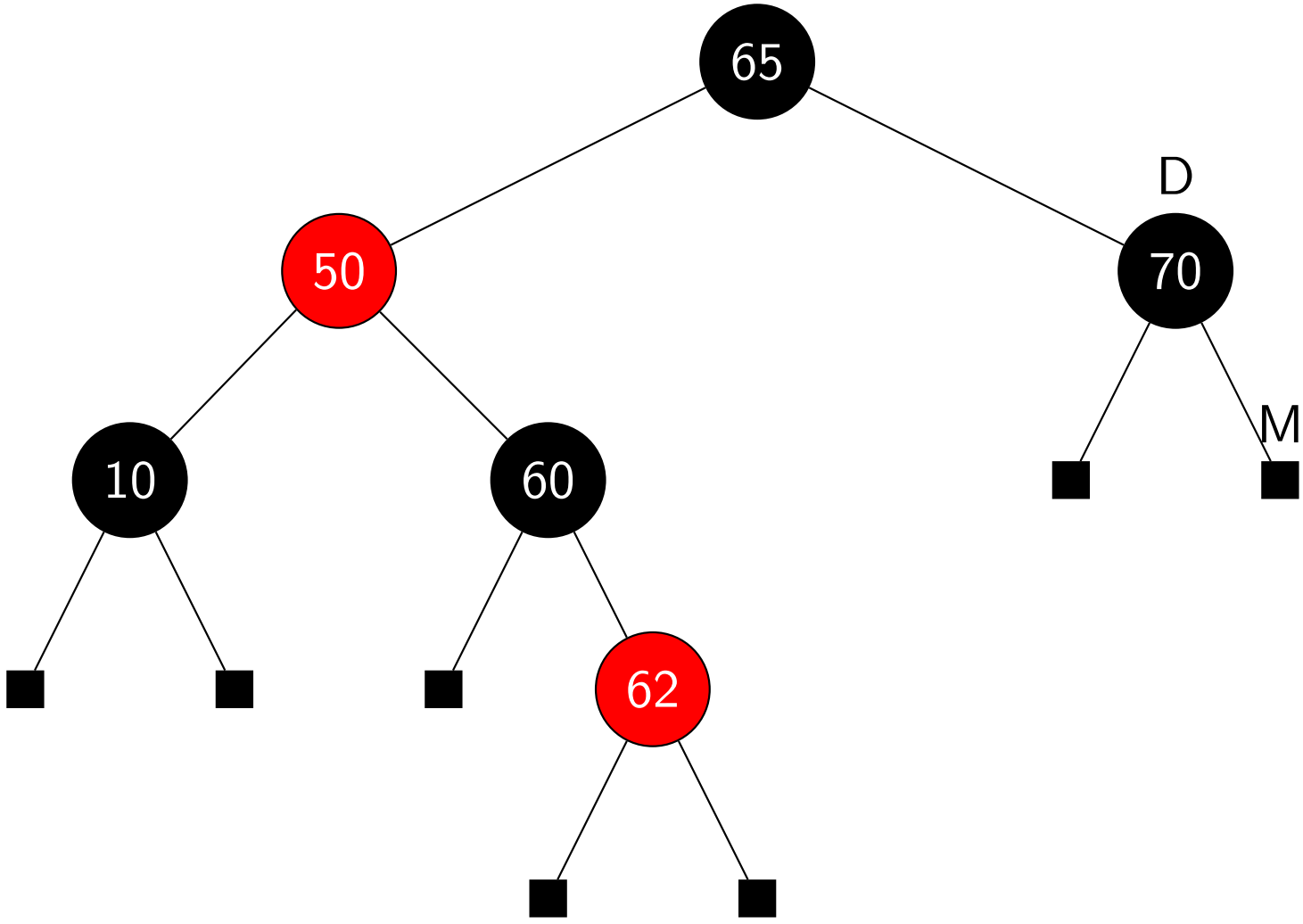
Deletion - Example

- 80 Deleted



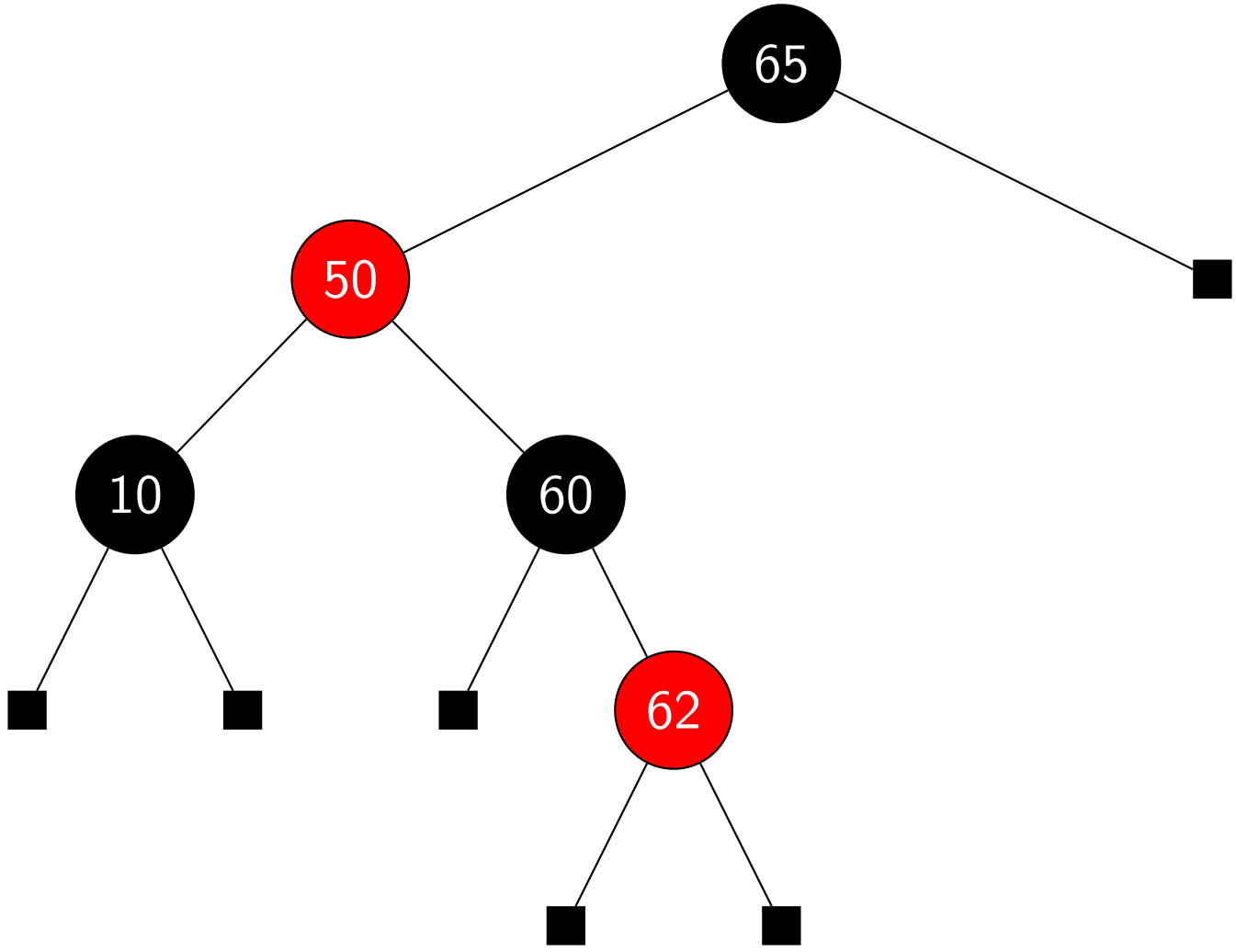
Deletion - Example

- Delete 70



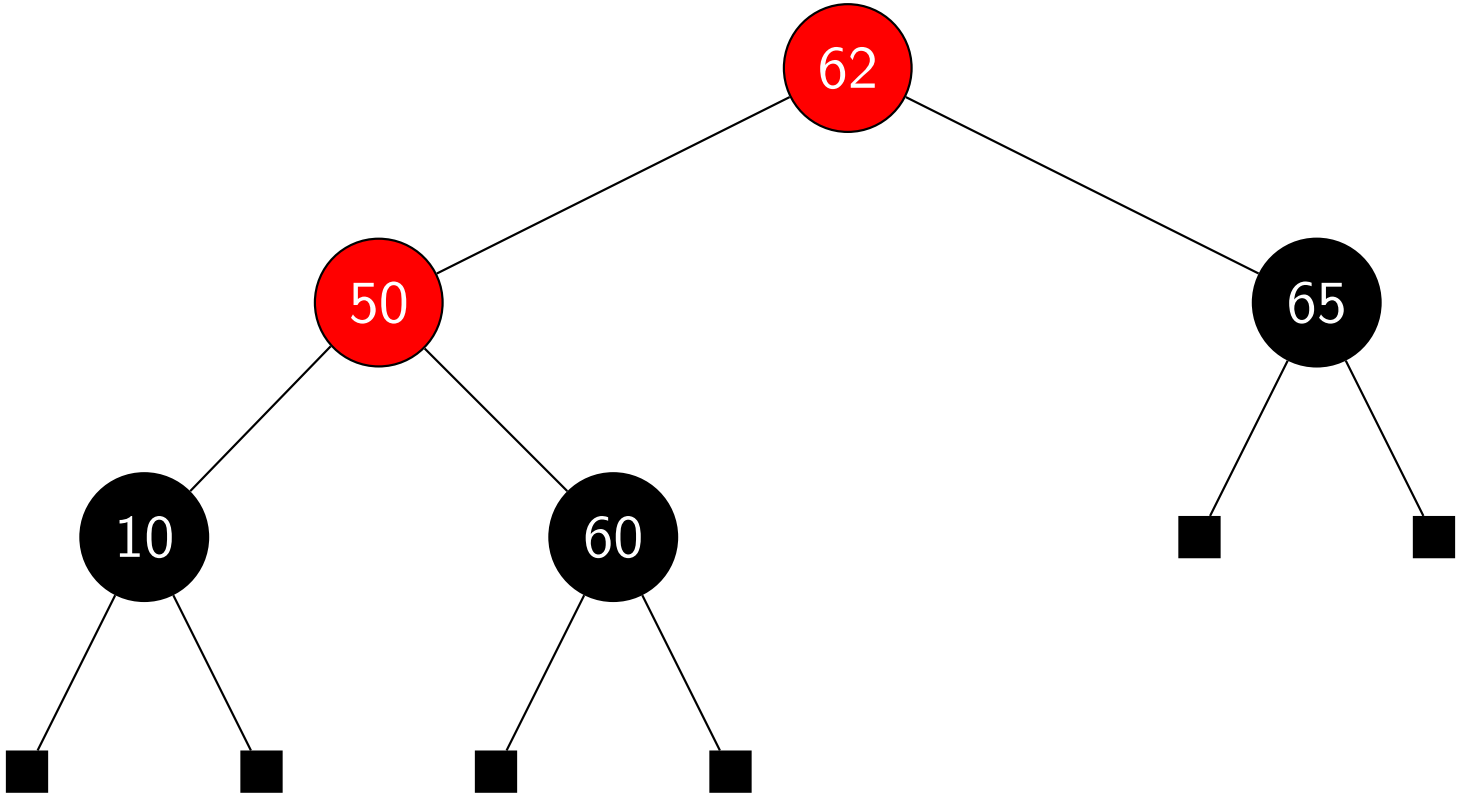
Deletion - Example

- 70 Deleted - Imbalanced Tree



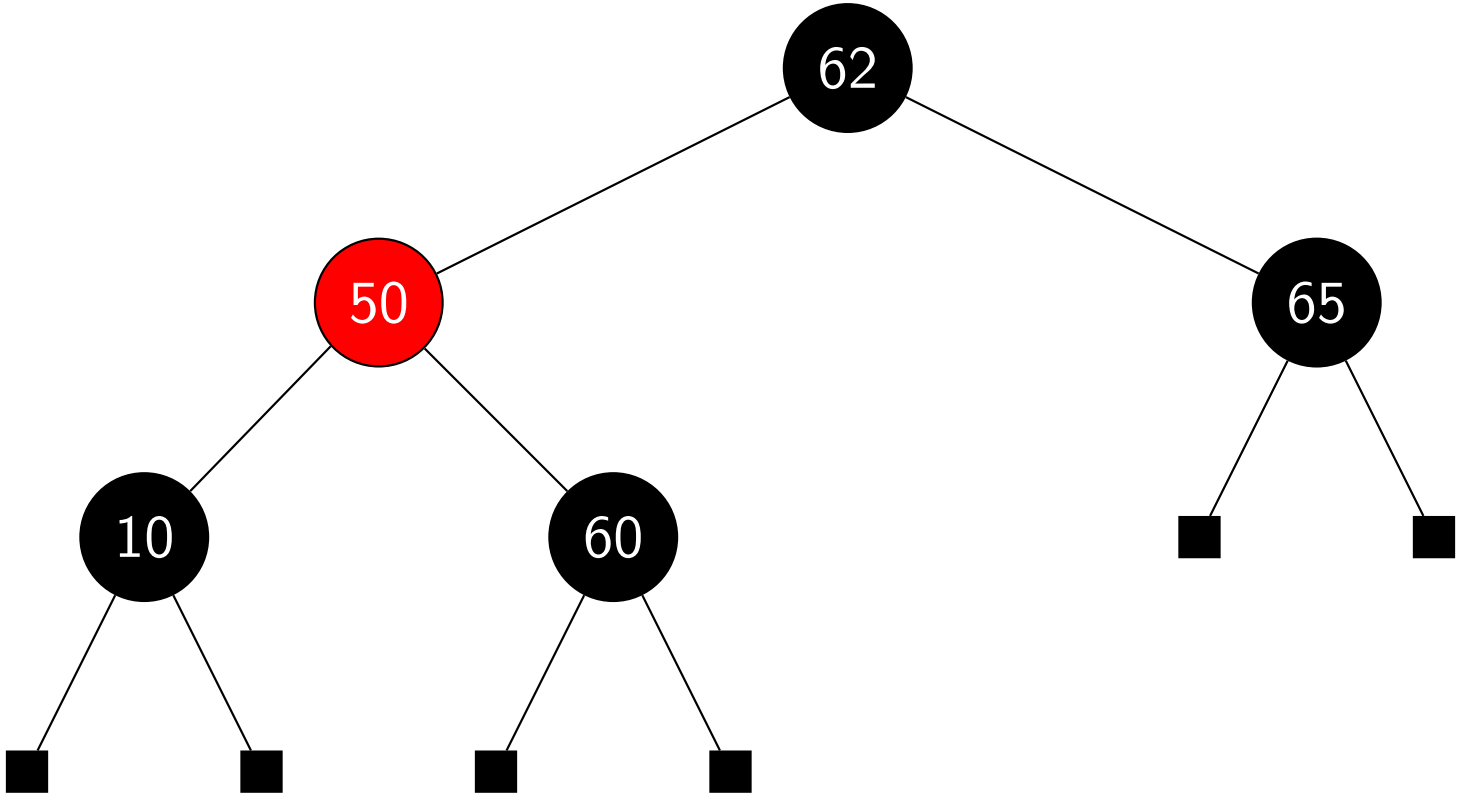
Deletion - Example

- Right Rotate about Node 65 - Imbalance Still There



Deletion - Example

- Colour Change - Imbalance is Removed



B-Tree

- A B-Tree generalises the binary search tree, allowing for nodes with more than two children.

Definition

A B-tree of order m is a tree that is either empty or is of height ≥ 1 which satisfies the following properties:

1. Every node has at most m children.
2. The root node has at least 2 children.
3. All nodes other than the root node and failure nodes have at least $\lceil m/2 \rceil$ child nodes.
4. All failure nodes are at the same level.
5. A node with k children contain k-1 keys.

B-Tree

Definition - continues

Keys in a node satisfies the following properties.

- All keys in a node will be in the increasing order.
- A node's keys act as separators between the keys in its child nodes.
- For example, let a_1 and a_2 be the keys of a node with 3 child nodes.
- Then, all keys in the left child will be less than a_1
- All keys in the middle child will be between a_1 and a_2
- All keys in the right child will be greater than a_2

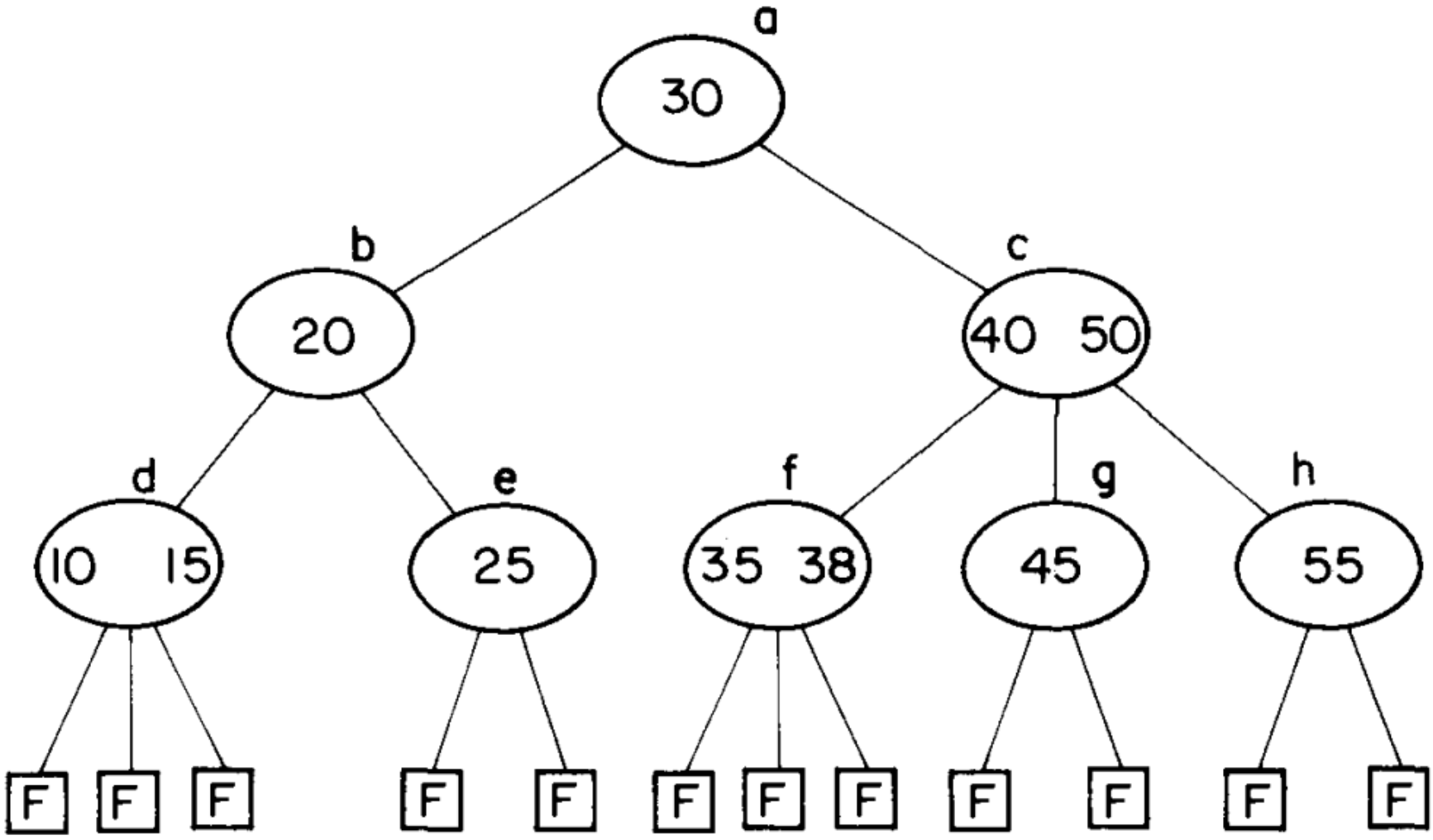
B-Tree

Failure Nodes

- Every null link in a B-Tree is replaced by a failure node.
- A failure node is a hypothetical node drawn square and marked with F.
- It is called a failure node, since all unsuccessful searches in the tree will terminate at this node.

B-Tree

- Example

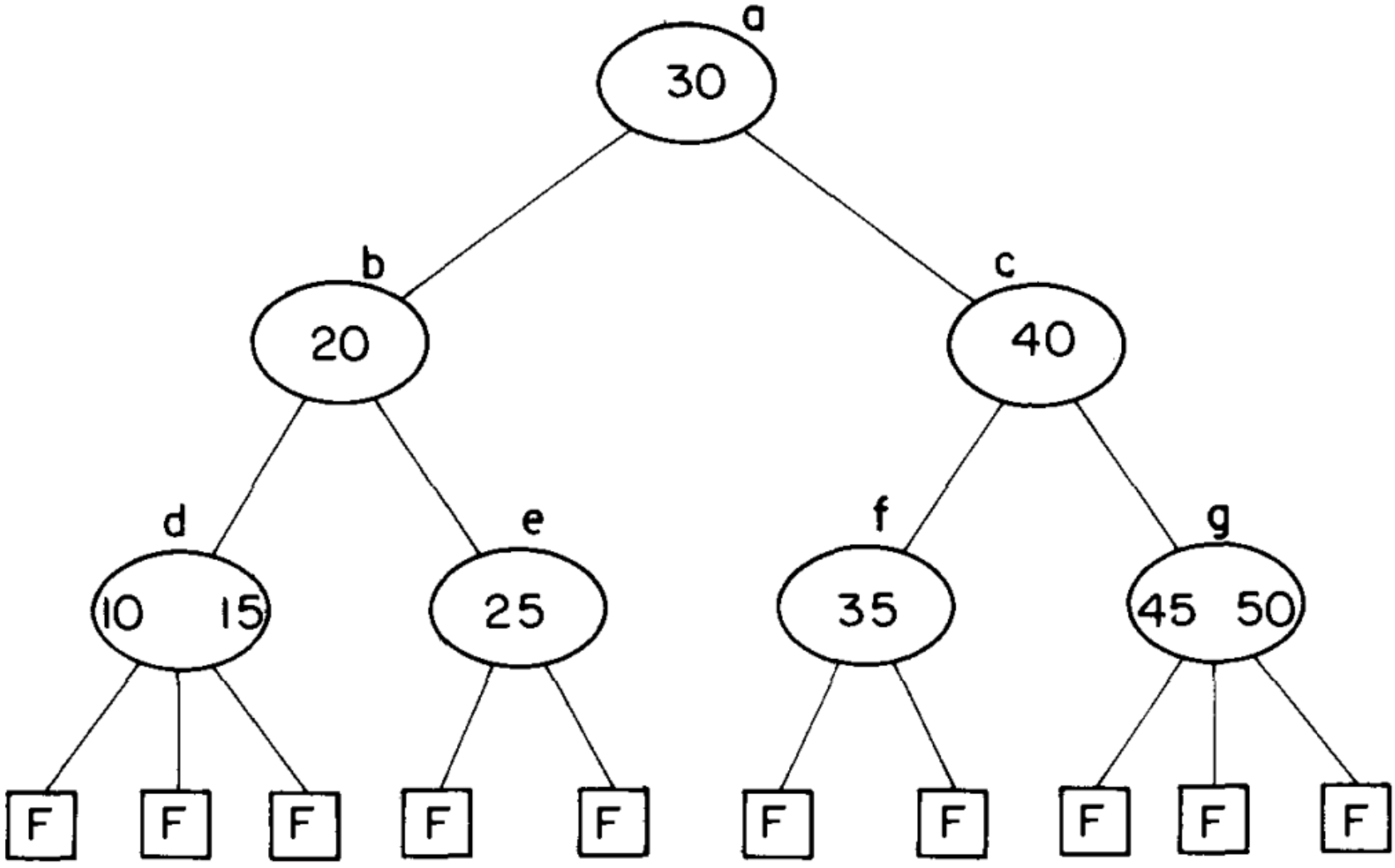


Insertion

1. Search the tree to find the leaf node where the new element should be added.
2. If the node contains fewer than the maximum allowed number of elements, then there is room for the new element. Insert the new element in the node, keeping the node's elements ordered.
3. Otherwise the node is full, evenly split it into two nodes so:
   - 3.1 A single median is chosen from among the leaf's elements and the new element.
   - 3.2 Values less than the median are put in the new left node and values greater than the median are put in the new right node, with the median acting as a separation value.
   - 3.3 The separation value is inserted in the node's parent, which may cause it to be split, and so on. If the node has no parent (i.e., the node was the root), create a new root above this node (increasing the height of the tree).

