# PRINCIPLES OF DATA STRUCTURES THROUGH C/C++
# BCS-12

**Contact Hours Lecture : 3, Tutorial : 1 , Practical: 2 (online 4 lectures/week)**
**Number of Credits :5**

**B.Tech (Computer Science & Engg.)** **Semester: III**

By
**Muzammil Hasan**
(Asst. Prof)

Department of Computer Science & Engineering
Madan Mohan Malaviya University of Technology
Gorakhpur, India

# Course Outcomes

The students are expected to be able to demonstrate the following knowledge, skills and attitudes after completing this course

1. Describe how arrays, records, linked lists, stacks, queues, trees, and graphs are represented in memory, used by the algorithms and their common applications.
2. Write programs that use arrays, records, linked structures, stacks, queues, trees, and graphs.
3. Compare and contrast the benefits of dynamic and static data structures implementations.
4. Identity the alternative implementations of data structures with respect to its performance to solve a real world problem.
5. Demonstrate organization of information using Trees and Graphs and also to perform different operations on these data structures.
6. Design and implement an appropriate organization of data on primary and secondary memories for efficient its efficient retrieval. .
7. Discuss the computational efficiency of the principal algorithms for sorting, searching and hashing.
8. Describe the concept of recursion, its application, its implementation and removal of recursion.

**"Get your data structures correct first, and the rest of the program will write itself."**

**- *David Jones***

# EXPERIMENTS

Write C/C++ Programs to illustrate the concept of the following:

- 1. Sorting Algorithms-Non-Recursive
- 2. Sorting Algorithms-Recursive
- 3. Searching Algorithm
- 4. Stack
- 5. Queue
- 6. Linked List
- 7. Graph

# Textbooks


- Horowitz and Sahani, Fundamentals of Data Structures, Galgotia Publication, New Delhi.
- R. Kruseetal, Data Structure and Pragram Design in C, Pearson Education Asia Delhi
- A. M.Tenenbaum, Data Structures using C & C++, PHI, India
- K Loudon, Mastering Algorithms with C, Shroff Publication and Distributor Pvt. Ltd.
- Bruno R Preiss, Data Structure and Algorithms with Object Oriented Design Pattern in C++, John Wiley & Sons
- Adam Drozdek, “Data Structures and Algorithms in C++”, Thomson Asia Pvt. Ltd. Singapore

# Definition

- Data structure is representation of the logical relationship existing between individual elements of data.
- In other words, a data structure is a way of organizing all data items that considers not only the elements stored but also their relationship to each other.

- All programs manipulate data
  - programs *process*, *store*, *display*, *gather*
  - data can be *information*, *numbers*, *images*, *sound*
- Each program must decide how to store data
- Choice influences program at every level
  - execution speed
  - memory requirements
  - maintenance (debugging, extending, etc.)

# Introduction

- Data structure affects the design of both structural & functional aspects of a program.

Program=algorithm + Data Structure

- You know that a algorithm is a step by step procedure to solve a particular function.
- That means, algorithm is a set of instruction written to carry out certain tasks & the data structure is the way of organizing the data with their logical relationship retained.
- To develop a program of an algorithm, we should select an appropriate data structure for that algorithm.
- Therefore algorithm and its associated data structures from a program.

