MALLA REDDY UNIVERSITY

MR22-1CS0104

ADVANCED DATA STRUCTURES

II YEAR B.TECH. (CSE) / II – SEM

Unit-1

Introduction - List ADT

Array List: Dynamic Arrays – Implementation and Amortized Analysis.

Positional List: Positions – Positional List ADT Singly Linked List Implementation.

Iterators: The Iterable interface and java's For-Each Loop – Implementing Iterators.

Applications of Stack – Infix to Prefix and infix to Postfix Conversions.

Array

- Array is a collection of elements of same data type referred to by a common name.
- Arrays may be stored in contiguous memory.
- Locations within an array are an integer index.
- the first element of a sequence has index 0, and the last has index n− 1.
- ‘n’ denotes the total number of elements.
- Arrays are objects in Java, we can find their length using the object property “**length”.**

Array

Declare an array:

```
int[] num;
int num[];
```

Initialize an array:

```
int[] num = {10, 20, 30, 40,50};
```

Array Length:

```
System.out.println(num.length);
```

Access the Elements of an Array:

```
num[1];

for (int i = 0; i < num.length; i++)
 System.out.println("Element : "+ arr[i]);
```

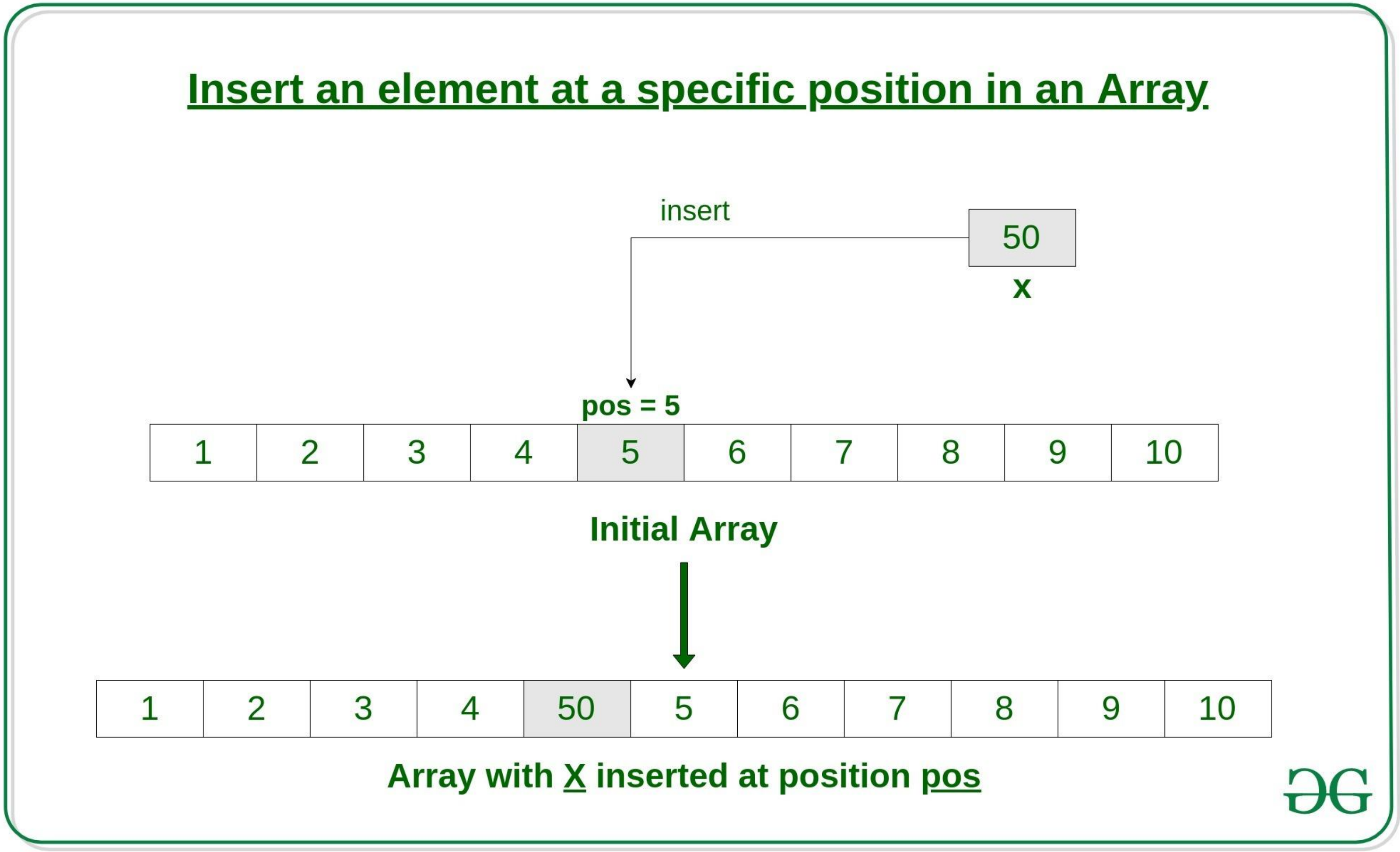
Array

Example:

```
import java.io.*;
class Arr {
        public static void main (String[] args) {
        int [] array1=new int [4];
        array1[0]=10;
        array1[1]=20;
        array1[2]=30;
        array1[3]=40;
        for (int i = 0; i < arr.length; i++)
         System.out.println("Element : "+array1[i]);
        }  }
```

Array - Insert

Insert at any position:



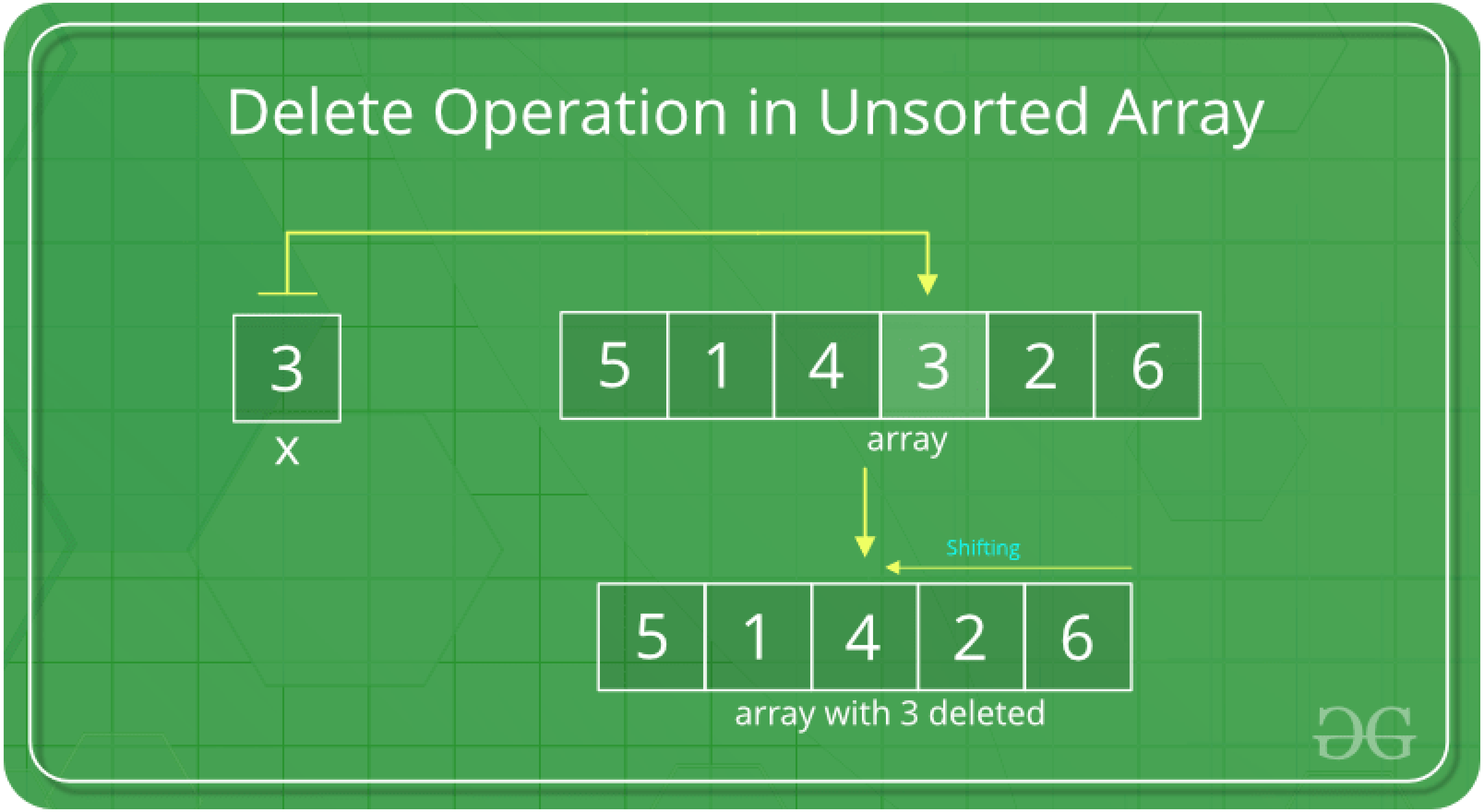
Array - Insert

Example:

```
import java.io.*;
class ArrayInsert {
        static void insertElement(int arr[], int n, int x, int pos)
        {       for (int i = n - 1; i >= pos; i--)
                arr[i + 1] = arr[i];}
                arr[pos] = x;
        public static void main(String[] args)
        {       int arr[] = new int[15];
                arr[0] = 2;
                arr[1] = 4;
                arr[2] = 1;
                arr[3] = 8;
                arr[4] = 5;
                int n = 5;
                int x = 10, pos = 2;
                insertElement(arr, n, x, pos);
                n += 1;
                System.out.print("\n\nAfter Insertion: ");
                for (int i = 0; i < n; i++)
                        System.out.print(arr[i] + " ");    }     }
```

Array - Delete

Delete at any position:



Array - Delete

Example:

```
class Main {
        static int findElement(int arr[], int n, int key)
        {               int i;
                        for (i = 0; i < n; i++)
                        if (arr[i] == key)
                        return i;
                        return -1;    }
        static int deleteElement(int arr[], int n, int key)
        {               int pos = findElement(arr, n, key);
                        if (pos == -1) {
                                System.out.println("Element not found");
                                return n;                       }
                        for (int i = pos; i < n - 1; i++)
                                arr[i] = arr[i + 1];
                        return n - 1;                   }
        public static void main(String args[])
        {               int i;
                        int arr[] = { 10, 50, 30, 40, 20 };
                        int n = arr.length;
                        int key = 30;
                        n = deleteElement(arr, n, key);
                        System.out.println("\n\nArray after deletion");
                        for (i = 0; i < n; i++)
                                System.out.print(arr[i] + " ");         }    }
```

List ADT

- Java defines a general interface, java.util.List, that includes the following index-based methods,
    - size( )
    - isEmpty( )
    - get(i)
    - set(i, e)
    - add(i, e)
    - remove(i)

List ADT

Example:

Method	Return Value	List Contents
add(0, A)	–	(A)
add(0, B)	–	(B, A)
get(1)	A	(B, A)
set(2, C)	"error"	(B, A)
add(2, C)	–	(B, A, C)
add(4, D)	"error"	(B, A, C)
remove(1)	A	(B, C)
add(1, D)	–	(B, D, C)
add(1, E)	–	(B, E, D, C)
get(4)	"error"	(B, E, D, C)
add(4, F)	–	(B, E, D, C, F)
set(2, G)	D	(B, E, G, C, F)
get(2)	G	(B, E, G, C, F)

List ADT

A simplified version of the java.util.List interface

```
public interface List<E>
{
int size( );
boolean isEmpty( );
E get(int i)     throws IndexOutOfBoundsException;
E set(int i, E e)     throws IndexOutOfBoundsException;
void add(int i, E e)  throws IndexOutOfBoundsException;
E remove(int i)     throws IndexOutOfBoundsException;
}
```

List ADT - Example

```
import java.util.*;
class ListAdtExample {
public static void main(String args[])
{      List<String> adt = new ArrayList<>();
       adt.add("Data");
       adt.add("Structures");
       adt.add(1, "Using");
       adt.set(1,"Java");
       System.out.println("Size of the List :" +adt.size());
       System.out.println("Elements of List :" +adt);
       adt.remove(1);
       System.out.println("Elements after removing :" +adt);
      System.out.println("Element in the List at position 1 : "+adt.get(1));
}  }
```

List ADT - Example

Output:

Size of the List :3

Elements of List :[Data, Java, Structures]

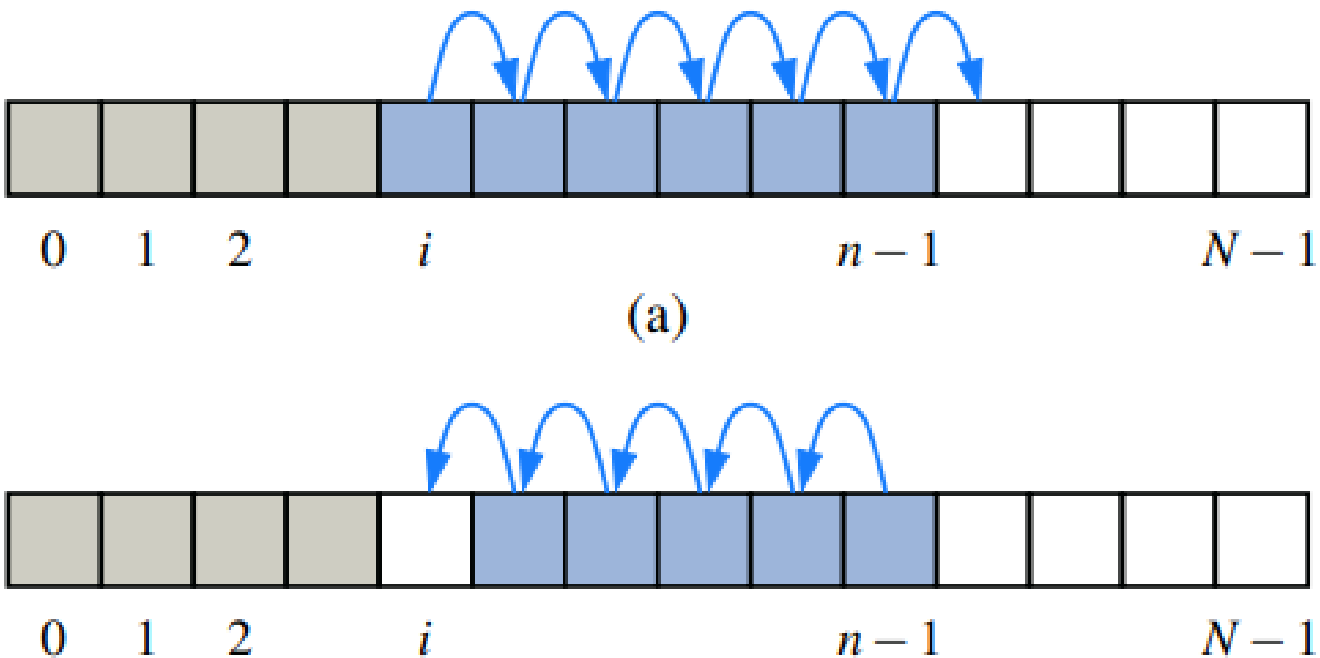
Elements after removing :[Data, Structures]

Element in the List at position 1 : Structures

ArrayList

With a representation based on an array A,

- get(i) and set(i, e) methods are easy to implement
- add(i, e) and remove(i) are more time consuming



ArrayList

The Performance of a Simple Array-Based Implementation:

Method	Running Time
size()	$O(1)$
isEmpty()	$O(1)$
get(i)	$O(1)$
set(i, e)	$O(1)$
add(i, e)	$O(n)$
remove(i)	$O(n)$