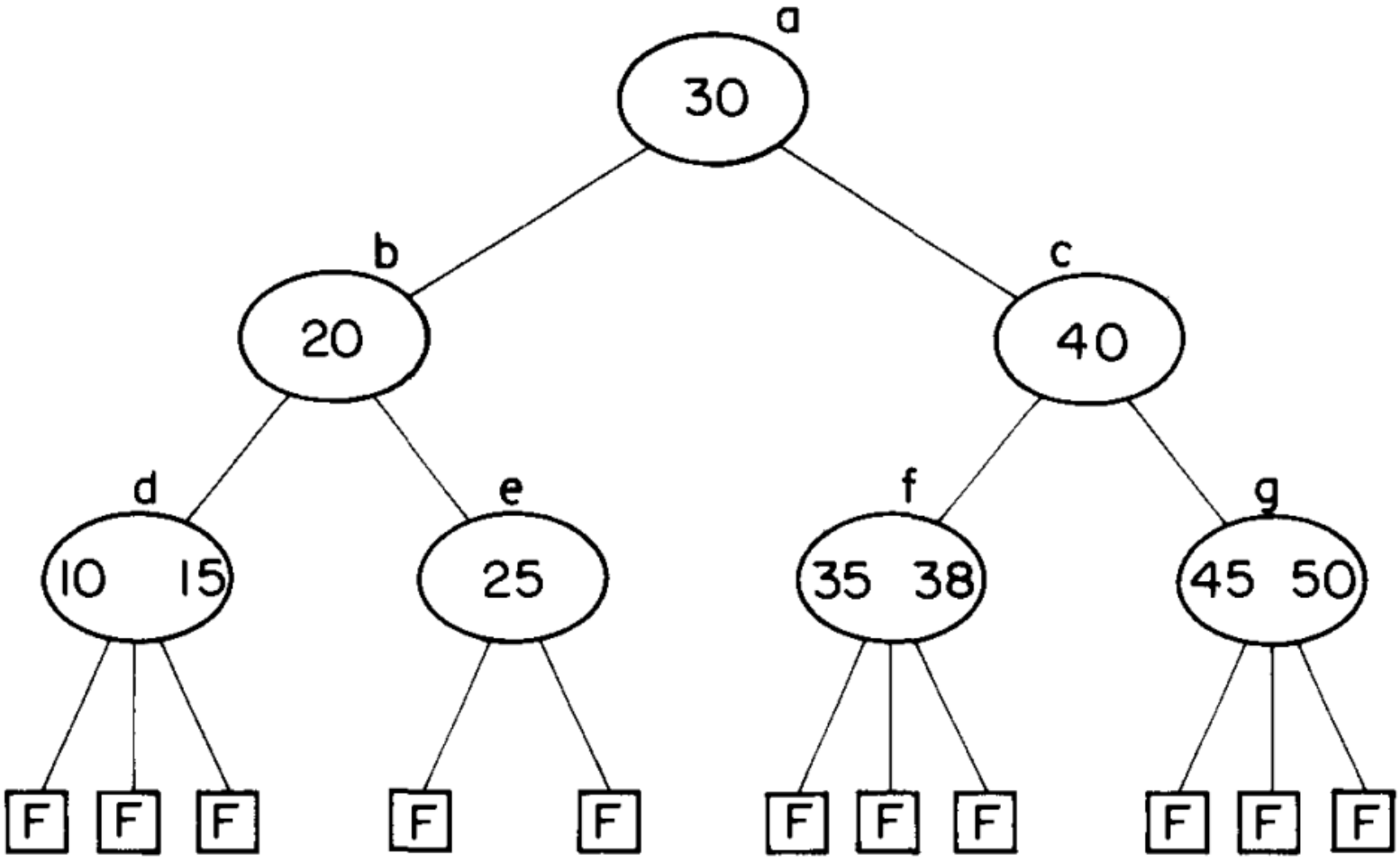
Insertion - Example

- Initial



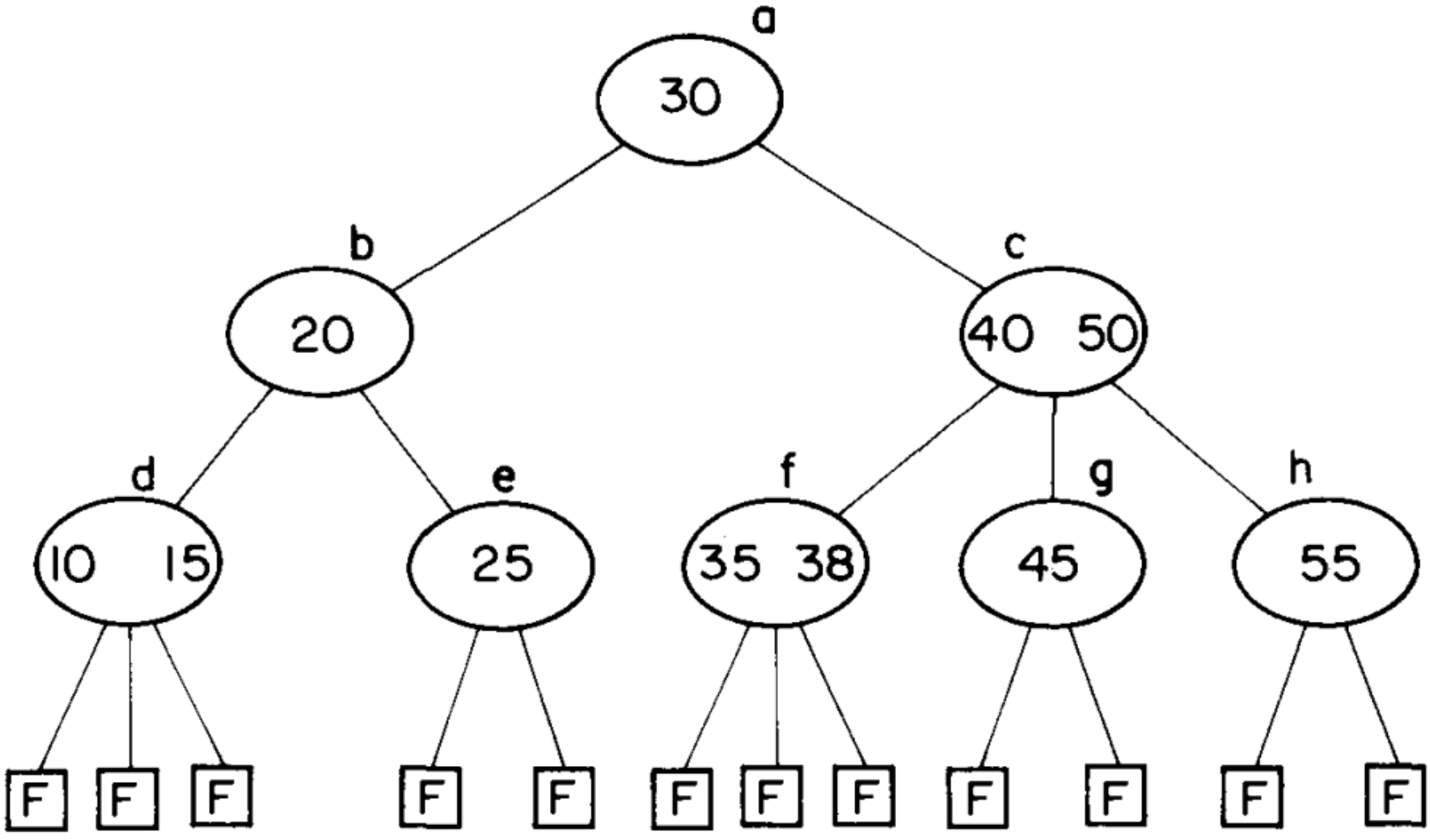
Insertion - Example

- Insert 38



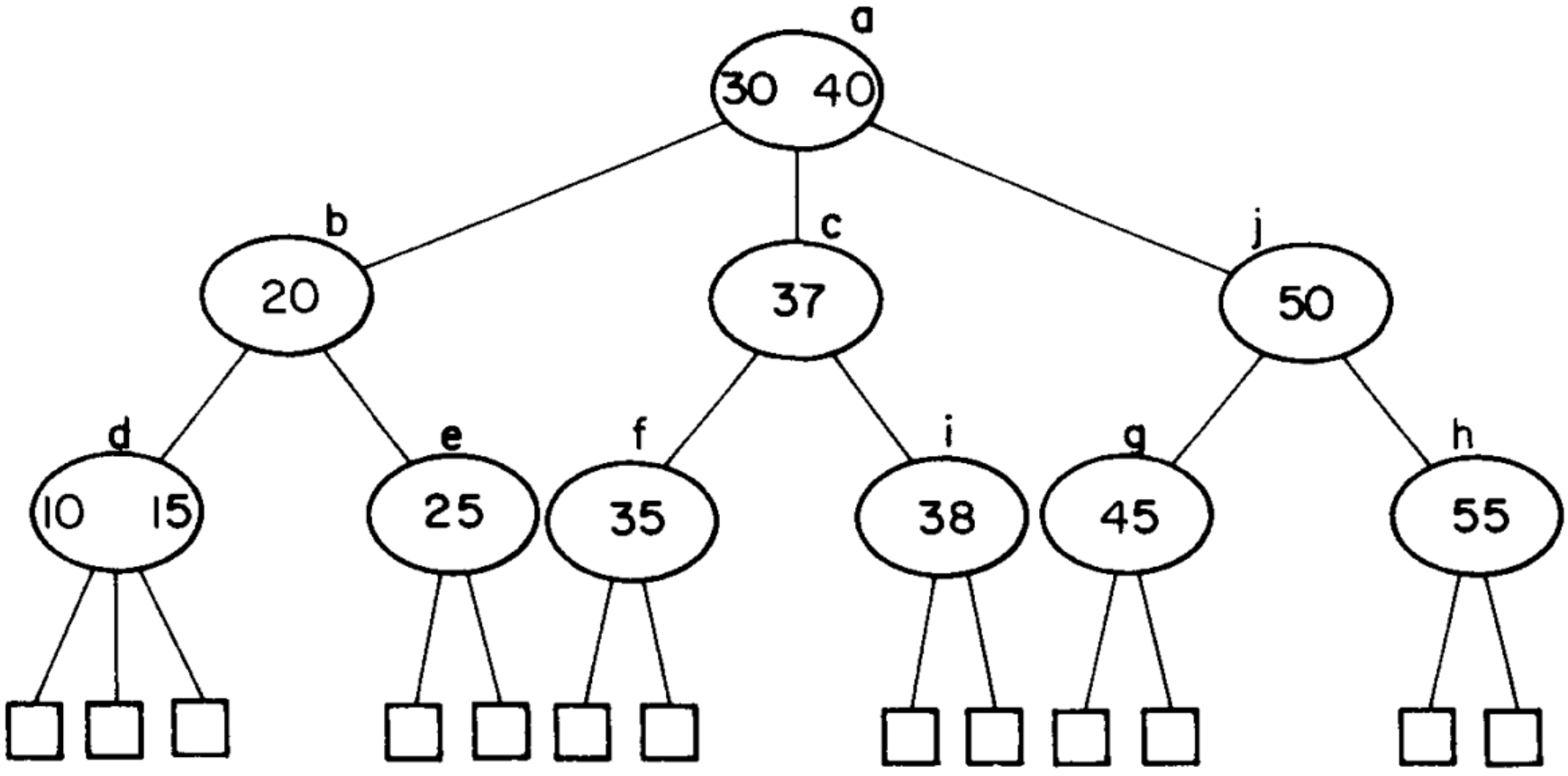
Insertion - Example

- Insert 55



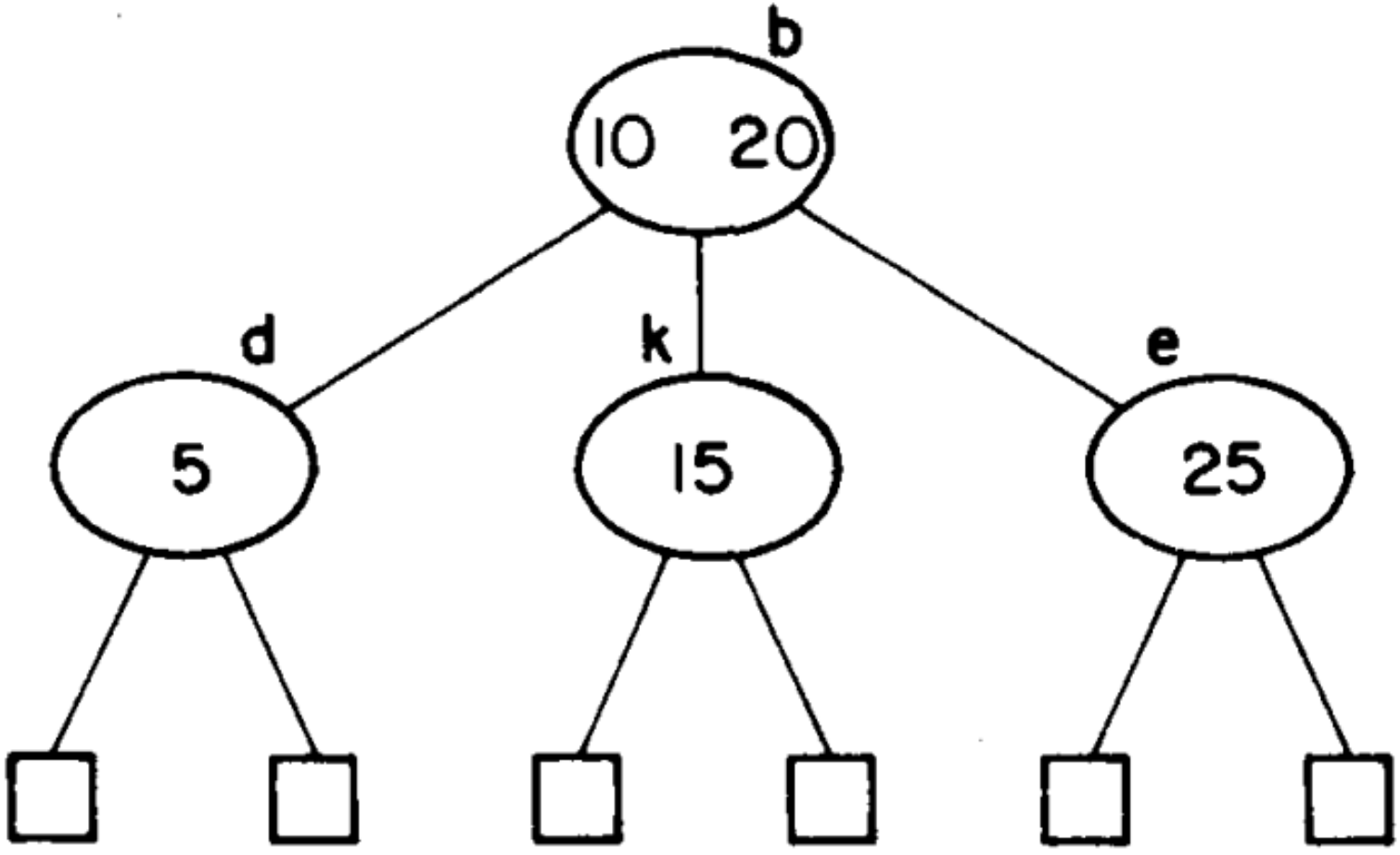
Insertion - Example

- Insert 37



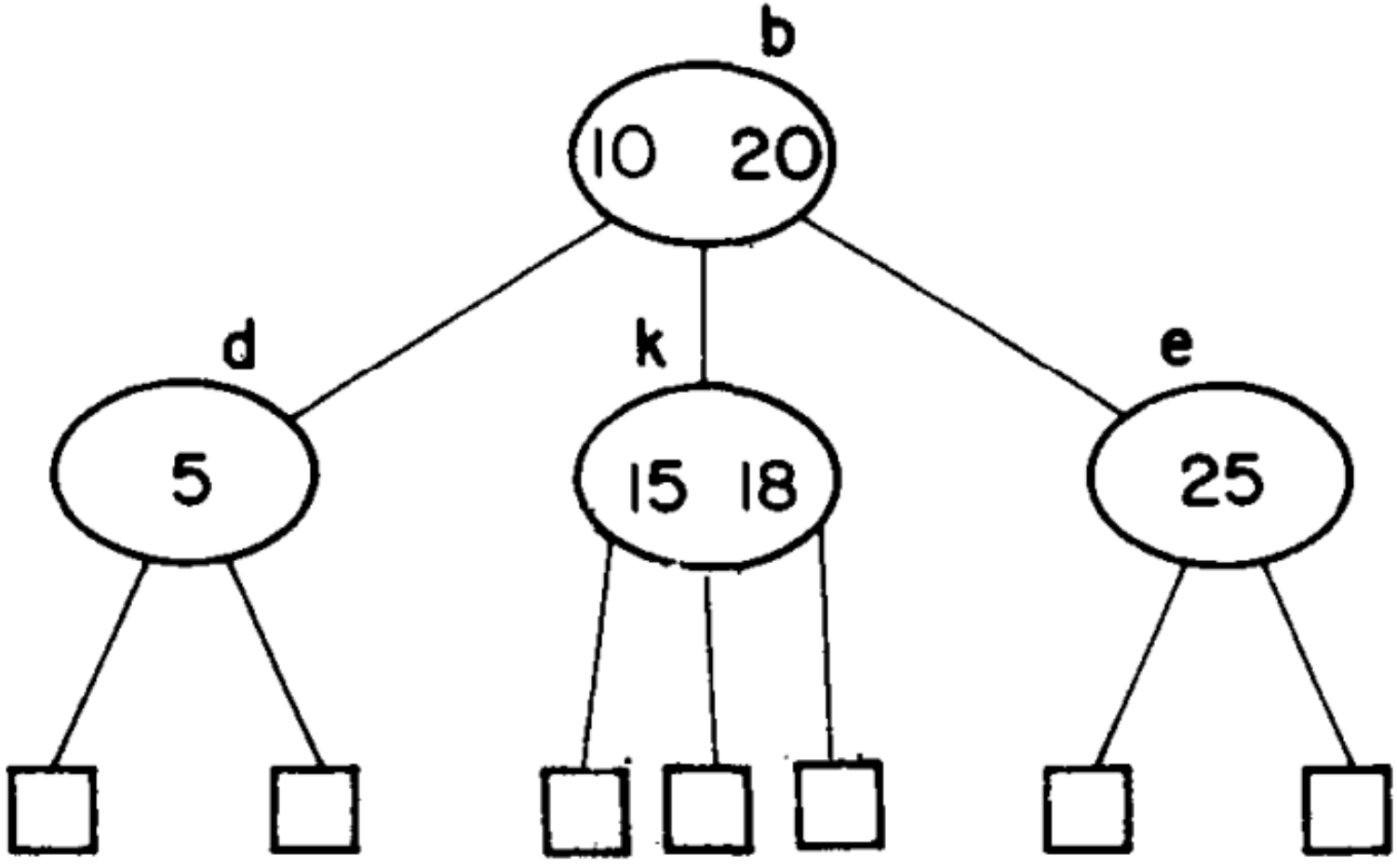
Insertion - Example

- Insert 5



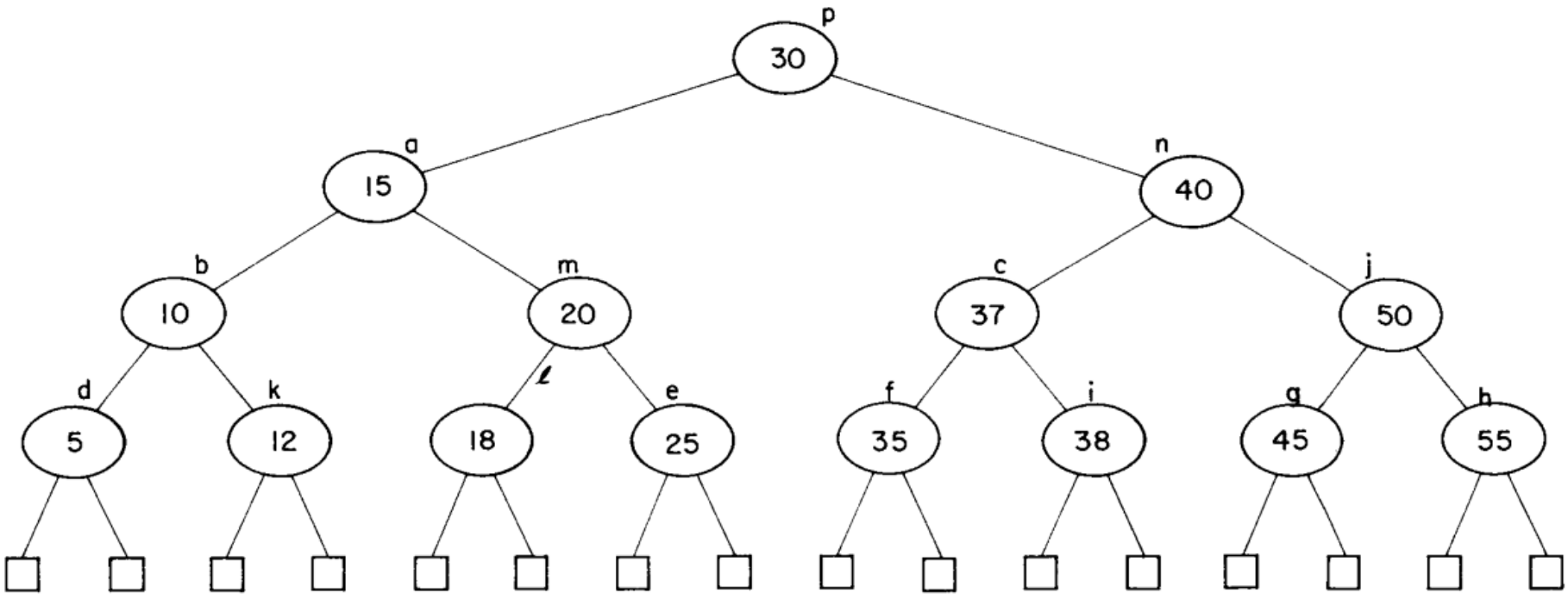
Insertion - Example

- Insert 18



Insertion - Example

- Insert 12



Deletion

1. Search for the value to delete.
2. The value is either in a leaf node or in a non-leaf node.

- Deletion From A Leaf Node

1. Delete the value from the node.
2. If underflow happens, rebalance the tree.

- Deletion From A NonLeaf Node

1. Choose a new separator (either the largest element in the left subtree or the smallest element in the right subtree),remove it from the leaf node it is in, and replace the element to be deleted with the new separator.
2. If that leaf node is now deficient, then rebalance the tree.

Deletion

- Rebalancing After Deletion

1. If the deficient node's right sibling exists and has more than the minimum number of elements, then rotate left
   1.1 Copy the separator from the parent to the end of the deficient node
   1.2 Replace the separator in the parent with the first element of the right sibling
2. Otherwise, if the deficient node's left sibling exists and has more than the minimum number of elements, then rotate right
   2.1 Copy the separator from the parent to the start of the deficient node
   2.2 Replace the separator in the parent with the last element of the left sibling

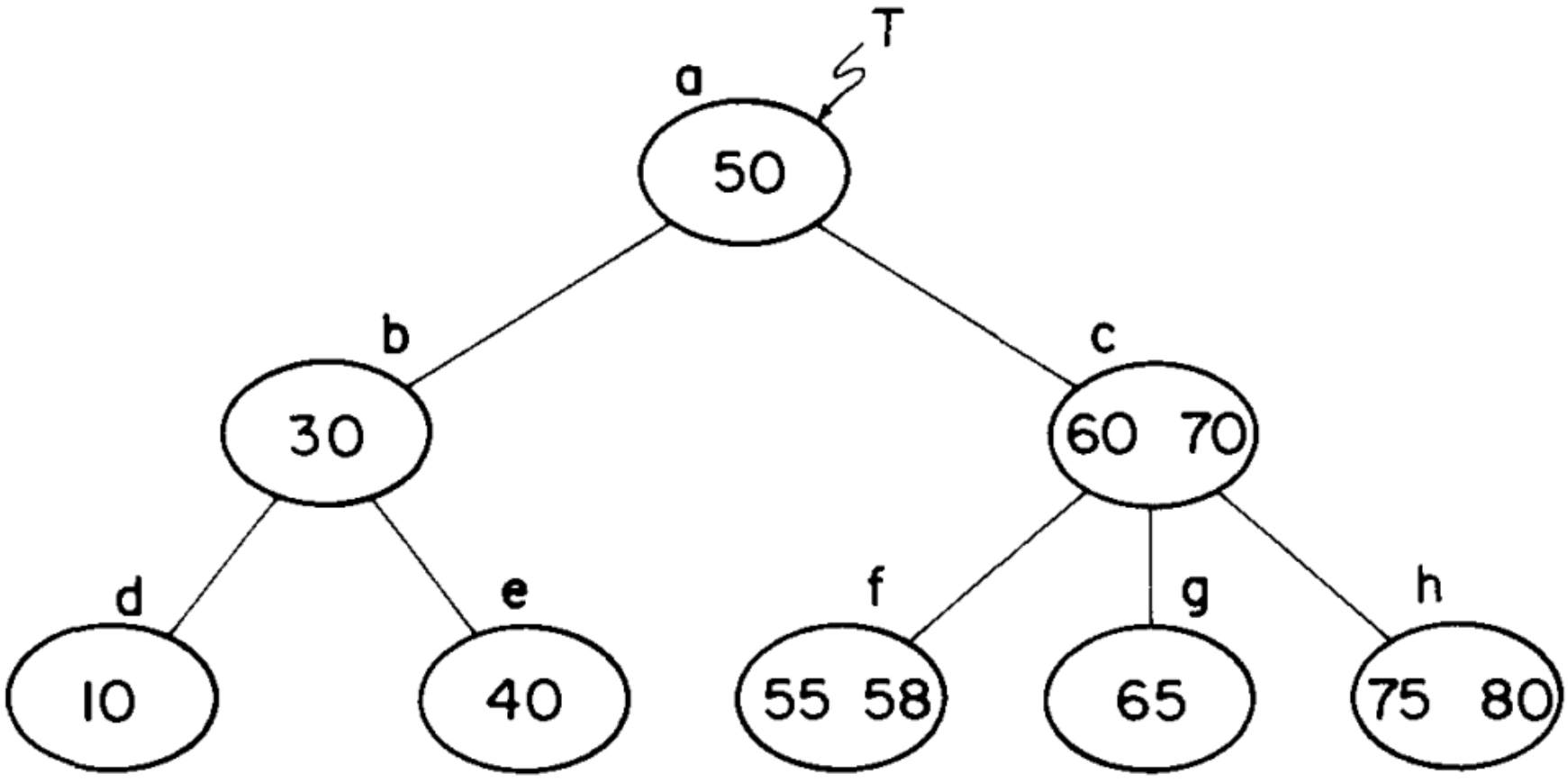
Deletion

- Rebalancing After Deletion - continued

3 Otherwise, if both immediate siblings have only the minimum number of elements, then merge with a sibling sandwiching their separator taken off from their parent.
   - 3.1 Copy the separator to the end of the left node
   - 3.2 Move all elements from the right node to the left node
   - 3.3 Remove the separator from the parent along with its empty right child
   - 3.4 If the parent is the root and now has no elements, then free it and make the merged node the new root
   - 3.5 Otherwise, if the parent has fewer than the required number of elements, then rebalance the parent