# Abstract Data Type (ADT)

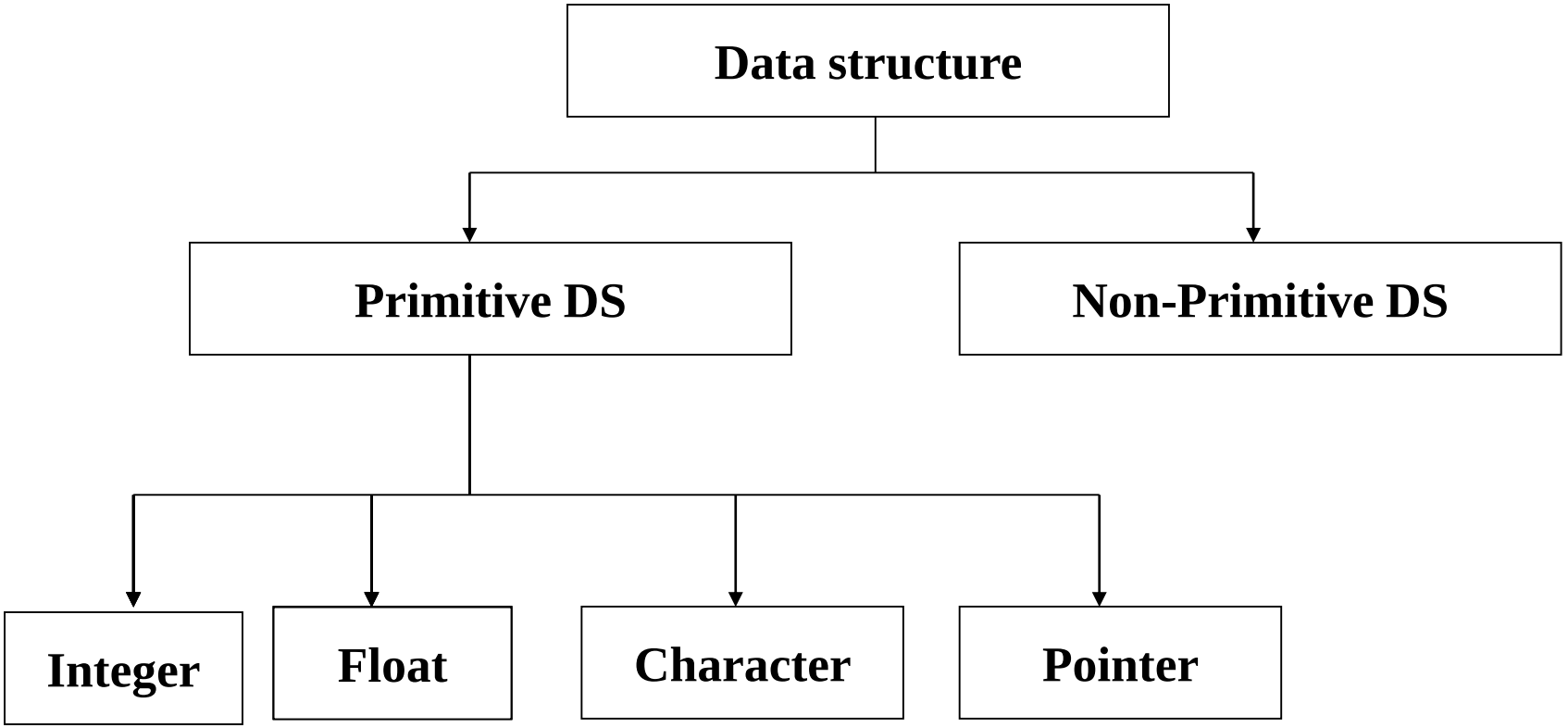
1) An opportunity for an acronym

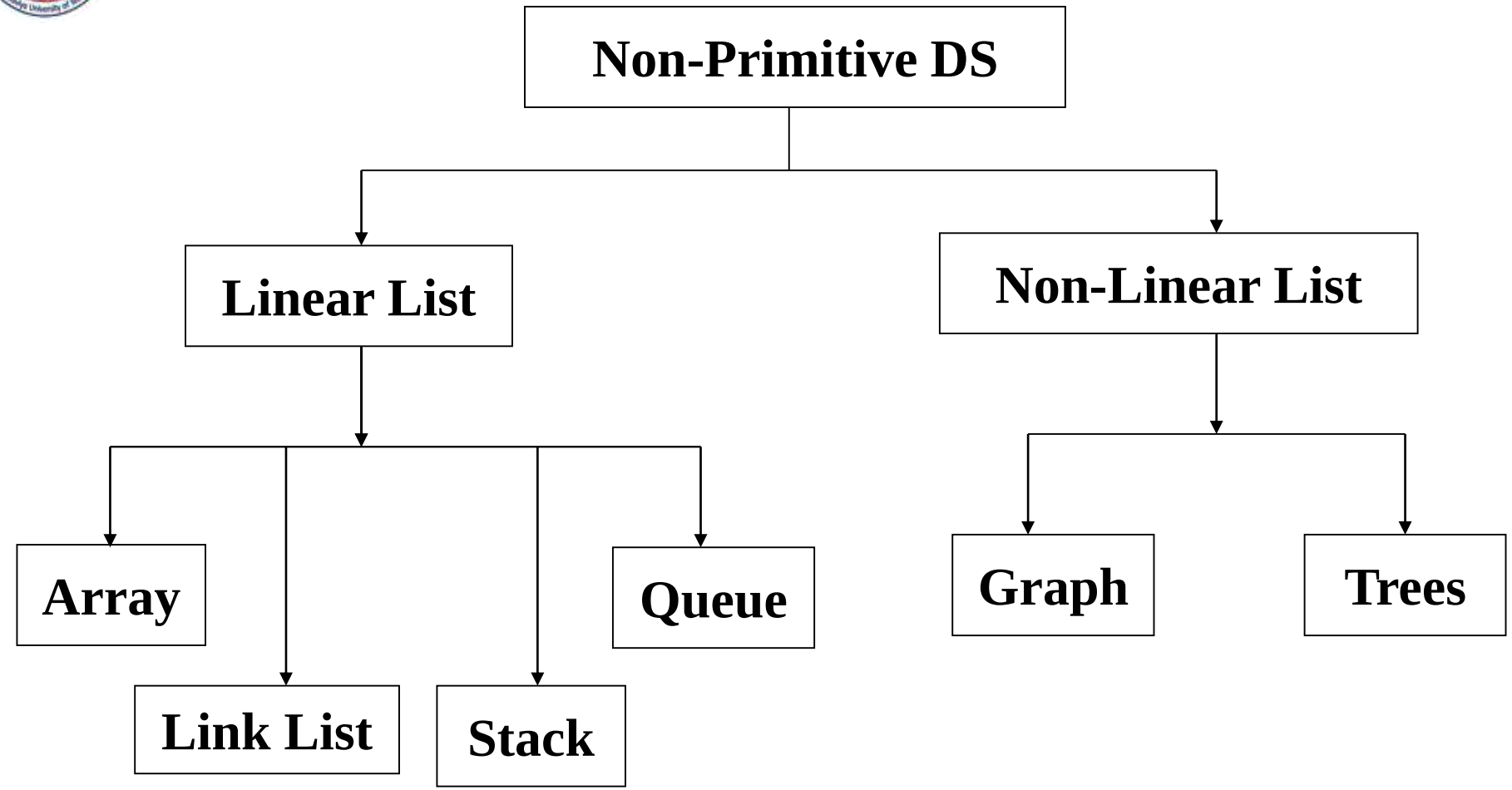
2) Mathematical description of an object and the set of operations on the object

# Classification of Data Structure

Data structure are normally divided into two broad categories:

- Primitive Data Structure
- Non-Primitive Data Structure




Non-Primitive DS
Linear List
Non-Linear List
Array
Link List
Stack
Queue
Graph
Trees

# Primitive Data Structure

- There are basic structures and directly operated upon by the machine instructions.
- In general, there are different representation on different computers.
- Integer, Floating-point number, Character constants, string constants, pointers etc, fall in this category.

# Non-Primitive Data Structure

- There are more sophisticated data structures.
- These are derived from the primitive data structures.
- The non-primitive data structures emphasize on structuring of a group of homogeneous (same type) or heterogeneous (different type) data items.
- Lists, Stack, Queue, Tree, Graph are example of non-primitive data structures.
- The design of an efficient data structure must take operations to be performed on the data structure.

The most commonly used operation on data structure are broadly categorized into following types:

- Create
- Selection
- Updating
- Searching
- Sorting
- Merging
- Destroy or Delete

# Primitive Vs Non Primitive

- A primitive data structure is generally a basic structure that is usually built into the language, such as an integer, a float.
- A non-primitive data structure is built out of primitive data structures linked together in meaningful ways, such as a or a linked-list, binary search tree, AVL Tree, graph etc.

# Array

- An array is defined as a set of finite number of homogeneous elements or same data items.
- It means an array can contain one type of data only, either all integer, all float-point number or all character.
- Simply, declaration of array is as follows:

```
int arr[10]
```

- Where int specifies the data type or type of elements arrays stores.
- “arr” is the name of array & the number specified inside the square brackets is the number of elements an array can store, this is also called sized or length of array.

**Array Representation:(Storagestructure)**

Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.


Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.

As per the above illustration, following are the important points to be considered.

- Index starts with 0.
- Array length is 10 which means it can store 10 elements.
- Each element can be accessed via its index. For example, we can fetch an element at index 6 as 9.

**Basic Operations**

Following are the basic operations supported by an array.

- **Traverse – print all the array elements one by one.**
- **Insertion – Adds an element at the given index.**
- **Deletion – Deletes an element at the given index.**
- **Search – Searches an element using the given index or by the value.**
- **Update – Updates an element at the given index.**

Following are some of the concepts to be remembered about arrays:

- The individual element of an array can be accessed by specifying name of the array, following by index or subscript inside square brackets.

- The first element of the array has index zero[0]. It means the first element and last element will be specified as : arr[0] & arr[9] respectively.

- The elements of array will always be stored in the consecutive (continues) memory location.

- The number of elements that can be stored in an array, that is the size of array or its length is given by the following equation:

  (Upperbound-lowerbound)+1

For the above array it would be

- (9-0)+1=10,where 0 is the lower bound of array and 9 is the upper bound of array.
- Array can always be read or written through loop. If we read a one-dimensional array it require one loop for reading and other for writing the array.
- For example: Reading an array

```
for(i=0;i<=9;i++)
    scanf("%d",&arr[i]);
```

- For example: Writing an array

```
For(i=0;i<=9;i++)
    printf("%d",arr[i]);
```

If we are reading or writing two-dimensional array it would require two loops and similarly the array of a N dimension would required N loops.

**Insertion Operation**

- Insert operation is to insert one or more data elements into an array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array.

**Algorithm**

Let LA be a Linear Array (unordered) with N elements and K is a positive integer such that K<=N.

Following is the algorithm where ITEM is inserted into the Kth position of LA

1. Start
2. Set J = N
3. Set N = N+1
4. Repeat steps 5 and 6 while J >= K
5. Set LA[J+1] = LA[J]
6. Set J = J-1
7. Set LA[K] = ITEM
8. Stop

## Deletion Operation

- Deletion refers to removing an existing element from the array and re-organizing all elements of an array.

## Algorithm

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**.

Following is the algorithm to delete an element available at the Kth position of LA.

1. Start
2. Set J = K
3. Repeat steps 4 and 5 while J < N
4. Set LA[J] = LA[J + 1]
5. Set J = J+1
6. Set N = N-1
7. Stop

## SearchOperation

- You can perform a search for an array element based on its value or its index. **Algorithm**
- Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**.

Following is the algorithm to find an element with a value of ITEM using sequential search.

1. Start
2. Set J = 0
3. Repeat steps 4 and 5 while J < N
4. IF LA[J] is equal ITEM THEN GOTO STEP 6
5. Set J = J +1
6. PRINT J, ITEM
7. Stop

## UpdateOperation

- Update operation refers to updating an existing element from the array at a given index.

## Algorithm

- Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**.

Following is the algorithm to update an element available at the Kth position of LA.

1. Start
2. Set LA[K-1] = ITEM
3. Stop

# Application

## Sparse Matrix and its representations

- A matrix is a two-dimensional data object made of m rows and n columns, therefore having total m x n values. If most of the elements of the matrix have **0 value**, then it is called a sparse matrix.

## Why to use Sparse Matrix instead of simple matrix ?

- **Storage:** There are lesser non-zero elements than zeros and thus lesser memory can be used to store only those elements.
- **Computing time:** Computing time can be saved by logically designing a data structure traversing only non-zero elements..

Example:

0 0 3 0 4
0 0 5 7 0
0 0 0 0 0
0 2 6 0 0

- Representing a sparse matrix by a 2D array leads to wastage of lots of memory as zeroes in the matrix are of no use in most of the cases. So, instead of storing zeroes with non-zero elements, we only store non-zero elements. This means storing non-zero elements with **triples- (Row, Column, value).**

Sparse Matrix Representations can be done in many ways following are two common representations:

- 1. Array representation
- 2. Linked list representation

**Using Arrays** #include<stdio.h>

```
int main()
{
// Assume 4x5 sparse matrix
int sparseMatrix[4][5] ={
{0 , 0 , 3 , 0 , 4 },
{0 , 0 , 5 , 7 , 0 },
{0 , 0 , 0 , 0 , 0 },
{0 , 2 , 6 , 0 , 0 }
};

int size = 0;
for (int i = 0; i < 4; i++) for (int j = 0; j < 5; j++)
if (sparseMatrix[i][j] != 0) size++;
int compactMatrix[3][size]; // Making of new matrix
```

```
int k = 0;
for (int i = 0; i < 4; i++) for (int j = 0; j < 5; j++)
if (sparseMatrix[i][j] != 0)
 {
compactMatrix[0][k] = i;
compactMatrix[1][k] = j;
compactMatrix[2][k] = sparseMatrix[i][j];
k++;
}
for (int i=0; i<3; i++) {
for (int j=0; j<size; j++)
printf("%d ", compactMatrix[i][j]); printf("\n");
}
return 0;
}
```