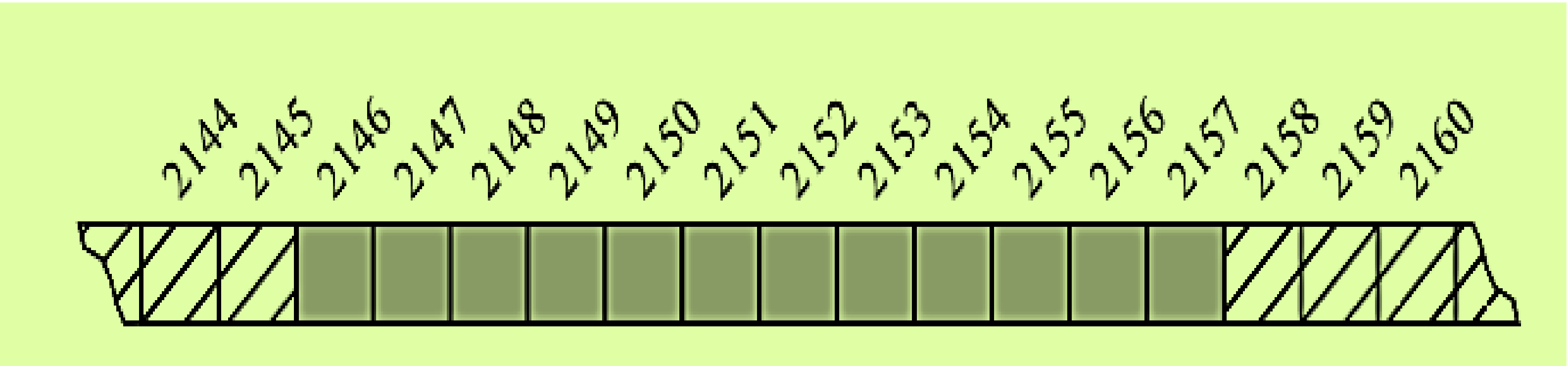
Dynamic Arrays

- An user is unsure of the size of a collection
    - causing an inefficient waste of memory, or
    - causing a fatal error when exhausting that capacity.
- Java’s **ArrayList** class provides a more robust abstraction,
- allows a user to add elements to the list with no limit on the capacity.
- **Dynamic array** - to provide this abstraction.

Dynamic Arrays

- In reality, elements of an ArrayList are stored in a traditional array,
- The precise size of that traditional array must be internally declared.

For example,



- an array with 12 cells that might be stored in memory locations 2146 through 2157 on a computer system.

Dynamic Arrays

- array list instance maintains an **internal array** that often has greater capacity than the current length of the list.
- **if all reserved capacity is exhausted,**
    - the class requests a new, larger array from the system
    - copies all references from the smaller array into the new array
- **the old array** can be **reclaimed** by the system.

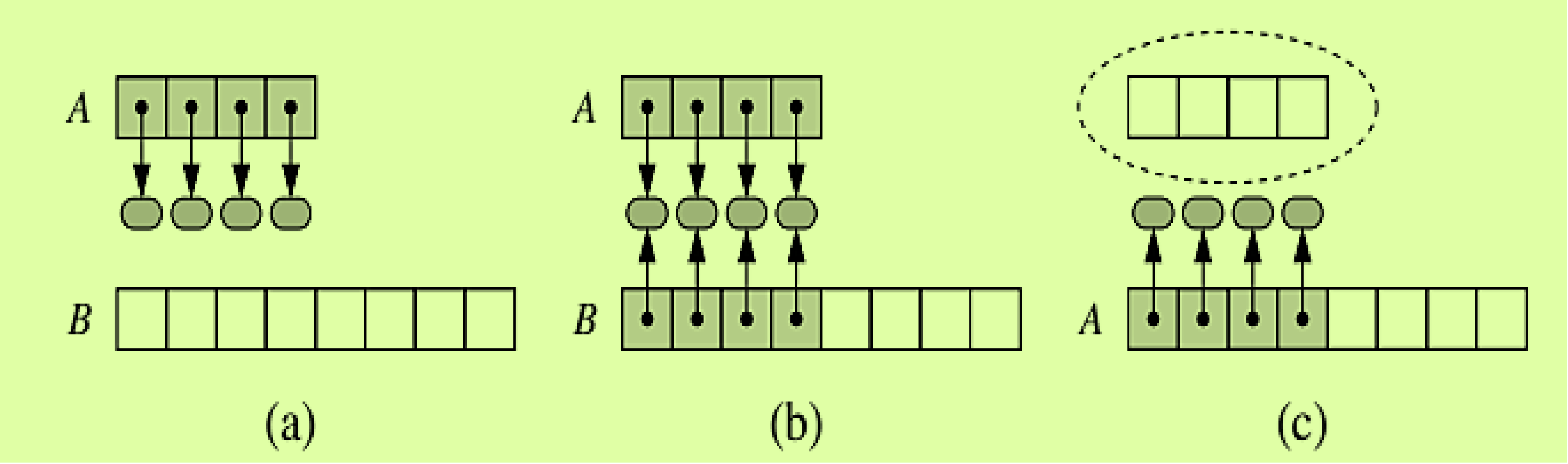
Implementing a Dynamic Array

1. Allocate a new array B with larger capacity.

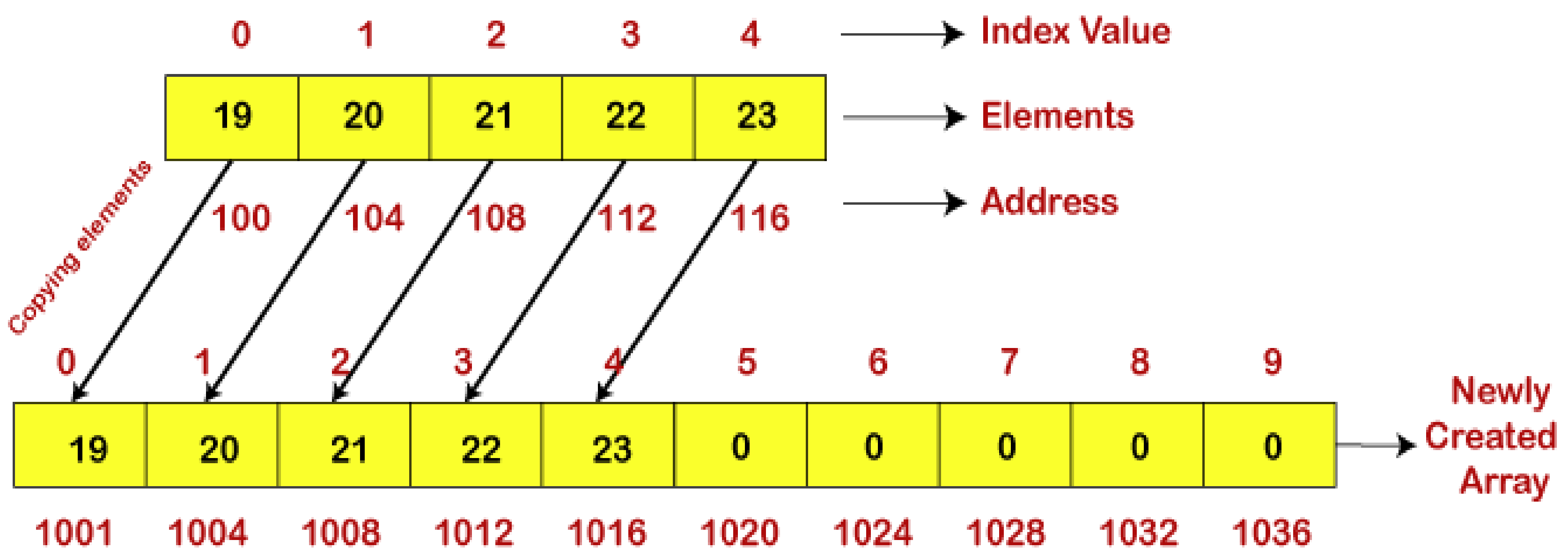
2. Set B[k] = A[k], for k = 0,...,n−1, where n denotes current number of items.

