Data Structures

- A way of organizing data so that it can be used effectively
- Makes it possible to create fast and powerful algorithms
- Help to manage and organize data
- Help make computer code easy to understand
- The two main categories of data structures are ***linear*** and ***non-linear*** data structures.

Course Content

Non Linear Data Structures

- Trees
    - Tree Traversals
    - Conversion of expressions from one form to another
    - Internal representation of Trees (linked list and array)
    - Binary Trees
    - Binary Search Trees
    - Height Balanced Trees
- Hashing
    - Division/remainder
    - Mid square
    - Folding at boundaries

Course Content

- Linear Data Structures
    - Arrays
    - List
    - Linked List
    - Queues
    - Stacks
- Sorting and Searching
    - Bubble Sort
    - Selection Sort
    - Insertion
    - Radix / Bucketing
    - Shell Sort

Course Content

- Collision and Collision Handling
    - Collision
    - Linear Probing
    - Chaining
    - Bucketing

Arrays

- An array is a collection of object of the same type stored in contiguous memory under a common name
- Each component of an array is called element of an array
- Elements of arrays are accessed by specifying the position with an index value or subscript.
- The address of an array is the memory location of the first element
- An array is stored so that the position of each element can be computed from its index using a mathematical formular.
- Generally, arrays have three parts namely address, base type and size
- The base type of an array determines the amount of memory used by each element

Arrays (How multi-dimensional arrays are stored in memory)

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39

A. Assume the elements are to be stored in row major order

Let us consider the array name to be X
Let us also consider the element 30 that is X[4,7]
If the base address is 200 and the width is

and the element 30. 30 is X[4,7] element and is on 4 (I) column 7 (J) of the 5 (n) by 8 (m) array
To deduce its location, we need to cover 3 (I-1) rows of 8 column (m) plus (7-1) (J) columns
The memory location will therefore be given as X[I,J] = m(I-1) + J-1
If the base type (B) is 2 bytes, then the address will be X[I,J] = B[m(I-1)+J-1]
Assume the base address (BA) is 200, then we can deduce the location as

X[I,J] = BA + B[m(I-1)+J-1]

Hence the address of 30 will be = 200+ 1[8(4-1)+(6-1) = 200+(24+5) = 229

Arrays (How multi-dimensional arrays are stored in memory)

- B. Storing in Column major order

Let us consider the array name to be X and the element 30. X[4,7] element is on row 4 (I) column 6 (J) of the 5 (n) by 8 (m) array

To deduce its location, we need to cover 6 (I-1) columns of 5 rows (n) plus 3 (J) rows

The memory location will therefore be given as X [I,J] = n(J-1) + I-1

If the base type (B) is 2 bytes, then the address will be X[I,J] = B[n(J-1)+I-1]

Assume the base address (BA) is 200, then we can deduce the location as

$$30=X[4,7] = BA + B[n(J-1)+I-1]$$

Hence the address of X[30] will be = 200+ 1[5(7-1)+(4-1)] = 200+(30+3) = 233

Array operations

The operations that can be performed on arrays are :

- Searching for an element
- Inserting an element
- Deleting an element

- The main disadvantages of arrays are as follows:

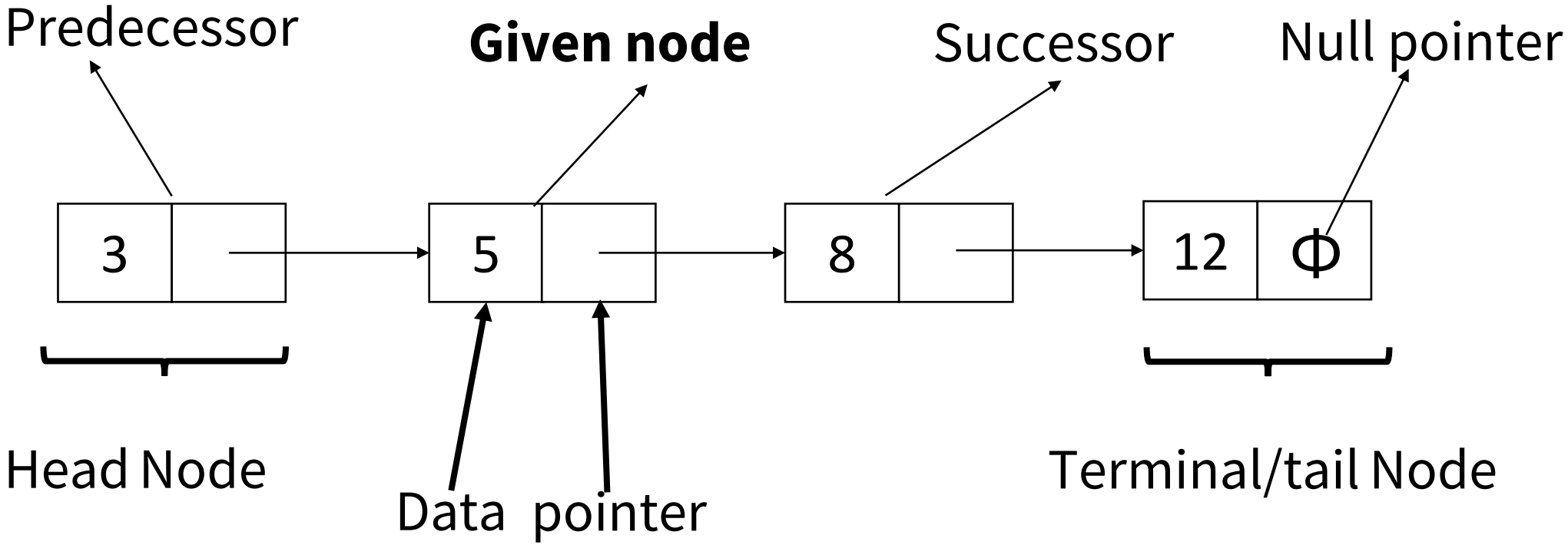
- Size of static arrays remains constant in memory
- Insertions and deletions between elements require movement of other elements

Linked List

- Linked list is a sequence of **nodes** that contain **data items** and **links** to subsequent nodes.
- A node consists two parts; the **data field**, and the **pointer**, **link** or **index field.**
- The very first node is known as the **head node**, and the last node is known as the **tail or terminal node**.
- All terminal nodes have the link field as null, lamda or Φ (zero).
- Each node is always related to the next adjacent node.

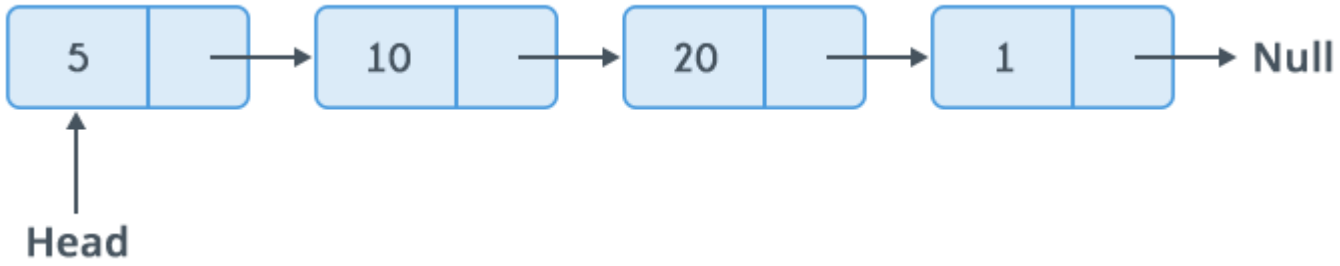
Linked List

- The node before a given node is known as its **predecessor.**
- The node after a given node is known as its **successor**.

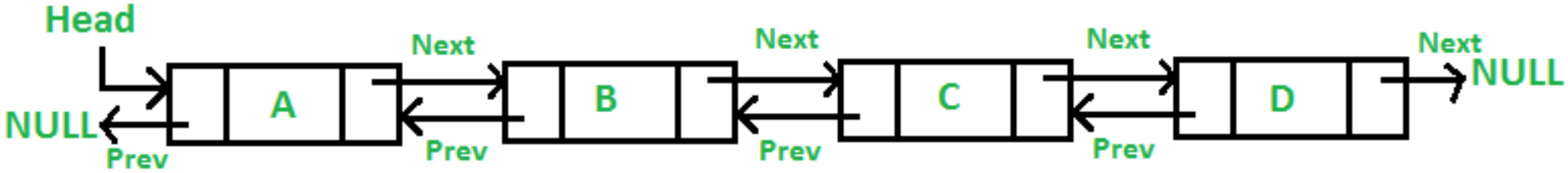


Types of Linked Lists

- A **singly linked list** also called a **simple linked** list is characterized by item navigation in the forward direction only. All navigations start with the Head node

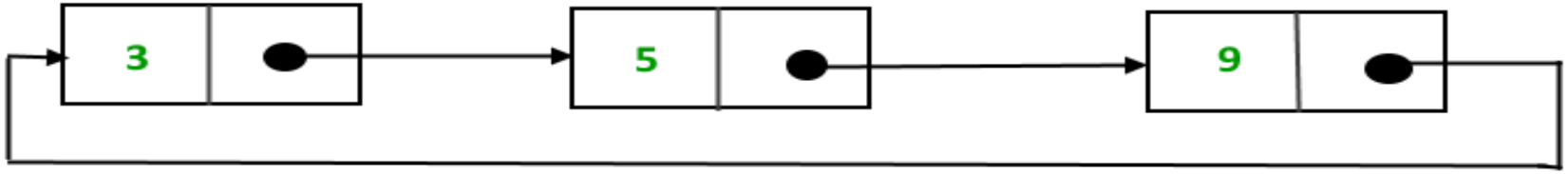


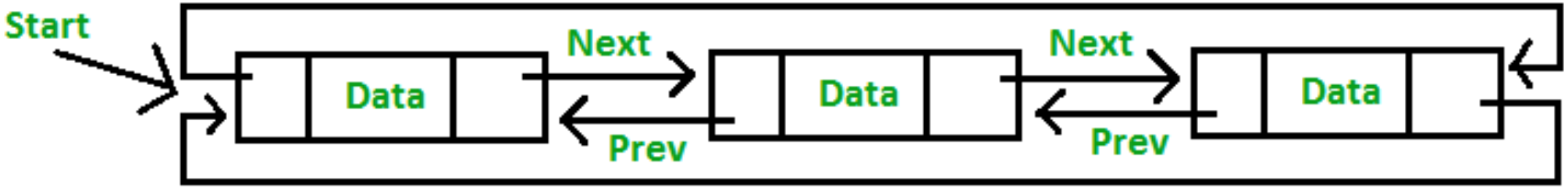
- **Doubly linked list** can navigate through items in both forward and



Types of Linked Lists

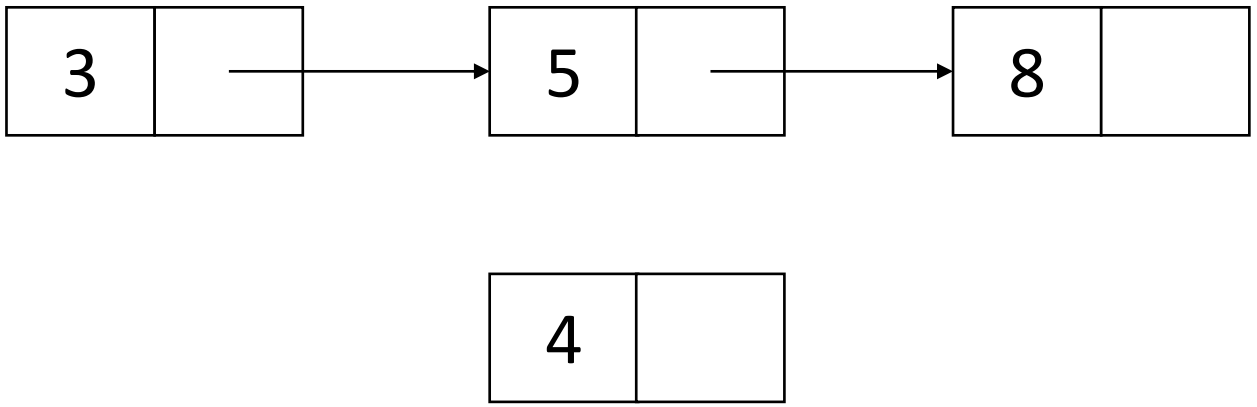
- A **circular linked list** is a list in which the last node points to the very first node. A circular linked list can either be singly or doubly linked.