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{bmatrix}$$

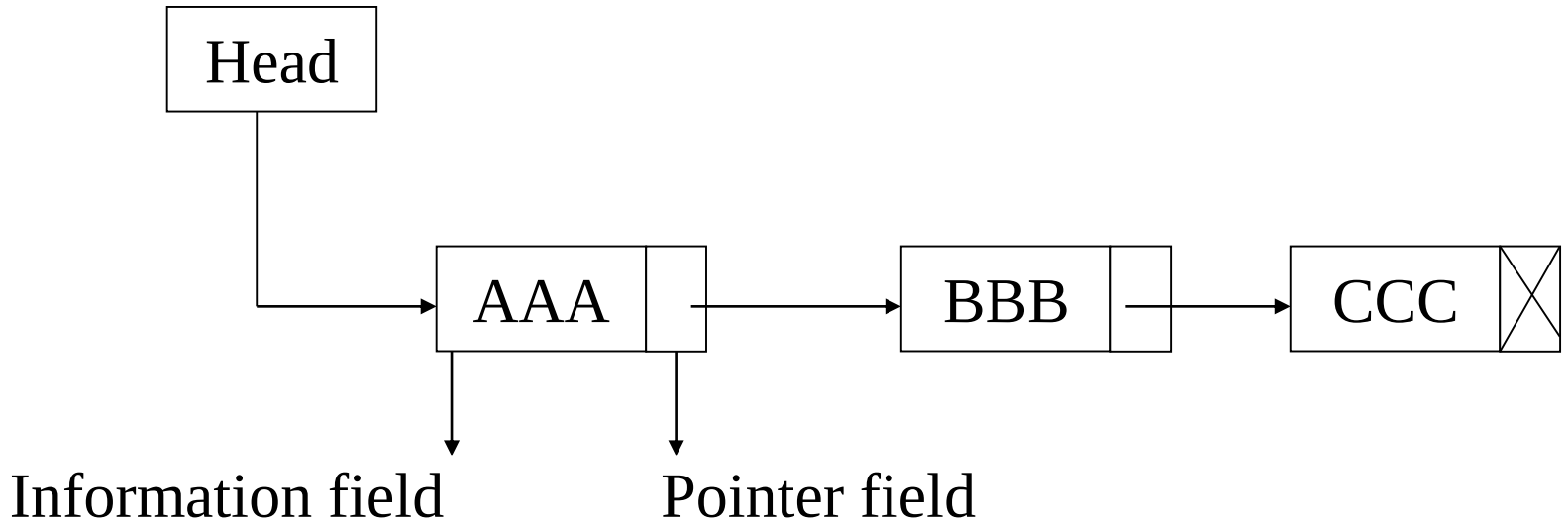
| Row | 0 | 0 | 1 | 1 | 3 | 3 |
|---|---|---|---|---|---|---|
| Column | 2 | 4 | 2 | 3 | 1 | 2 |
| Value | 3 | 4 | 5 | 7 | 2 | 6 |

# List

- A lists (Linear linked list) can be defined as a collection of variable number of data items.
- Lists are the most commonly used non-primitive data structures.
- An element of list must contain at least two fields, one for storing data or information and other for storing address of next element.
- As you know for storing address we have a special data structure of list the address must be pointer type

Technically each such element is referred to as a node, therefore a list can be defined as a collection of nodes as show bellow:

[Linear Liked List]



# List ADT

AbstractDataType *LinearList*

{

instances

ordered finite collections of zero or more elements

operations

*isEmpty()*: return true iff the list is empty, false otherwise

size(): return the list size (i.e., number of elements in the list)

*get(index)*: return the *index*th element of the list

*indexO f(x)*: return the index of the first occurrence of *x* in the list, return -1 if *x* is not in the list

*remove(index)*: remove and return the *index*th element, elements with higher index have their index reduced by 1

*add(theIndex, x)*: insert x as the *index*th element, elements with theIndex >= *index* have their index increased by 1

*output()*: output the list elements from left to right

}

Types of linked lists:

- Single linked list
- Doubly linked list
- Single circular linked list
- Doubly circular linked list

# Stack

- A stack is also an ordered collection of elements like arrays, but it has a special feature that deletion and insertion of elements can be done only from one end called the top of the stack (TOP)
- Due to this property it is also called as last in first out type of data structure (LIFO).
- It could be through of just like a stack of plates placed on table in a party, a guest always takes off a fresh plate from the top and the new plates are placed on to the stack at the top.
- It is a non-primitive data structure.
- When an element is inserted into a stack or removed from the stack, its base remains fixed where the top of stack changes.

# Operations

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations −

- **push()** − Pushing (storing) an element on the stack.
- **pop()** − Removing (accessing) an element from the stack.

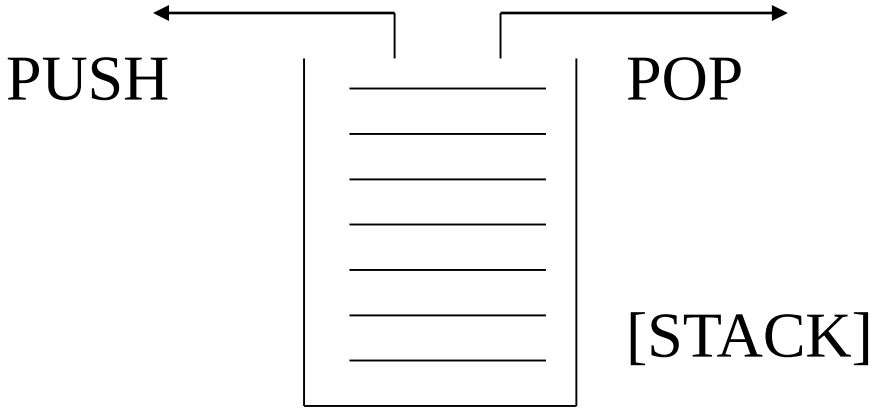
To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks −

**peek()** − get the top data element of the stack, without removing it. ·

**isFull()** − check if stack is full.

**isEmpty()** − check if stack is empty.

- Insertion of element into stack is called PUSH and deletion of element from stack is called POP.
- The bellow show figure how the operations take place on a stack:



The stack can be implemented into two ways:

- Using arrays (Static implementation)
- Using pointer (Dynamic implementation)

At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer always represents the top of the stack, hence named **top**. The **top** pointer provides top value of the stack without actually removing it.

First we should learn about procedures to support stack functions − **peek()**

**Algorithm** of peek() function −

- begin procedure peek
- return stack[top]
- end procedure

**Implementation** of peek() function in C programming language

```
int peek() {
return stack[top]; }
```

## isfull()

**Algorithm** of isfull() function −

- begin procedure isfull
- if top equals to MAXSIZE return true
- else
- return false endif
- end procedure

Implementation of isfull() function in C language

```
bool isfull()
 {
if(top == MAXSIZE) return true;
else
return false;
 }
```

## isempty()

**Algorithm** of isempty() function −

- begin procedure isempty
- if top less than 1 return true
- else
- return false endif
- end procedure

Implementation of isempty() function in C programming language is slightly different. We initialize top at -1, as the index in array starts from 0. So we check if the top is below zero or -1 to determine if the stack is empty.

```
bool isempty()
 { if(top == -1)
return true; else
return false; }
```

## PushOperation

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps

**Step 1** − Checks if the stack is full.

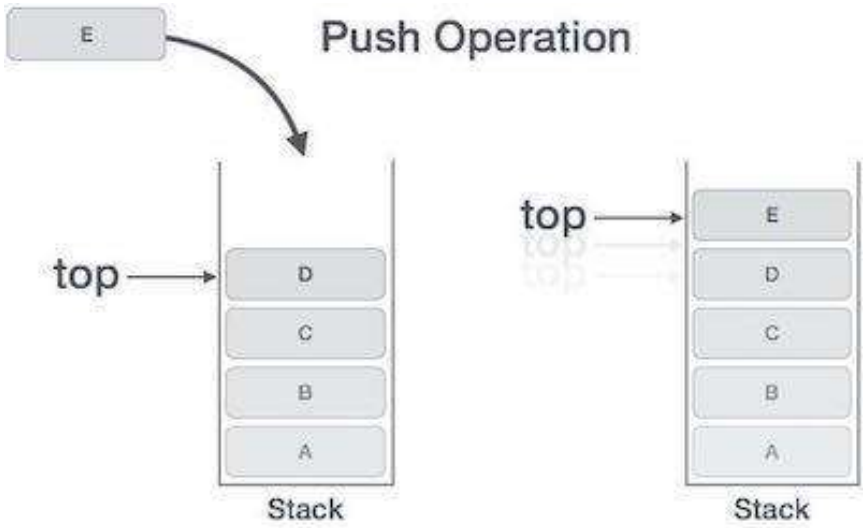
**Step 2** − If the stack is full, produces an error and exit.