3. Set A = B, that is, we henceforth use the new array to support the list.

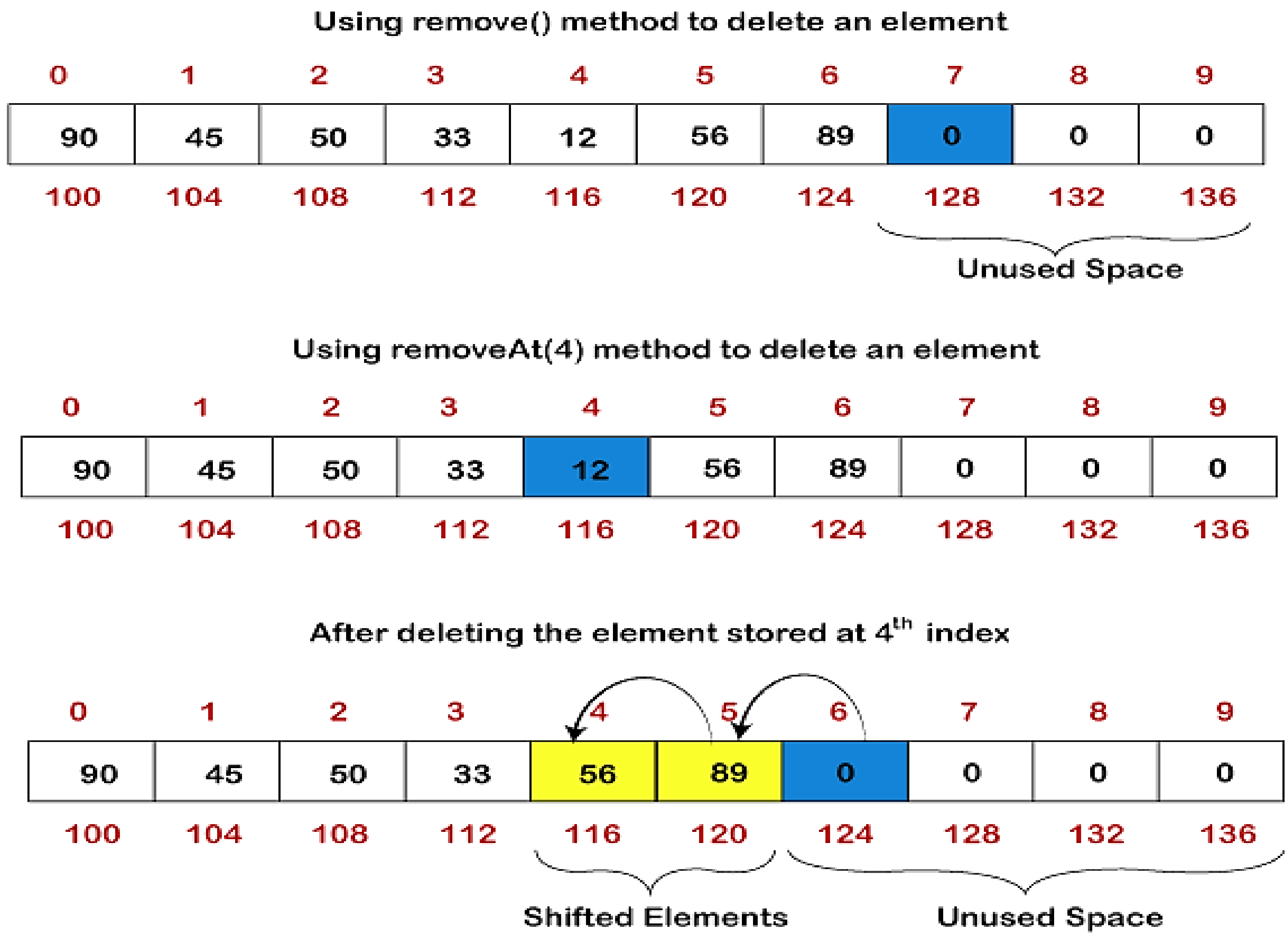
4. Insert the new element in the new array.



Add Element in a Dynamic Array



Delete an element in a Dynamic Array



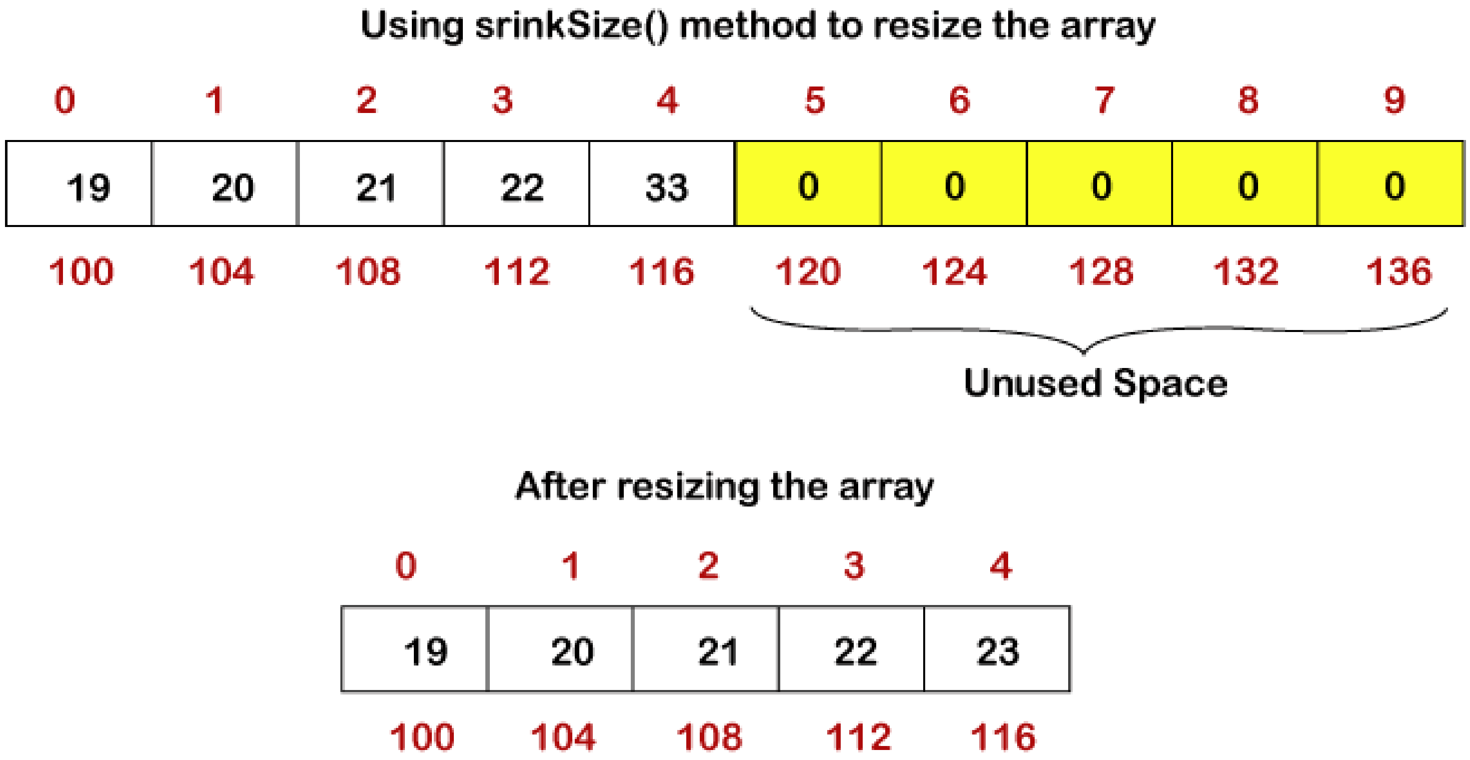
Resizing a Dynamic Array

We need to resize an array in two scenarios if,

- The array uses extra memory than required.
- The array occupies all the memory and we need to add elements.
  - **srinkSize()**
  - **growSize()**