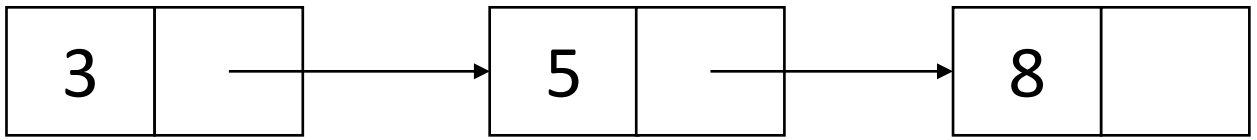
Basic Linked List Operations

- **Insertion:** To insert an item or node into a linked list, the link of the position's predecessor node is directed towards the data field of the new node to be inserted, while its pointer is directed to its successor node. Insertion can be at the front, middle or at the end.



Linked List

- **Deletion:** To delete a node from a linked list, the pointer of the predecessor node is removed from the node and redirected to its successor node. And that eliminates the node from the list. We can delete start node, a middle node or an end node.



Stack

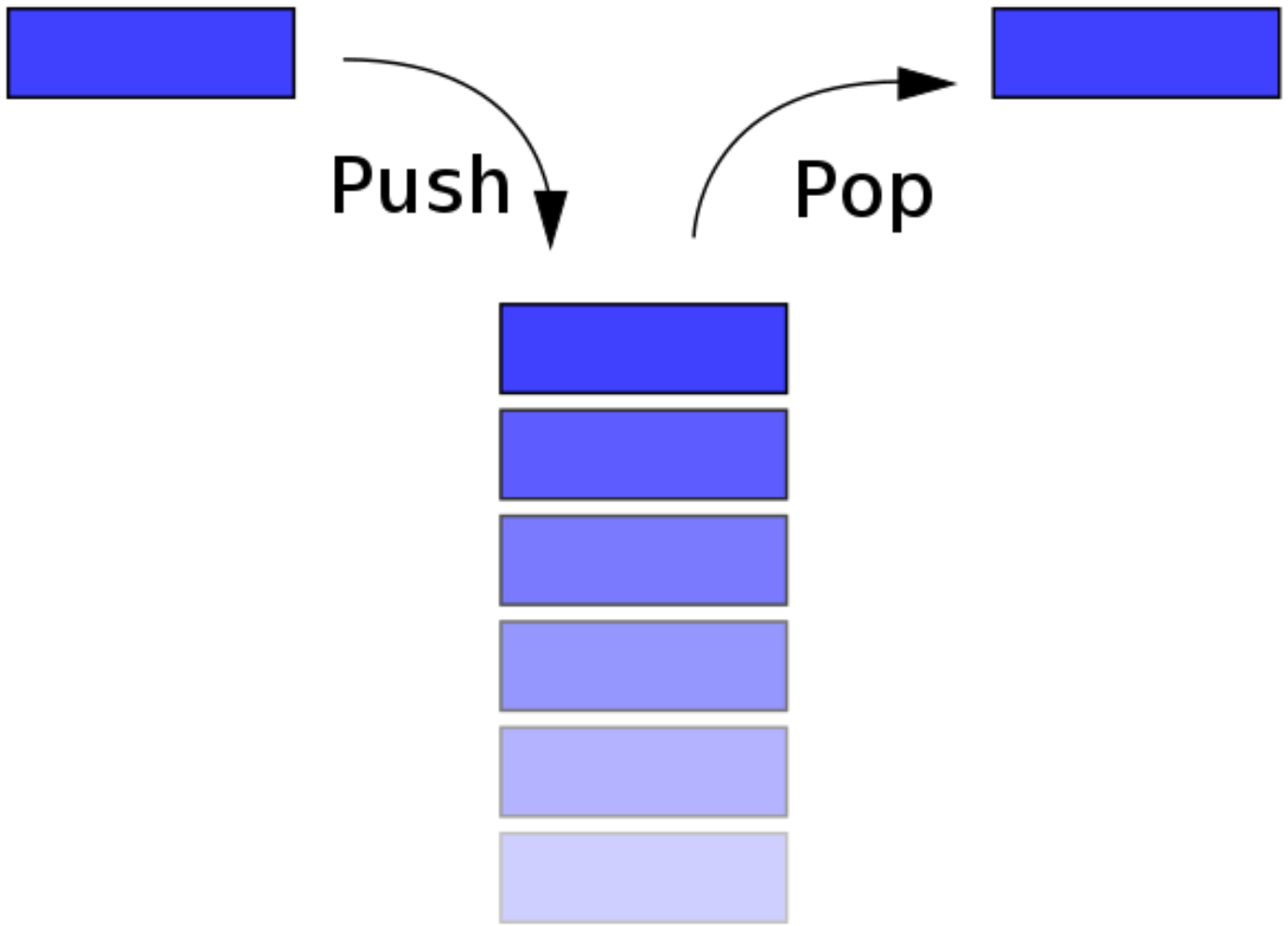
- The **stack** is a linear ordered list in which all insertions and deletions take place at one end; the top.
- The stack data structure is also referred to as the LIFO (last in, first out) structure.
- Two main operations are used; **PUSH** and **POP.**
- In all stack operations, we need to guard against ***overflow*** and ***underflow.***

Stack

- An **overflow** occurs when a program attempts to add an element to a full stack.

- An **underflow** occurs when a program attempts to rmove or delete an element from an empty stack.

Stack



Working Functions Associated with Stacks

- **CREATE(S)** – Create an empty stack.
- **PUSH(i,S)** – Insert an element, i, onto a stack, S, and return a new stack.
- **POP(S)** – Remove the top element of stack S and return a new stack.
- **PEEK(S)/TOP(S)** – Access the topmost element on stack S. Note that it does not remove the topmost item.

Working Functions Associated with Stacks

- **ISFULL(S)** – Check if the stack is full or not. This function returns True/False.

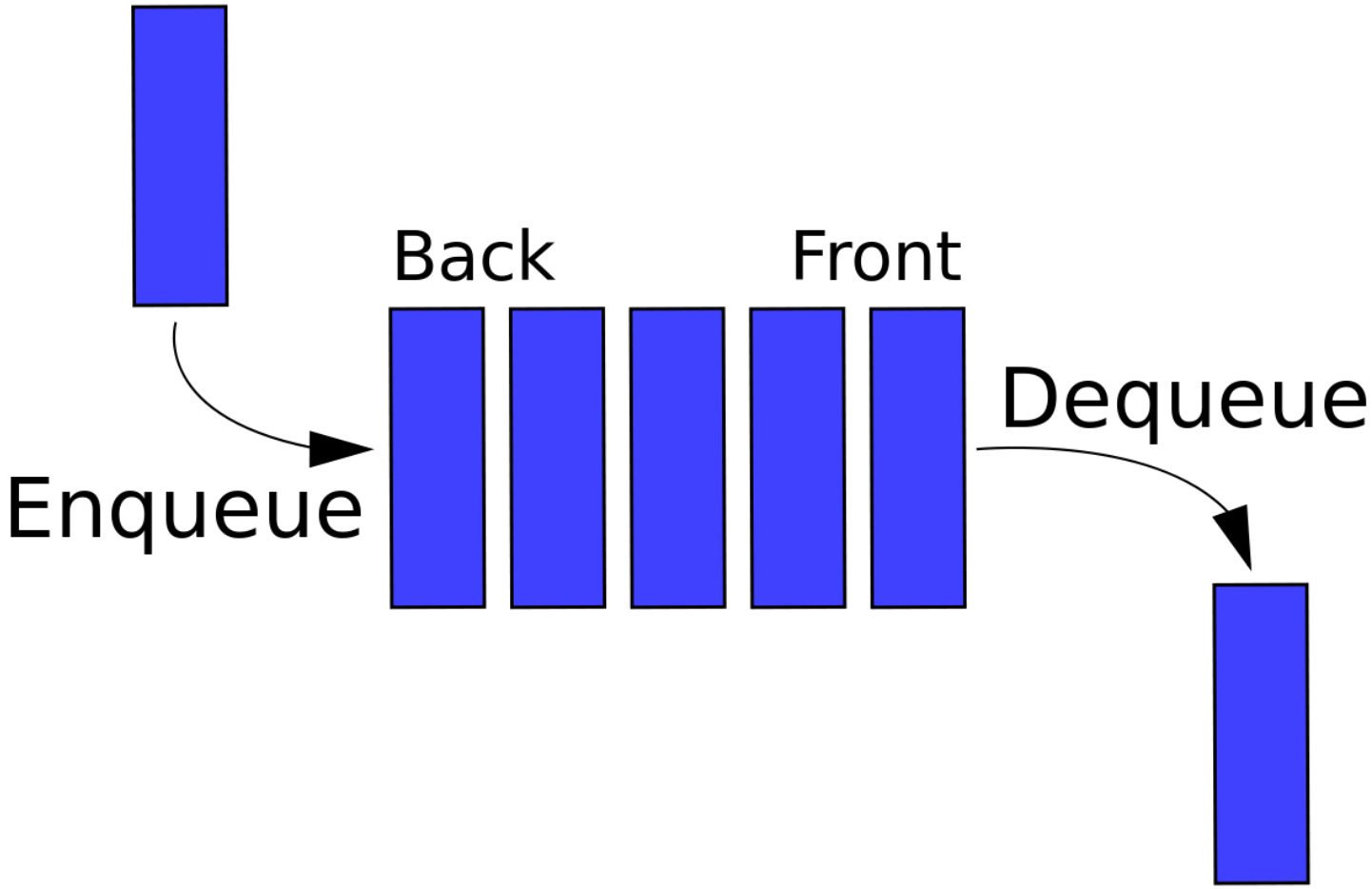
- **ISEMPTY** – Check if the stack is empty or not. Returns True/False.

Queues

- A **queue** is a linear data structure similar to a stack, but open at both ends; one end is always used to insert data (enqueue) and the other is used to remove data (dequeue).

- Queues follow a First-In-First-Out (FIFO) or Last-In-Last-Out (LILO) methodology, i.e., the data item stored first will be accessed first.

Queues



Basic Functions in Queues

- **ENQUEUE()** – Add an item to the queue.
- **DEQUEUE()** – Remove an item from the queue.
- **PEEK()** – Get the element at the front of the queue without removing it.
- **ISFULL()** – Check id the queue is full.
- **ISEMPTY()** – Check if the queue is empty.

Sorting

- A way of arranging items systematically
- The items can be arranged in ascending or descending order
- Some of the sorting techniques are as follows:

- Bubble Sort
- Selection Sort
- Insertion Sort
- Bucket Sort
- Quick Sort, etc

Bubble Sort

step 1: Compare the first and the second elements in the set.

step 2: If the 1st element is greater than the second, interchange the positions of the two elements so that the first becomes the second and the second the first otherwise maintain their positions.

step 3: Next compare the second and the third elements, third and fourth, etc. until the last but one element has been compared with the last, interchanging positions where necessary.

Step 4: After completing step 3, one item will be at its correct position. Assume it is not part of the list.

Step 5: Repeat step 1 to 4 until there is only one item left

Bubble Sort (Sorting N items in ascending order)

```
Input n
For pass = 1 to N
For i = 1 to N-pass
    If x[i] >x[i+1] then
       temp = x[i]
       x[i]=x[i+1]
       x[i+1] = temp
    endif
End for I
End for pass
```

Selection Sort (ascending order)

Set 1: Let *pass* = 1

Step 2: For the current *pass*, start from the element at position *pass* and scan through to the end of the entire data elements for the minimum entry depending on the order of sorting

Step 3: Interchange the positions of the minimum element and the element at the *pass* position such that the minimum entry selected in step 2 is moved to location *pass* and the element at location *pass* to the location where the minimum entry was found.

Step 4: Increase the value of pass by 1.

Step 5: If the current value of pass is less than the total number of items in the original list then repeat steps 2-4 otherwise stop as the list will be sorted by now.

Algorithm for Selection

```
Input N
For i=1 to N
     pos = i
    for j = i+1 to N-1
        if min>x[j] then
            pos = j
    end for j
    temp = x[i]
    x[i]=x[pos]
    x[pos] = temp
End for i
```

Insertion Sort

Assume the number of elements to be N

Step 1:Set a variable *pass* to 1.

Assume the first entry in the list to be at its correct position when the entire list is sorted. We shall refer to this part of the list as sorted segment, hence we are assuming that the first entry is a sorted segment.

Step 2: Increase the value of *pass* by 1 so that it points to the element at position *pass*.

Step 3: Compare the *passth* entry with the entries in the sorted segment starting with element at location to that at (pass-1) and insert the *passth* element in its proper position of the sorted segment.

Step 4: Repeat steps 2 and 3 until the counter *pass* has a value equal to *N*.