**Step 3** − If the stack is not full, increments **top** to point next empty space.

**Step 4** − Adds data element to the stack location, where top is pointing.

**Step 5** − Returns success.

```
begin procedure push:stack, data
if stack is full
return null endif
top ← top + 1
stack[top] ← data
end procedure
```



## PopOperation

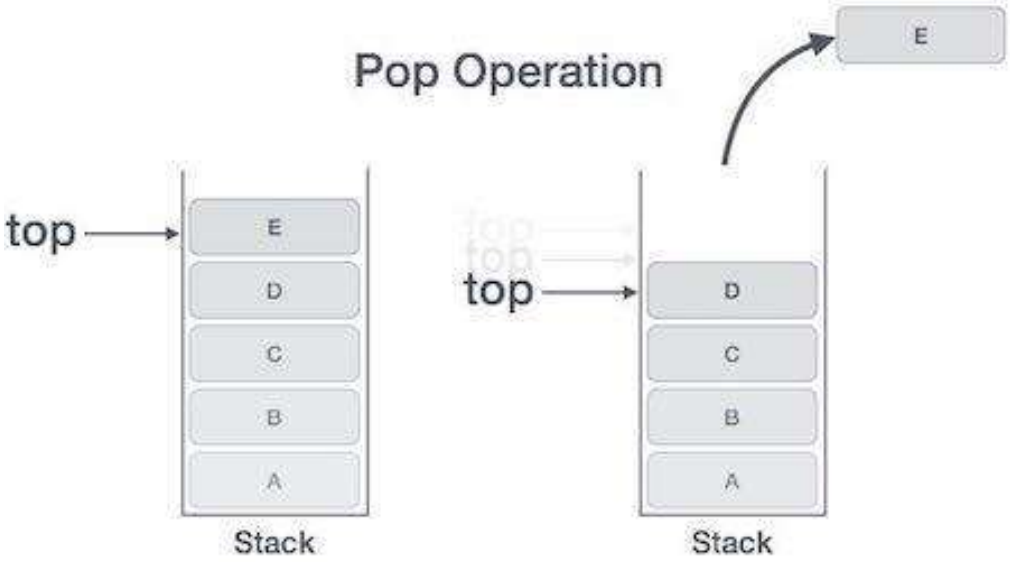
Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead **top** is decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop() actually removes data element and deallocates memory space.

A Pop operation may involve the following steps −

**Step 1** − Checks if the stack is empty.
**Step 2** − If the stack is empty, produces an error and exit.
**Step 3** − If the stack is not empty, accesses the data element at which **top** is pointing.
**Step 4** − Decreases the value of top by 1.
**Step 5** − Returns success.

A simple algorithm for Pop operation can be derived as follows −

- begin procedure pop: stack
- if stack is empty return null
- endif
- data ← stack[top] top ← top - 1
- return data
- end procedure



# Stack Applications

Three applications of stacks are presented here. These examples are central to many activities that a computer must do and deserve time spent with them.

1. Expression evaluation
2. Backtracking (game playing, finding paths, exhaustive searching)
3. Memory management, run-time environment for nested language features.

# Expression evaluation

- In particular we will consider arithmetic expressions. Understand that there are boolean and logical expressions that can be evaluated in the same way. Control structures can also be treated similarly in a compiler.
- This study of arithmetic expression evaluation is an example of problem solving where you solve a simpler problem and then transform the actual problem to the simpler one.

# Infix, Prefix and Postfix Notation

- We are accustomed to write arithmetic expressions with the operation between the two operands: a+b or c/d. If we write a+b*c, however, we have to apply precedence rules to avoid the ambiguous evaluation (add first or multiply first?).
- There's no real reason to put the operation between the variables or values. They can just as well precede or follow the operands. You should note the advantage of prefix and postfix: the need for precedence rules and parentheses are eliminated.

# Infix to postfix conversion

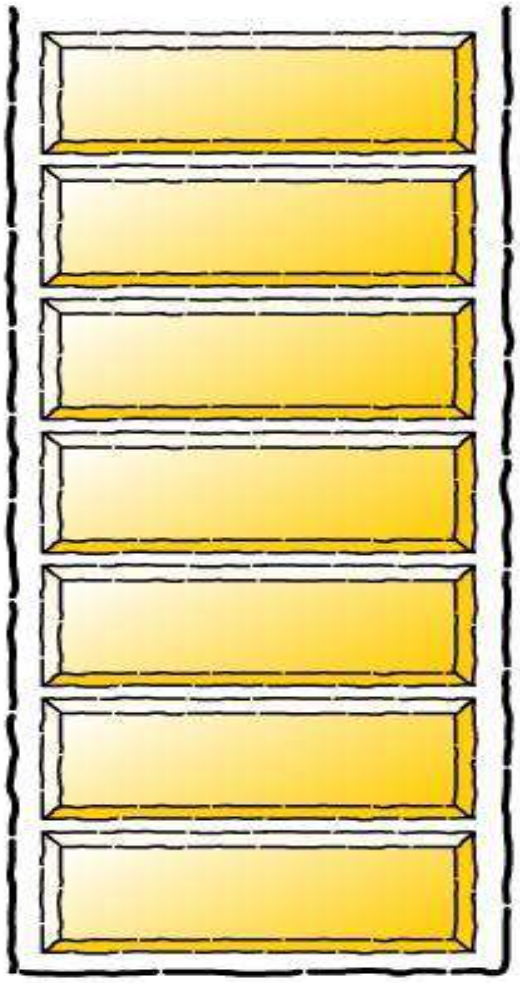
Read the tokens from a vector infixVect of tokens (strings) of an infix expression

- When the *token* is an operand
  - Add it to the end of the vector postfixVect of token (strings) that is used to store the corresponding postfix expression
- When the *token* is a left or right parenthesis or an operator
  - If the *token x* is "("
    - Push_back the token x to the end of the vector stackVect of token (strings) that simulates a stack
  - if the *token* x is ")"
    - Repeatedly pop_back a token y from stackVect and push_back that token y to postfixVect until "(" is encountered in the end of stackVect. Then pop_back "(" from stackVect.

- If stackVect is already empty before finding a "(", that expression is not a valid expression.

- if the *token x* is a regular operator
  - **Step 1**: Check the token y **currently** in the end of stackVect.
  - **Step 2**: **If** (case 1) stackVect is **not** empty **and** (case 2) y is **not** "(" **and** (case 3) y is an operator of **higer or equal** precedence than that of x, then pop_back the token y from stackVect and push_back the token y to postfixVect, and **go to Step 1 again**.
  - **Step 3**: **If** (case 1) stackVect is already empty **or** (case 2) y is "(" **or** (case 3) y is an operator of **lower precedence** than that of x, then push_back the token x into stackVect.

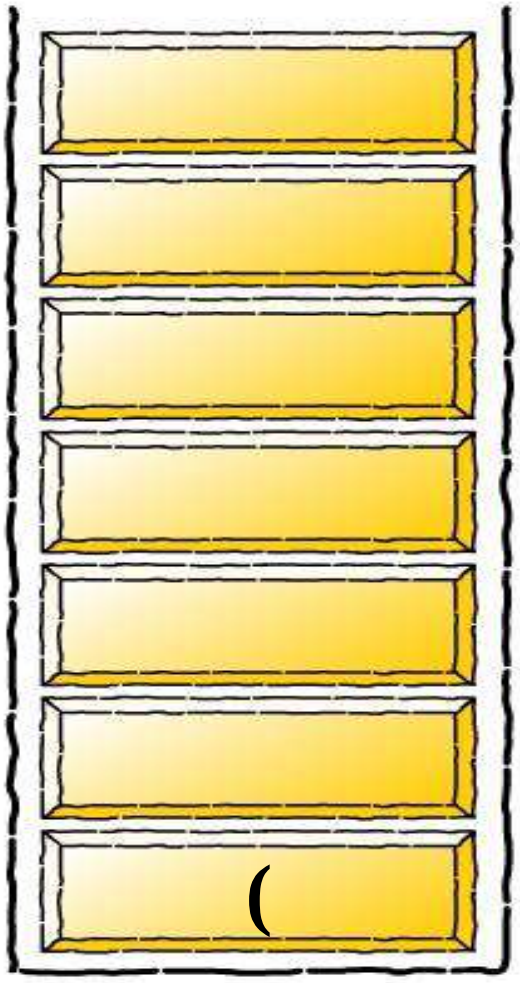
- When all tokens in infixVect are processed as described above, repeatedly pop_back a token y from stackVect and push_back that token y to postfixVect until stackVect is