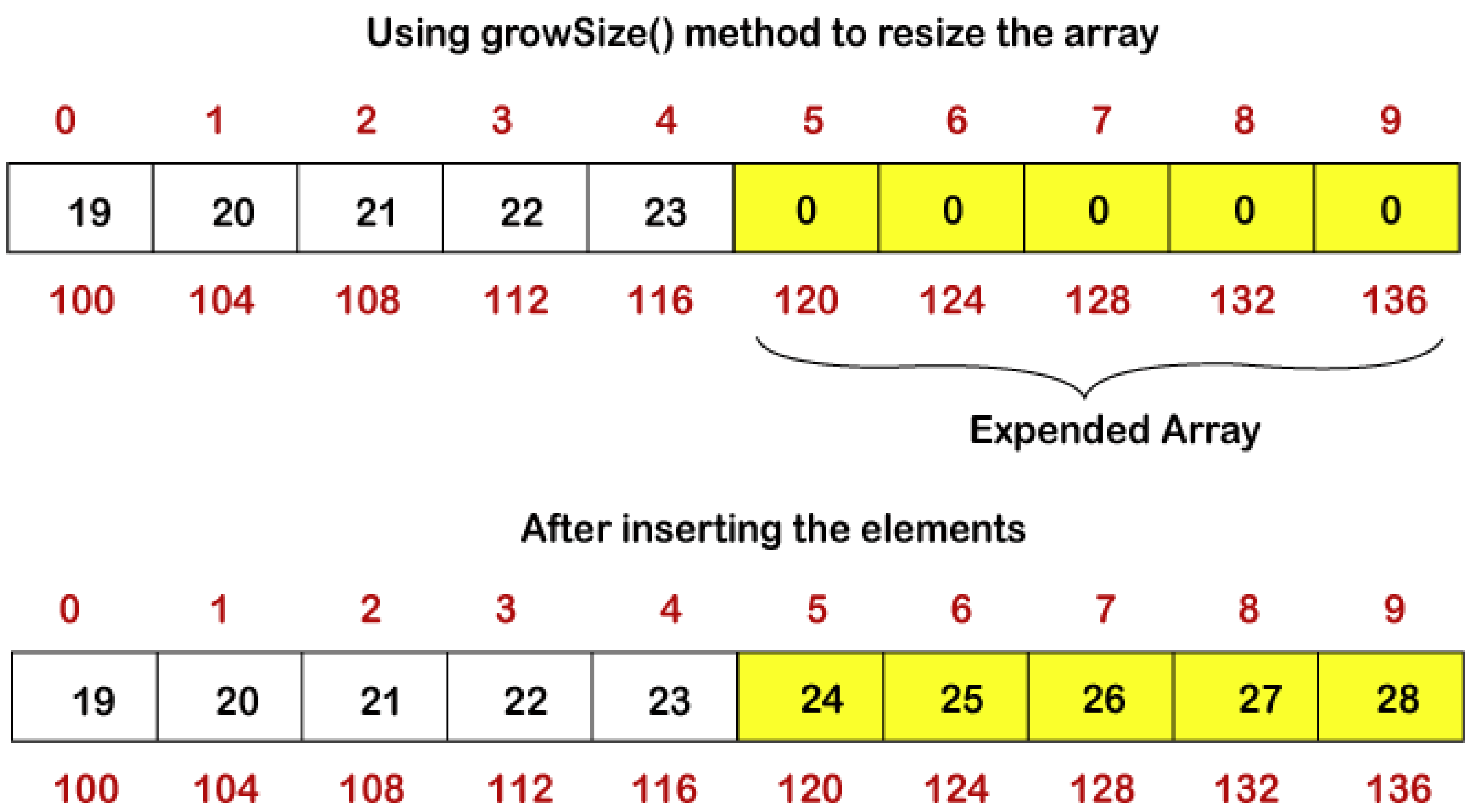
Resizing a Dynamic Array

- **srinkSize()**



Resizing a Dynamic Array

- **growSize()**



Dynamic Array

Creating a Dynamic Array:

```
ArrayList<data-type> arr=new ArrayList<>();
```

Add Element in a Dynamic Array:

```
arr.add(elem);
```

Example:

```
import java.util.ArrayList;
public class Main {
  public static void main(String[] args) {
    ArrayList<Integer> arr = new ArrayList<>();
    arr.add(89);
    arr.add(15);
    System.out.println("The dynamic array is: " + arr);
    System.out.println("Size " + arr.size());  }  }
```

Resizing a Dynamic Array

Internal implementation using array:

```
public void resize() {
    if (arr.length == capacity) {
            int[] narr = new int[2 * capacity];
    }
    for (int i = 0; i < capacity; i++) {
            narr[i] = arr[i];
    }
    arr = narr;
}
```

Dynamic Array

```
import java.util.*;
public class ArrayListExample {
  public static void main(String[] args) {
    ArrayList<Integer> arr = new ArrayList<>();
    System.out.println("Initial Size: " + arr.size());
    arr.add(10);
    arr.add(15);
    arr.add(20);
    arr.add(25);
    arr.add(30);
    System.out.println("Size after addition: " + arr.size());
    System.out.println("The dynamic array is: " + arr);
    System.out.println("First element: " + arr.get(0));
    arr.remove(1);
    System.out.println("The dynamic array is: " + arr);
    System.out.println("Size after deletion: " + arr.size());    }  }
```

Dynamic Array

Output:

Initial Size: 0

Size after addition: 5

The dynamic array is: [10, 15, 20, 25, 30]

First element: 10

The dynamic array is: [10, 20, 25, 30]

Size after deletion: 4

Amortized Analysis of Dynamic Array

- detailed analysis of the running time of operations on dynamic arrays.
- refer to the insertion of an element in an array list.

There are two possible behaviors,

- There is still room in the array and we can simply add the value into the array.
- There is no more room in the array so we need to make a new, larger array, copy over all the elements into the new array, and then finally add the new value.

Amortized Analysis of Dynamic Array

- a single insert operation may require O(n) time to perform.
- amortized order of appending to a dynamic array will depend on how often we have to resize.

When we resize, we have two options for large to make the new internal array:

  - We could increase the size of the internal array by a constant k
  - We could increase the size of the internal array by a constant factor k

Amortized Analysis of Dynamic Array

Increase by a Constant k:

- Start with an array of size k
- Try to add the k+1th element to the list.
- Need to resize.
- Creating a new array of size 2k and copy the k elements over into the new array.
- Add the 2k+1th element, we’ll resize again and copy over 2k elements.
- Add up how many copies we’ll need to make in the process of adding N+1 elements to the array.

Amortized Analysis of Dynamic Array

- Let's assume that worst case:

$$\text{Number of copies} = k + 2k + 3k + \ldots + N = k\left(1 + 2 + 3 + \ldots + \frac{N}{k}\right)$$

- By factoring out the k,

$$k\left(1 + 2 + 3 + \ldots + \frac{N}{k}\right) = k\left(\frac{(\frac{N}{k})(\frac{N}{k} + 1)}{2}\right) \approx O(N^2)$$

- the process of copying over all of the elements while resizing takes O(N2) time.
- Dividing this by the N+1 insertions, the amortized cost for each call to add() would be:

$$\frac{O(N^2)}{N + 1} \approx O(N)$$

Amortized Analysis of Dynamic Array

Increase by a Constant Factor k:

- Generalized to Any Factor (k>1)

$$k^0 + k^1 + k^2 + \ldots + N = \sum_{i=0}^{\log_k N} k^i$$

- This is a geometric sum and the formula for a partial geometric sum is:

$$\sum_{i=0}^{\log_k N} k^i = \frac{k^{\log_k N+1} - 1}{k-1} = \frac{k^{\log_k N} k + 1}{k-1} = \frac{Nk+1}{k-1} \approx O(N)$$

- Thus, resizing by any factor k also results in O(N) time for resizing.

Positional Lists

- indices are not a good abstraction for describing a more local view of a position in a sequence.
- because the index of an entry changes over time due to insertions or deletions.
- goal is to design an abstract data type that provides a user a way to refer to elements anywhere in a sequence.
- **Example**, a text document.
- develop our own abstract data type that we denote as a **positional list**.

Positional Lists

- Positional list is an abstraction, and need not rely on a linked list for its implementation.
- In defining the positional list ADT, the concept of a **position** is introduced.
- **Position** formalizes the intuitive notion of the “location” of an element relative to others in the list.

Positions

- **getElement():** Returns the element stored at this position.
- A position p, associated with some element e in a list L, does not change, even if the index of e changes in L due to insertions or deletions elsewhere in the list.
- The only way in which a position becomes invalid is if that position and its element are explicitly removed from the list.

The Positional List Abstract Data Type

- The accessor methods includes,
  - first( ): Returns the position of the first element of L or null
  - last( ): Returns the position of the last element of L or null
  - before(p): Returns the position of L immediately before position p
  - after(p): Returns the position of L immediately after position p
  - isEmpty( ): Returns true if list L does not contain any elements.
  - size( ): Returns the number of elements in list L.
  - An error occurs if a position p is not a valid position for the list.