Insertion Sort

```
Input N
For i=1 to N
   j=i+1
  while j >1 && x[j]>x[i]
       x[j]=x[j-i]
       j=j-1
  end while
   x[i]=x[j+1]
End for
```

Radix or Bucket Sort

- Assume the largest number to have N digits, we will need N passes
- Numbers with fewer digits than N should be preceded with zeros to make up
- Label 10 buckets from 0 to 9

Pass 1: Place the numbers in the buckets based on the Nth digits

Pass 2: Place the numbers in the buckets from Pass 1 based on the (N-1)th digits

- Repeat, similar to pass 2 for up to pass 1
- Elements will be sorted

Searching

- A process of locating an item from a list or the process of locating a target date from a set.
- The two main types of searching are namely Linear (or Sequential) and the Binary search. Other types of searching includes Jump Search, Fibonacci Search, exponential Search, etc.
- The Binary search is faster than the Linear Search except when dealing with small data sets.
- The Linear search can be applied to ordered and unordered data sets while the Binary search works on ordered sets

Sequential or Linear Search

- Searching for an item requires traversing the list from start to the end
- Searching for an item results in either finding a match or whole list searched without a match.
- For an ordered list, the search stops when a match is found or when an item greater than or less than the search item is encountered, depending on the sort order. The latter implies the item is not on the list.

Linear Search on Unordered items

```
Declare and store elements in X
n=len(X)
Start=1
While (start<=n)
    if(target!=X[start])
          start=start+1
 endWhile
If start<=n
   targetPosition=start
Else
   display “Target element not on the list”
```

Linear Search on Ordered Items

```
Declare and store elements in X
n=len(X)
Start=1
While (start<=n && target!=X[start])
        start=start+1
 endWhile
If start<=n
   targetPosition=start
Else
   display “Target element not on the list”
```

Binary Search

- It is also known as half-interval search.
- It compared the target item to the middle element
- If the target item is not the middle item, half of the elements in the list are excluded from subsequent search
- It works by the use of two pointers. The following algorithm explains the processes:

Binary search Algorithm

```
Declare and store elements in array, say X
n=len(X)
start=0
end = n-1
Found = false
While (start<=end)
    middlePosition=0.5*(start+end)
    if (X[middlePosition]<target)
          start=middlePosition+1
    else if (X[middlePosition]>target)
         end = middlePosition-1
    else
          found=true
           exitLoop
    endIf
If (Found=true)
       targetPosition=MiddlePosition
else
    display “target element not found”
```

Trees

- A tree is a non-Liner data structure.
- It has a hierarchical structure consisting of nodes
- A node is a structure that contains either a value or condition
- Each node has zero or more child nodes
- On top of a tree is a root node with subtrees

Advantages of Trees

- Trees are used to represent hierarchies
- Trees provide an efficient way of inserting and searching of data elements
- Trees have structural relationships in the data elements

Terminologies used for trees

- **Sibling nodes**: Nodes of the same parent
- **Internal Node**: Any node that has child node(s)
- **External node**: Also known as leaf node or terminal node, is any node that has no child nodes.
- From the root node, all other nodes can be reached by following **edges** or **links**.
- **Height of a tree**: Is the longest downward path to a leaf node from the root node.
- **Height of a node:** Is the number of edges from the root to the node.
- Depth of a node: The number of nodes from the root to the node
- **Level of a node**: Is the number of edges from the root to the node. The root has a level of zero.
- **Degree of a node**: Is the number of children the node has

Terminologies

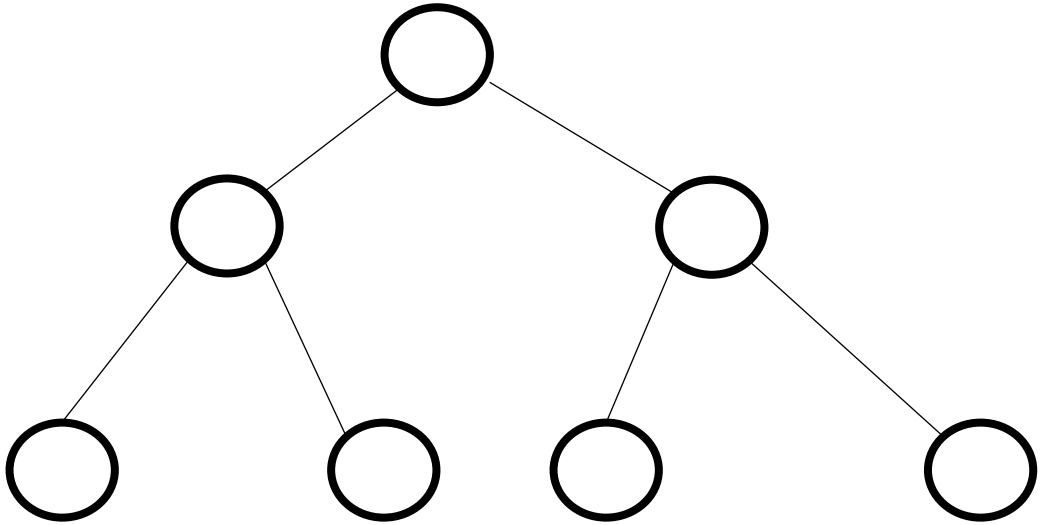
- Width is the number of nodes in a given level
- Breadth is the number of leaves of a tree
- Subtree: The subtree of a tree T is a tree in T consisting of all descendant nodes of a given node.
- Forest is two or more disjoint trees.
- Tree traversal is the process of visiting each node of a tree only once

Binary Trees

- A tree in which each node has at most two children referred to as the left child and the right child.
- The left child and its descendants nodes are referred as as Left Subtree and the of the right as Right Subtree,
- Each node has three parts and these are pointer to the left subtree, pointer to the right subtree and the data element.

Types of Trees

- **Perfect Binary tree:** A tree in which all non-leaf nodes have two children and all leaves are at the same level.



Types of trees

- **Rooted Binary Tree**: A tree having a root with each non terminal node having at most two children
- **Full Binary Tree**: Is a tree in which every node has either 0 or two children. The least number of nodes in such tree is n=2h-1 and the maximum number of nodes is $n=2^{h+1}-1$ where h is the height of the tree.
- **Complete Binary tree**: Is a tree in which every level, except probably the last, is completely filled and all nodes are to the left as far as possible.

Types of trees

- **Height Binary Balanced tree:** Any tree with the following characteristics

- The left and right subtrees must differ in heights by at most 1 and
- The left and right subtrees are each balanced

- **Degenerate tree:** Any tree in which each node has only one child along the same path.

Tree Traversals

- Tree traversal is the process of visiting each node of a tree only once
- Traversing a tree always start from the root node
- There are three main methods of tree traversals and these are
    - In-order traversal
    - Pre-order traversal
    - Post order traversal

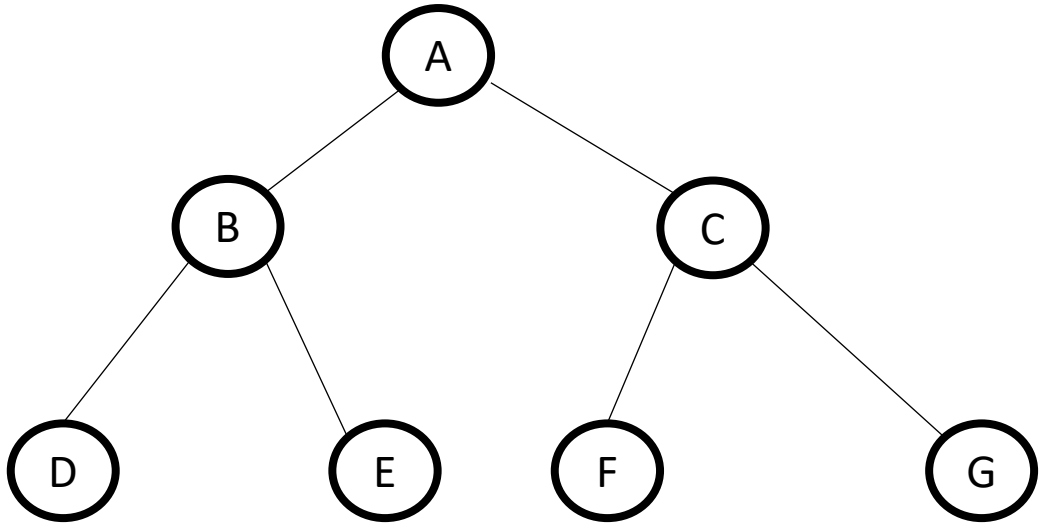
In-Order

In this traversal method, the steps involved are as follows:

- Recursively traverse the **left subtree**
- Visit the **root node**
- Recursively traverse the **right subtree**

Note that in traversing a subtree, the subtree is considered as a tree of it own.

Example of in-order traversal



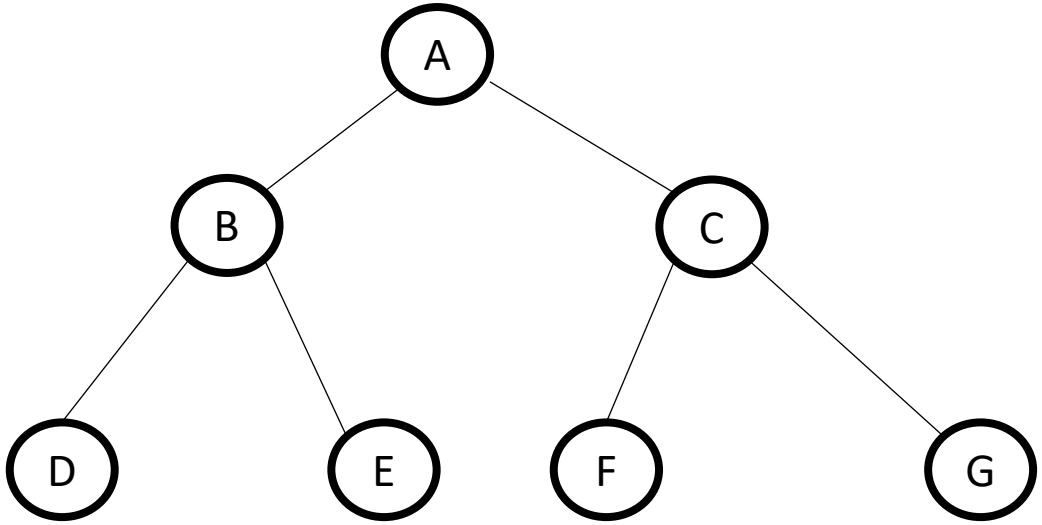
The in-order traversal gives D → B → E → A → F → C → G

Pre-Order

In this traversal method, the steps involved are as follows:

- Visit the **root node**
- Recursively traverse the **left subtree**
- Recursively traverse the **right subtree**

Example of Pre-order traversal



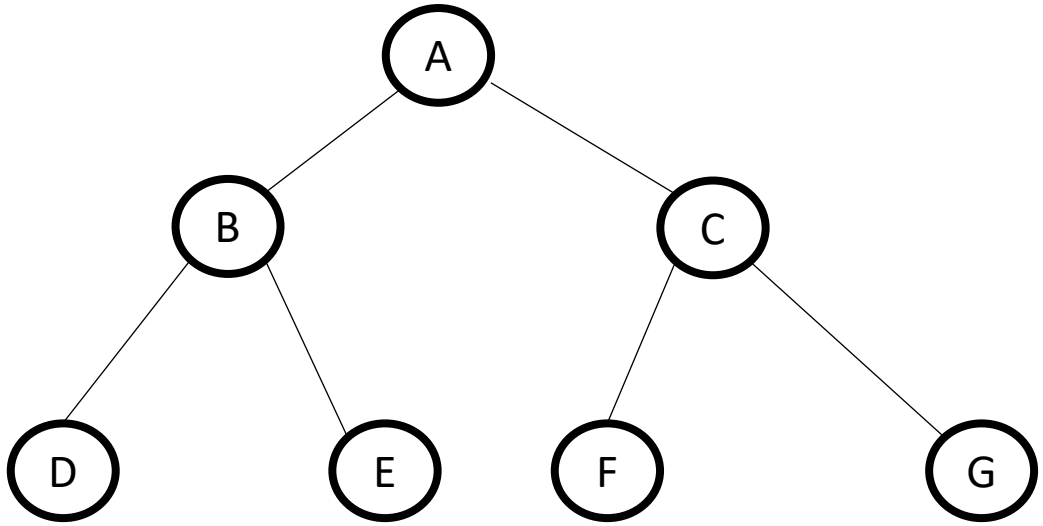
The Pre-order traversal gives A→B→D→E→C→F→G

Post-Order

In this traversal method, the steps involved are as follows:

- Recursively traverse the **left subtree**
- Recursively traverse the **right subtree**
- Visit the **root node**

Example of Post-order traversal



The Post-order traversal gives D → E → B → F → G → C → A

Conversion of Expressions from one form to another

Conversion from Infix to Postfix

- Apply the following operator precedence

-	1	unary minus or negation
^	2	Exponentiation
* / %	3	multiplication division and modulus
+ -	4	addition and subtraction
=	5	equal to

- Fully parenthesize the expression to reflect the operator precedence
- Move each operator to replace the right parenthesis to which it belongs
- Remove the parenthesis

Example of Infix to Postfix

Consider the expression A+B^C-D*E+F/G

Fully parenthesizing gives

(((A+(B^C))-(D*E))+(F/G))

Using the order of evaluation, place each operator to the right of the left operand and remove the parenthesis

step 1	(((A+BC^)-(D*E))+(F/G))
Step 2	(((A+BC^)-DE*)+FG/)
Step 3	ABC^+DE*-FG/+

Infix to Prefix

- Fully parenthesize the expression to reflect the operator precedence
- Move each operator to replace the left parenthesis to which it belongs
- Remove the parenthesis

Example of Infix to Prefix

Consider the expression A+B^C-D*E+F/G

Fully parenthesizing gives

(((A+(B^C))-(D*E))+(F/G))

Using the order of evaluation, place each operator to the left of the left operand and remove the parenthesis

step 1	((A+^BC-(D*E))+(F/G))
Step 2	(((A+^BC)-*DE)+/FG)
Step 3	+-+A^BC*DE/FG

Postfix to Infix

Bear in mind that postfix algorithm is left, right, root

- Scan from right to left for an operator preceded by two consecutive operands
- Place the operator between the two operands end enclosed them by parenthesis.
- Repeat the steps above until all operators are between operands.