**infixVect**

**( a + b - c ) * d – ( e + f )**

**postfixVect**

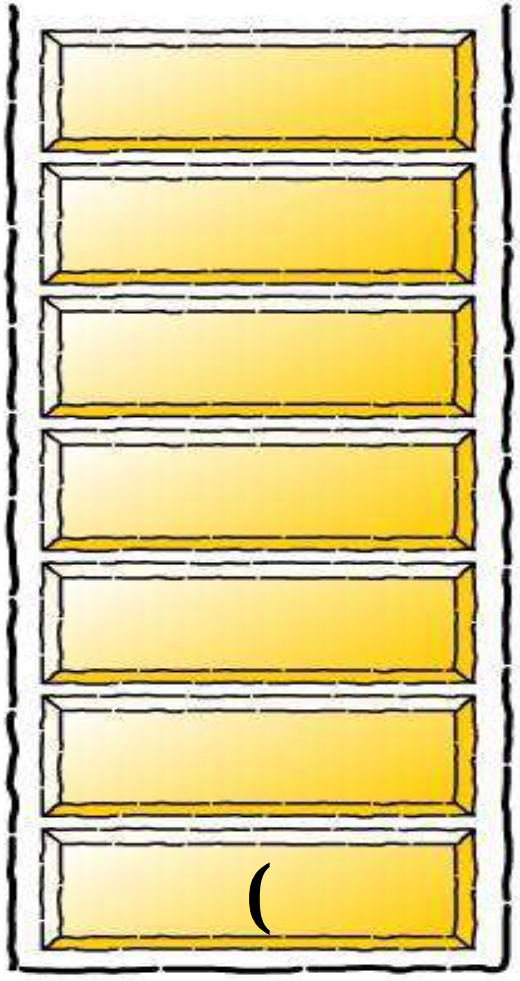



**infixVect**

**a + b - c ) * d – ( e + f )**

**postfixVect**

**stackVect**



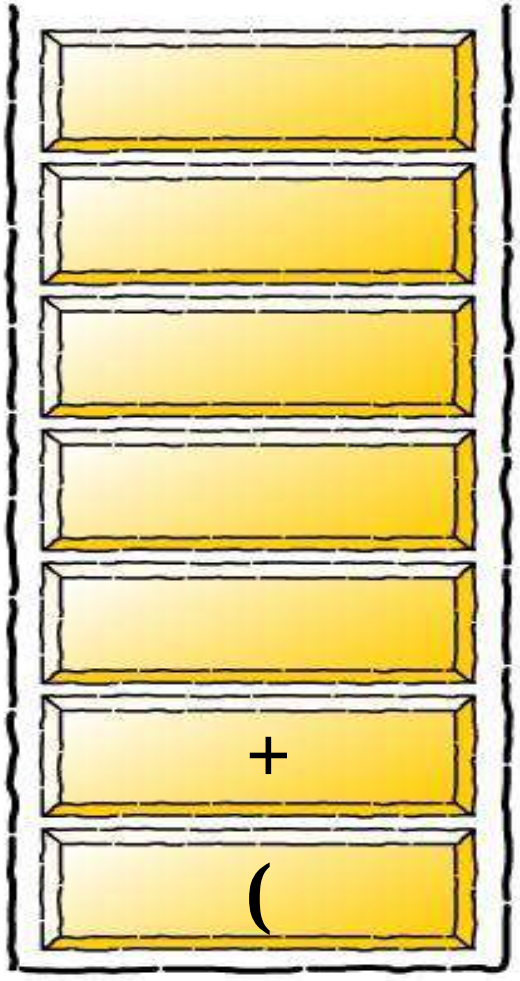

**infixVect**

**+ b - c ) * d – ( e + f )**

**postfixVect**

**a**

**stackVect**



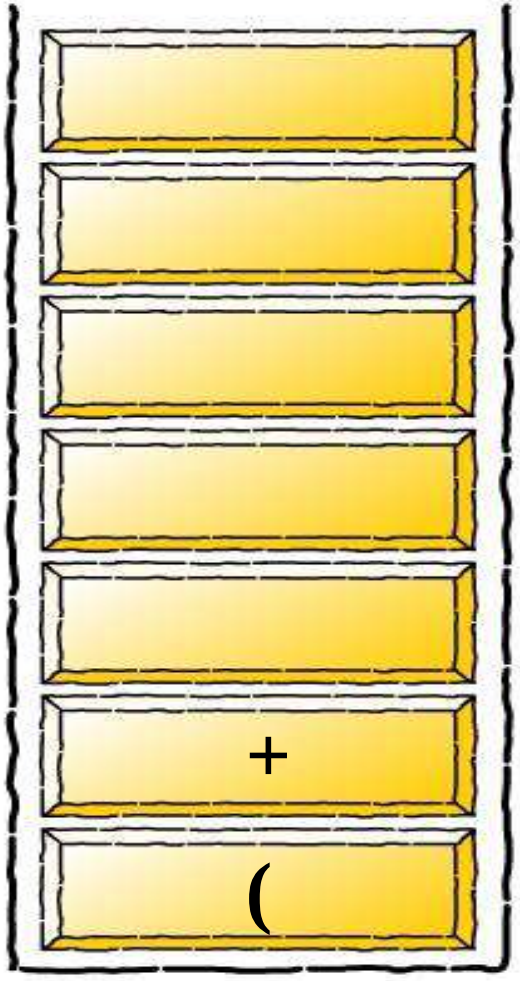

**infixVect**

**b - c ) * d – ( e + f )**

**postfixVect**

**a**

**stackVect**



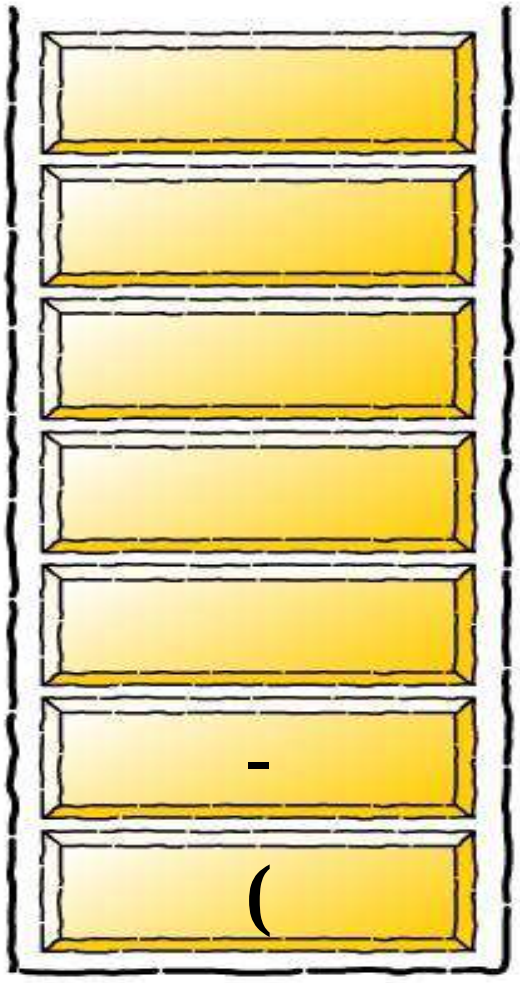

**infixVect**

- c ) * d – ( e + f )