The Positional List Abstract Data Type

- first( ) and last( ) methods return the associated positions, not the elements.
- The first element of a positional list can be determined by,

  first( ).getElement;

- A typical traversal of a positional list

```
Position<String> cursor = guests.first( );
while (cursor != null)
{
System.out.println(cursor.getElement( ));
cursor = guests.after(cursor);
}
```

The Positional List Abstract Data Type

Updated Methods of a Positional List,

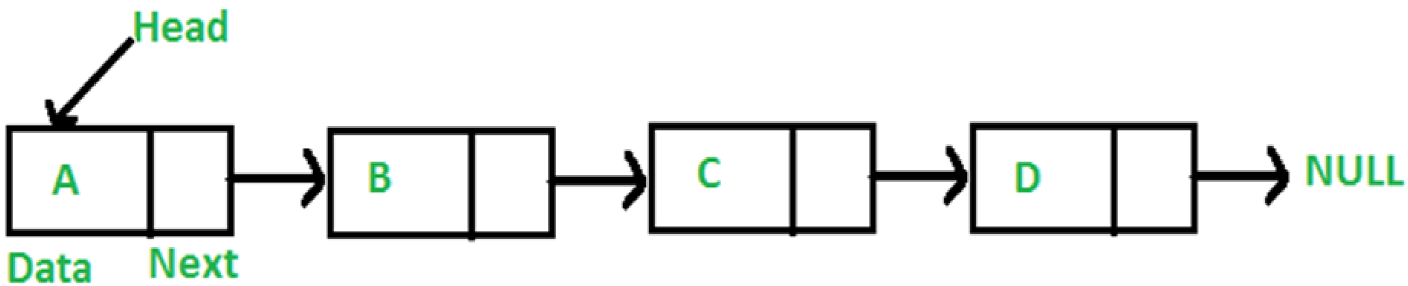
- addFirst(e)
- addLast(e)
- addBefore(p, e)
- addAfter(p, e)
- set(p, e)
- remove(p)

The Positional List Abstract Data Type

Method	Return Value	List Contents
addLast(8)	p	(8_p)
first()	p	(8_p)
addAfter(p, 5)	q	$(8_p, 5_q)$
before(q)	p	$(8_p, 5_q)$
addBefore(q, 3)	r	$(8_p, 3_r, 5_q)$
r.getElement()	3	$(8_p, 3_r, 5_q)$
after(p)	r	$(8_p, 3_r, 5_q)$
before(p)	null	$(8_p, 3_r, 5_q)$
addFirst(9)	s	$(9_s, 8_p, 3_r, 5_q)$
remove(last())	5	$(9_s, 8_p, 3_r)$
set(p, 7)	8	$(9_s, 7_p, 3_r)$
remove(q)	"error"	$(9_s, 7_p, 3_r)$

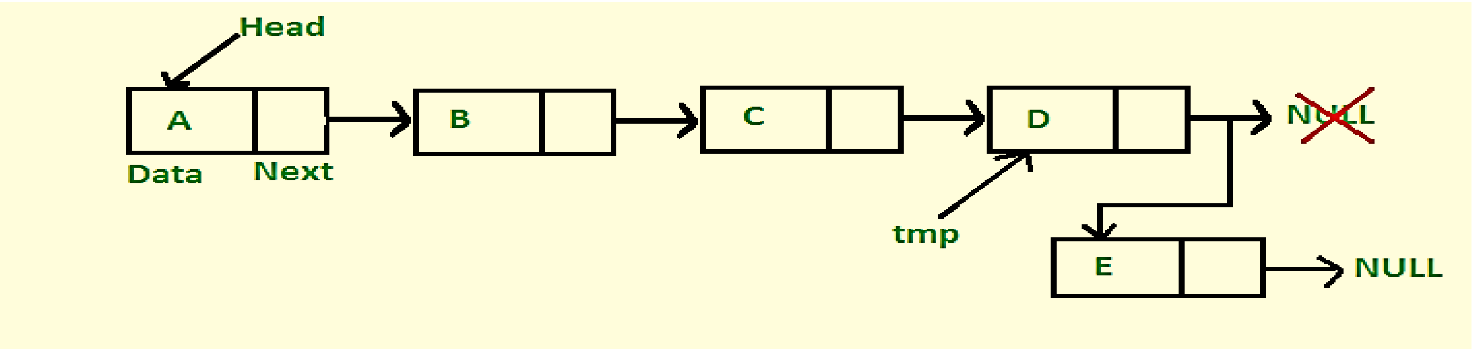
Implementing a Single Linked List

- linear data structure in which each element of the list contains a pointer which points to the next element in the list.
- Each element in the singly linked list is called a node.



Implementing a Single Linked List

Creation and Insertion:




```
class Node {
        int data;
        Node next;
        public Node(int d)
        {       this.data = d;
                this.next=null;
        }   }
```

Implementing a Single Linked List

Insertion:

```
public void insert(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
        } else {
            Node current = head;
            while (current.next != null) {
                current = current.next;
            }
            current.next = newNode;
        }    }
```

Implementing a Single Linked List

Traversal:

For traversal, below is a general-purpose function printList() that prints any given list by traversing the list from head node to the last.

```
public void traverse() {
    Node current = head;
    while (current != null) {
        System.out.print(current.data + " ");
        current = current.next;
    }
    System.out.println();
  }     }
```

Implementing a Single Linked List

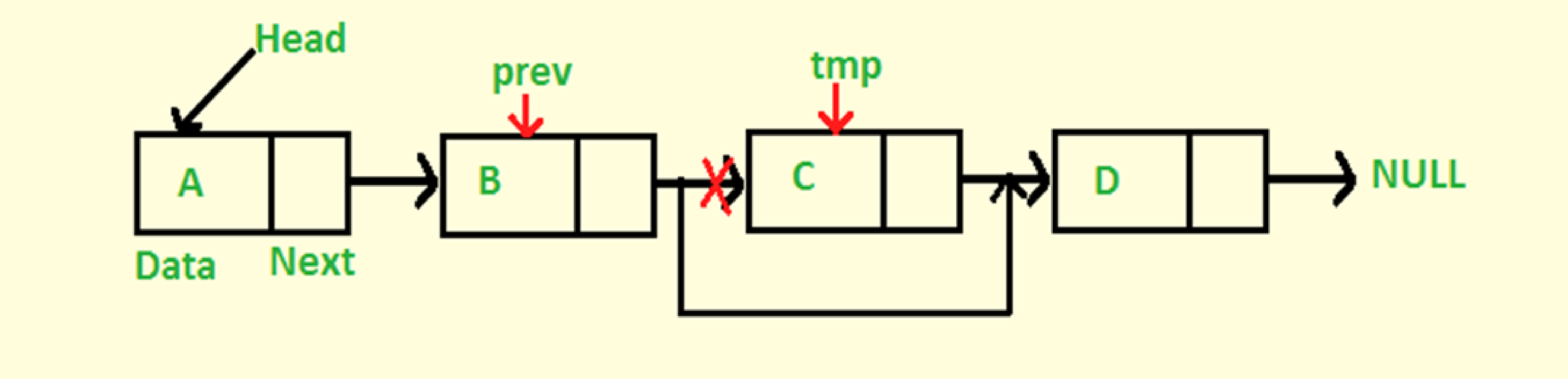
Deletion By KEY:

1. Search the key for its first occurrence in the list
2. Now, Any of the 3 conditions can be there:

Case 1: The key is found at the head

Case 2: The key is found in the middle or last, except at the head

Case 3: The key is not found in the list



Implementing a Single Linked List

Deletion By KEY:

```
public void delete(int data) {
        if (head == null) {
          return;            }
        if (head.data == data) {
        head = head.next;
        return;   }
        Node current = head;
         while (current.next != null && current.next.data != data) {
        current = current.next;        }
        if (current.next != null) {
        current.next = current.next.next;
            }    }
```

Iterators

- the process of scanning through a sequence of elements, one element at a time.
- Java defines the **java.util.Iterator** interface with the following methods:
    - **hasNext( )**
    - **next( )**
    - **remove( )**
- If the next( ) method of an iterator is called, when no further elements are available, a **“NoSuchElementException”** is thrown.
- hasNext( ) method can be used to detect that condition before calling next( ).
- The combination of these two methods allows a general loop construct for processing elements of the iterator.

Iterators

Example:

```
while (iter.hasNext( ))
{
String value = iter.next( );
System.out.println(value);
}
```

The Iterable Interface and Java's For-Each Loop

- A single iterator instance supports only one pass through a collection
- there is no way to "reset" the iterator back to the beginning of the sequence.
- Java defines another parameterized interface, named Iterable.

  iterator( )
- Java's Iterable class also supports the **"for-each"** loop syntax.

```
for (ElementType variable : collection) {
        loopBody        // may refer to "variable"        }
```

Syntax is shorthand for,

```
Iterator<ElementType> iter = collection.iterator( );
while (iter.hasNext( )) {
ElementType variable = iter.next( );
loopBody // may refer to "variable"     }
```

The Iterable Interface and Java's For-Each Loop

- Iterator's remove method cannot be invoked when using the for-each loop syntax. Instead, we must explicitly use an iterator.

```
ArrayList<Double> data;
Iterator<Double> walk = data.iterator( );
while (walk.hasNext( ))
if (walk.next( ) < 0.0)
walk.remove( );
```

Implementing Iterators

- Two general styles for implementing iterators,
- **A snapshot iterator -** maintains copy of the sequence of elements, which is constructed at the time the iterator object is created
- It requires O(n) time and O(n) auxiliary space.
- **A lazy iterator -** does not make an upfront copy, instead performing a piecewise traversal only when the next( ) method is called.
- It requires only O(1) space and O(1) construction time.