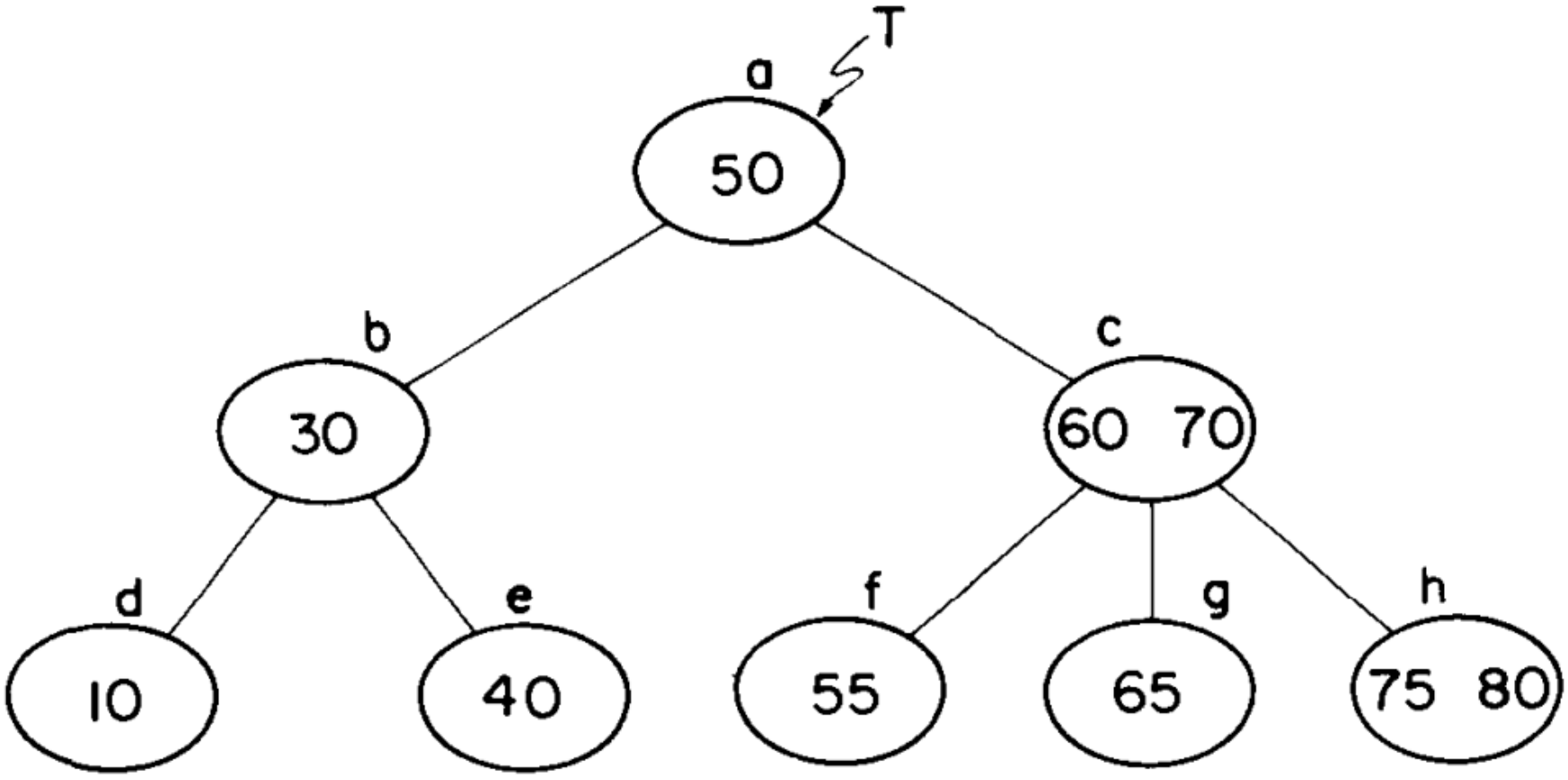
Delete - Example

- Initial (Failure nodes are not shown here)



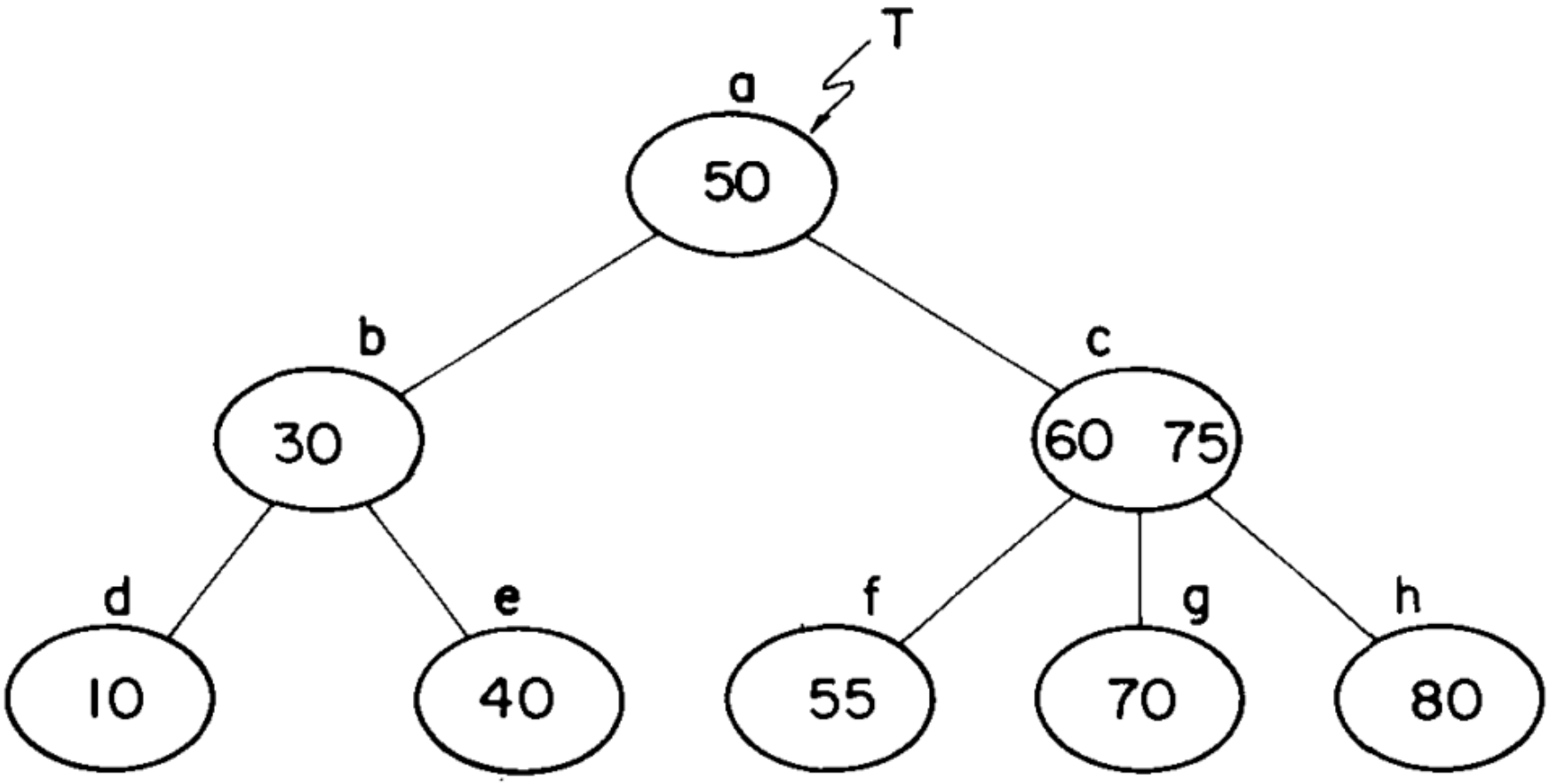
Delete - Example

- Delete 58 (Leaf Node)



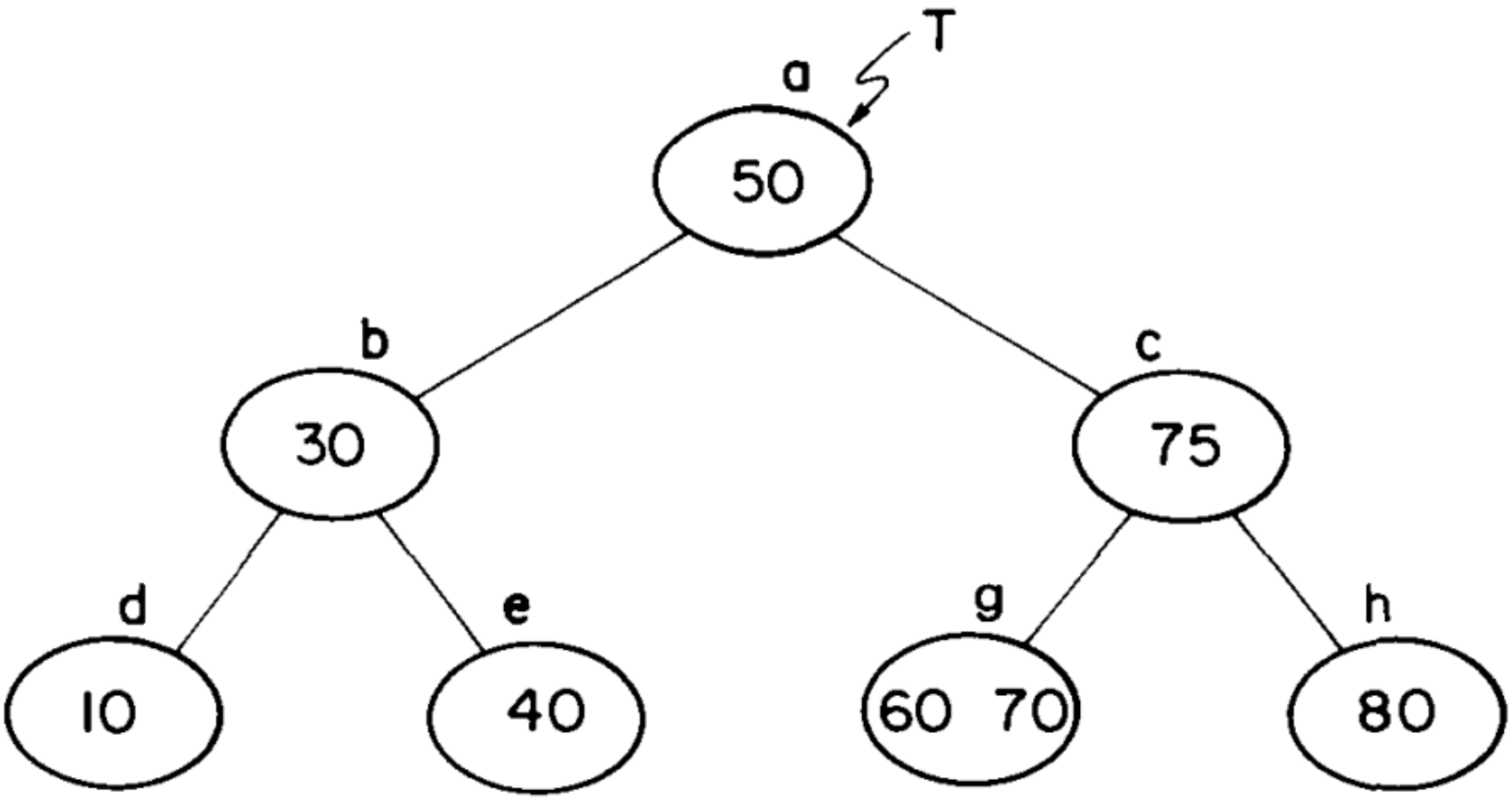
Delete - Example

- Delete 65 (Leaf Node)



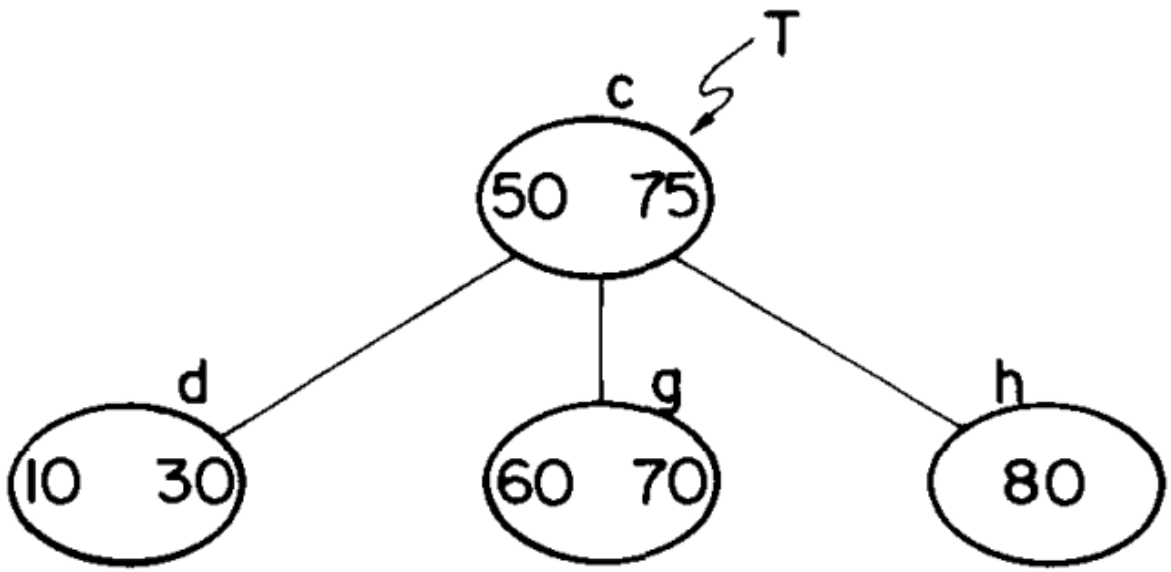
Delete - Example

- Delete 55 (Leaf Node)



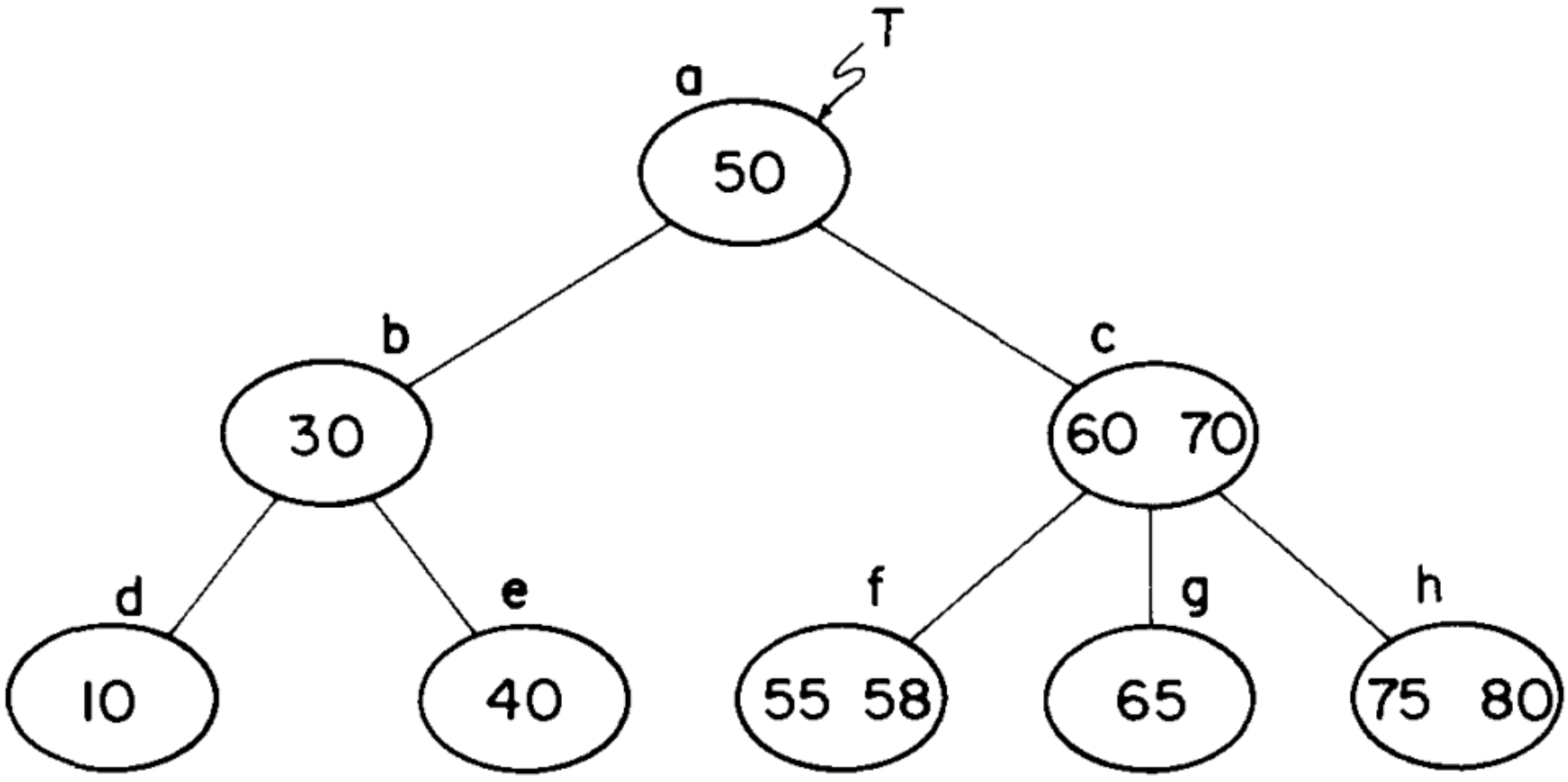
Delete - Example

- Delete 40 (Leaf Node)



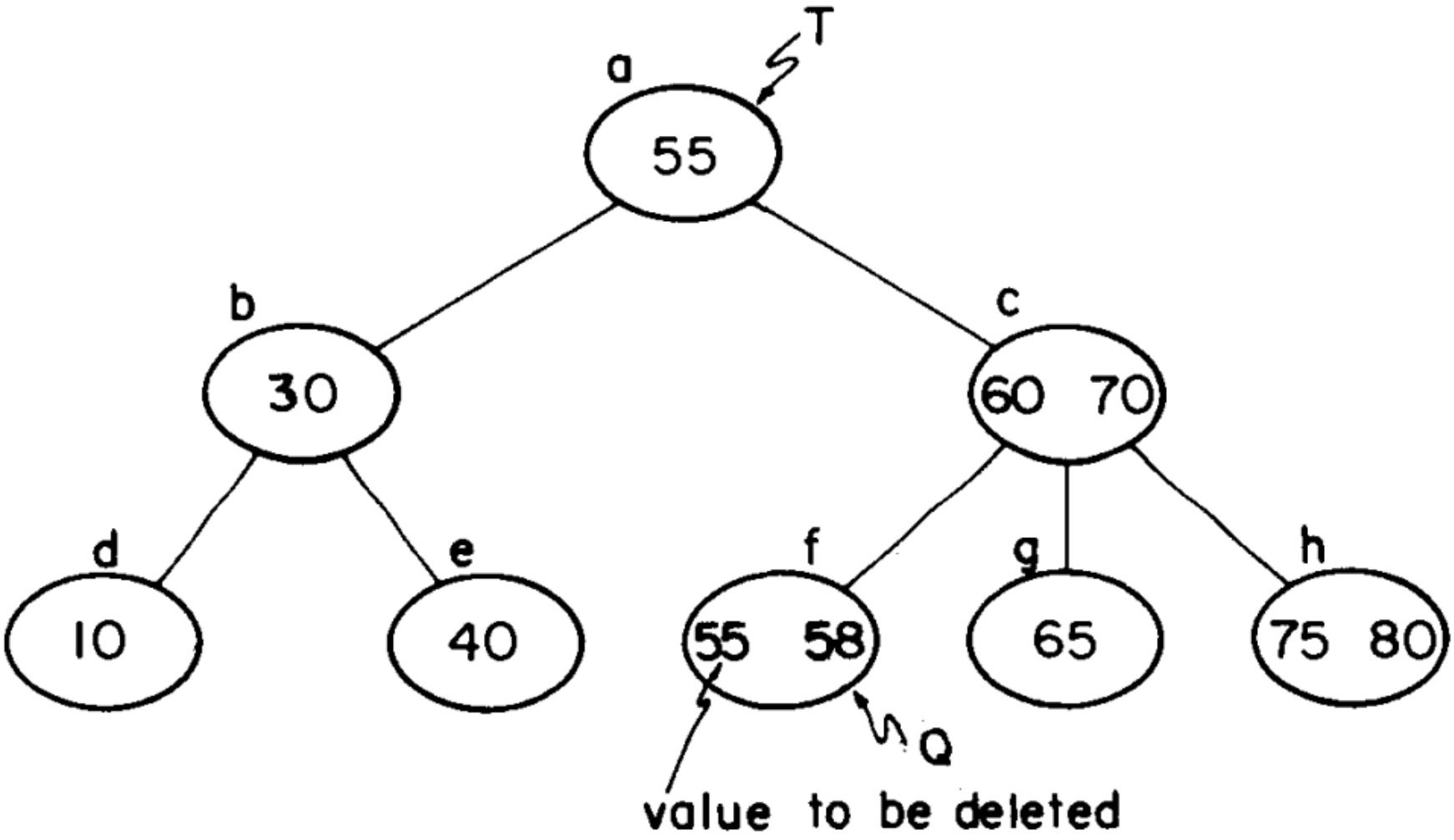
Delete - Example

- Initial



Delete - Example

- Delete 50 (NonLeaf Node)

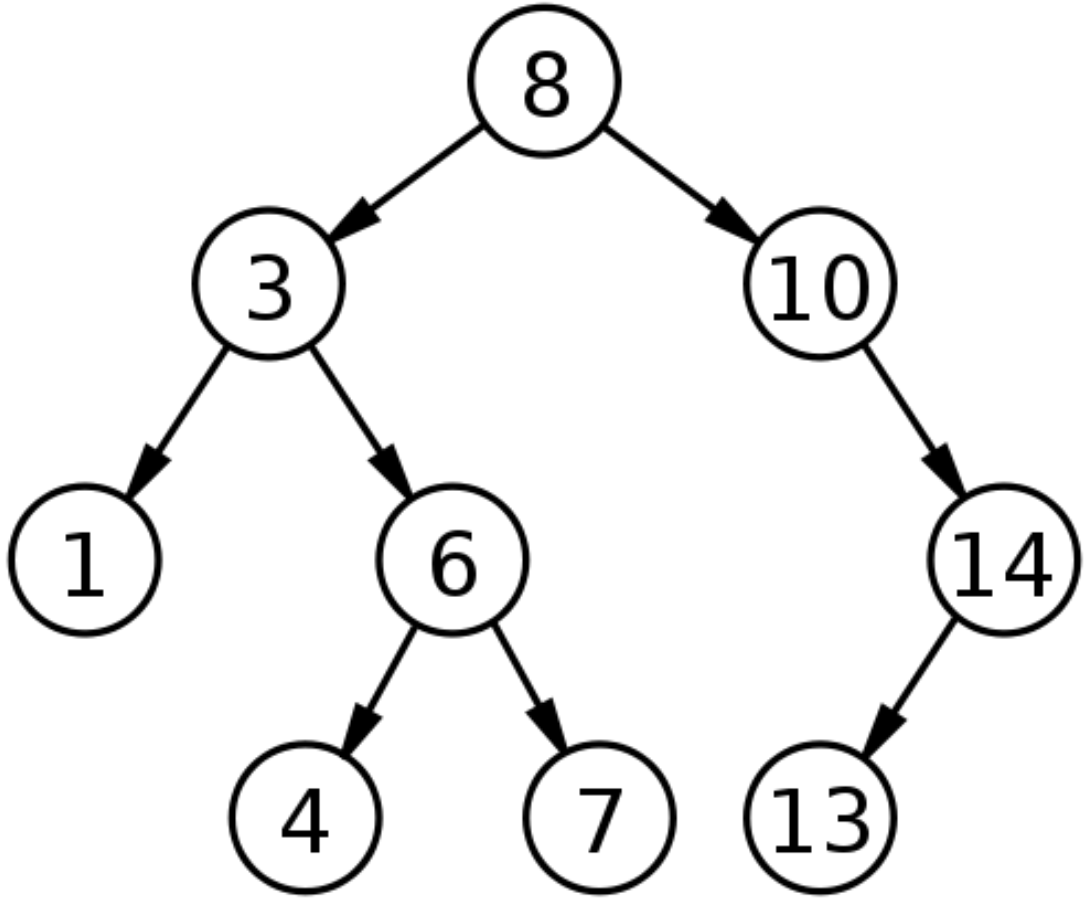


Introduction to Splay Trees

- A splay tree is a binary search tree with the additional property that recently accessed elements are quick to access again.
- The splay tree was invented by Daniel Sleator and Robert Tarjan in 1985
- The amortised complexity of a find, insert or delete operation performed on a splay tree with n elements is O(log n)
- Here a splay operation is performed to balance the tree.
- This is performed on a node called splay node in the tree.
- A splay node is the deepest level node that was searched in the splay tree.
- After performing the splay operation, the splay node will become the root of the splay tree.

Introduction to Splay Trees

- For Find(8) - Splay Node is 8
- For Find(13) - Splay Node is 13
- For Find(5) - Splay Node is 4

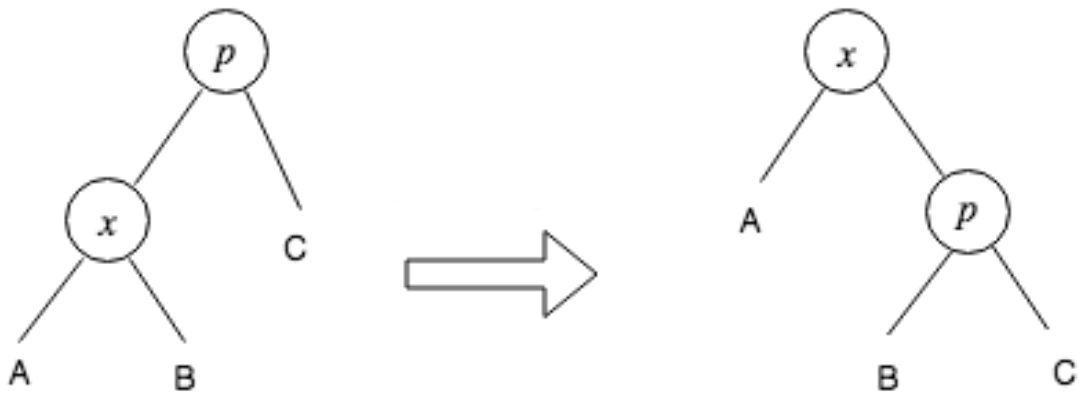


Introduction to Splay Trees

- The splay operation consists of a sequence of splay steps.
- When the splay node is the root of the splay tree, this sequence of steps is empty.
- When the splay node is not the root node, each splay step moves the splay node either one level or two levels up the tree.
- A one level move is made only when the splay node is at level 2 of the splay tree.
- A two level move is made when the level of the splay node is > 2.