Example of postfix to infix

Given the postfix expression ABC^+DE*-FG/+

Step 1: ABC^+DE*-(F/G)+

Step 2: ABC^+(D*E)-(F/G)+

Step 3: A(B^C)+(D*E)-(F/G)+

Step 4: (A+(B^C))(D*E)-(F/G)+

Step 5: ((A+(B^C))-(D*E))(F/G)+

Step 6: (((A+(B^C))-(D*E))+(F/G))

Prefix to Infix

Bear in mind that postfix algorithm is left, right, root

- Scan from left to right for an operator followed by two consecutive operands
- Place the operator between the two operands end enclosed them by parenthesis.
- Repeat the steps above until all operators are between operands.

Example of Prefix to infix

Given the postfix expression +-+A^BC*DE/FG

Step 1: +-+A(B^C)*DE/FG

Step 2: +-(A+(B^C))*DE/FG

Step 3: +-(A+(B^C))(D*E)/FG

Step 4: +((A+(B^C))-(D*E))/FG

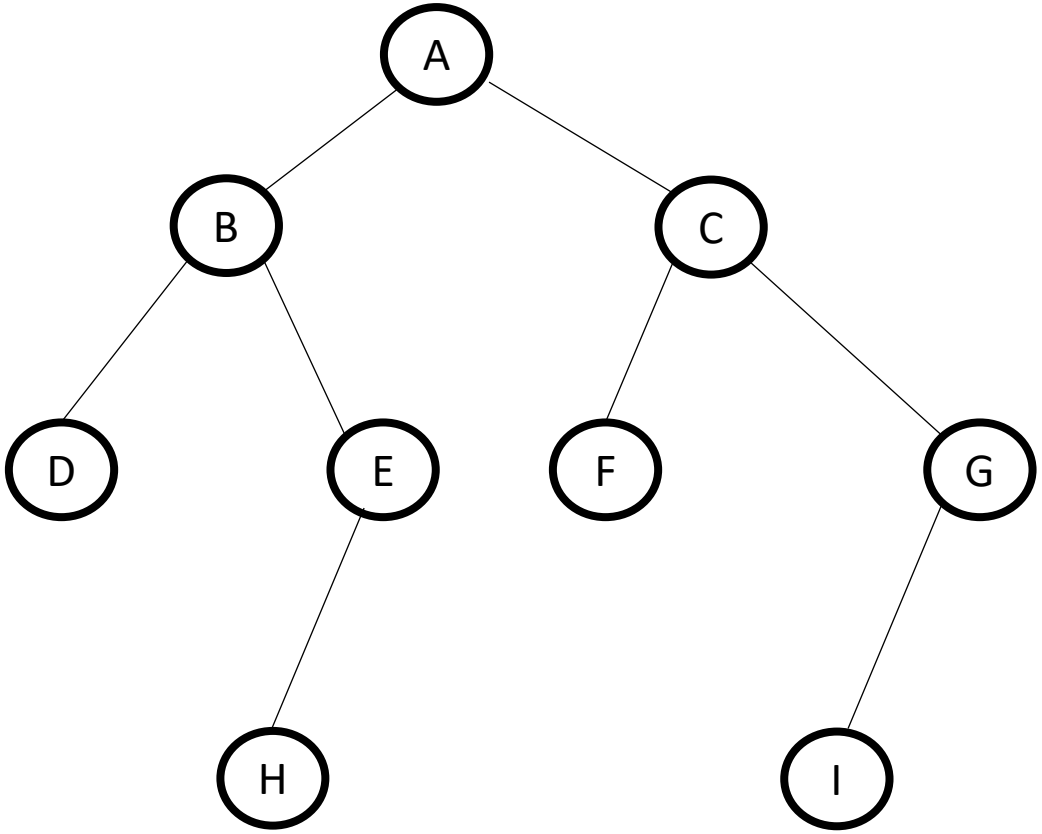
Step 5: +((A+(B^C))-(D*E))(F/G)

Step 6: (((A+(B^C))-(D*E))+(F/G))

Internal Representation of Trees

There are two main ways of Binary Tree-Representation in memory. These are

- Array representation and
- Linked list



Array Representation of Binary trees

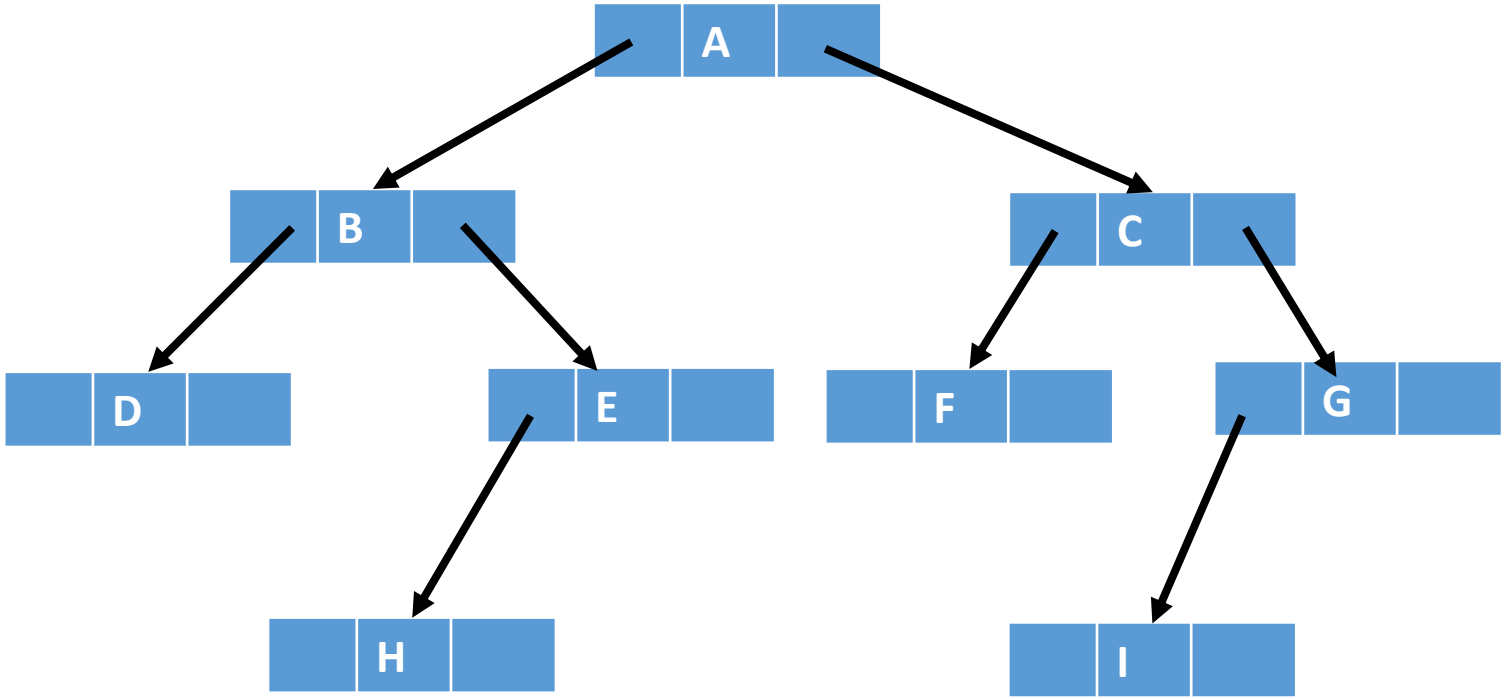
- Binary trees are represented as one-dimensional array
- The root is the first element of the array
- Nodes are stored level by level
- Index of a left child = 2 *index of parent
- Index of a right child = 2*index of parent +1

Thus, the tree above will be represented as follows

1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	B	C	D	E	F	G			H				I

Linked List Representation of Binary Trees

Each node is represented as | Left Child Address | Data | Right Child Address |. The above tree can be represented as follows:



Binary Search Trees

A Binary search tree is a that has the following characteristics

- For each node A, all items in the left sub-tree are smaller than node A
- For each node A, all items in the right sub-tree are bigger than node A

Operations performed on Binary Search Trees

- Searching for a node
- Insertion into a Binary Search Tree
- Deletion from a Binary Search Tree

Searching a Binary Search Tree

i. Compare search data with that of the root

ii. If search data is less then search from the left sub-tree else search from the right sub-tree

iii. Repeat step (ii) until either the node in question is found or a leaf node is encountered. If a leaf node is encountered, it implies there is no node with the data being search for.

Insertion into a Binary Search Tree

i. Compare search data with that of the root

ii. If search data is less then search from the left sub-tree else search from the right sub-tree

iii. Repeat step (ii) until a leaf node is encountered.

iv. If the data is less than that of the leaf node insert a node to the left with the data else insert the node into the right.

Deletion from a Binary Search Tree

To delete a node from a Binary Search Tree there are 4 possible cases

1. No node contains the specified data
2. The node containing the data is a leaf node
3. The node containing the data has only one child
4. The node containing the data has two children

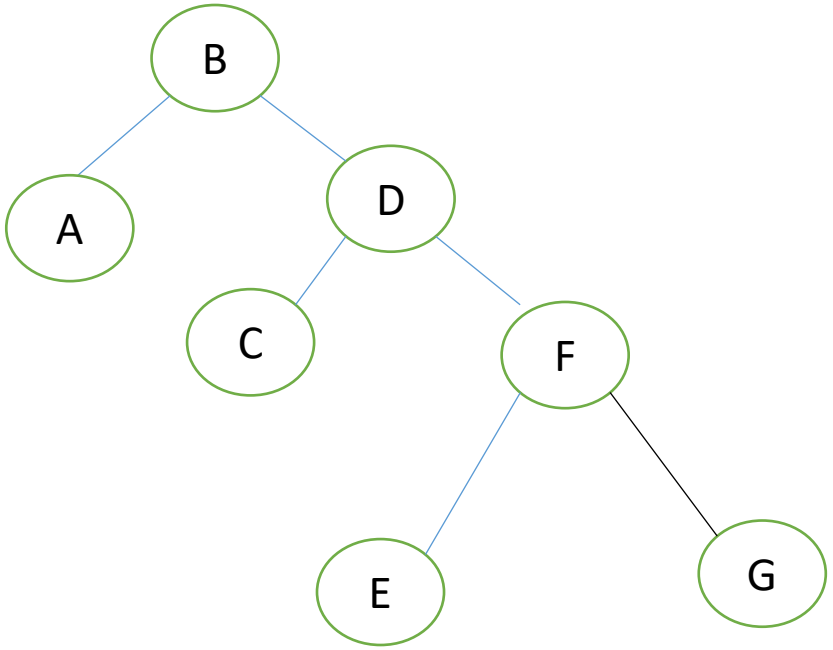
Deletion from a Binary Search Tree

The first three cases are straight forward.