**postfixVect**

a b

**stackVect**



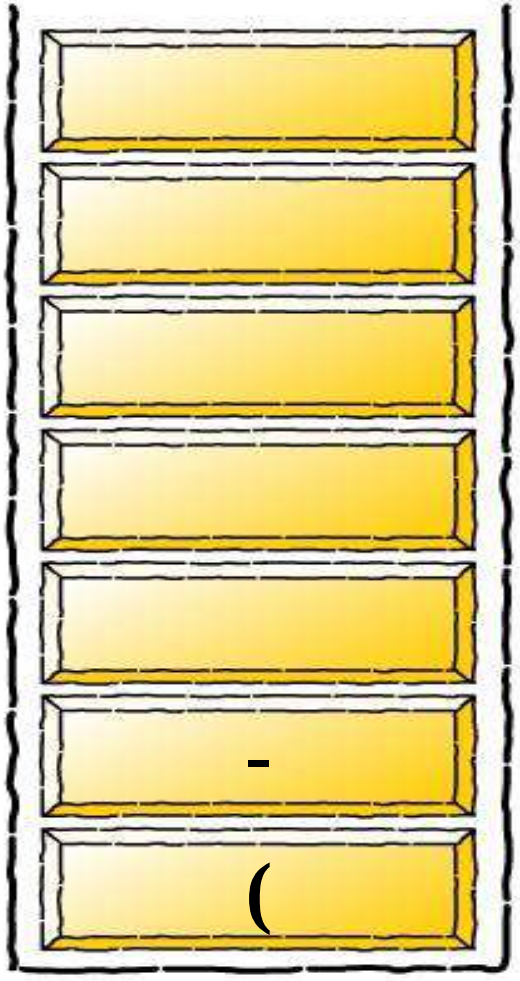

**infixVect**

**c ) * d – ( e + f )**

**postfixVect**

**a b +**

**stackVect**



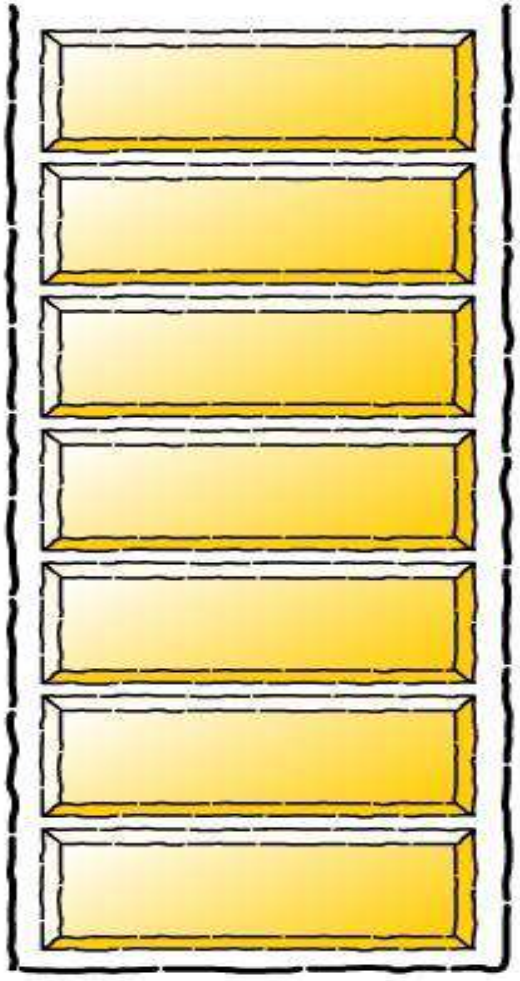

**infixVect**

) * d – ( e + f )