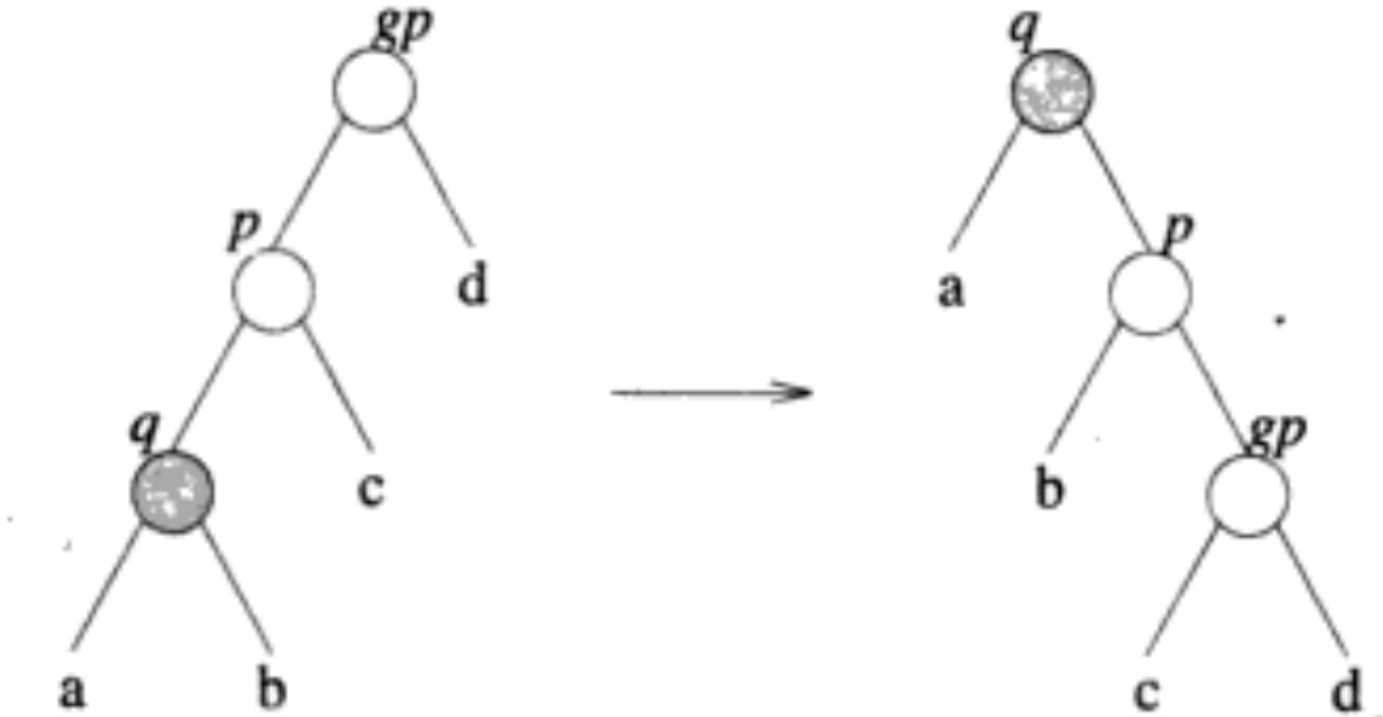
Introduction to Splay Trees

- Two types of one level splay steps
- Type L - done when the splay node is the left child of its parent
- Type R - done when the splay node is the right child of its parent
- Example - Type L - Here x is the splay node, p its parent and A, B, C are the subtrees



Introduction to Splay Trees

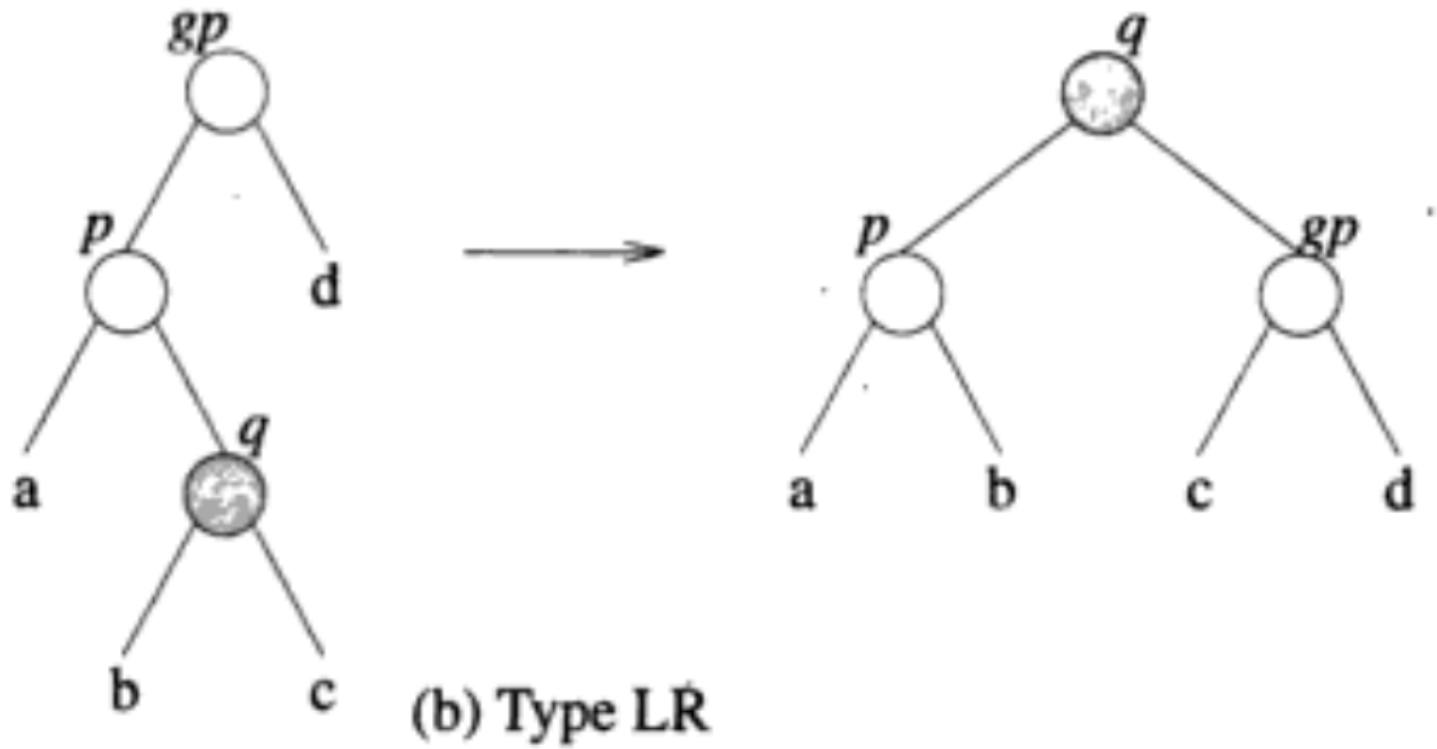
- Four types of two level splay steps
- LL, LR, RL, RR
- Type LL - done when the splay node is the left child of its parent, parent is the left child of its grand parent
- Example - Here q is the splay node, p its parent, gp its grand parent and a, b, c, d the subtrees



(a) Type LL

Introduction to Splay Trees

- Four types of two level splay steps
- LL, LR, RL, RR
- Type LR - done when the splay node is the right child of its parent, parent is the left child of its grand parent
- Example - Here q is the splay node, p its parent, gp its grand parent and a, b, c, d the subtrees



(b) Type LR

Introduction to Suffix Trees

Definition

- The suffix tree for the string S of length n is a tree such that:

1. The tree has exactly n leaves numbered from 1 to n
2. Except for the root, every non leaf node has at least two children
3. Each edge is labelled with a non-empty substring of S
4. No two edges starting out of a node can have string-labels beginning with the same character
5. Each suffix of S corresponds to exactly one path from the tree's root to a leaf

- It was first introduced by Peter Weiner in 1973

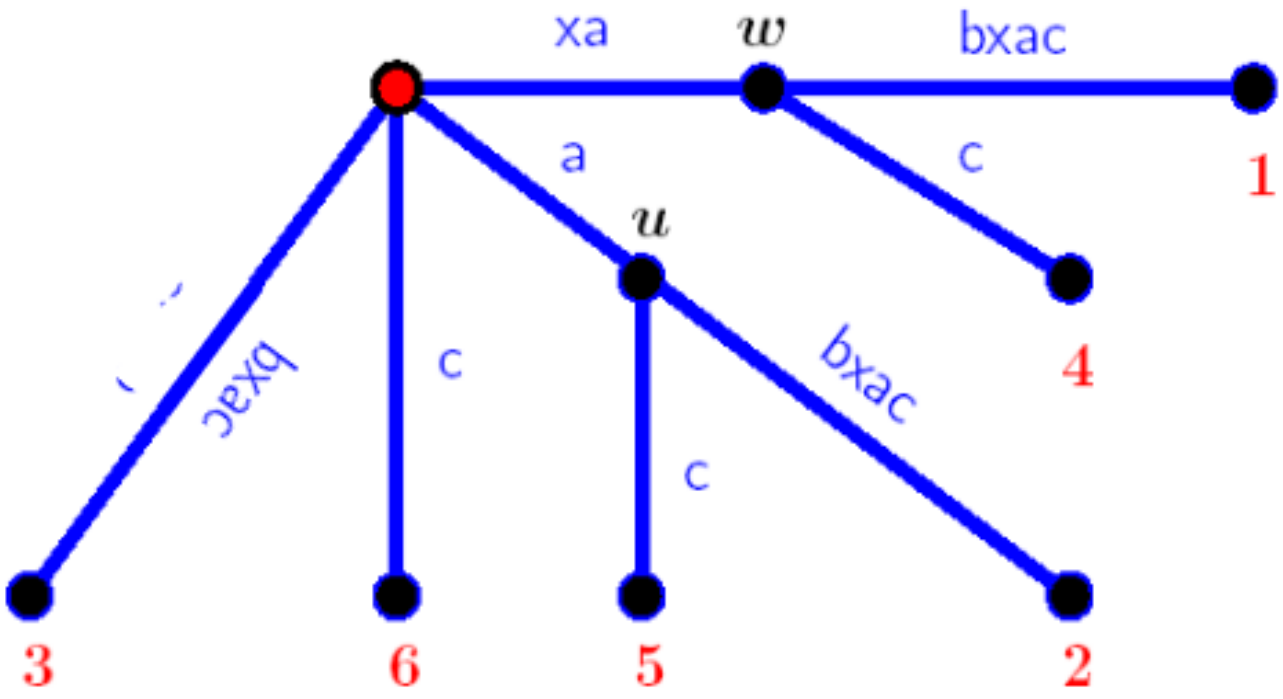
Introduction to Suffix Trees

- A Suffix is a special case of a Substring
- A String S is a suffix of String T if there exists a String P such that $T = PS$

```
Suffixes of xabxac
------------------
xabxac
 abxac
  bxac
   xac
    ac
     c
```

Introduction to Suffix Trees

- Example - Suffix Tree for the string xabxac



Introduction to Suffix Trees

- Applications - used to solve a large number of string problems

1. Bioinformatics - searching for patterns in DNA or protein sequences(represented as strings)
2. Finding the longest repeated substring
3. Finding the longest common substring in two strings
4. Finding the longest palindrome in a string

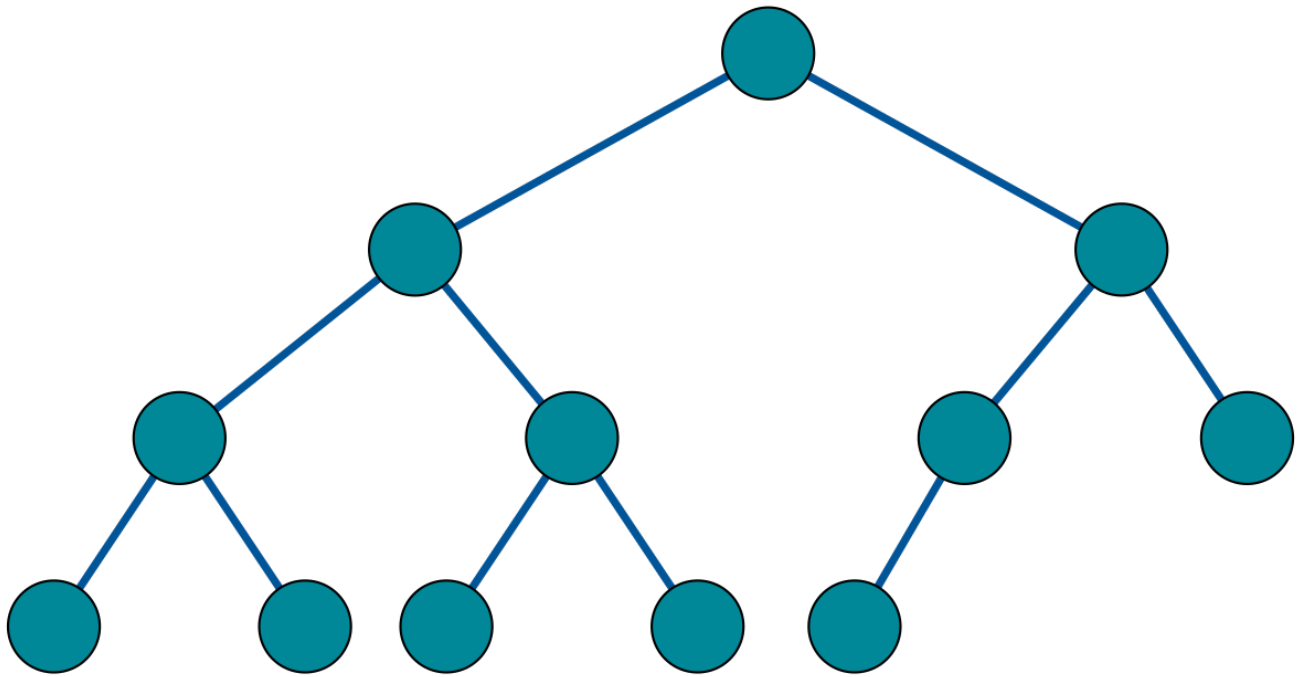
Module 3
Advanced Heap Structures

- Complete Binary Tree
- Heap
- Mergable Heaps
- Binomial Heaps
- Fibonacci Heaps

Complete Binary Tree

Definition

A binary tree is called a complete binary tree if all levels in it are completely filled except the last one. In the last level, the nodes are filled from the left.



Heap

Definition

A Heap is a complete binary tree.

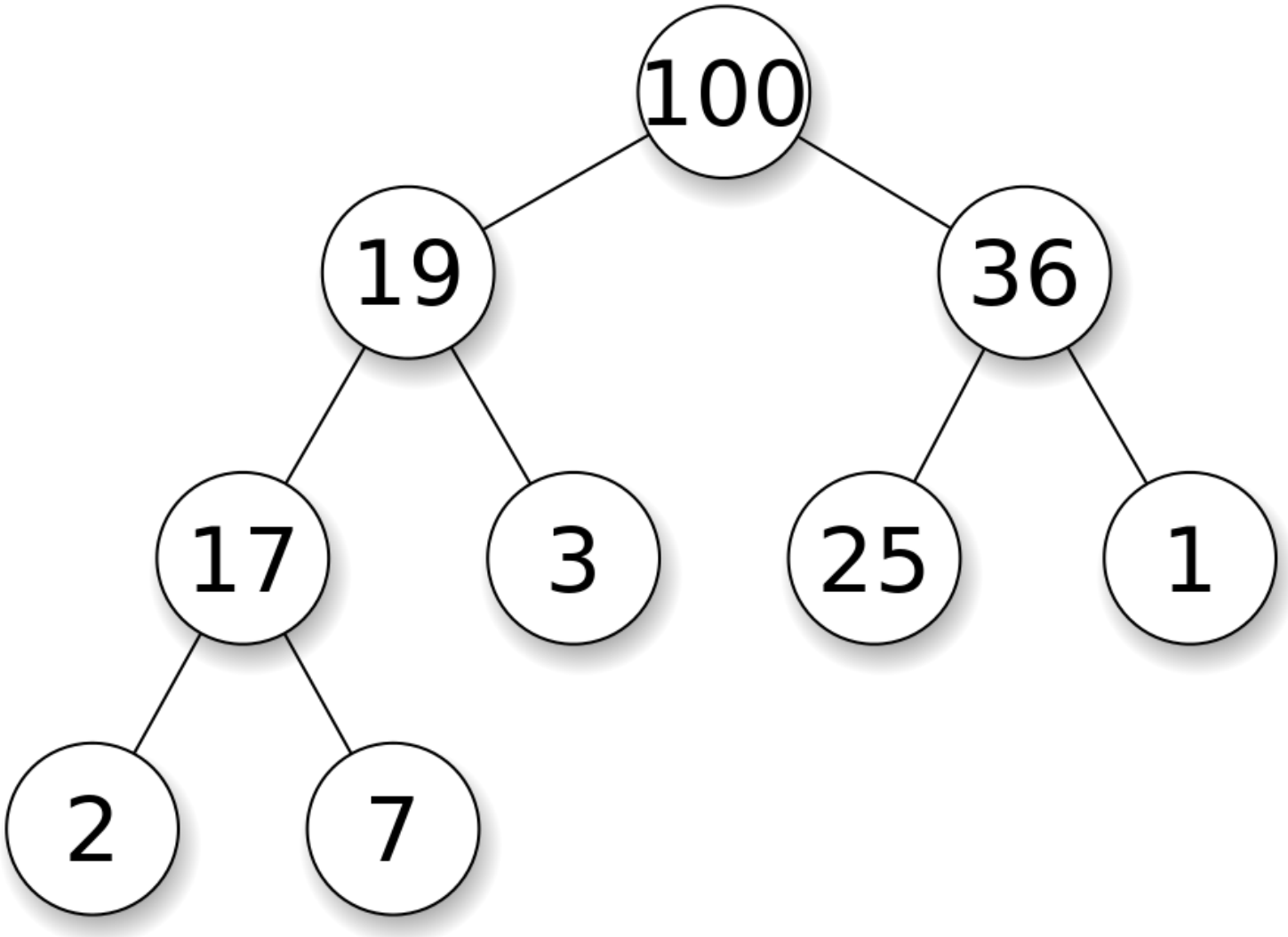
There are two types of heaps.

1. Max Heap
   In a max heap, the value in each node is greater than or equal to those in its children.
2. Min Heap
   In a min heap, the value in each node is less than or equal to those in its children.

- The root node will contain the largest(smallest) value in a max heap(min heap).
- The heap data structure was invented by J W J Williams in 1964.

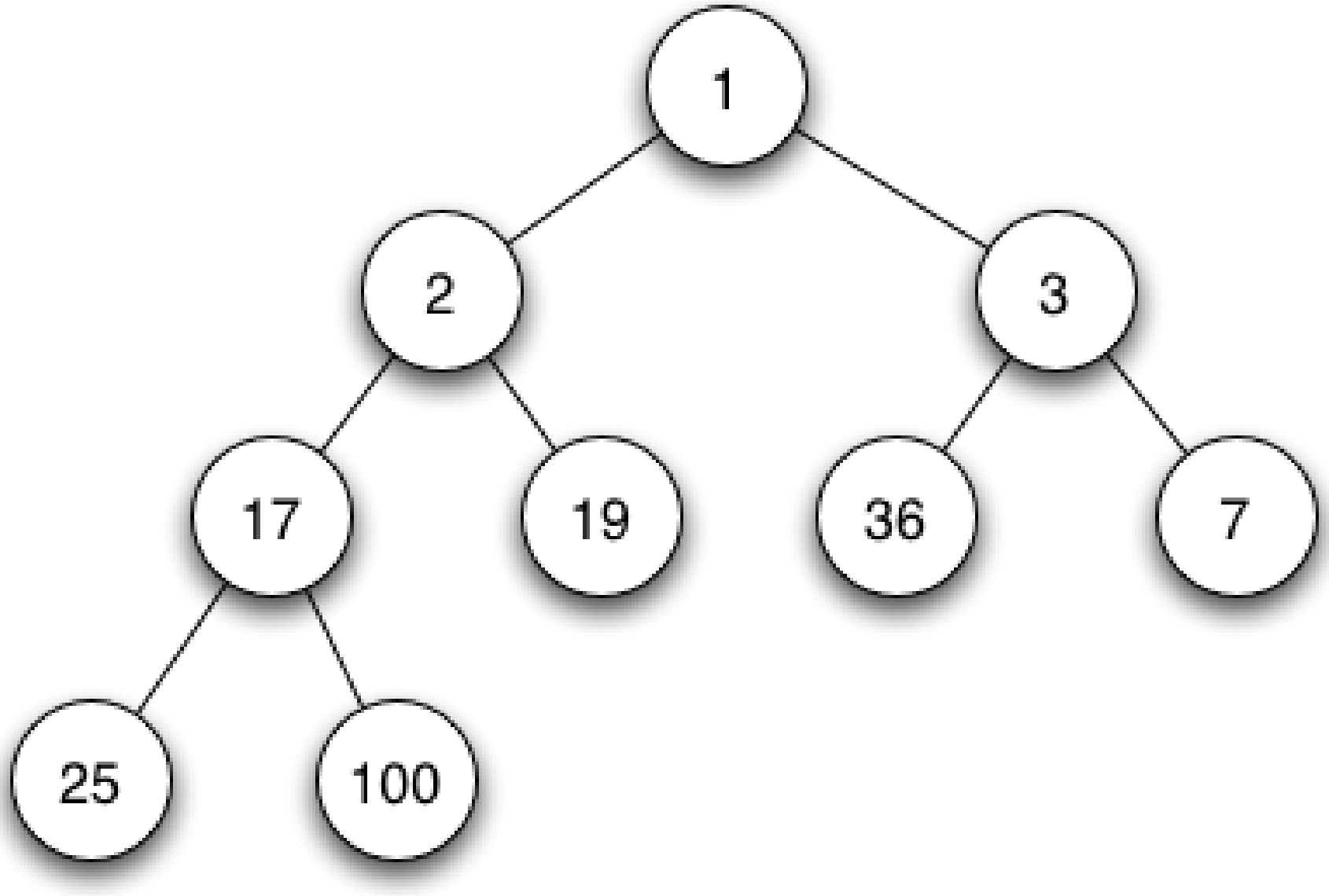
Heap

- Max Heap - Example



Heap

- Min Heap - Example



Mergable Heap

Definition

A mergable heap is a heap that satisfies the following operations.

1. Make-Heap(), creates an empty heap
2. Insert(H,x), inserts an element x into heap H
3. Max(H) or Min(H), return the maximum(minimum) element, or nil if heap is empty
4. Extract-Max(H) or Extract-Min(H), delete the maximum(minimum) element, or nil if heap is empty
5. Merge(H1,H2), combine the elements of heaps H1 and H2 into a single heap

Operations on Mergable Heaps

- Create Empty Heap

Algorithm

1. Create a heap object H, where H.n (number of nodes in H) = 0 and H.min (pointer to the root of a tree containing the minimum key) = NIL

Operations on Mergable Heaps

- Insert Operation

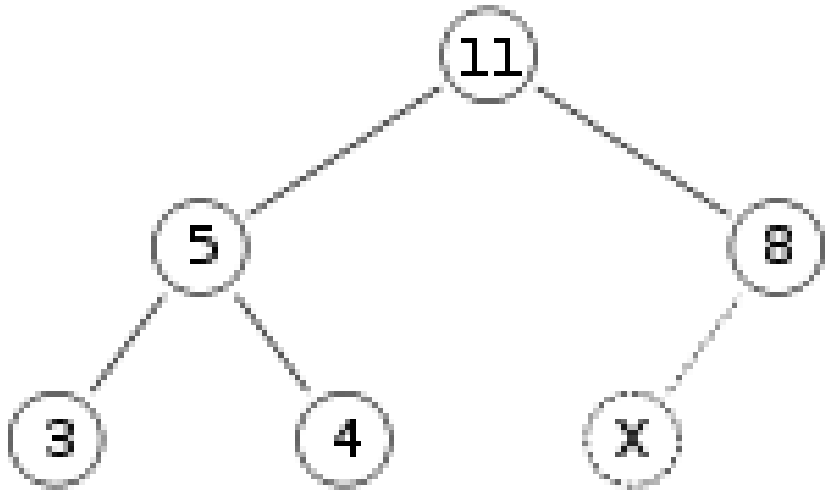
Algorithm

1. Add the element to the bottom level of the heap at the leftmost open space.
2. Compare the added element with its parent; if they are in the correct order, stop.
3. If not, swap the element with its parent.
4. Perform compare and swap operations in higher levels till heap property is restored.

- Steps which restore the heap property by comparing and possibly swapping a node with its parent, are called the up-heap operation

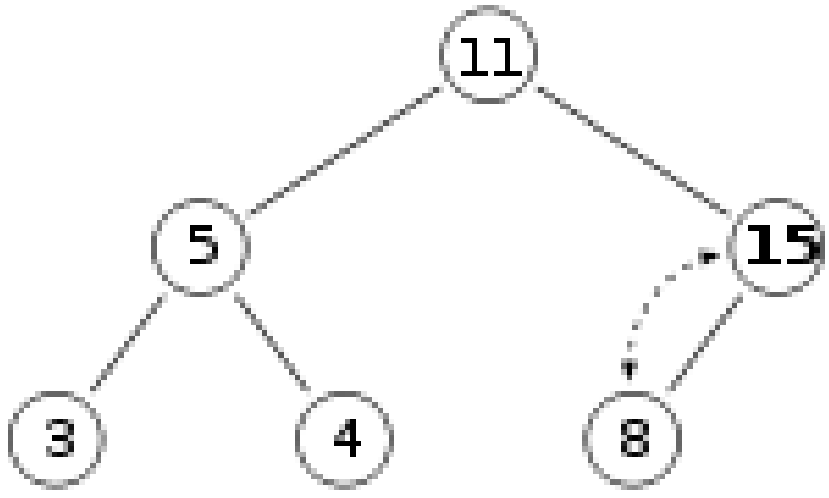
Operations on Mergable Heaps

- Insert 15 into a Max Heap



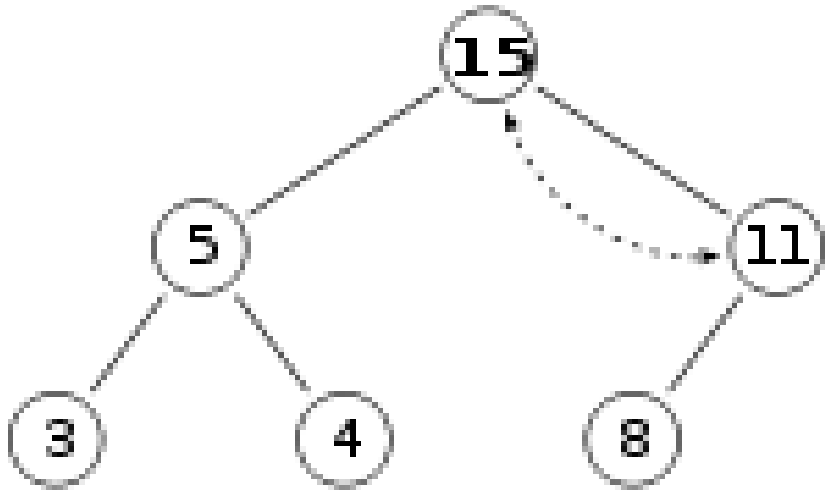
Operations on Mergable Heaps

- Insert 15 into a Max Heap



Operations on Mergable Heaps

- Insert 15 into a Max Heap



Operations on Mergable Heaps

- Max or Min Operation

Algorithm

1. If heap is not empty return the element in the root node, else return nil.

Operations on Mergable Heaps

- Extract Operation

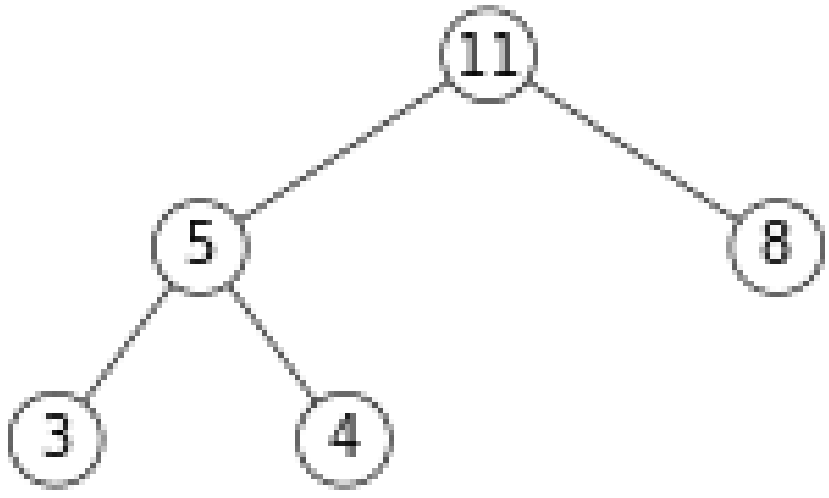
Algorithm

1. Replace the root of the heap with the last element on the last level.
2. Compare the new root with its children; if they are in the correct order, stop.
3. If not, swap the element with one of its children (Swap with its smaller child in a min-heap and its larger child in a max-heap.)
4. Perform compare and swap operations in lower levels till heap property is restored.