Case 4: When a node has two children then the deletion can be carried out as follows:

i. Find the item which is the inorder successor of the node to be deleted
ii. Delete the node found above and set the parent's pointer to NULL for this item
iii. Replace the node deleted in step (ii) with the one found in step i
iv. Give the left and the right pointers of the node to be deleted to the node now in step (iii)

Example of Deletion from Binary Search Tree



Example of Deletion

Let us consider the deletion of node F

i. The inorder successor of node F is node G

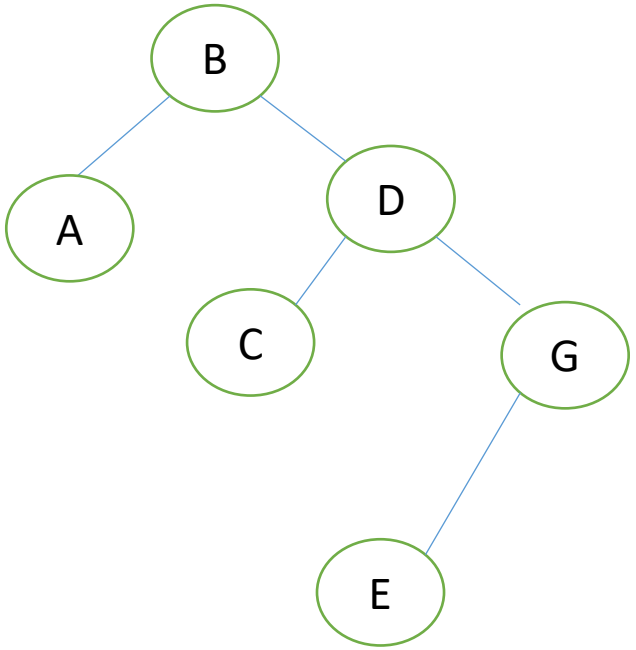
ii. We replace node F with node G.

iii. The left child of F becomes the left child of G

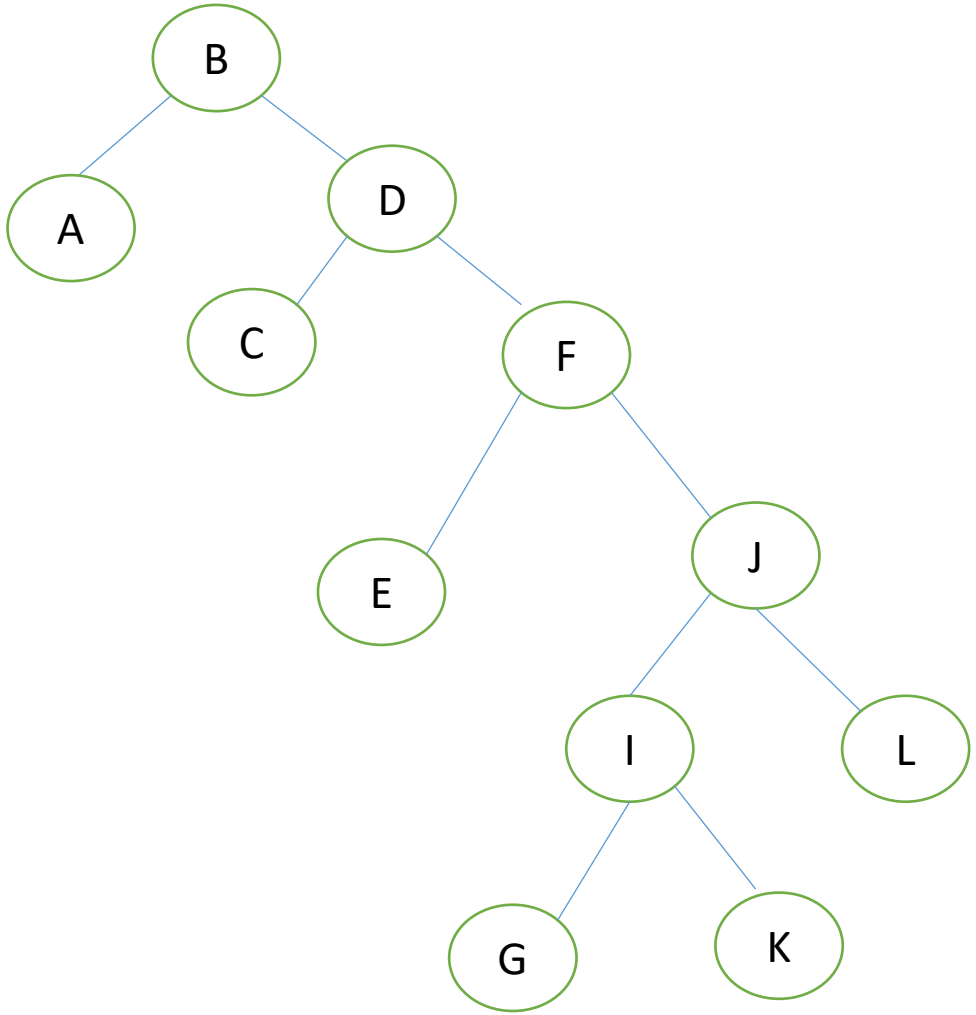
iv. The right link of D will now point to G

Deleting from a Search Tree

After deleting F the result is as follows:



Example Two on Deletion



Let us consider the deletion of note F.

Deleting node F

- The inorder successor of node F is node G

- Replace node F with Node G

- Right link of parent of F, that is D points to G

- The right link of G will now point to the right link of F

Height Balance Trees (AVL)

(i) Assuming we want to create a Binary Search Tree (BST) from three nodes A, B and C, this will give 3! (6) possible trees as shown on the next slide.

(ii) Four of the trees have a height of three and two have 2. The higher the height of a tree the higher the number of comparisons needed to locate a node in the worse case (or when the search node is not on the tree)

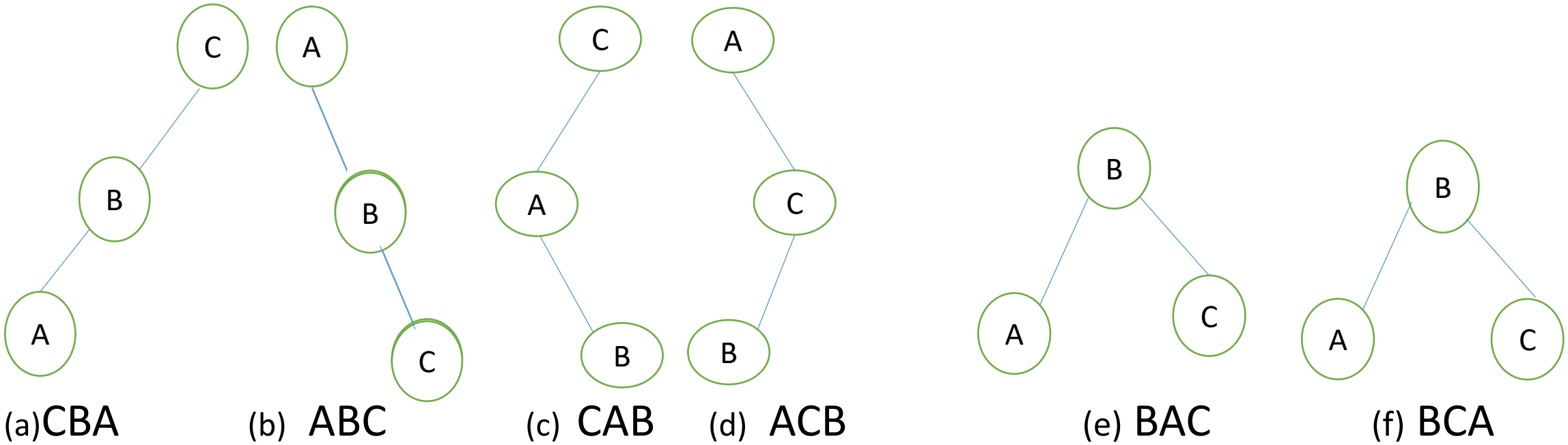
(iii) For the four trees that have a height of 3 (a-d), a single rotation is required or a double rotation (on e and f) can reduce the height of the trees to 2.

(iv) Rotating a node in a tree that results in balancing the height of the tree gives what is known as a height balanced tree.

(v) The main advantage of an AVL over BST is the fact that in most cases the maximum number of comparisons required in locating a node in AVL is lower than BST.

(v) The main disadvantage of AVL over BST is the fact that an additional field is required for each node to store the node's balance factor, hence more memory.

The six (6) possible Binary search trees from A, B and C

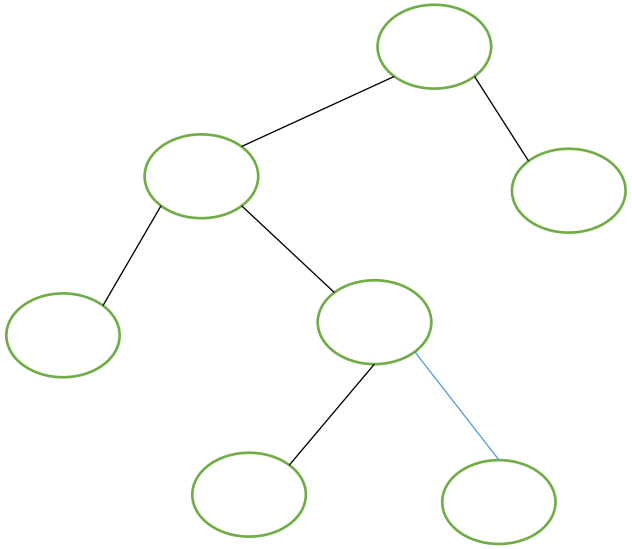


It can be seen from above that the trees (a) to (d) have a height of 3 hence the maximum number of comparisons (height) to locate a node is 3. Trees (e) and (f) have a height of 2 and hence require a maximum of 2 comparisons to locate any node. Since the maximum number of comparisons in (e) and (f) are lower than in (a) to (d), the trees (e) and (f) are said to be a better representation of the nodes for a Binary Search Tree. If we rotate (a) at node C to the right and (b) at node A to the left we obtain the same tree as in (e) or (f) thus reducing the height to 2. For (c), if we apply double rotation, first at A to the left and then at C to the right we obtain the same tree as in (e). Similarly, for (d), if we apply double rotation, first at node B to the right and then to the left at node A we obtain the same tree as (e).

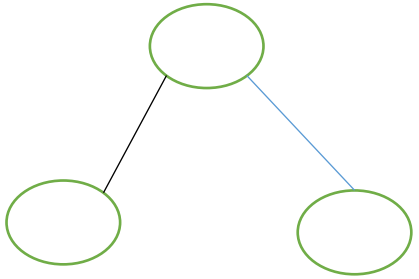
HEIGHT BALANCED TREES (AVL)

A Height Balance Tree is any Binary Tree in which the difference in height between the left and the right subtrees is not more than one in absolute terms for each node.

The height of a node is the longest path from the root to any leaf.



Height of tree is 4



Height of tree is 2

AVL

- For each sub-tree, the difference in height between its left and right subtrees is known as its Height Balance Factor (HBF). An HBF can be either -1, 0 or 1 and is calculated as follows:

  HBF= Height of left Subtree – Height of right subtree

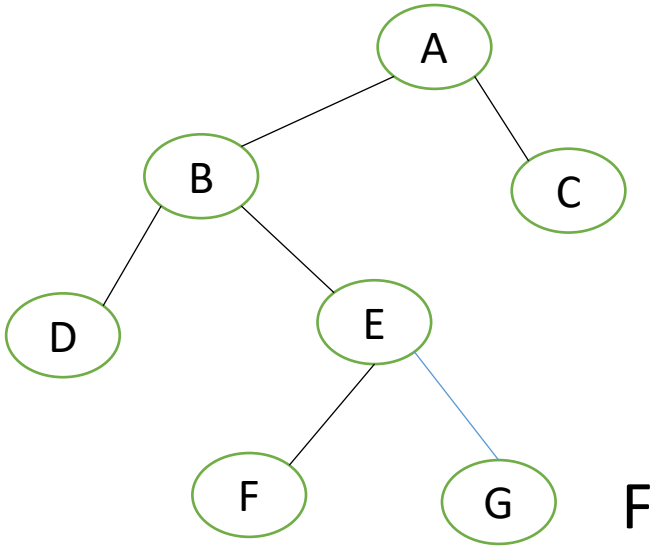


Fig 1.

AVL

From fig.1, the height balance factor of the nodes are

A 4-2=2

B 2-3=-1

C 0-0=0

D 0-0=0

E 1-1=0

F 0-0=1

G 0-0=0

Building an AVL

i. Start with the root
ii. Insert the next node
iii. Calculate the balance factor (BF) of each node
iv. If a node is found to have a $|BF| > 1$ then a rotation is required. Where two nodes have the same $|BF| > 1$ rotate the node on the higher level. To do a rotation, take the node with its $|BF| > 1$, its immediate child and grandchild along the path of insertion.
v. Repeat steps (ii) to (iv) until all nodes have been inserted.