**postfixVect**

a b + c

**stackVect**



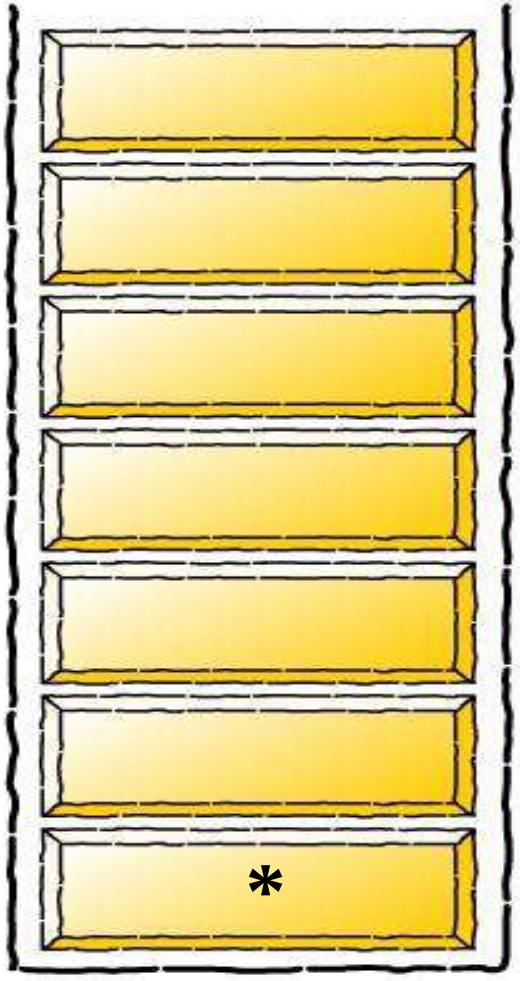
**infixVect**

*** d – ( e + f )**

**postfixVect**

**a b + c -**

**stackVect**



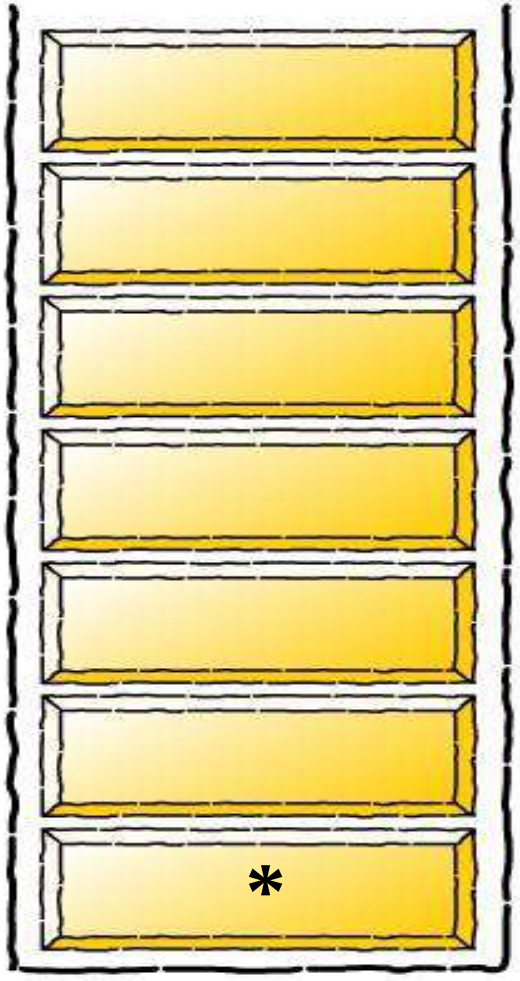

**infixVect**

**d – ( e + f )**

**postfixVect**

**a b + c -**

**stackVect**



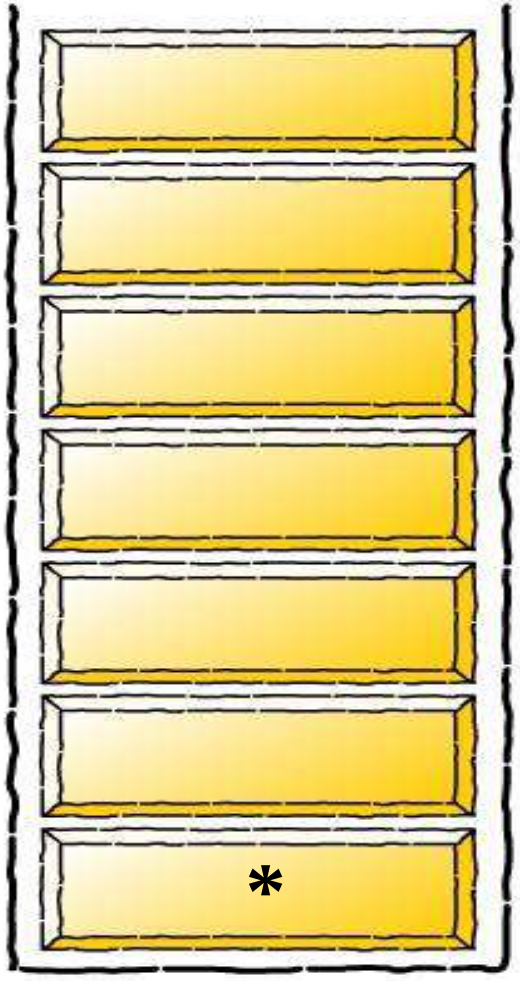

**infixVect**

**– ( e + f )**

**postfixVect**

**a b + c - d**

**stackVect**



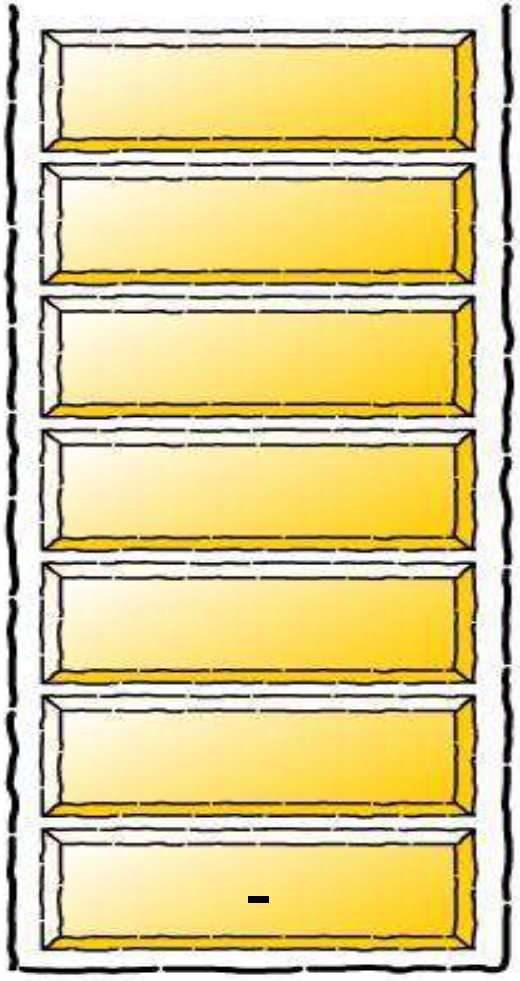

**infixVect**

**– ( e + f )**

**postfixVect**

**a b + c - d**

**stackVect**



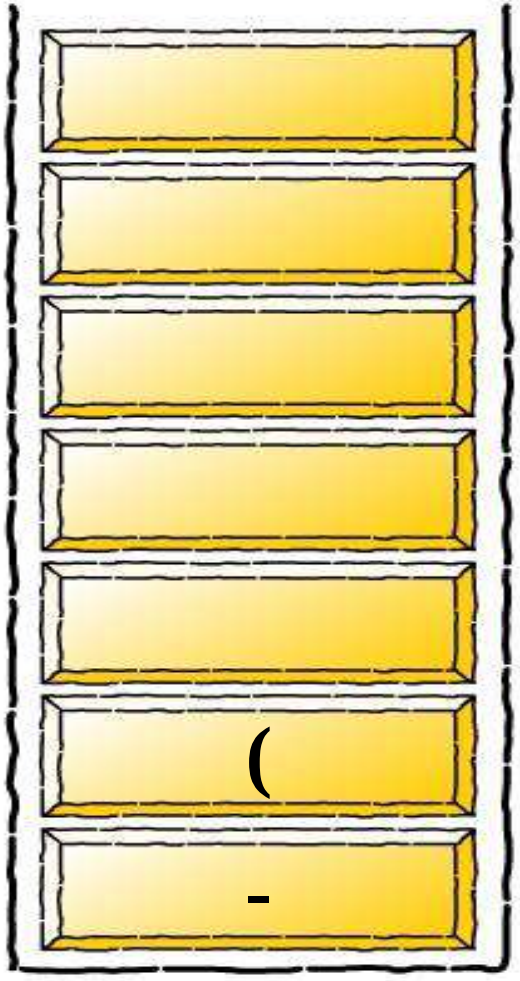

**infixVect**

**( e + f )**

**postfixVect**

**a b + c – d ***

**stackVect**



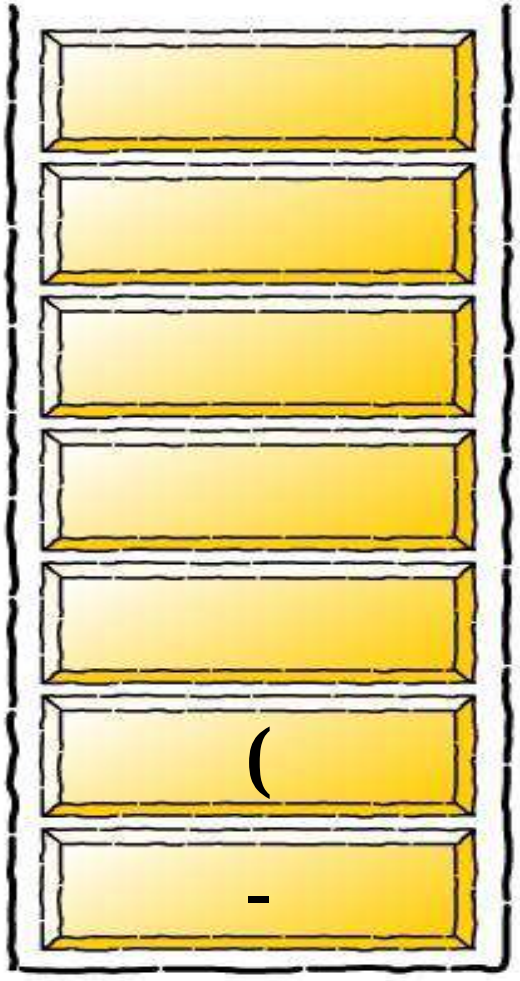

**infixVect**

**e + f )**

**postfixVect**

**a b + c – d ***

**stackVect**



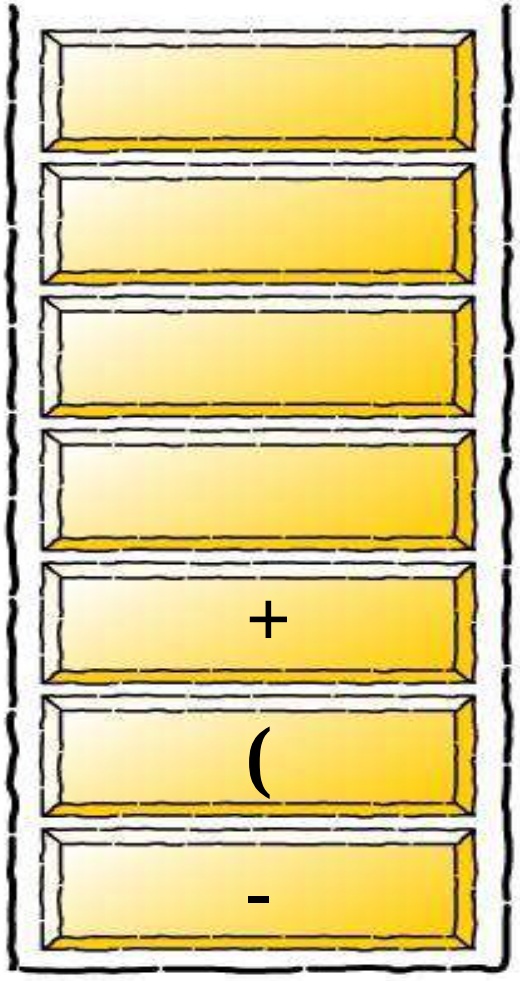

**infixVect**

**+ f )**

**postfixVect**

**a b + c – d * e**

**stackVect**



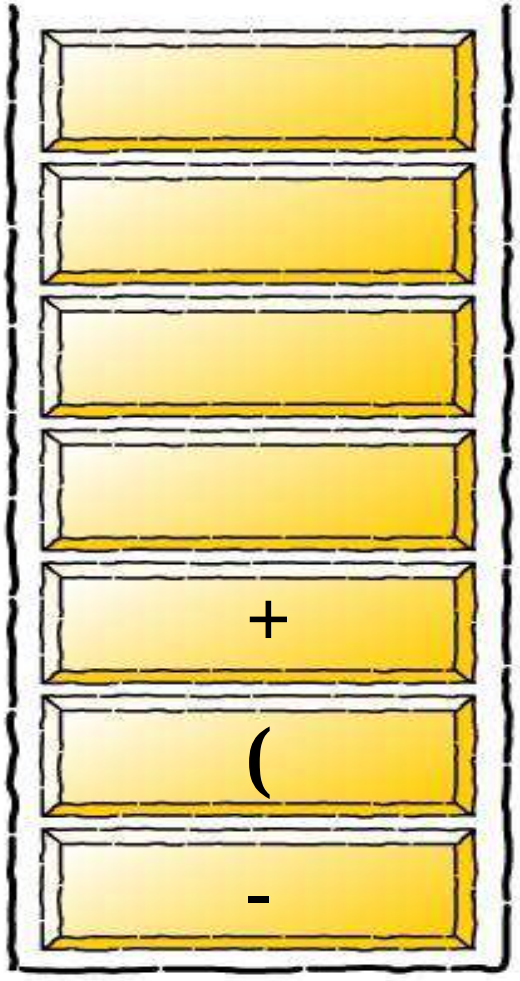

**infixVect**

**f )**

**postfixVect**

**a b + c – d * e**

**stackVect**



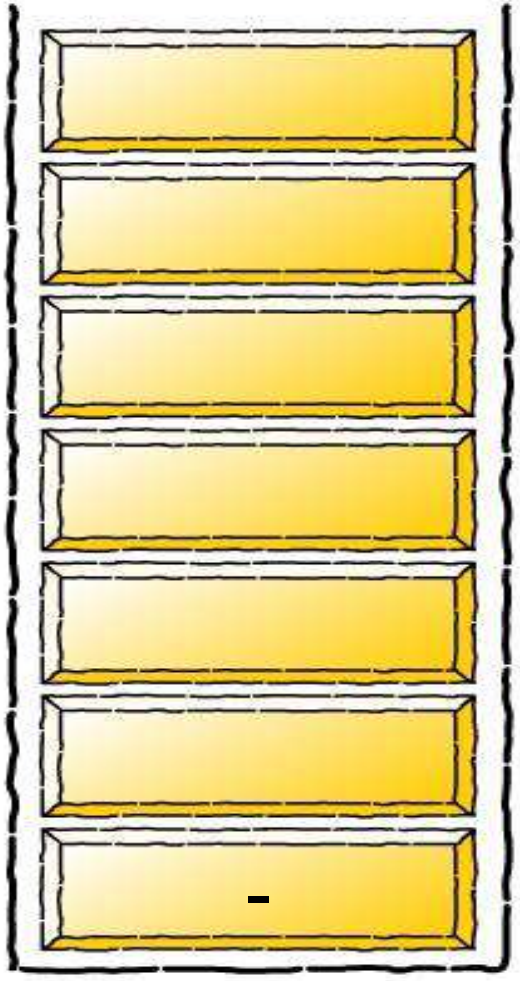

**infixVect**

)