- Steps which restore the heap property by comparing and possibly swapping a node with one of its children, are called the down-heap operation

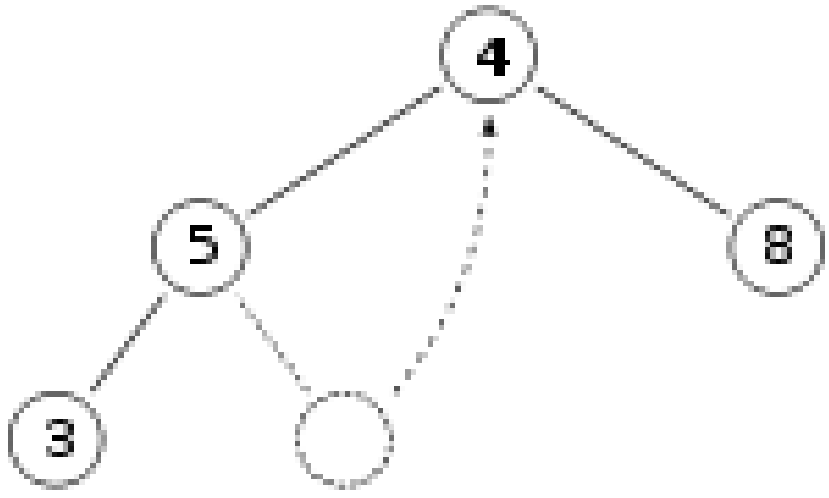
Operations on Mergable Heaps

- Extract 11 from a Max Heap



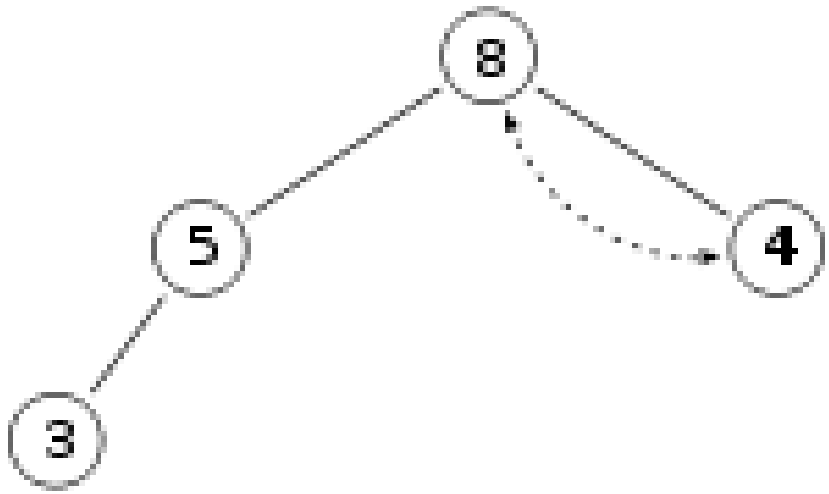
Operations on Mergable Heaps

- Extract 11 from a Max Heap



Operations on Mergable Heaps

- Extract 11 from a Max Heap



Operations on Mergable Heaps

- Merge Operation on Max Heaps

Algorithm

```
Merge(H1,H2)
1    x = Extract-Max(H2)
2    while x != Nil
3         Insert(H1,x)
4         x = Extract-Max(H2)
```

Operations on Mergable Heaps

- Merge Operation on Min Heaps

Algorithm

```
Merge(H1,H2)
1    x = Extract-Min(H2)
2    while x != Nil
3         Insert(H1,x)
4         x = Extract-Min(H2)
```

Binomial Heaps

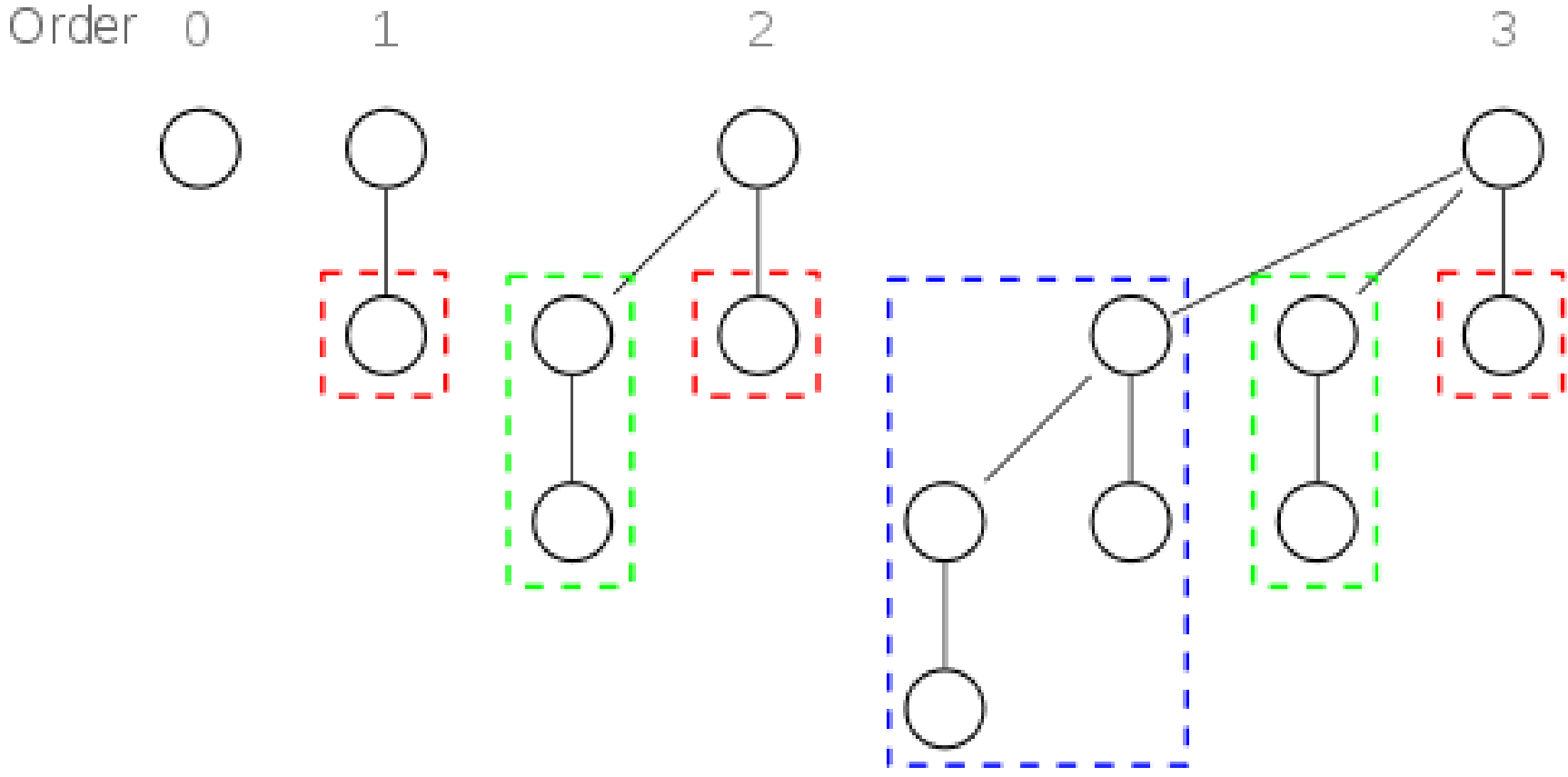
- A Binomial Tree of order k is a tree in which

1. The height of the tree is k
2. There are 2^k nodes
3. There are $\binom{k}{d}$ nodes at depth d
4. The root node has children which are roots of binomial trees of orders k-1, k-2, ..., 2, 1, 0 (from left to right).

- The height of the tree is the length of the longest path from the root node to any leaf node in the tree.
- The depth of a node is the length of the path from that node to the root node.
- $\binom{k}{d} = \frac{k!}{d!(k-d)!}$

Binomial Heaps

- Binomial Trees of Order 0 to 3



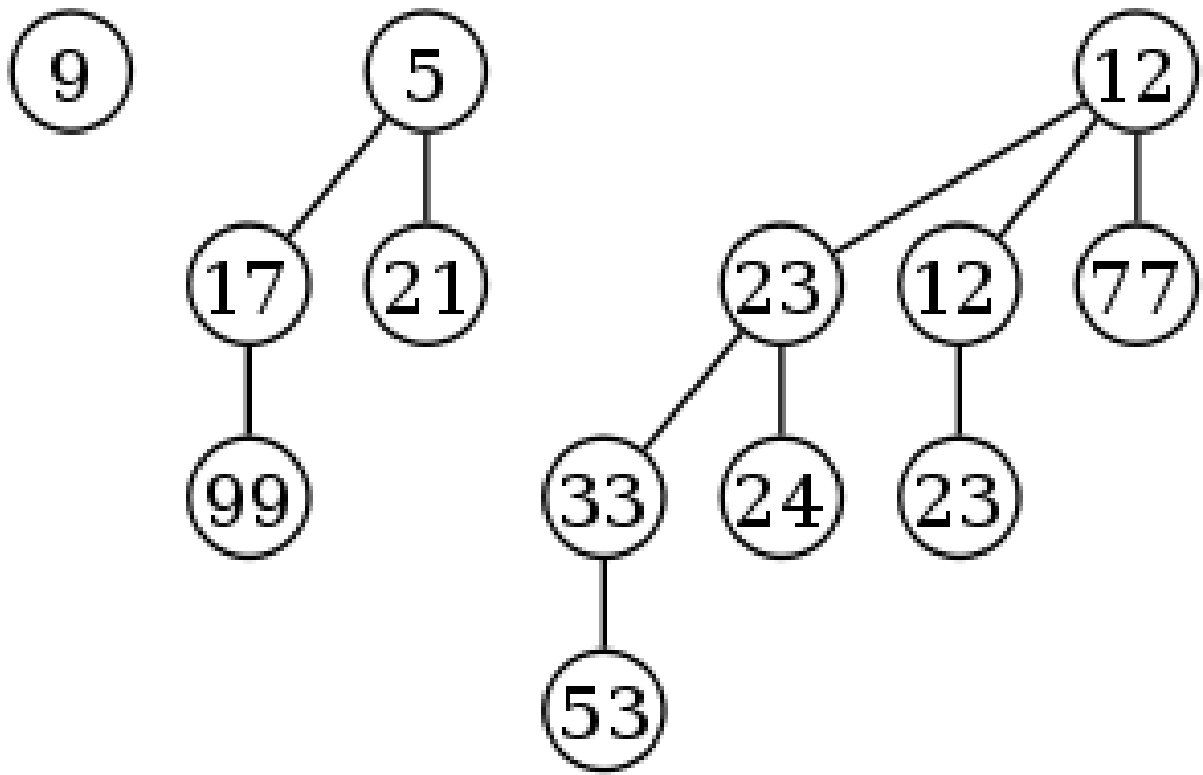
Binomial Heaps

- In the above Binomial Tree of order 3

1. The height of the tree is k=3
2. There are $2^3 = 8$ nodes
3. There are $\binom{3}{2} = \frac{3!}{2!(3-2)!} = \frac{3!}{2!1!} = 3$ nodes at depth 2
4. The root node has children which are roots of binomial trees of orders 2, 1, 0 (from left to right).

Binomial Heaps

- A Binomial Heap is a set of binomial trees where each tree is a min heap.There can be at most one binomial tree of any order.
- It was invented by the French computer scientist Jean Vuillemin in 1978.
- Example - A Binomial Heap containing 3 binomial trees of order 0, 2 and 3



Binomial Heap Operations and Analysis

The following operations are performed on a binomial heap H.

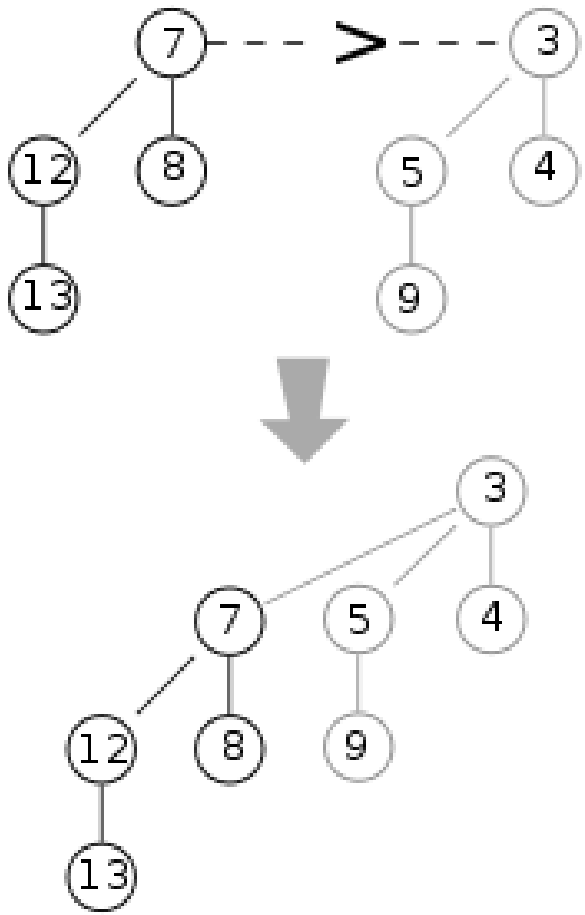
1. Make-Heap(), creates an empty heap
2. Insert(H,x), inserts an element x into H
3. Min(H), return the minimum element, or nil if H is empty
4. Extract-Min(H), delete the minimum element or nil if H is empty
5. Merge(H1,H2), combine the elements of binomial heaps H1 and H2 into a single binomial heap
6. Decrease-Key(H,x,k), assigns to element x within H a new key value k, which is less than its current key value
7. Delete(H,x), deletes element x from H

Binomial Heap Operations and Analysis

- Merge Operation of two binomial trees of the same order

Algorithm

1. Compare the keys of root nodes of the binomial trees
2. The root node with the larger key is made into a child of the root node with the smaller key



Binomial Heap Operations and Analysis

- Merge Operation of two binomial heaps

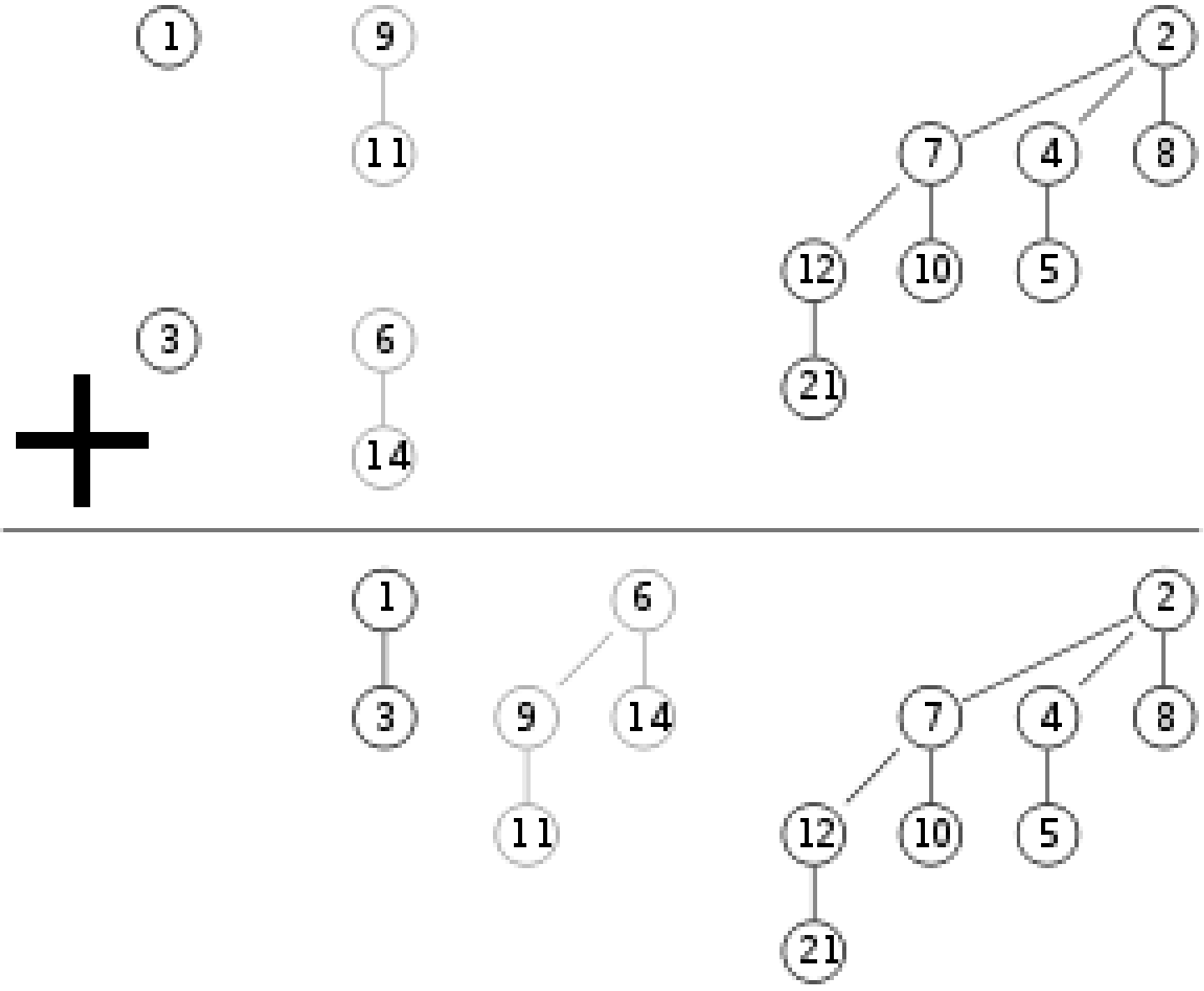
Algorithm

1. Merge two binomial trees of the same order one by one.
2. If the resulting merged tree has the same order as one binomial tree in one of the two heaps, then those two are merged again.
3. This process is continued till there are no two binomial trees of the same order.

- Since each binomial tree has order $log_2 n$, the running time is O(log n)

Binomial Heap Operations and Analysis

- Merge Operation of two binomial heaps



Binomial Heap Operations and Analysis

- Create Empty Heap

Algorithm

1. Create a heap object H, where H.n (number of nodes in H) = 0 and H.min (pointer to the root of a tree containing the minimum key) = NIL

- This operation will take time O(1), a constant time

Binomial Heap Operations and Analysis

- Insert Operation

Algorithm

1. Create a heap containing only the element to be inserted.
2. Merge this heap with the original heap.

- Because of the merge, the insert operation will take time O(log n)

Binomial Heap Operations and Analysis

- Find Minimum Operation

Algorithm

1. find the minimum among the roots of the binomial trees in the binomial heap.

- This can be done in O(log n) time as there are O(log n) tree roots to examine.

Binomial Heap Operations and Analysis

- Extract Minimum Operation

Algorithm

1. Find this element and remove it from the root of its binomial tree.
2. Transform its subtrees into a separate binomial heap by reordering them from smallest to largest order.
3. Merge this heap with the remaining binomial trees in the original heap.

- Since each root has at most $log_2 n$ children, creating this new heap takes time O(log n)
- Merging heaps takes time O(log n)
- So entire delete minimum operation takes time O(log n)

Binomial Heap Operations and Analysis

- Decrease Key Operation

Algorithm

1. After decreasing the key of an element, it may become smaller than the key of its parent, violating the minimum-heap property.
2. If this is the case, exchange the element with its parent, and possibly also with its grandparent, and so on, until the minimum-heap property is no longer violated.

- Each binomial tree has height at most $log_2 n$, so this takes O(log n) time.

Binomial Heap Operations and Analysis

- Delete Operation

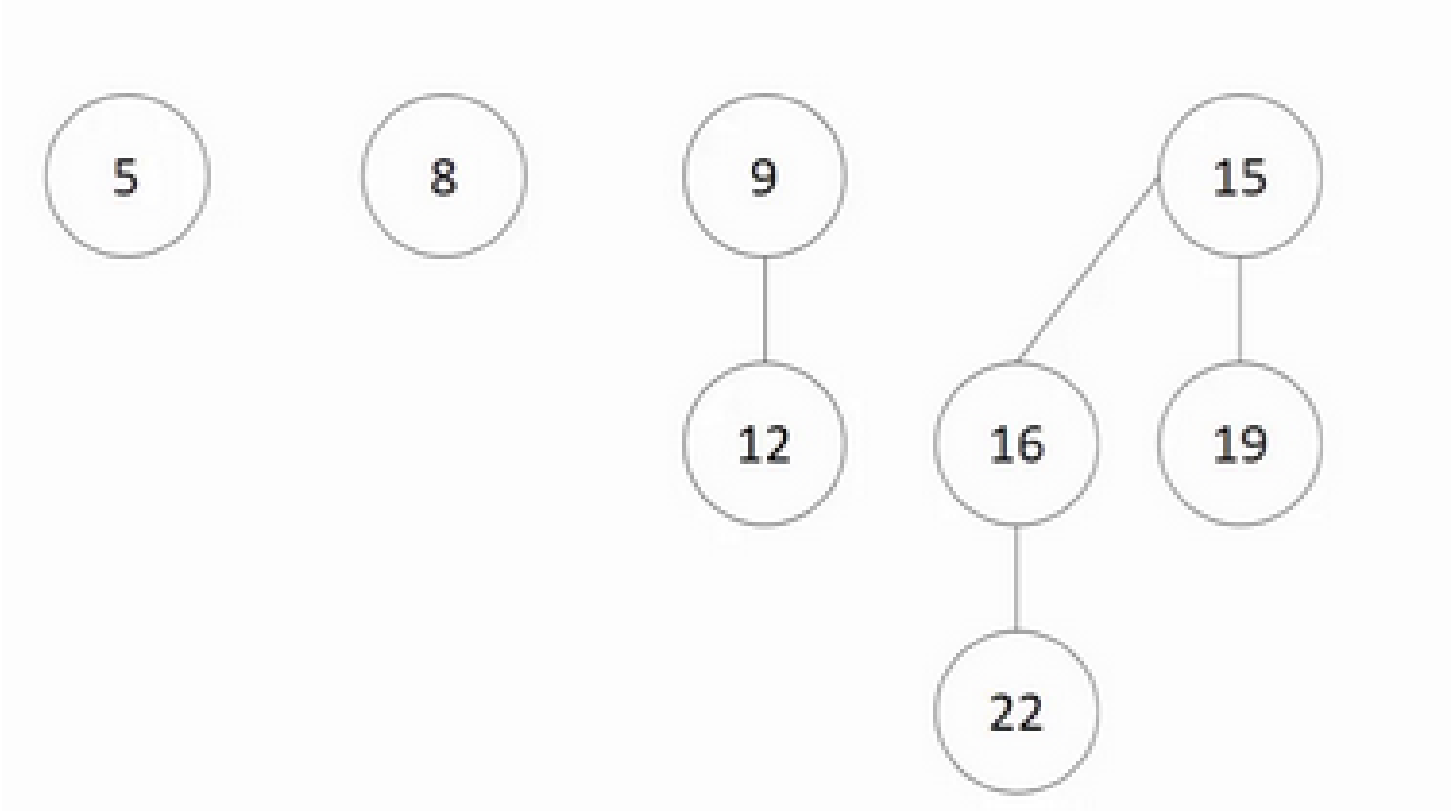
Algorithm

1. To delete an element from the heap, decrease its key to negative infinity.
2. Then extract the minimum element from the heap.