Iterations with the ArrayList class

Iterate through ArrayList using for-each loop:

```
import java.util.ArrayList;
class Main {
  public static void main(String[] args) {
    ArrayList<String> languages = new ArrayList<>();
    languages.add("Java");
    languages.add("JavaScript");
    languages.add("Python");
    System.out.println("ArrayList: " + languages);
    System.out.println("Iterating over ArrayList using for-each loop:");
    for(String language : languages) {
      System.out.print(language);
      System.out.print(", ");    }   }    }
```

Iterations with the ArrayList class

Iterate through ArrayList using listIterator():

```
import java.util.ArrayList;
import java.util.ListIterator;
class Main {
  public static void main(String[] args) {
    ArrayList<Integer> numbers = new ArrayList<>();
    numbers.add(1);
    numbers.add(3);
    numbers.add(2);
    System.out.println("ArrayList: " + numbers);
    ListIterator<Integer> iterate = numbers.listIterator();
    System.out.println("Iterating over ArrayList:");
    while(iterate.hasNext()) {
      System.out.print(iterate.next() + ", ");    }  }   }
```

Iterations with the LinkedPositionalList class

- standard iterator( ) method return an iterator of the elements of the list.
- positions( ) to iterate through the positions of a list.

for (Position<String> p : waitlist.positions( ))

- We define three new inner classes.
- PositionIterator, providing the core functionality of our list iterations.
- PositionIterable inner class
- ElementIterator class

Iterations with the LinkedPositionalList class

Methods:

- public int size()
- public boolean isEmpty()
- public Position<E> first()
- public Position<E> last()
- public Position<E> before(Position<E> p) throws IllegalArgumentException
- public Position<E> after(Position<E> p) throws IllegalArgumentException
- public Position<E> addFirst(E e)
- public Position<E> addLast(E e)
- public Position<E> addBefore(Position<E> p, E e) throws IllegalArgumentException
- public Position<E> addAfter(Position<E> p, E e) throws IllegalArgumentException
- public E remove(Position<E> p) throws IllegalArgumentException
- public Iterable<Position<E>> positions()
- public Iterator<E> iterator()

Applications of Stack

- Stack is a simple linear data structure used for storing data.
- Stack follows the LIFO
- It can be implemented through an array or linked lists.
- Some of its main operations are: push(), pop(), top(), isEmpty(), size(), etc.

Applications:

- Expression conversion
- Function calls and recursion
- Undo/Redo operations
- Expression evaluation
- Browser history
- Balanced Parentheses

Applications of Stack

Expression Conversion:

- Infix to Postfix conversion
- Infix to Prefix conversion

Notations for Arithmetic Expression:

There are three notations to represent an arithmetic expression:

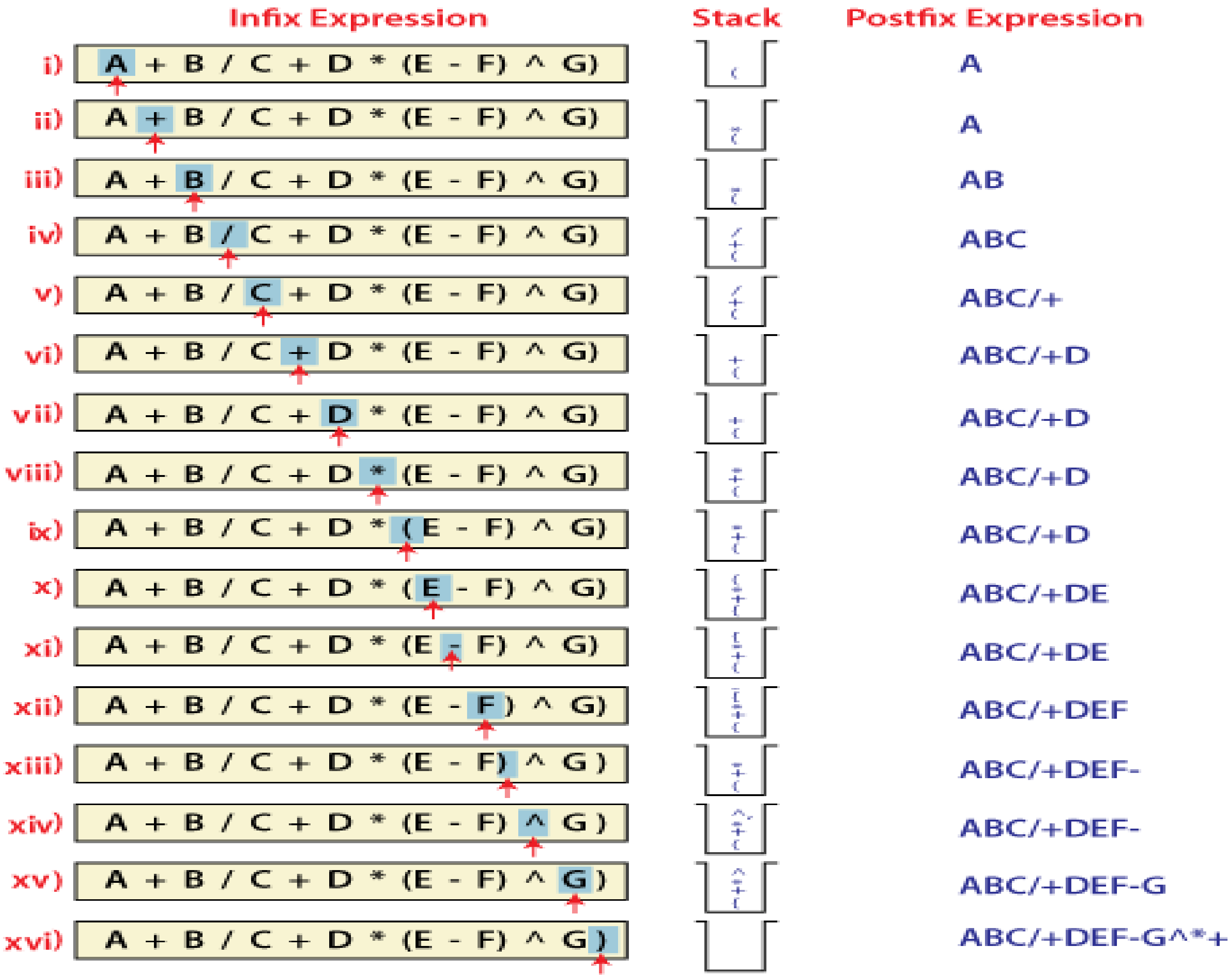
- Infix Notation - **A + B, (C - D)**
- Prefix Notation - **+AB, -CD**
- Postfix Notation - **AB+, CD+**

Conversion of Arithmetic Expression into various Notations:

Infix Notation	Prefix Notation	Postfix Notation
A * B	* A B	AB*
(A+B)/C	/+ ABC	AB+C/
(A*B) + (D-C)	+*AB - DC	AB*DC-+

Infix to Postfix Conversion

- Scan the infix expression from left to right.
- If the scanned character is an operand, put it in the postfix expression.
- Otherwise, do the following
  - If the precedence of the new operator is greater than the precedence of the operator in stack then push the new operator into stack.
  - Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the new operator.
    - After doing that Push the scanned operator to the stack.
- If the scanned character is a ‘(‘, push it to the stack.
- If the scanned character is a ‘)’, pop the stack and output it until a ‘(‘ is encountered
- Repeat the steps until the infix expression is scanned.
- Once the scanning is over, Pop the stack and add the operators in the postfix expression until it is not empty.
- Finally, print the postfix expression.