How to perform a rotation

- Let us look at the 4 basic trees that we may have to rotate
- The 4 possible trees that require rotations are as follows:
    - **LL** if the insertion took place along the left-left traversal

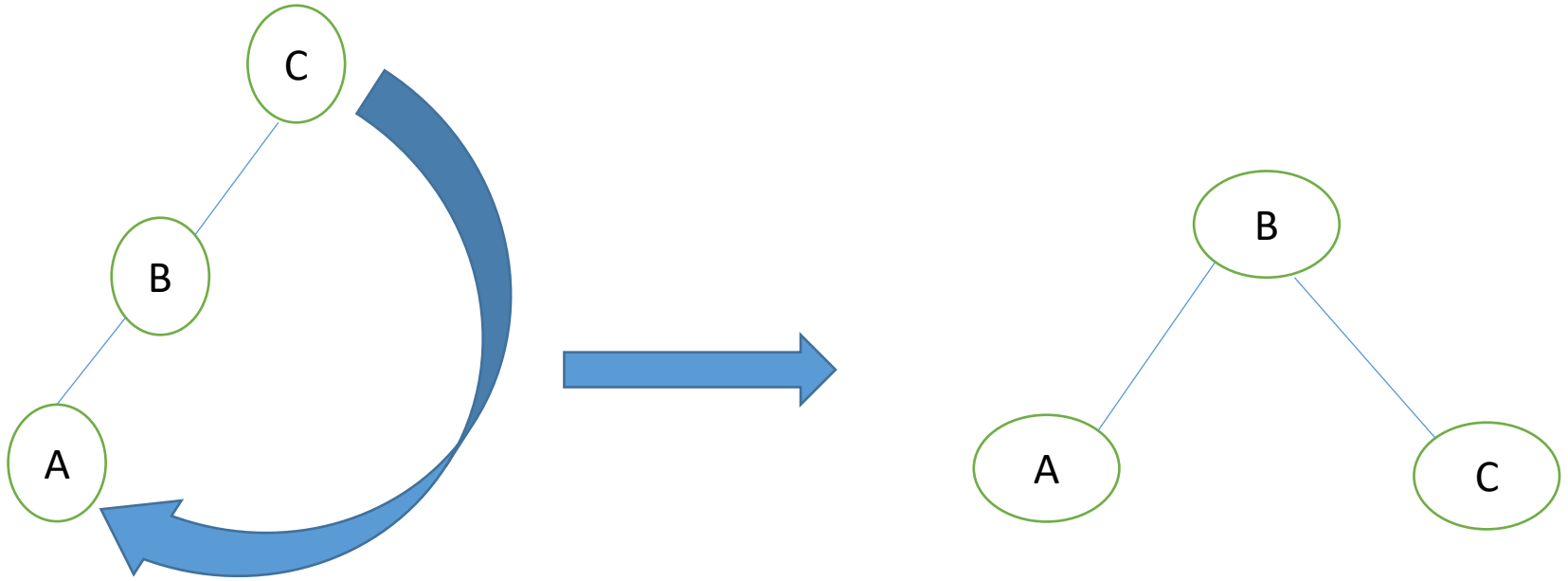
      This requires a single rotation to the right
    - **RR** if the insertion took place along the right-right traversal

      This requires a single rotation to the left
    - **LR** if the insertion is first to the left and then right

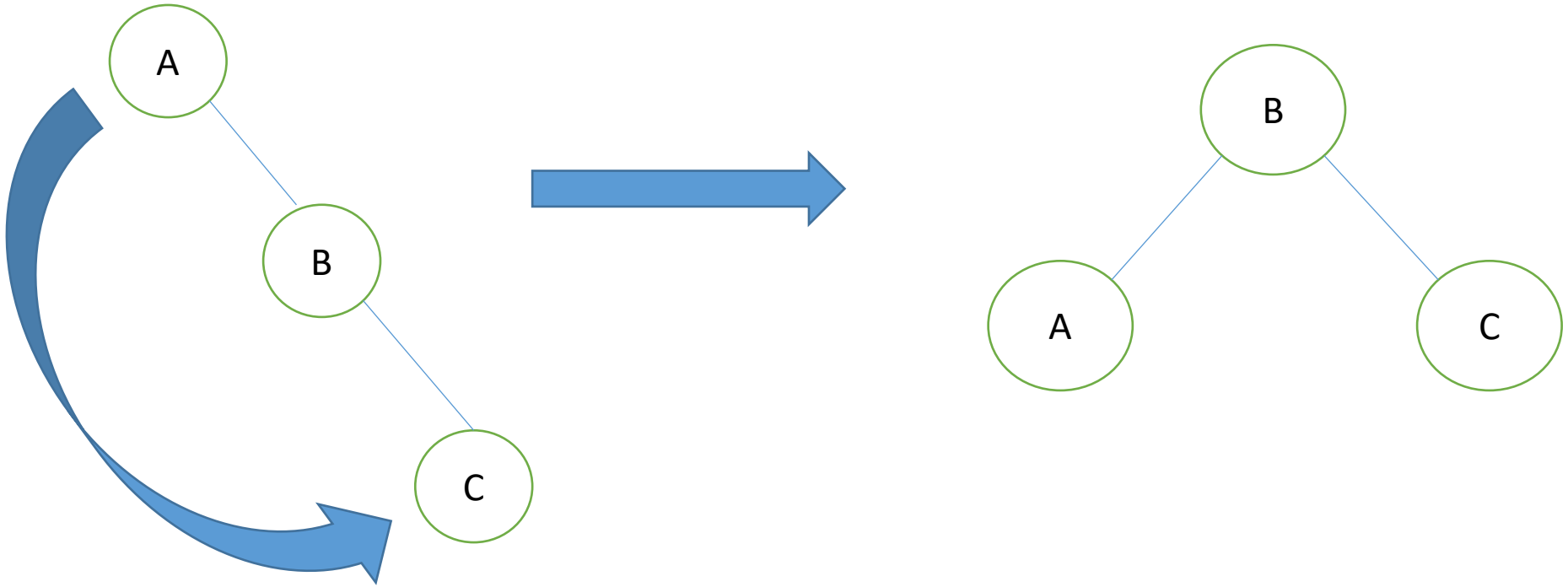
      This requires double rotation, first to the left and then to the right
    - **RL** if the insertion is first to the right and then right

      This requires double rotation, first to the right and then to the left
- The next flow slides shows how each rotation is carried out

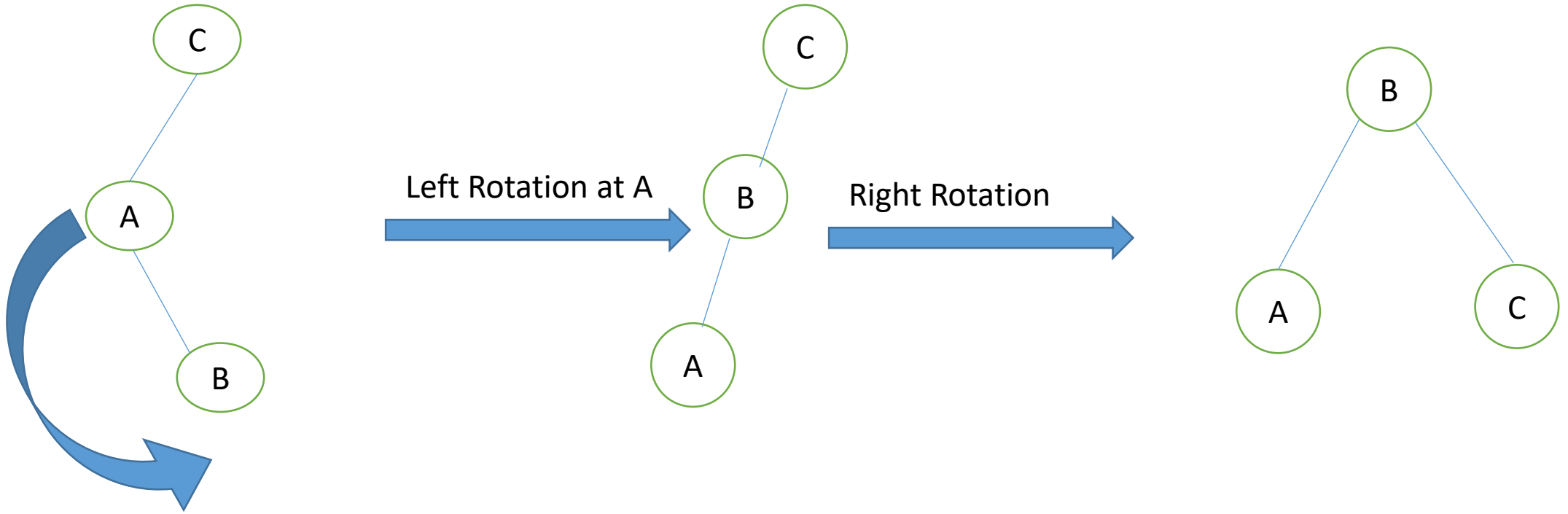
LL insertion requires single right rotation



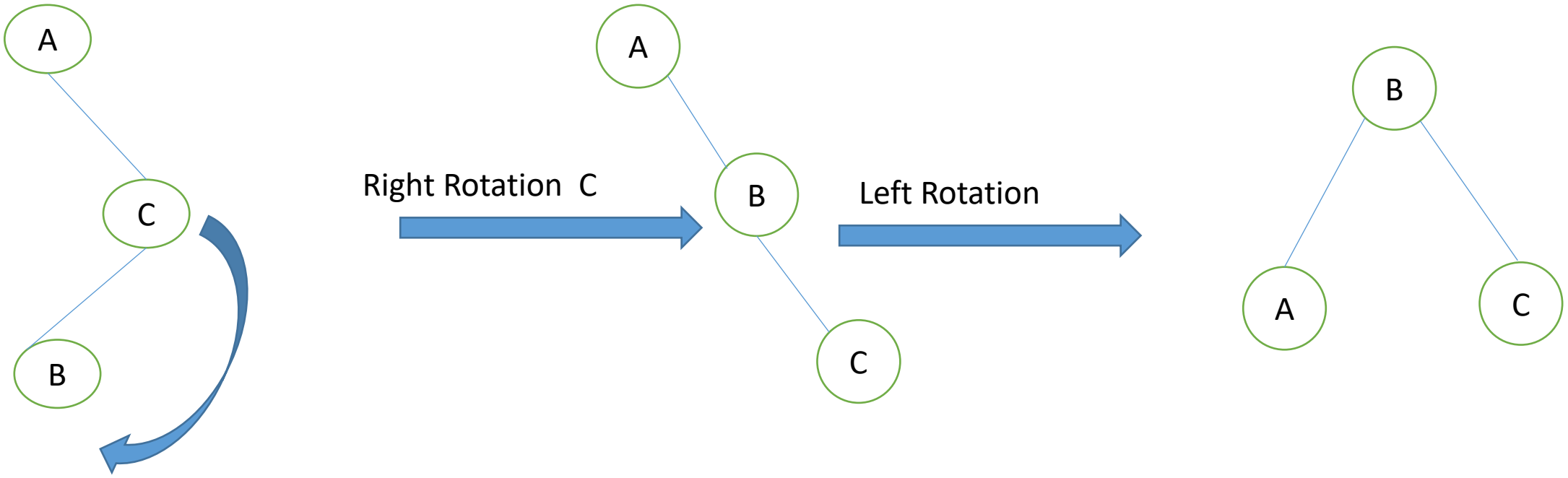
RR insertion requires single left rotation