- This involves Decrease Key and Extract Minimum operations, both of which takes O(log n) time
- Hence the total time for delete operation is O(log n)

Fibonacci Heaps

- A Fibonacci Heap is a set of trees where each tree is a min heap.
- It is called so, since it running time analysis involves a pattern of fibonacci numbers
- It was developed by Michael L. Fredman and Robert E. Tarjan in 1984.
- Example - A Fibonacci Heap containing 4 min heaps

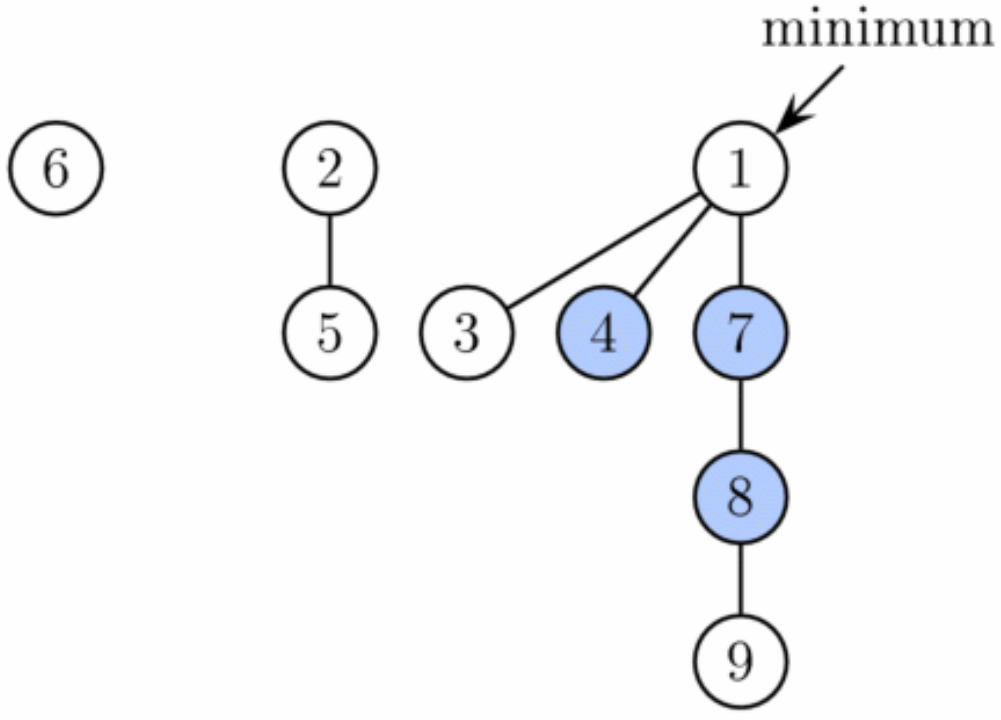


Fibonacci Heaps

- We mark some nodes in fibonacci heaps

Rules for Marking

1. All root nodes are unmarked
2. Whenever a child is deleted(cut) for the first time, the parent node is marked
3. When a second child is cut, the parent node itself is cut from its parent and it becomes the root of a new tree



Fibonacci Heap Operations and Analysis

The following operations are performed on a fibonacci heap H.

1. Make-Heap(), creates an empty heap
2. Insert(H,x), inserts an element x into H
3. Min(H), return the minimum element, or nil if H is empty
4. Extract-Min(H), delete the minimum element or nil if H is empty
5. Merge(H1,H2), combine the elements of fibonacci heaps H1 and H2 into a single fibonacci heap
6. Decrease-Key(H,x,k), assigns to element x within H a new key value k, which is less than its current key value
7. Delete(H,x), deletes element x from H

- To implement these operations, the roots of all trees in the fibonacci heap are linked using a circular doubly linked list

Fibonacci Heap Operations and Analysis

- The performance of these operations are measured using potential method in amortised analysis
- The potential function of the fibonacci heap is defined as $\Phi(H) = t(H) + 2m(H)$
- $t(H)$ is the number of trees in the heap
- $m(H)$ is the number of marked nodes in the heap

Fibonacci Heap Operations and Analysis

- Create Empty Heap

Algorithm

1. Create a heap object H, where H.n (number of nodes in H) = 0 and H.min (pointer to the root of a tree containing the minimum key) = NIL

- This operation will take time O(1), a constant time
- amortised cost = actual cost + change in potential due to the operation
- Since the heap is empty now, t(H) and m(H) are 0 now, hence there is no change in potential due to this operation
- amortised cost = O(1)

Fibonacci Heap Operations and Analysis

- Merge Operation

Algorithm

1. Concatenate the lists of tree roots of two fibonacci heaps

- This operation will take time O(1), a constant time
- During merge, t(H) and m(H) are not changed, hence there is no change in potential due to this operation
- amortised cost = O(1)

Fibonacci Heap Operations and Analysis

- Insert Operation

Algorithm

1. Create a heap containing only the element to be inserted.
2. Merge this heap with the original heap.

- Because of the merge, the insert operation will take time $O(1)$
- Since one tree(heap) is added, change in potential is 1
- amortised cost $= O(1) + 1 = O(1)$

Fibonacci Heap Operations and Analysis

- Find Minimum Operation

Algorithm

1. find the minimum among the roots of the trees in the fibonacci heap.

- Since we are maintaining a pointer to the root of the tree containing the minimum key, this operation is trivial and will take O(1) time
- As there is no change in potential due to this operation, amortised cost = O(1)

Fibonacci Heap Operations and Analysis

- Extract Minimum Operation

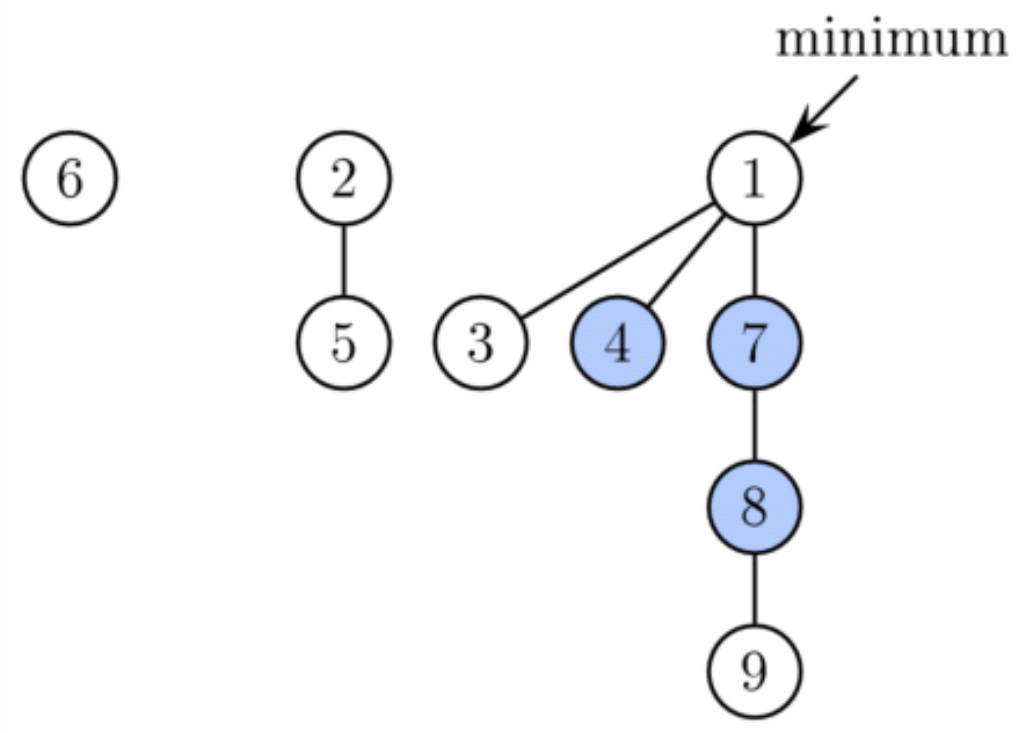
Algorithm

1. Find this element and remove it from the root of its tree.
2. Its children will become roots of new trees.
3. Decrease the number of roots by successively linking together roots of the same degree.
4. When two roots have the same degree, we make one of them a child of the other so that the one with the smaller key remains the root.
5. This is repeated until every root has a different degree.
6. Update the pointer to the root with the minimum key.

- Let N be the number of roots and D be the maximum degree of a node in the root list in the fibonacci heap, D = log n for a heap of size n
- Actual cost of this operation is O(N+D)
- Amortised cost is O(D) = O(log n)

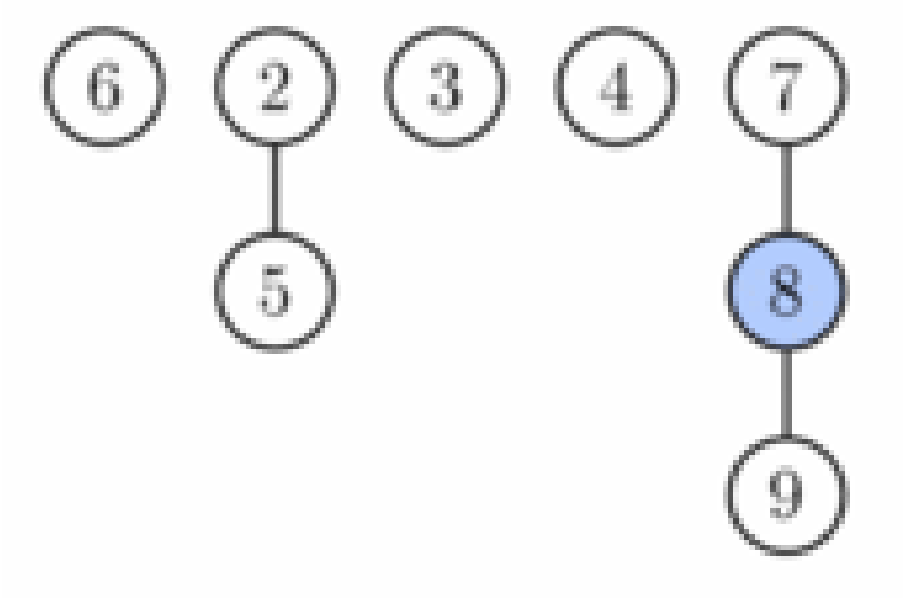
Fibonacci Heap Operations and Analysis

- Extract Minimum Operation
- Initial Fibonacci Heap



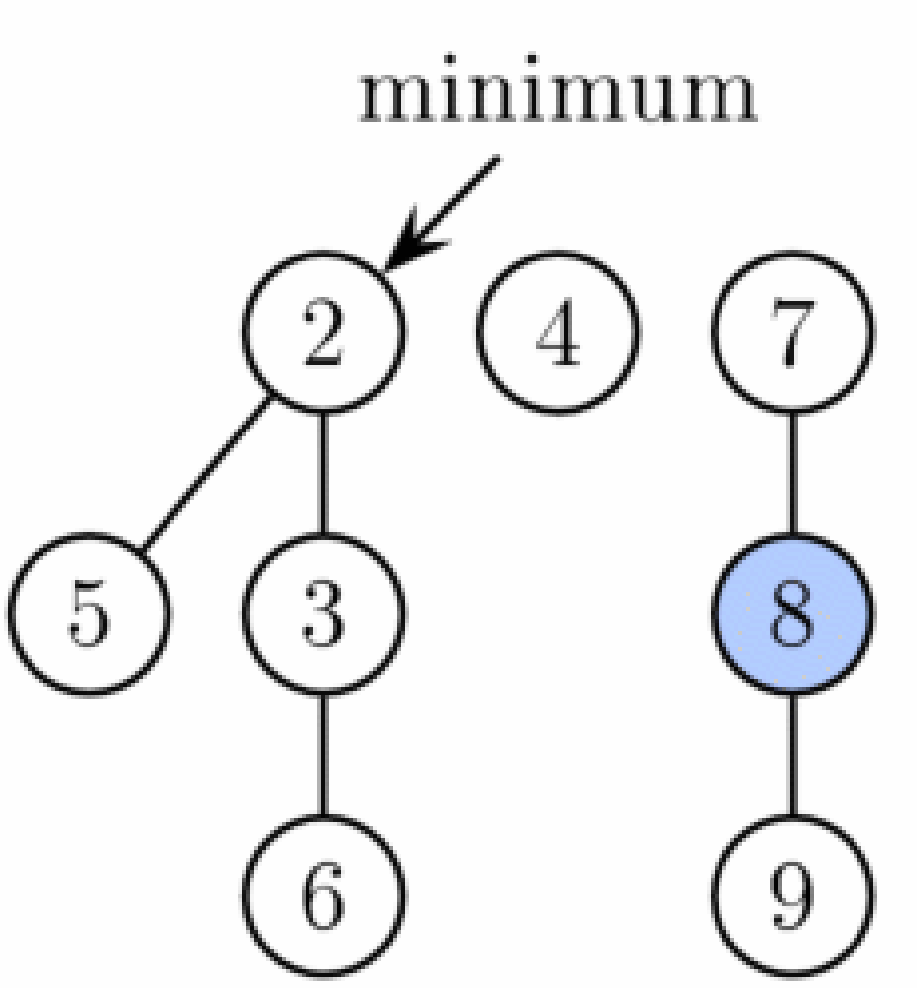
Fibonacci Heap Operations and Analysis

- Extract Minimum Operation
- Delete node 1 and make its subtrees as separate trees



Fibonacci Heap Operations and Analysis

- Extract Minimum Operation
- After Completion of Operation



Fibonacci Heap Operations and Analysis

- Decrease Key Operation

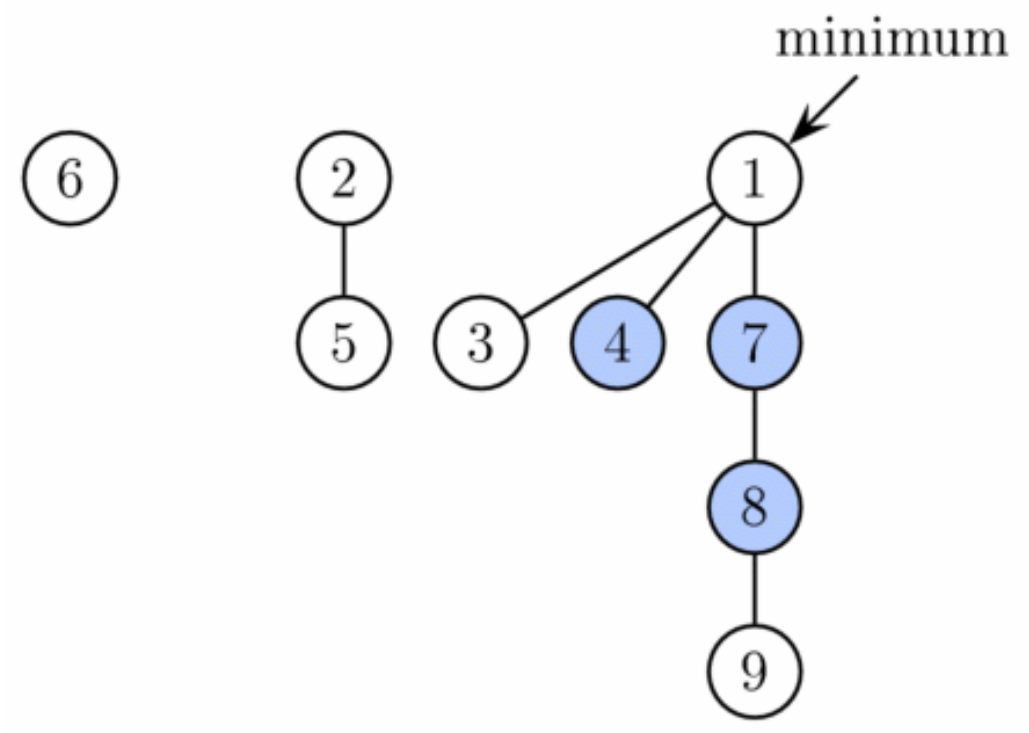
Algorithm

1. Decrease the key and if the heap property becomes violated, the node is cut from its parent.
2. If the parent is not a root, it is marked.
3. If it has been marked already, it is cut as well and its parent is marked.
4. We continue upwards until we reach either the root or an unmarked node.
5. Now we set the minimum pointer to the decreased value if it is the new minimum.

- Since there is no change in the number of nodes, it will take O(1) time.
- As the potential also changes by a constant value, amortised cost = $O(1)$

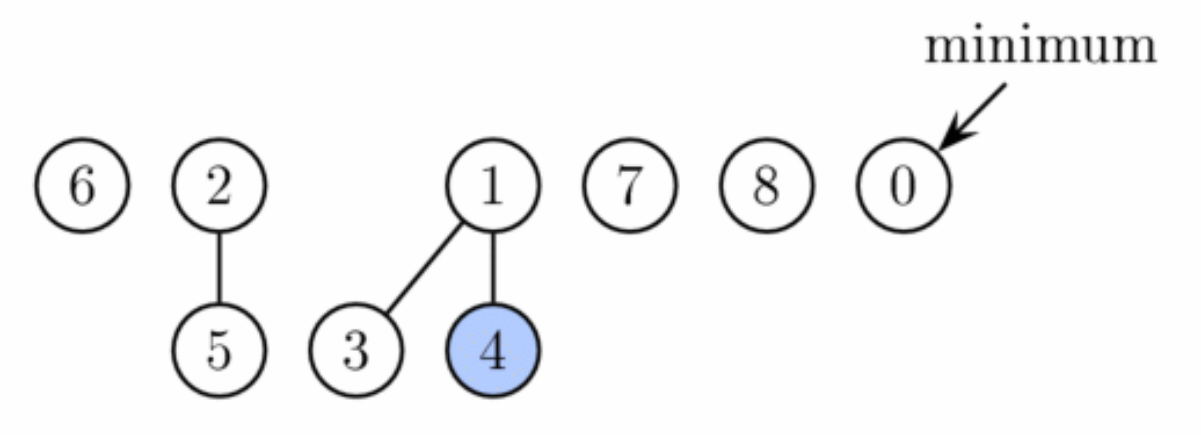
Fibonacci Heap Operations and Analysis

- Decrease Key Operation
- Initial Fibonacci Heap



Fibonacci Heap Operations and Analysis

- Decrease Key Operation
- After Decreasing Key from 9 to 0



Fibonacci Heap Operations and Analysis

- Delete Operation

Algorithm

1. To delete an element from the heap, decrease its key to negative infinity.
2. Then extract the minimum element from the heap.

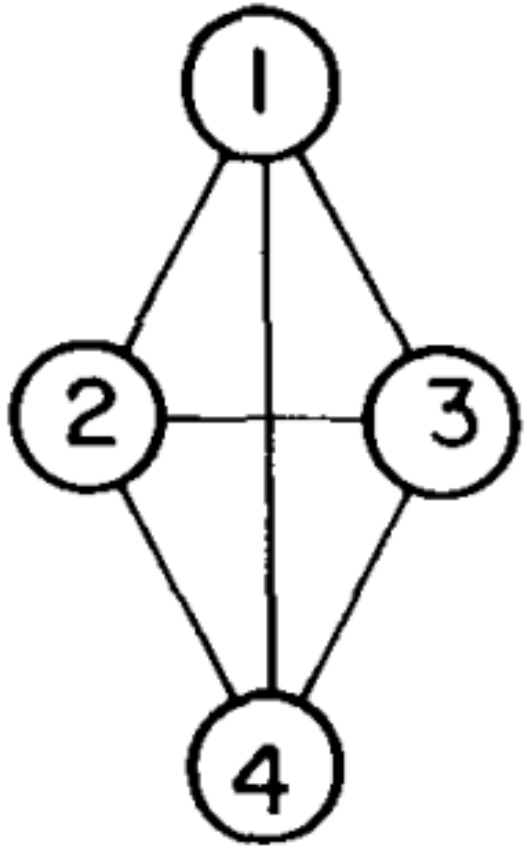
- This involves Decrease Key and Extract Minimum operations, which takes O(1) and O(log n) amortised times respectively
- Hence the total amortised time for delete operation is O(log n)

Module 4
Advanced Graph Structures

- Representation of Graph
- Graph Traversals
- Topological Sorting
- Connected Components
- Minimum Cost Spanning Tree Algorithms
- Shortest Path Finding Algorithm

Graph

- A graph is a set of vertices and a set of edges that connects these vertices



G_1

Representation of A Graph

- Using Adjacency Matrix
- It is a 2 dimensional matrix A in which an element
  $A_{ij} = 1$, if there is an edge between vertex i and vertex j and
  $A_{ij} = 0$, if there is no edge between vertex i and vertex j
- Adjacency Matrix for Graph G_1

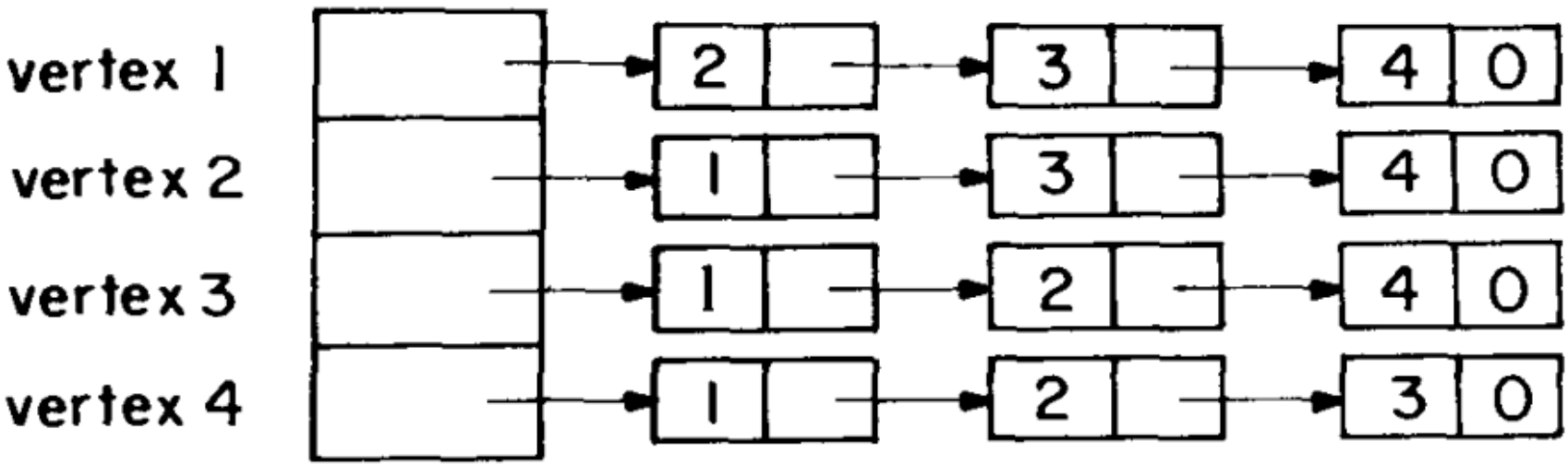
	1	2	3	4
1	0	1	1	1
2	1	0	1	1
3	1	1	0	1
4	1	1	1	0

Representation of A Graph

- Using Adjacency Lists
- Here the n rows of the adjacency matrix are represented as n linked lists
- There is one list for each vertex in the graph
- Each list has a head node
- The nodes in list i represent the vertices that are adjacent from vertex i
- Each node has 2 fields - VERTEX and LINK
- The VERTEX fields contain the indices of the vertices adjacent to vertex i

Representation of A Graph

- Adjacency Lists for Graph G_1



Traversal in a Graph

- Traversal is the process of visiting every vertex in the graph
- It can be done in 2 ways

1. Depth First Traversal
2. Breadth First Traversal

Depth First Traversal

- In depth first traversal, the depth of a graph is explored before its breadth
- Here the child vertices of a vertex are visited before visiting the sibling vertices
- Algorithm

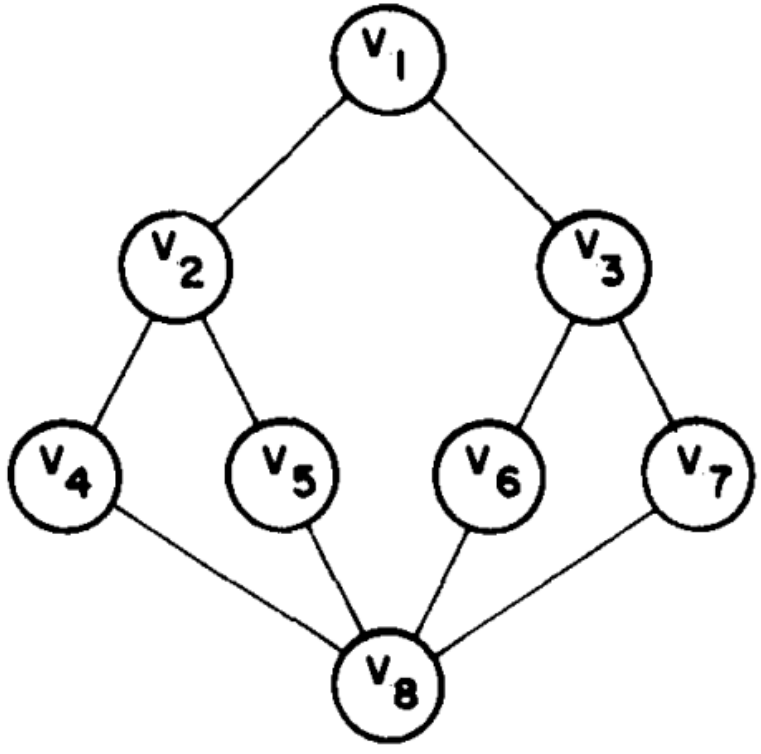
1. The start vertex v is visited
2. Visit a vertex w adjacent to v
3. Next visit a vertex adjacent to w, and so on
4. When a vertex u is reached and all its adjacent vertices are visited, we back up to the previous vertex and search for any adjacent unvisited vertex
5. This process is continued till all vertices are visited

Breadth First Traversal

- In breadth first traversal, the breadth of a graph is explored before its depth
- Here the sibling vertices of a vertex are visited before visiting the child vertices
- Algorithm

1. The start vertex v is visited
2. Visit all vertices adjacent to v
3. Next visit all vertices adjacent to these visited vertices and so on
4. This process is continued till all vertices are visited

Traversal - Example



- Depth First Traversal - $V_1, V_2, V_4, V_8, V_5, V_6, V_3, V_7$
- Breadth First Traversal - $V_1, V_2, V_3, V_4, V_5, V_6, V_7, V_8$