Infix Expression
Stack
Postfix Expression
i) A + B / C + D * (E - F) ^ G)
A
ii) A + B / C + D * (E - F) ^ G)
A
iii) A + B / C + D * (E - F) ^ G)
AB
iv) A + B / C + D * (E - F) ^ G)
ABC
v) A + B / C + D * (E - F) ^ G)
ABC/+
vi) A + B / C + D * (E - F) ^ G)
ABC/+D
vii) A + B / C + D * (E - F) ^ G)
ABC/+D
viii) A + B / C + D * (E - F) ^ G)
ABC/+D
ix) A + B / C + D * (E - F) ^ G)
ABC/+D
x) A + B / C + D * (E - F) ^ G)
ABC/+DE
xi) A + B / C + D * (E - F) ^ G)
ABC/+DE
xii) A + B / C + D * (E - F) ^ G)
ABC/+DEF
xiii) A + B / C + D * (E - F) ^ G)
ABC/+DEF-
xiv) A + B / C + D * (E - F) ^ G)
ABC/+DEF-
xv) A + B / C + D * (E - F) ^ G)
ABC/+DEF-G
xvi) A + B / C + D * (E - F) ^ G)
ABC/+DEF-G^*+

Infix to Postfix Conversion

	Stack	postfix
$(A+B)/C$	(	
	(	A
	(+	A
	(+	AB
	~~(+)~~	AB
		AB+
	/	AB+
	/	AB+C
		AB+C/

Infix to Postfix Conversion

(A*B)+(D-C)

	Stack	postfix
(	(	
A	(	A
*	(*	A
B	(*	AB
)	(*)	AB
		AB*
+	+	AB*
(	+(	AB*
D	+(	AB*D
-	+(-	AB*D
C	+(-	AB*DC
)	+(-)	AB*DC
		AB*DC-+

Infix to Postfix Conversion

Example:

Input: A + B * C + D

Output: ABC*+D+

Input: ((A + B) – C * (D / E)) + F

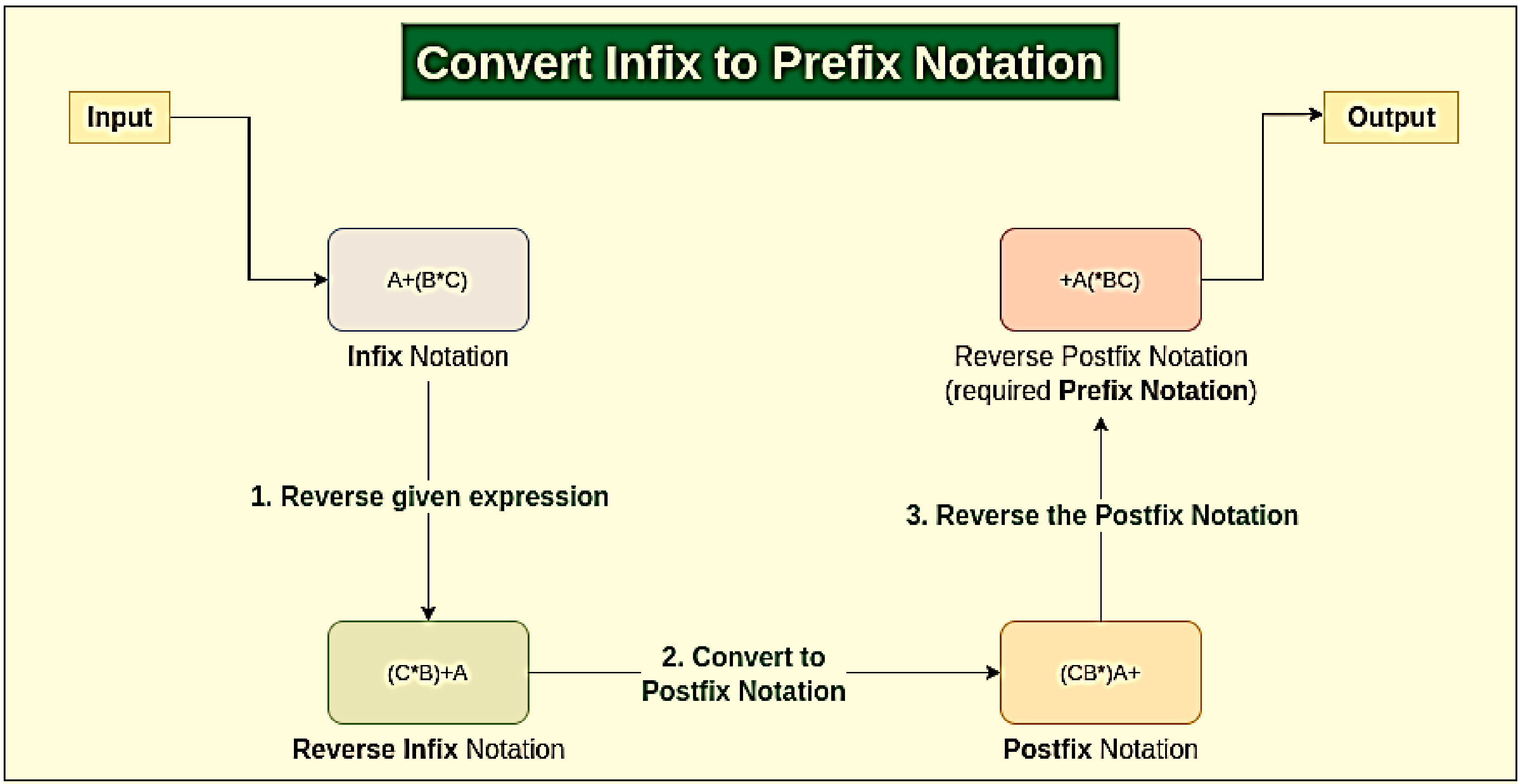
Output: AB+CDE/*-F+

Infix to Postfix Conversion

```
public static String infixToPostfix(String infixExpression) {
       StringBuilder postfix = new StringBuilder();
       Stack<Character> operatorStack = new Stack<>();
       for (char ch : infixExpression.toCharArray()) {
           if (Character.isLetterOrDigit(ch)) {
               postfix.append(ch);
           } else if (ch == '(') {
               operatorStack.push(ch);
           } else if (ch == ')') {
               while (!operatorStack.isEmpty() && operatorStack.peek() != '(') {
                   postfix.append(operatorStack.pop());                }
               operatorStack.pop();             }
               else if (isOperator(ch)) {
               while (!operatorStack.isEmpty() && getPrecedence(ch) <=
getPrecedence(operatorStack.peek())) {
                   postfix.append(operatorStack.pop());                }
               operatorStack.push(ch);           }       }
while (!operatorStack.isEmpty()) {
           postfix.append(operatorStack.pop());       }
       return postfix.toString();    }
```

Infix to Prefix Conversion

- Reverse the infix expression. Note while reversing each
  ‘(‘ will become ‘)’ and each ‘)’ becomes ‘(‘.
- Convert the reversed infix expression to “nearly” postfix expression
- Reverse the postfix expression.



Infix to Prefix Conversion

Page No. ....

Prefix. (A*B)+(D-C)

Step 1 ⇒ (C-D)+(B*A)

	Scan	Stack	Expression ~~Prefix~~
Step2 ⇒	(	(	
	C	(	C
	–	(–	C
	D	(–	CD
	)	(–)	CD–
	+	+	CD–
	(B	+(	CD–B
	*	+(*	CD–B
	A	+(*	CD–BA
	)	+(*)	CD–BA
			CD–BA*+

Prefix ⇒ $\boxed{+*AB-DC}$

Infix to Prefix Conversion

Example:

Input: A * B + C / D

Output: + * A B/ C D

Input: (A – B/C) * (A/K-L)

Output: *-A/BC-/AKL

Infix to Prefix Conversion

```
public static String infixToPrefix(String infixExpression) {
        StringBuilder prefix = new StringBuilder();
        Stack<Character> operatorStack = new Stack<>();
        String reversedInfix = reverse(infixExpression);
        for (char ch : reversedInfix.toCharArray()) {
            if (Character.isLetterOrDigit(ch)) {
                prefix.append(ch); // Append operands directly to the result
            } else if (ch == ')') {
                operatorStack.push(ch);
            } else if (ch == '(') {
         while (!operatorStack.isEmpty() && operatorStack.peek() != ')') {
                    prefix.append(operatorStack.pop());                }
                operatorStack.pop(); } else if (isOperator(ch)) {
         while (!operatorStack.isEmpty() && getPrecedence(ch) <
getPrecedence(operatorStack.peek())) {
                    prefix.append(operatorStack.pop());                }
                operatorStack.push(ch);            }        }
        while (!operatorStack.isEmpty()) {
            prefix.append(operatorStack.pop());        }
return reverse(prefix.toString());    }
```