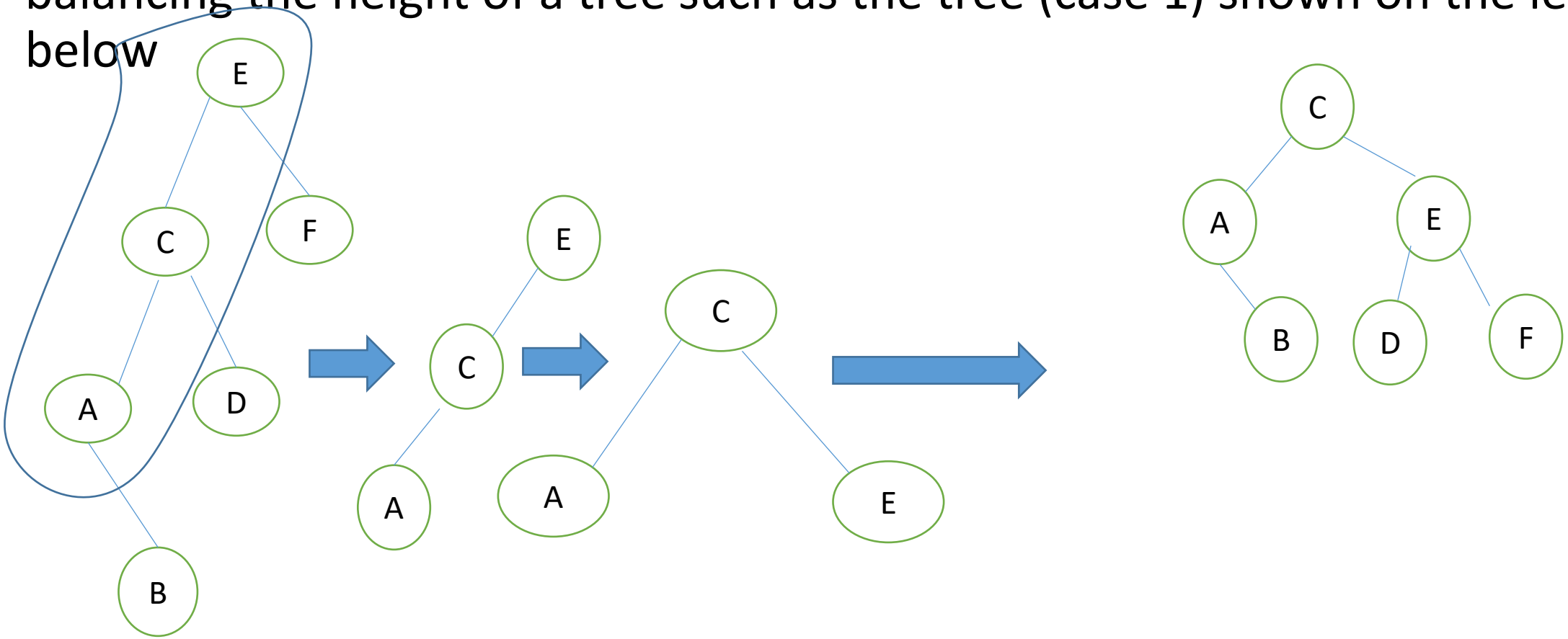
LR insertions require double (LR)rotation



RL insertions require double (RL)rotation



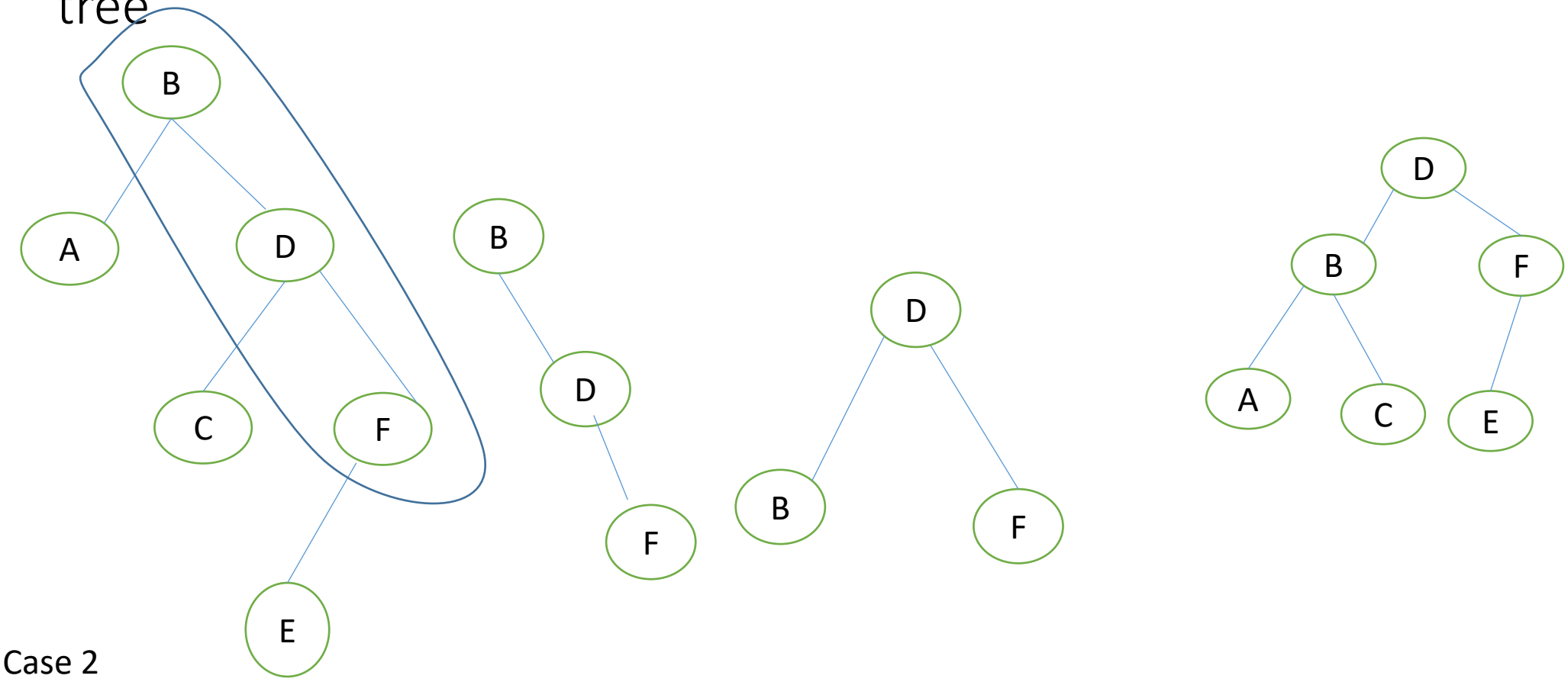
There are two additional possible scenarios that one may encounter in balancing the height of a tree such as the tree (case 1) shown on the left below



Case 1

From the above, it can be seen that the tree is unbalanced at left subtree of node C (C has a balance factor of 2), hence we need to balance the left subtree of node E, that is nodes E, C and A. This is because the last node to be inserted is B. Since ECA is LL, we rotate to the right. After rotation, E becomes the right child of C. The right child of C before the rotation is D, D then becomes the left child of E after rotation

Case 2 is as shown below where the tree is unbalanced at the right sub-tree



Case 2

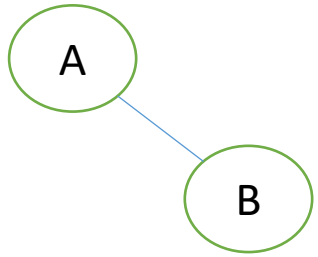
From the above, it can be seen that the tree is unbalanced at RIGHT subtree of node B (B has a balance factor of -2), hence we need to balance the left subtree of node B, that is nodes B, D and F. This is because the last node to be inserted is E. Since ECA is RR, we rotate to the left. After rotation, B becomes the left child of D. The left child of C before the rotation is C, C then becomes the right child of B after rotation

As an example, let us create or build an AVL for the nodes ABCDE

Step 1: insert A

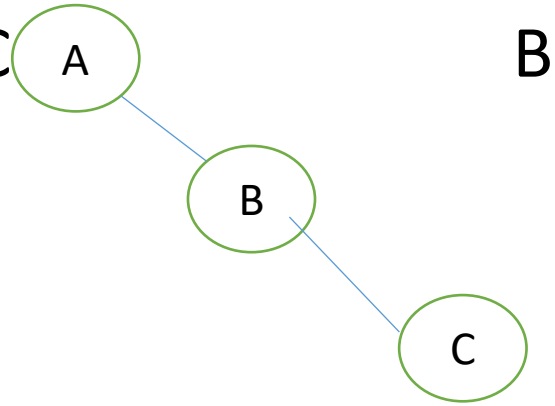
BF of A is 0

Step 2: insert B



BF of A is -1 and that of B is 0

Step 3: Insert C



BF of A = -2

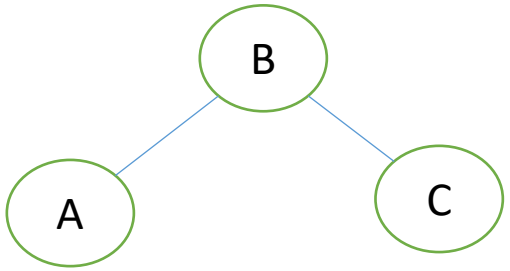
B = -1

C = 0

AVL

Since A has a BF of -2, there is the need for rotation to balance the height of the tree

Step 4: Rotating nodes A,B and C to the left (anti-clockwise) makes A the left child B and C the right child, that is

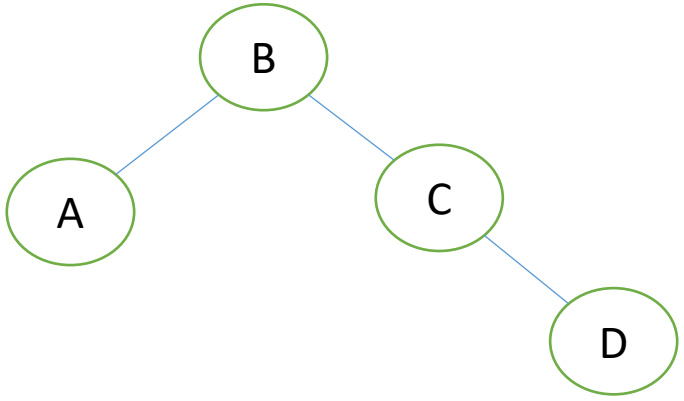


BF of A = 0

B = 0

C = 0

Step 5: Insert D



BF of A = 0

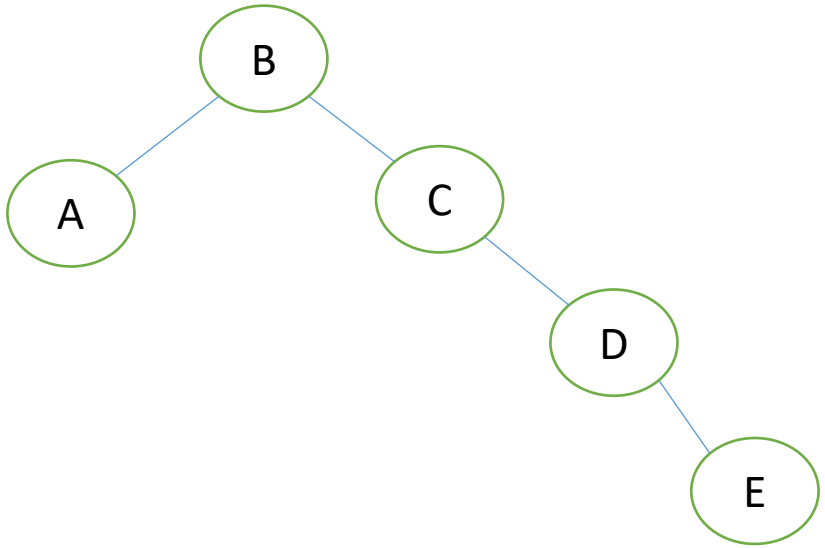
B = -1

C = -1

D = 0

AVL

Step 6: Insert E



BF of A = 0

B = -2

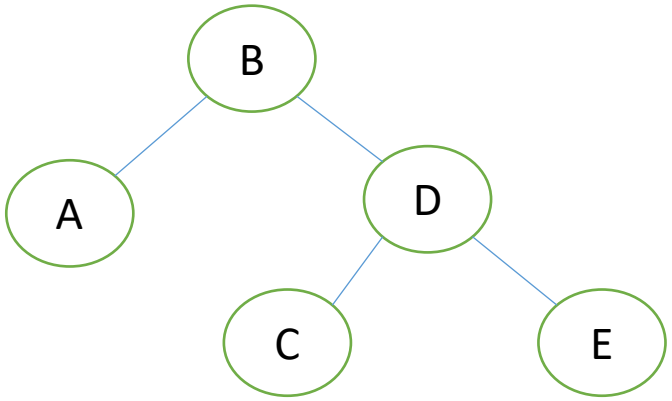
C = -2

D = -1

E = 0

As can be seen from above, the tree is not balanced and hence the need for rotation. Both B and C have BF of -2, however, in such situations, we rotate the one at the higher level, in this case node C. We have a situation of RR hence the need for left rotation. This therefore gives the tree below

AVL



BF of A = 0

B = -1

C = 0

D = 0

E = 0

Thus we have successfully created an AVF tree for ABCDE. Note that the height of the tree is 3. If we had create an ordinary Binary Search Tree the height would have been 5

OPERATIONS ON AVL TREES

The following operations can be performed on an AVL Tree

- Searching an AVL
- Insertion into AVL
- Deletion from AVL

Searching and Insertion

Searching an AVL is not different from searching a Binary Tree

Insertion into an AVL is also not different from Binary Search tree, however, once an insertion occurs, you must ensure that the resulting tree is also an AVL

Deletion from an AVL

The steps involved are as follows:

- Search for the node to be deleted

- Delete the node as you would have done with a Binary tree

- The deletion can cause the tree to lose its AVL property. If that happens, ensure that the AVL property is maintained. Note that the processes of rebalancing the tree is the same as inserting into an AVL

Exercise 1

i. Build an AVL tree from JAMESBNHYFROCQU. Show the step by step generation indicating clearly the height balance factor of each node in each step
ii. Redraw the AVL tree from (i) after removing the root
iii. Redraw the AVL tree from (i) after removing the right child of the root
iv. Redraw the AVL tree from (i) after removing the left child of the root
v. What is the inorder traversal of the tree in (i)
vi. Comment of your result in (v)

HASHING SCHEMES

- Hashing refers to the means of deriving a storage address from a record key

- This type of data organisation is stored in a file known as hash file

- The importance of hashing is to avoid unnecessary comparisons or searches. Each key is usually retrieved by a single search.