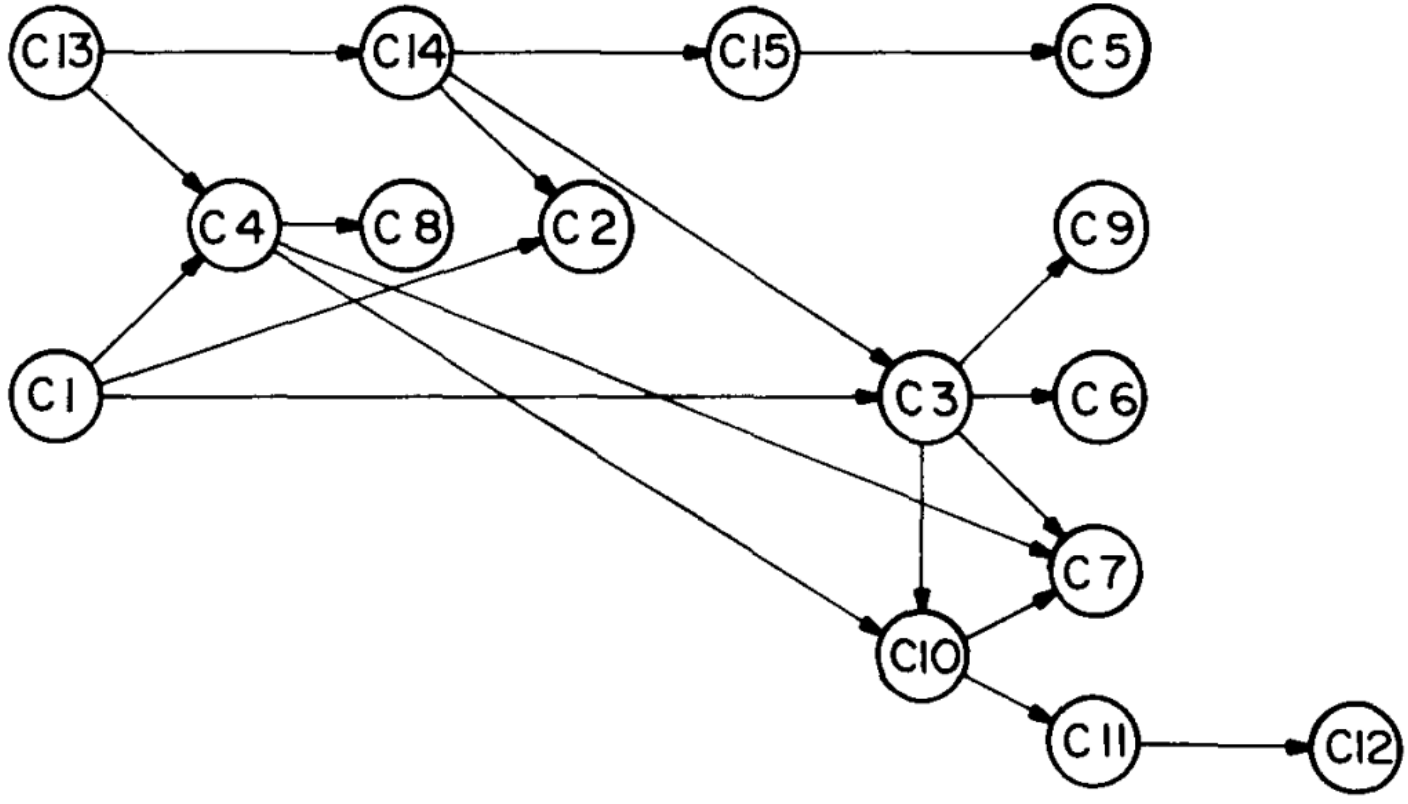
Topological Sorting

- In a directed graph, vertices can be used to represent activities or tasks to be performed and edges can be used to represent precedence relations between tasks
- Example - Courses Needed for a Computer Science Degree

Course Number	Course Name	Prerequisites
C1	Introduction to Programming	None
C2	Numerical Analysis	C1, C14
C3	Data Structures	C1, C14
C4	Assembly Language	C1, C13
C5	Automata Theory	C15
C6	Artificial Intelligence	C3
C7	Computer Graphics	C3, C4, C10
C8	Machine Arithmetic	C4
C9	Analysis of Algorithms	C3
C10	Higher Level Languages	C3, C4
C11	Compiler Writing	C10
C12	Operating Systems	C11
C13	Analytic Geometry and Calculus I	None
C14	Analytic Geometry and Calculus II	C13
C15	Linear Algebra	C14

Topological Sorting

- Directed Graph in which vertices are courses and prerequisites as edges



Topological Sorting

- The order in which these courses can be taught is determined by topological sorting
- The topological sorting of a directed graph is a linear ordering of its vertices such that for every directed edge from vertex u to vertex v, u comes before v in the ordering
- Two possible topological sorting of the above graph
- C1, C13, C4, C8, C14, C15, C5, C2, C3, C10, C7, C11, C12, C6, C9
- C13, C14, C15, C5, C1, C4, C8, C2, C3, C10, C7, C6, C9, C11, C12

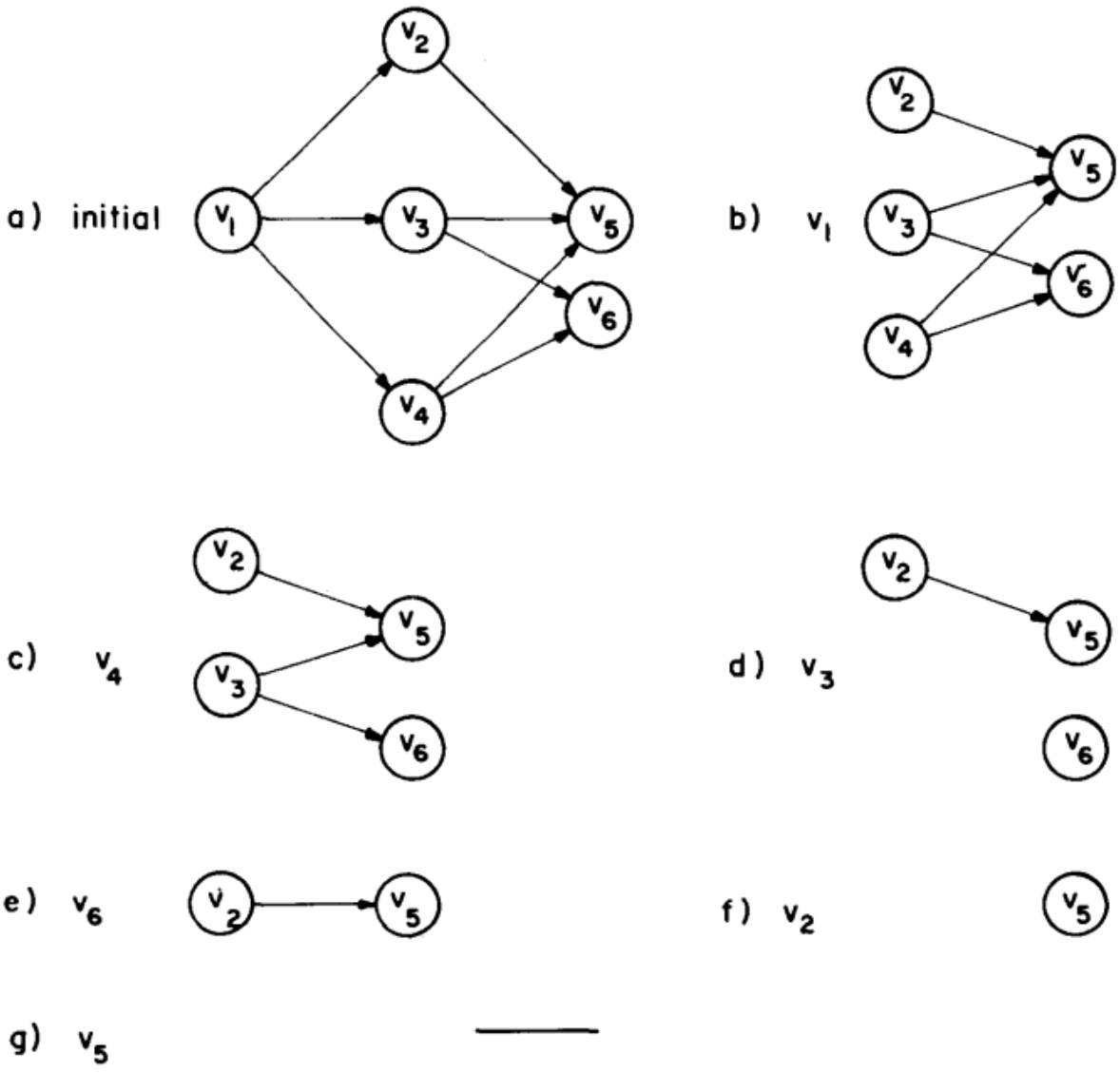
Topological Sorting

- Algorithm

1. If every vertex has a predecessor, the graph has a cycle and it is infeasible to topologically sort
2. Otherwise pick a vertex v which has no predecessors
3. Output v
4. Delete v and all edges leading out of v
5. This process is repeated for all vertices in the graph

Topological Sorting

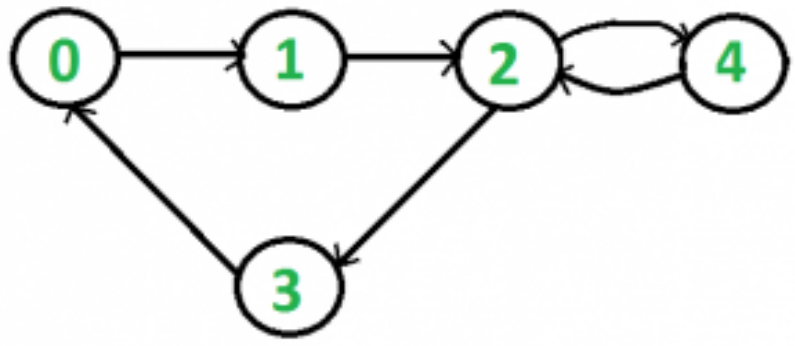
- Example



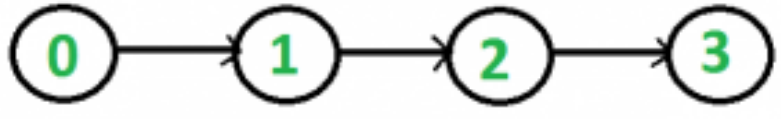
Topological order generated: $v_1, v_4, v_3, v_6, v_2, v_5$

Strongly Connected Components

- A graph is said to be connected if there is a path between every pair of vertices in the graph
- A directed graph is strongly connected if there is a path between every pair of vertices in the graph



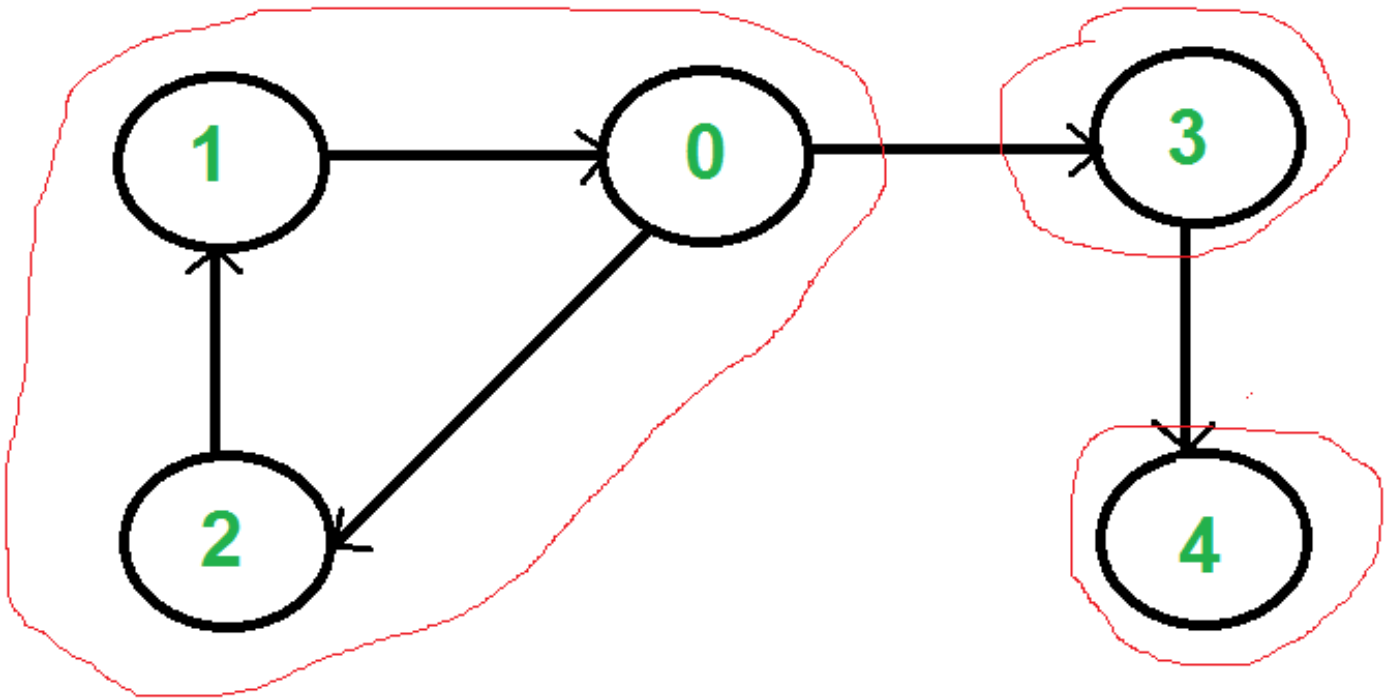
Strongly Connected



Not Strongly Connected

Strongly Connected Components

- A strongly connected component of a directed graph is a strongly connected subgraph of it
- There are 3 strongly connected components in the below given graph

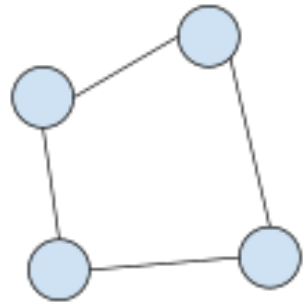


Biconnected Components

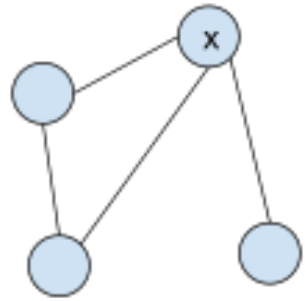
- A graph is said to be biconnected if

1. It is connected
2. Even after removing any vertex it is connected

- BiConnected Graph

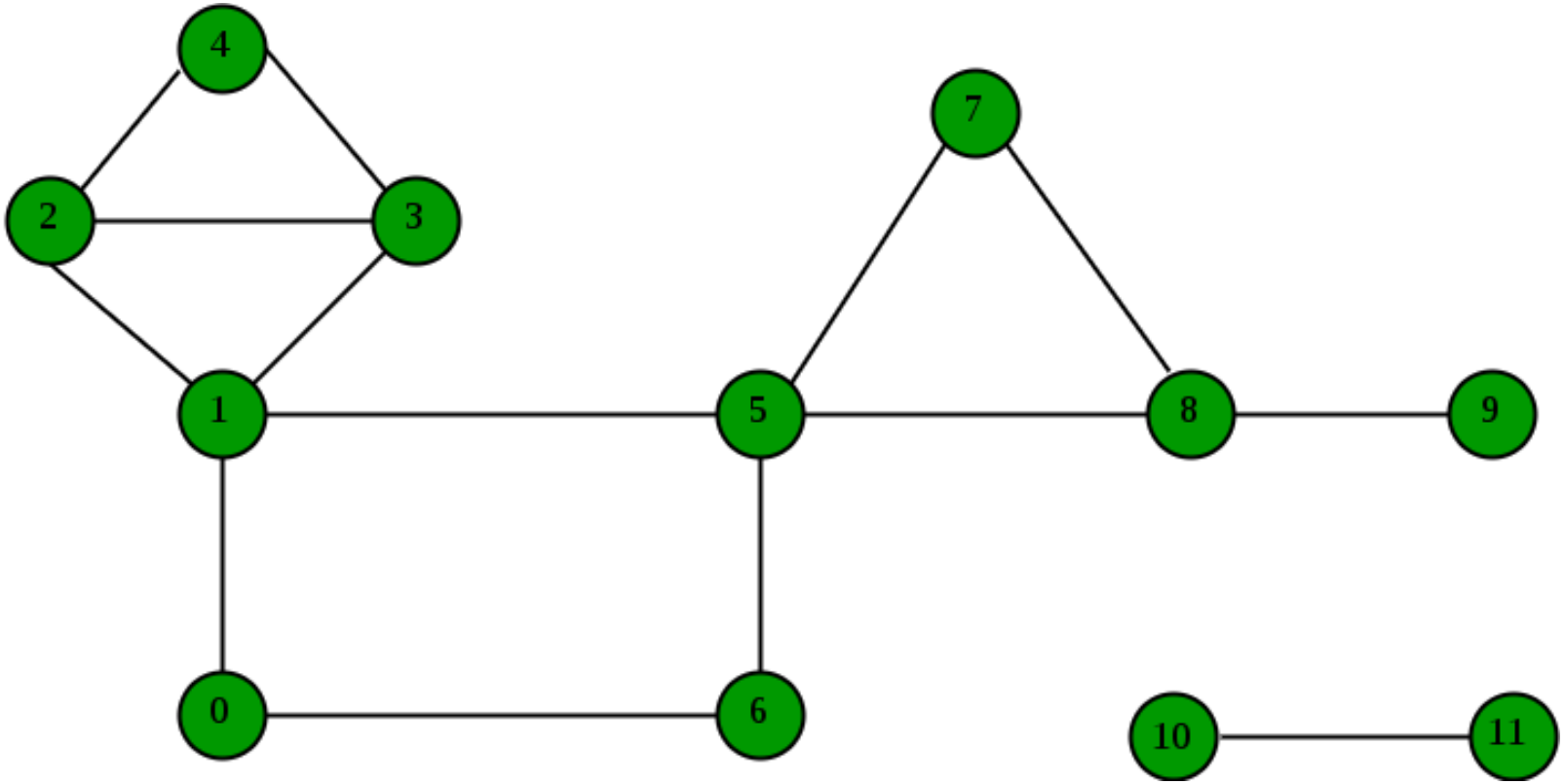


- A graph that is not BiConnected



Biconnected Components

- A biconnected component of a graph is a biconnected subgraph of it
- Example

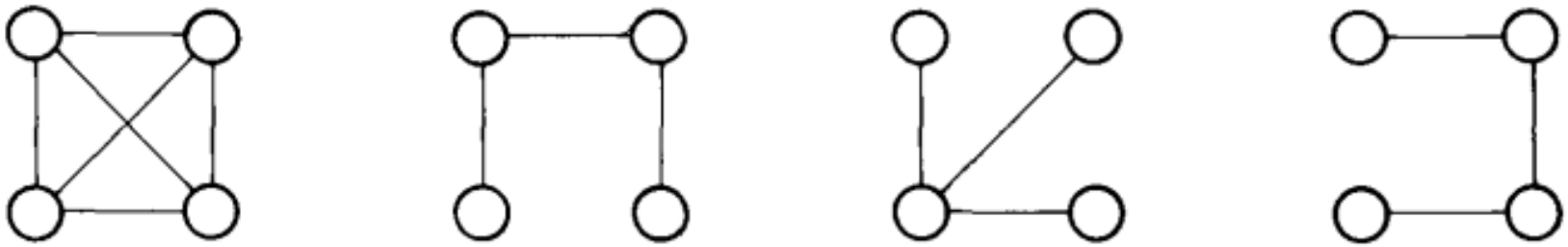


Biconnected Components

- The biconnected components in the above graph are
- 1-2, 1-3, 2-3, 2-4, 3-4
- 0-1, 0-6, 1-5, 5-6
- 5-7, 5-8, 7-8
- 8-9
- 10-11

Spanning Trees

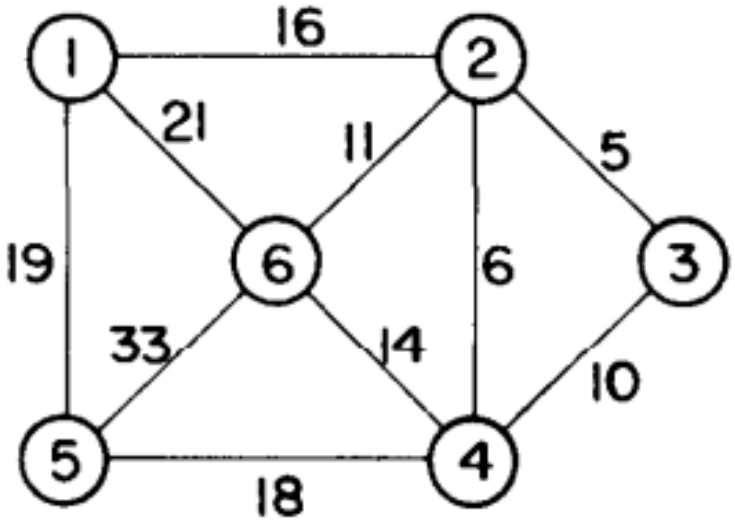
- The spanning tree of an undirected graph G is a subgraph of G that is a tree which contains all its vertices

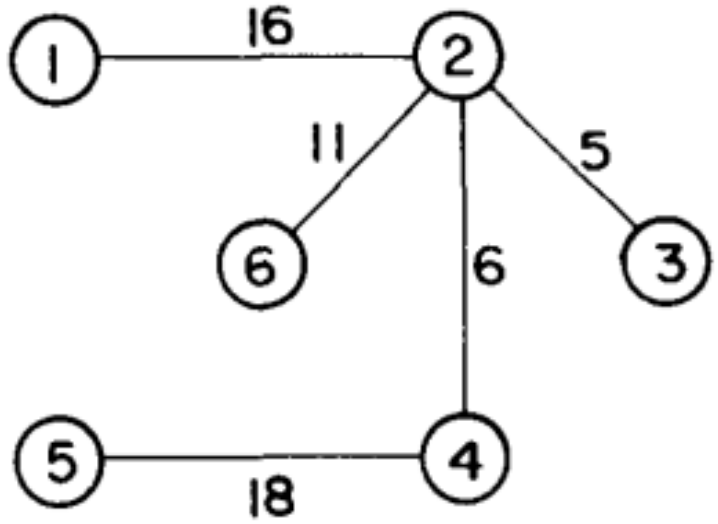


- In a practical situation, the edges in a graph will have weights assigned to them
- Here the vertices in a graph represent cities and edges represent links that connect them
- These links can be communication links or transportation links
- The weights correspond to cost of construction of links or length of links

Minimum Cost Spanning Trees

- The cost of a spanning tree is the sum of the costs of the edges in the tree
- We can find an optimal way of building communication links or transportation links between cities by finding the minimum cost spanning tree of a graph
- A Graph and its Minimum Cost Spanning Tree





Minimum Cost Spanning Trees

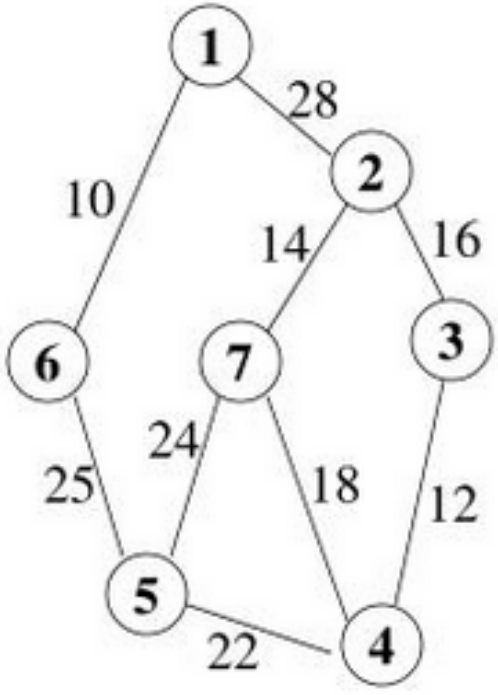
- Two popular algorithms for building minimum cost spanning trees are

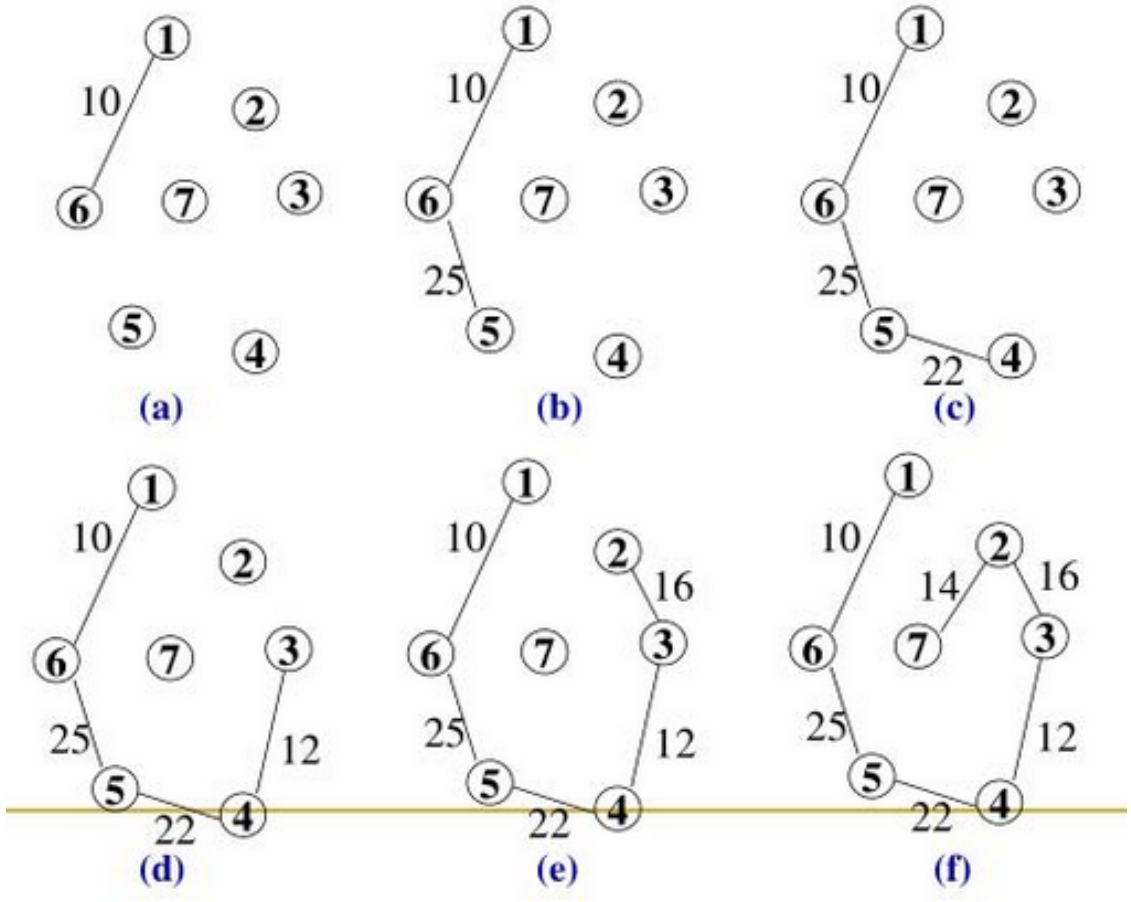
1. Prim's Algorithm
   Developed by Robert Prim in 1957
2. Kruskal's Algorithm
   Developed by Joseph Kruskal in 1956

Prim's Algorithm

1. Select all the vertices from the graph
2. Initially no edge is selected
3. Start with a random vertex from the graph
4. Select a least cost edge whose addition to the set of selected edges forms a tree
5. Repeat the above step till the minimum cost spanning tree is formed

Prim's Algorithm

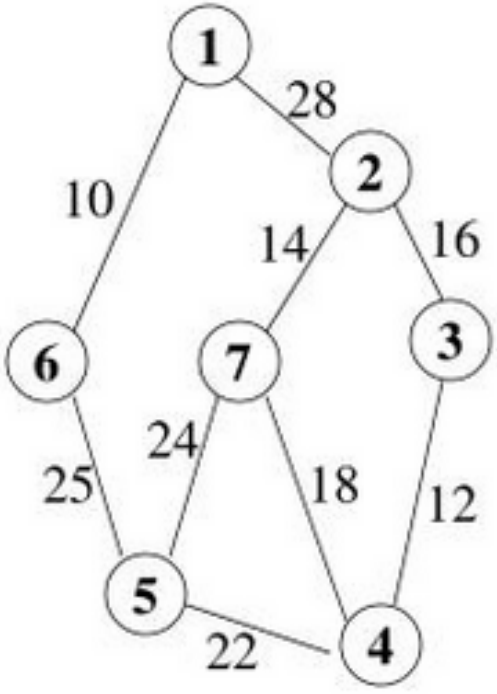


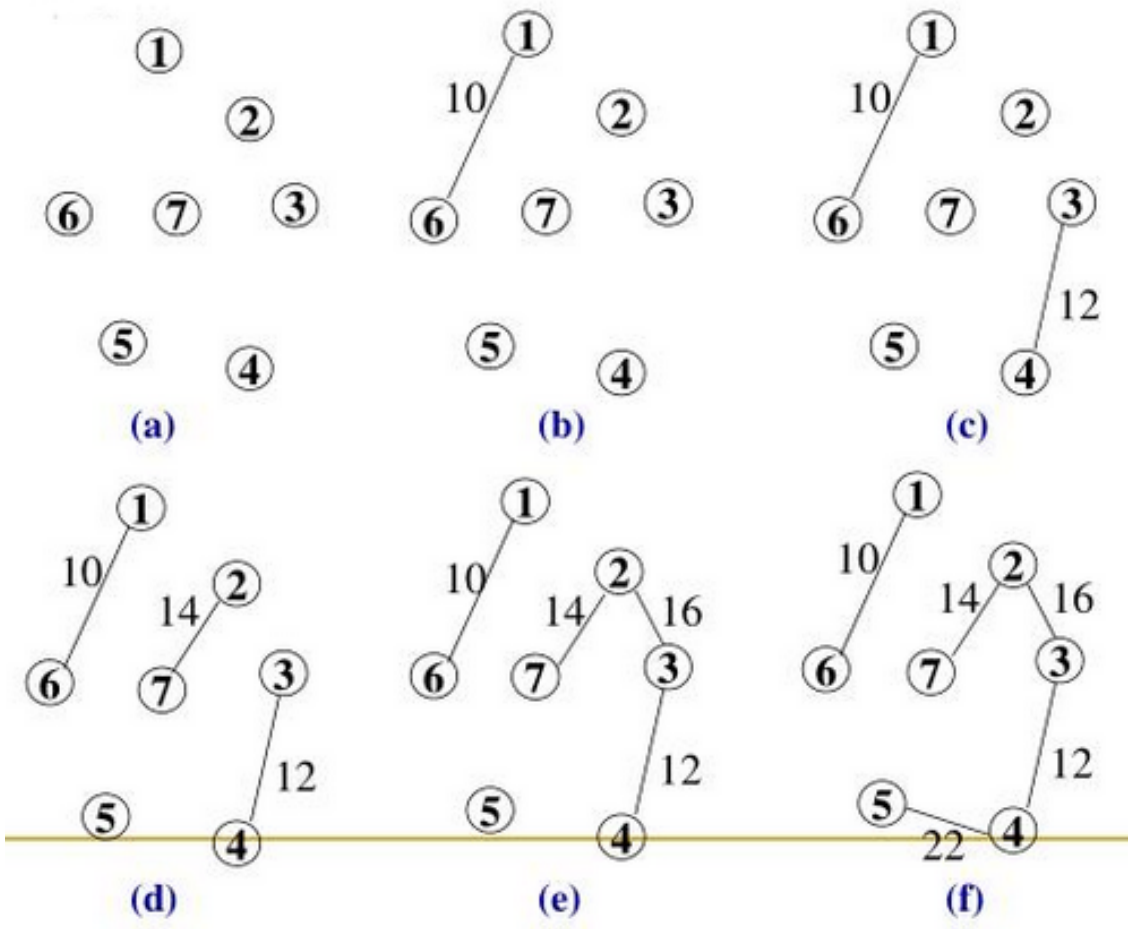


Kruskal's Algorithm

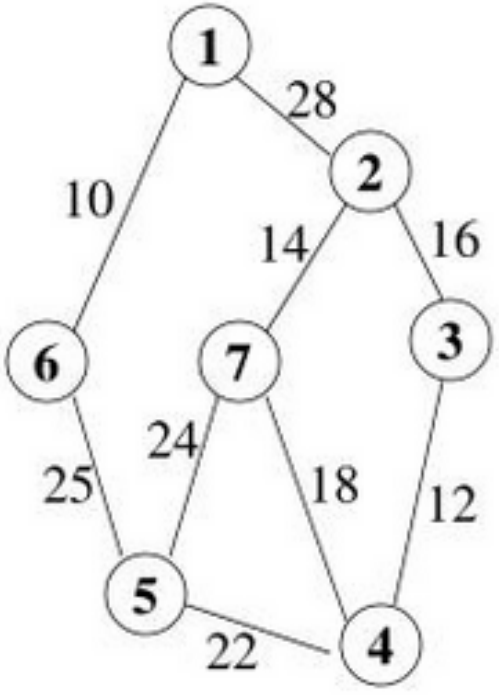
1. Select all the vertices from the graph
2. Initially no edge is selected
3. Select a least cost edge whose addition to the set of selected edges does not form a cycle
4. Repeat the above step till the minimum cost spanning tree is formed

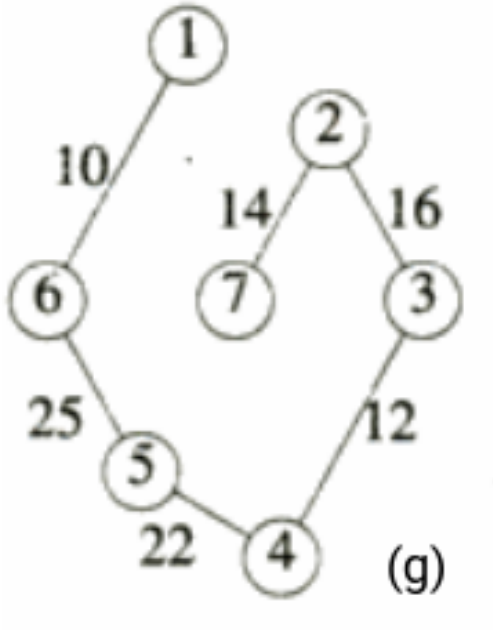
Kruskal's Algorithm





Kruskal's Algorithm





- Kruskal's algorithm is easier to implement compared to Prim's
- Prim's algorithm executes faster compared to Kruskal's

Shortest Path Finding Algorithms

- The shortest path problem is the problem of finding the path between any two vertices in a graph in such a way that the sum of the weights of its constituent edges is minimised
- We can represent a road network using a graph
- Here vertices in a graph represent cities (or road junctions)
- The edges represent road segments between cities (or road junctions)
- A directed graph can be used for representing one way traffic
- Using shortest path finding algorithms we can find the shortest path between two locations in a road network

Shortest Path Finding Algorithms

- There are three classes of shortest path problems

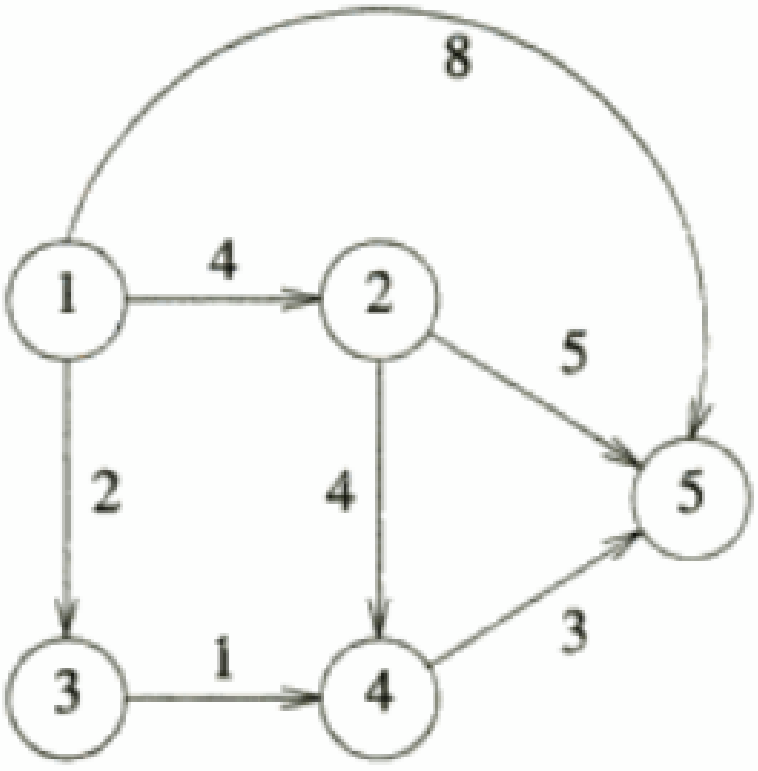
1. Single Source Shortest Path Problem
   Here we find the shortest paths from a single source vertex to all other vertices in the graph
2. Single Destination Shortest Path Problem
   Here we find the shortest paths to a single destination vertex from all other vertices in the graph
3. All Pairs Shortest Path problem
   Here we find the shortest paths between every pair of vertices in the graph

Dijkstra's Single Source Shortest Path Algorithm

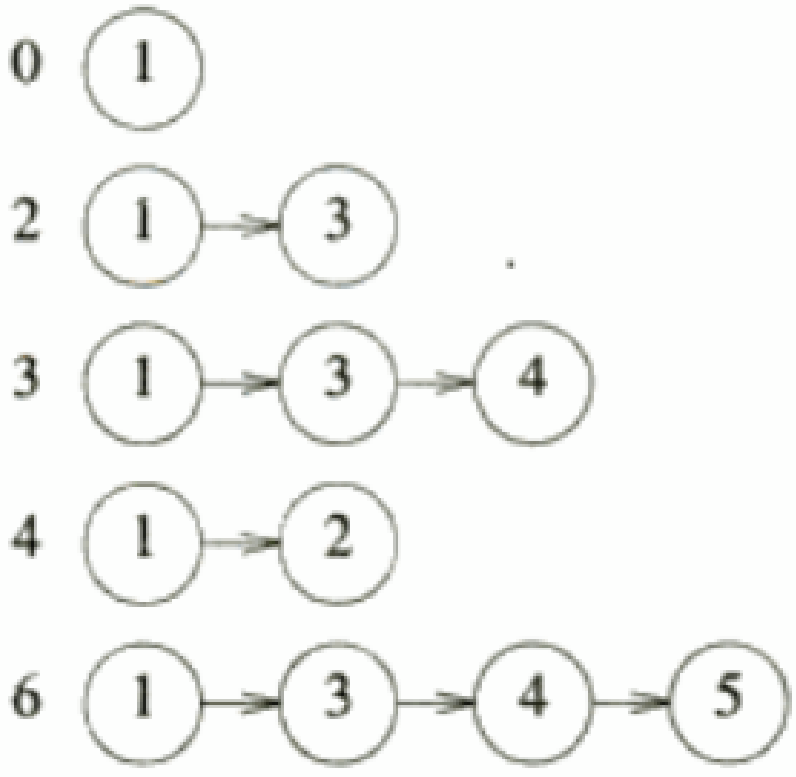
- E W Dijkstra developed this algorithm in 1959

1. We begin with a trivial path from the source vertex to itself. This path has no edges and has a length of 0.
2. In each stage the shortest path to a new destination vertex is generated.
3. This next shortest path is the shortest possible one edge extension of an already generated shortest path.
4. This process is continued till the shortest paths to all vertices are generated.

Dijkstra's Single Source Shortest Path Algorithm



(a) Graph



(b) Shortest paths

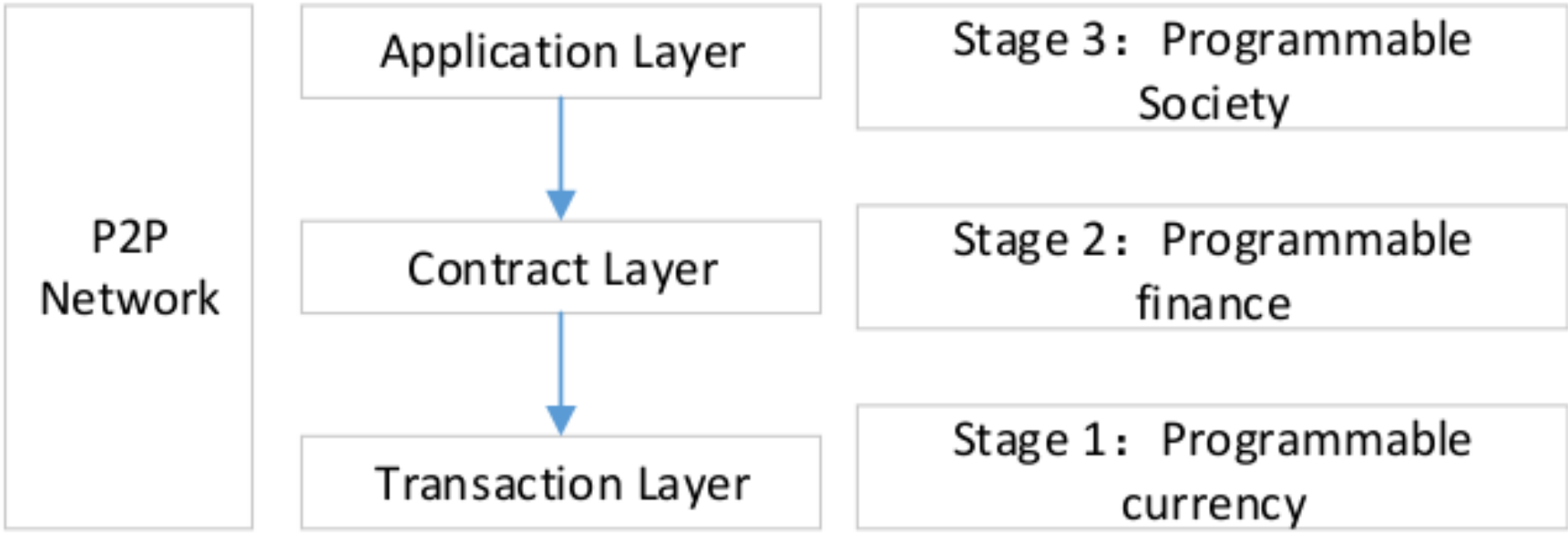
Module 5
Blockchain Data Structure

- A blockchain is a chain of records called blocks that contain certain information and are linked together using a peer-to-peer network
- In a peer-to-peer network, computers share information without a centralised server
- Each block contains a unique code called hash which is determined using a hashing algorithm
- It also contains the hash of the previous block in the chain.
- This data structure was invented by a person (or a group of persons) referred to by a fictitious name called Satoshi Nakamoto for using in the cryptocurrency bitcoin

Bitcoin

- Bitcoin is a cryptocurrency built using the blockchain technology
- A cryptocurrency is a digital currency that relies on cryptography to prevent fraudulent transactions
- There are only 21 million bitcoins available, as set in its source code
- Among these 18.5 million are in circulation now
- The remaining 2.5 million are yet to be released for circulation
- Bitcoin Mining is the process through which bitcoins are released into circulation
- This involves solving a computationally difficult puzzle
- By solving this puzzle, a number of bitcoins are given as reward
- 1 bitcoin is equivalent to 43 lakh rupees now

Blockchain Architecture

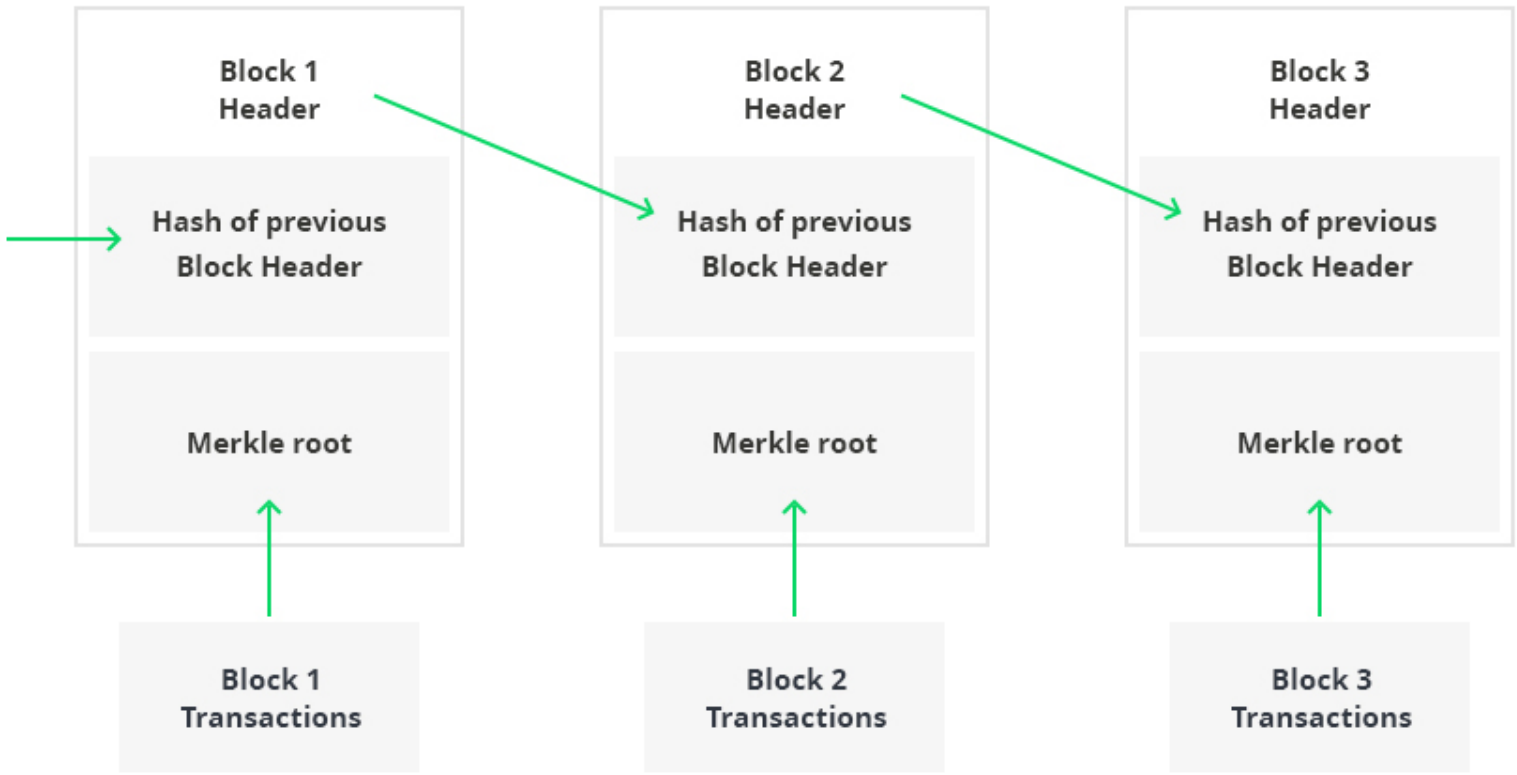


Blockchain Architecture

- It contains 3 layers

1. Transaction Layer
   - bottom layer
   - corresponds to blockchain development stage 1
   - It indicates the programmed currency represented by bitcoin
2. Contract Layer
   - middle layer
   - corresponds to blockchain development stage 2
   - It indicates the programmable finance represented by Ethereum
   - Ethereum is an open source blockchain with smart contract functionality
3. Application Layer
   - top layer
   - corresponds to blockchain development stage 3
   - It indicates the programmable society represented by the decentralised autonomous organisation

Blockchain Structure



Blockchain Structure

- Each block contains a header and a list of transactions corresponding to the block
- The block header contains metadata about the block, such as hash of the previous block and merkle tree root
- Every transaction in a block has a hash associated with it
- These hashes are stored in a tree called merkle tree
- The merkle tree root or merkle root contains the hash of all the hashes of various transactions of a block
- In other words, the merkle root provides a summary of all the transactions in a block
- The hash value of a block is obtained by performing a SHA256 (Secure Hash Algorithm) hash operation on various metadata in the block header

Contract Data

- Smart contract is a commercial contract written in a programming language that automatically enforces the terms of the contract when the predetermined conditions are met
- If we are implementing a cryptocurrency using a blockchain network, the constraints associated with this currency are stored in the smart contract
- It includes

1. The characteristics of the money (how many units there are, who initially owns it, etc)
2. How users can interact with the money (ask for a balance, make a payment, etc).

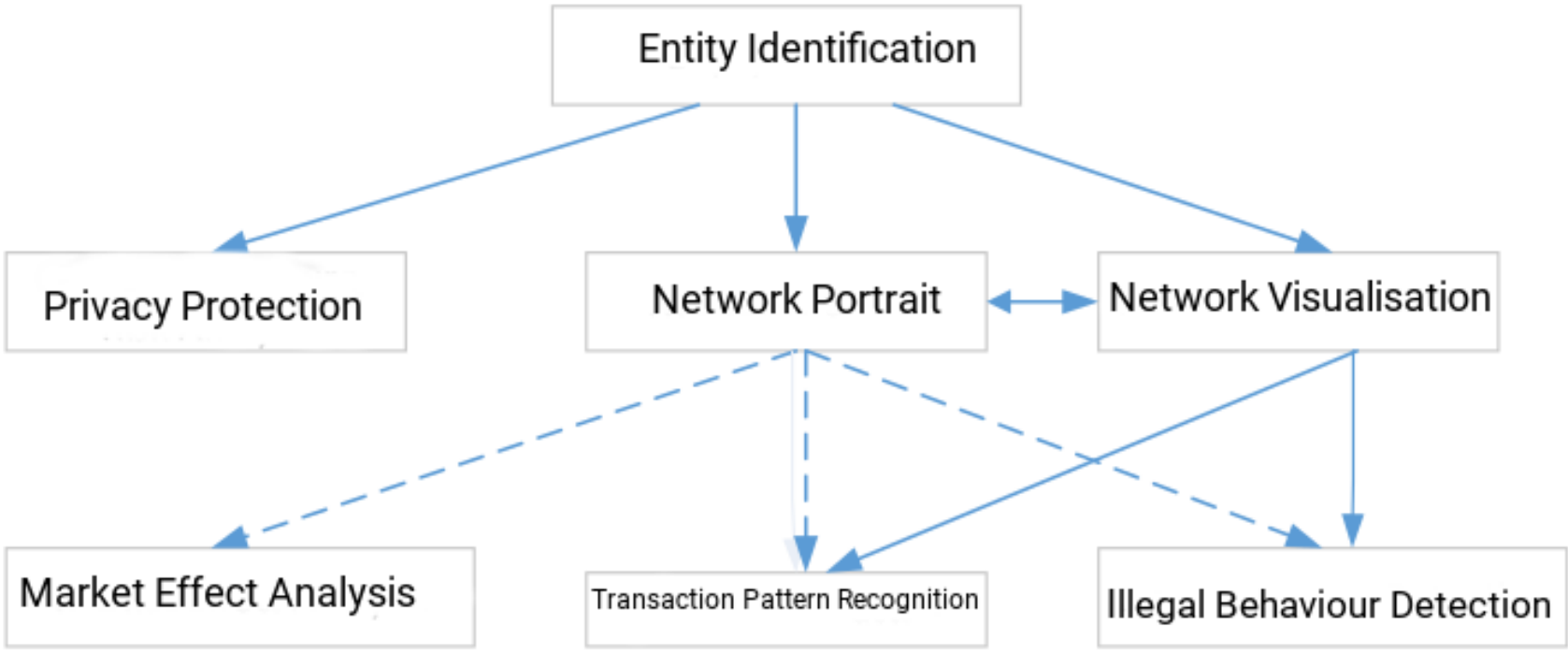
Problems to be Solved in Blockchain Data Analysis

1. entity identification
   - Entity can be a user or an organisation
   - In bitcoin transactions entities are anonymous
   - We need to identify which entity performs which transaction
2. privacy protection
   - It can be divided into identity privacy protection and transaction privacy protection
   - Identity privacy protection refers to protecting the identity of a user by hiding privacy details such as ip address
   - Transaction privacy protection refers to protecting various transaction related knowledge such as transaction amount, address of the recipient etc.
3. network portrait
   - We need to analyse the characteristics of a bitcoin network
   - How many users are involved in the transaction?
   - What are the characteristics of these users?

Problems to be Solved in Blockchain Data Analysis

4. network visualization
   - We need to have visualisation tools for tracking bitcoin transactions in real time
5. market effect analysis
   - We need to analyse the effect of bitcoin on other cryptocurrencies and traditional currencies
6. illegal behaviour detection
   - Anonymous features of bitcoin can lead to illegal activities
   - We need to detect illegal behaviour in the bitcoin network
7. transaction pattern recognition
   - We need to recognise specific patterns in bitcoin transactions for detecting illegal behaviour in the network

Problems to be Solved in Blockchain Data Analysis



Problems to be Solved in Blockchain Data Analysis

- Solid arrows indicate that the previous study is the basis for the next one
- Dotted arrows indicate that the latter study is a specialisation of the previous one
- Two-way arrows indicate different aspects of the same problem

Bibliography

1. Cormen T.H., Leiserson C.E, Rivest R.L. and Stein C, Introduction to Algorithms, Prentice Hall India, New Delhi, 2004
2. Ellis Horowitz, Sartaj Sahni, Fundamentals of Data Structures, Galgotia BookSource
3. Sartaj Sahni, Data Structures, Algorithms and Applications in C++, Universities Press
4. Yang, Xiaojing, Jinshan Liu, and Xiaohe Li. Research and Analysis of Blockchain Data, Journal of Physics: Conference Series. Vol. 1237. No. 2. IOP Publishing, 2019.
5. https://en.wikipedia.org
6. https://anirvana11.wordpress.com/2013/04/24/164/