- The function that generates the storage address is known as a hashing function. Some of the most commonly used hashing functions are as follows:

Hashing functions

- Division/remainder/modulus
- Mid Square
- Folding (Shift folding and folding at boundaries)

DIVISION/REMAINDER METHODS

- The Primary key is divided by an integer and the remainder used as the storage address for the record.
- If N is the divisor, then the remainder will always generate a value between 0 to N-1 inclusive. Generally, one is added to the remainder, thus the remainder will be between 1 and N inclusive.
- The method works better if N is a prime number by reducing the number of keys generating the same storage address.

DIVISION/REMAINDER METHOD

Example: Given the following primary keys: 20, 15, 40, 28, 44, 52, 23, 16, 25 and 33.

Since the numbers are 10, the closest prime number to 10 is 11. Therefore, using 11 as the divisor gives the following storage addresses:

Original Keys	20	15	40	28	44	52	23	16	25	33
Modulus 11	9	4	7	6	0	8	1	5	3	2
Plus 1	10	5	8	7	1	9	2	6	4	3

Mid Square Method

- In this method, first, the key is multiplied by itself (or squared)
- Select middle few digits of the squared number as the storage address. For example if the records are 100 we can select the two middle keys, for 1000 records we can select the middle 3, etc.

For example, if we have 100 records and some of the keys are 790, 542, 998 and 764 then the storage addresses will be as follows if we use the middle 2 keys

Original Keys	**790**	**542**	**998**	**764**
Square of keys	624100	293764	996004	583696
Middle 2 digits	**41**	**37**	**60**	**36**

Mid Square method – Example two

Let us assume there are 100 records and some of the record keys are 521, 86, 7881 and 108.

The storage addresses will be generated as follows:

Original keys	521	86	998	108
Square of the keys	271441	7396	996004	11664
Possible addresses 1	14	39	60	**16**
possible addresses 2	14	39	60	**66**

Advantage of Mid Square: Its flexible and can easily be modified when necessary

Disadvantage of Mid Square: Does not generate uniform hash values or keys.

Folding Method

- There are two methods of folding namely
- Shift folding and
- folding at boundaries.

SHIFT FOLDING METHOD

If you consider a bank, usually, the primary keys (account numbers) are usually long. Let us consider a typical account number, 5031028902112

To apply shift folding in generating the storage address,

- Each primary key is divided into a number of parts with each part having the same length except the last part
- The numbers in the different parts are then added. Note that the last part which usually have fewer digit is right justified (or shifted to the right)
- If there is a carry over such that the number of digits in generated address is more than that of the parts, the carry over digit is ignored or dropped.

Let us consider the key above, that is 5031028902112. Let us assume there are 1000 records, then 3-digits storage keys can be used for the records, hence we can divide the keys into 3-digits from left to right.

- Dividing the number into five parts gives 503, 102,890,211,2
- Add the parts up as follows:

503
102
890
211
2 → Right shifted
1708

-Since there is a carry over of 1, we drop the one and use 708 as the storage address.

Folding at boundaries method

- First, each primary key is divided into a number of parts with each part having the same length except the last part
- Fold the partition boundaries
- Reverse the digits at the partition boundaries
- Add the digits falling into the same position together
- If there is a carry over, drop the carry over and use the remaining number as the storage key.

Example of Folding at boundaries

Let us consider the key 50360289020012

- We divide the key into five parts as 503, **602,890,200,12.** The parts in red are considered the parts at the boundaries.
- Reverse the digits in the parts at the boundaries, 602 becomes 206 and 200 becomes 2.
- Add up the parts as follows:

503
206
890
2 → Right shifted
12 → Right shifted
1513

- We drop the carry over one and use the 513 as the storage key

Hash Collision

- In hashing, we don’t expect two or more keys to generate the same storage but it does happen, hence the major problem with using hashing schemes.
- When tow or more keys generate the same storage address we say there is a collision.
- Whenever collision occurs we must deal with the hash collision. There are three basic methods used in resolving hash collisions. These are:

  Linear Probing

  Chaining

  Bucketing

Linear Probing Method

The method works by storing the colliding keys into the next available space. If the record keys are 100 then we must set up a hash table consisting of 100 storage locations. The hash table is usually considered as circular. Thus, for N keys, the hash table locations will be 0 to N-1. After the (N-1)th position it will search from the 0^{th} position. The process works as follows:

- Generate the hash table of size N where N is the number of records
- Use the hash function to generate the storage key of a primary key
- Store the primary key at the generated key in the hash table. If the location is already occupied, search sequentially starting from where the collision occurs for a vacant cell.
- Repeat the above steps for the remaining records

Example of Linear probing

Let record keys be 100,56, 78, 2, 45, 39, 229, 387, 254, 22, 48, 53, 62, 49 and 55

- Assume the hash function to be the division or modulus. Since the keys are 15, the closest prime number is 17 hence we use 17 as the divisor.

Record Keys	100	56	78	2	45	39	229	387	254	22	48	53	62	49	55
Modulus 17 + 1	16	6	11	3	12	6	9	14	17	6	15	3	12	16	5

- Since we used modulus 17, the generated storage keys will be between 0 and 16 inclusive. Therefore, our hash table will have its cells labelled from 1 to 17.
- Applying linear probing will give the following:

1	49	Hashed to cell 16 but was already occupied
2		
3	2	
4	53	Hashed to cell 3 but was already occupied
5	5	
6	56	
7	39	Hashed to cell 6 but was already occupied
8	22	Hashed to cell 6 but was already occupied
9	229	
10		
11	78	
12	45	
13	62	Hashed to cell 12 but was already occupied
14	387	
15	48	
16	100	
17	254	

Chaining method

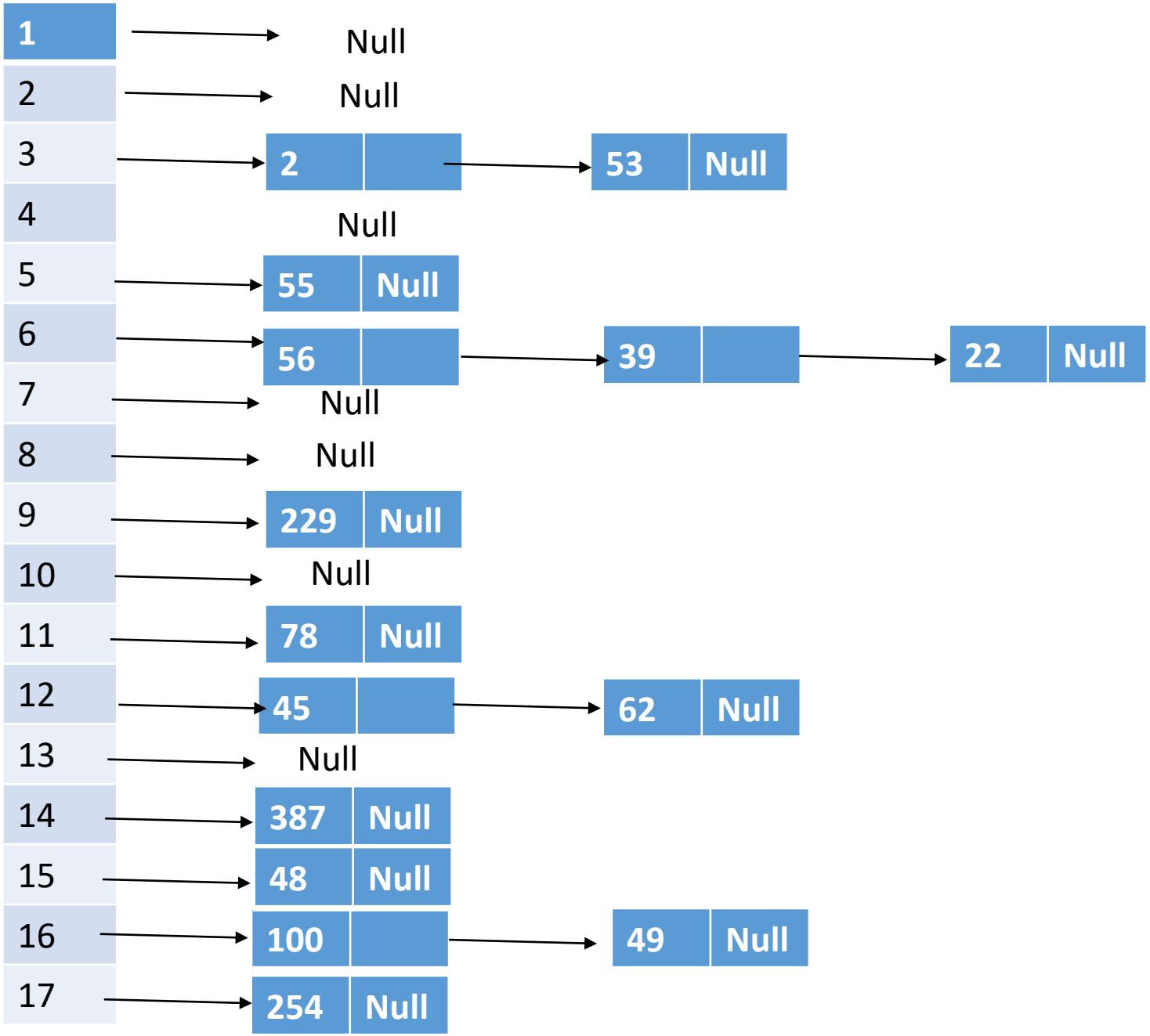
This method builds a linked list of all keys that hash to the same address. The method works better if the linked list are sorted.
During searching, the short linked list is traverse sequentially for the required key.

As an example, let us consider records with the following keys:

100,56, 78, 2, 45, 39, 229, 387, 254, 22, 48, 53, 62, 49 and 55

Assume the hash function to be the division or modulus. Since the keys are 15, the closest prime number is 17 hence we use 17 as the divisor.

	100	56	78	2	45	39	229	387	254	22	48	53	62	49	55
Generated Keys	16	6	11	3	12	6	9	14	17	6	15	3	12	16	5


1
Null
2
Null
3
2
53
Null
4
Null
5
55
Null
6
56
39
22
Null
7
Null
8
Null
9
229
Null
10
Null
11
78
Null
12
45
62
Null
13
Null
14
387
Null
15
48
Null
16
100
49
Null
17
254
Null

BUCKETING METHOD

- This method is very similar to the Chaining but instead of Linked list the bucketing method uses an array at each location in the hash table.
- N (where N is an integer) buckets are usually used. The buckets are labelled from 0 to N-1. The number of keys affects the value of N
- The storage addresses are stored into the buckets based on the last few digits in the generated keys.

Let us consider the primary keys 100,56, 78, 2, 45, 39, 229, 387, 254, 22, 48, 53, 62, 49 and 55 and a bucket size of 10.

Record Keys

Modulus 17 + 1

100	56	78	2	45	39	229	387	254	22	48	53	62	49	55
16	6	11	3	12	6	9	14	17	6	15	3	12	16	5

Since we are only using 10 buckets in this example, our buckets will be labelled from 0 to 9. We will place all primary keys ending with the same digit in the same bucket. Numbers in red are the hash keys.

0					
1	78 **(11)**	empty	empty	empty	empty
2	45 **(12)**	62 **(12)**	empty	empty	empty
3	2 **(3)**	53 **(3)**	empty	empty	empty
4	387 **(14)**	Empty	empty	empty	empty
5	48 **(15)**	55 **(5)**	empty	empty	empty
6	100 **(16)**	56 **(6)**	39 **(6)**	22 **(6)**	49 **(16)**
7	254 **(17)**	Empty	empty	empty	empty
8	empty	empty	empty	empty	empty
9	229 **(9)**	empty	empty	empty	empty