**postfixVect**

a b + c – d * e f

**stackVect**

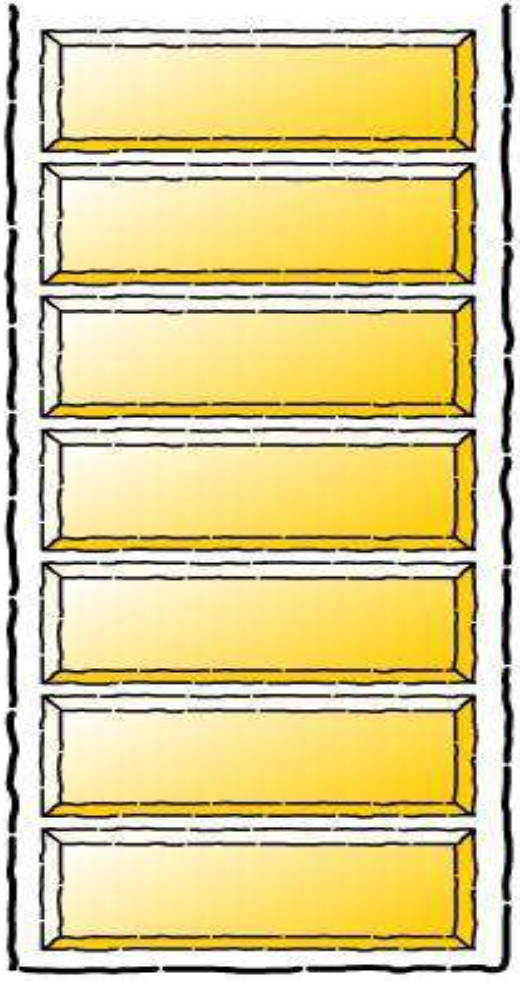



**infixVect**

**postfixVect**

**a b + c – d * e f +**

**stackVect**



**infixVect**

**postfixVect**

**a b + c – d * e f + -**

# From Postfix to Answer

Algorithm:

- Maintain a stack and scan the postfix expression from left to right
- If the element is a number, push it into the stack
- If the element is a operator O, pop twice and get A and B respectively. Calculate BOA and push it back to the stack
- When the expression is ended, the number in the stack is the final answer

# Reverse-Polish Notation

Alternatively, we can place the operands first, followed by the operator:

$$(3 + 4) \times 5 - 6$$

$$3\ 4\ +\ 5\ \times\ 6\ -$$

Parsing reads left-to-right and performs any operation on the last two operands:

$$3\ 4\ +\ 5\ \times\ 6\ -$$

$$7\quad 5\ \times\ 6\ -$$

$$35\quad 6\ -$$

$$29$$

Other example:

3 4 5 × + 6 –

3 20 + 6 –

23 6 –

17

3 4 5 6 – × +

3 4 –1 × +

3 –4 +

–1

Benefits:

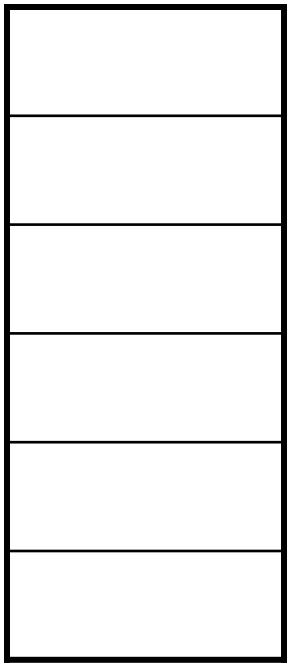
- No ambiguity and no brackets are required
- It is the same process used by a computer to perform computations:
  - operands must be loaded into registers before operations can be performed on them
- Reverse-Polish can be processed using stacks

The easiest way to parse reverse-Polish notation is to use an operand stack:

- operands are processed by pushing them onto the stack
- when processing an operator:
  - pop the last two items off the operand stack,
  - perform the operation, and
  - push the result back onto the stack

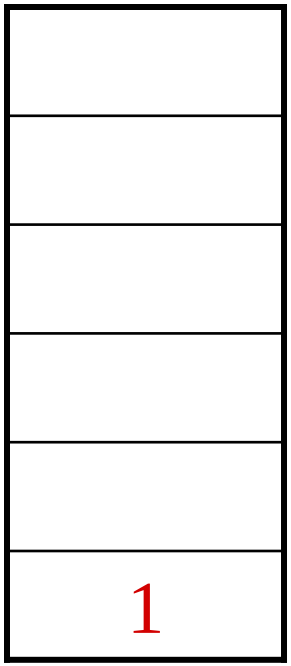
Evaluate the following reverse-Polish expression using a stack:

1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



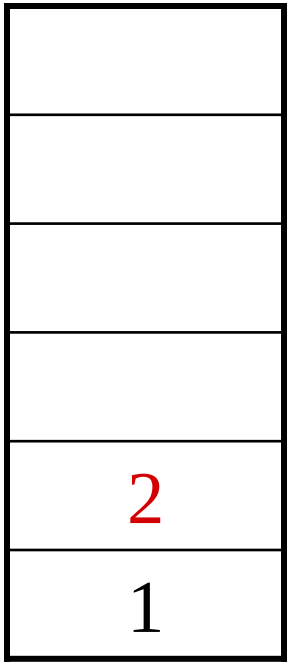
Push 1 onto the stack

1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



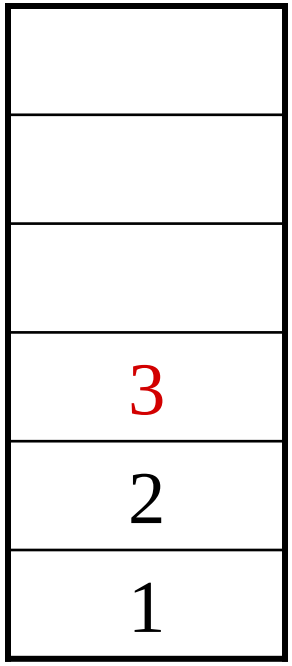
Push 1 onto the stack

1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



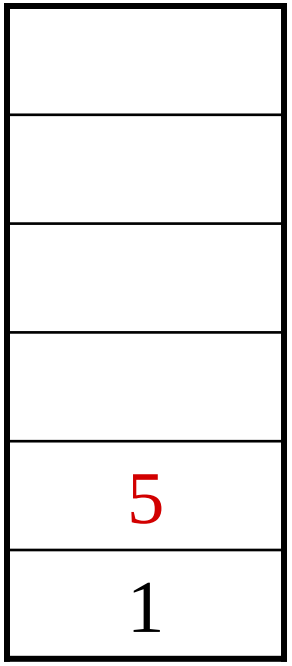
Push 3 onto the stack

1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



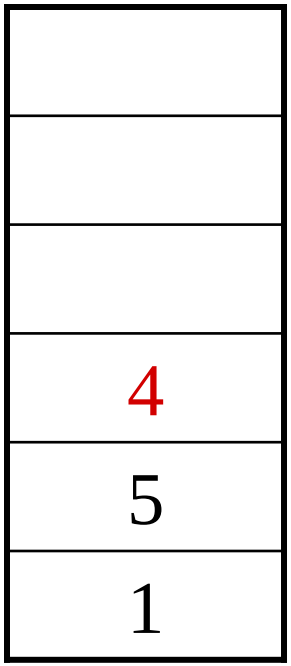
Pop 3 and 2 and push 2 + 3 = 5

1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Push 4 onto the stack

1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Push 5 onto the stack

1 2 3 + 4 5 6 × – 7 × + – 8 9 × +

| |
|---|
| |
| |
| 5 |
| 4 |
| 5 |
| 1 |

Push 6 onto the stack

1 2 3 + 4 5 6 × – 7 × + – 8 9 × +

| |
|---|
| 6 |
| 5 |
| 4 |
| 5 |
| 1 |

Pop 6 and 5 and push 5 × 6 = 30

1 2 3 + 4 5 6 × – 7 × + – 8 9 × +

| |
|---|
| |
| |
| 30 |
| 4 |
| 5 |
| 1 |

Pop 30 and 4 and push 4 – 30 = –26

1 2 3 + 4 5 6 × – 7 × + – 8 9 × +

| |
|---|
| |
| |
| |
| –26 |
| 5 |
| 1 |

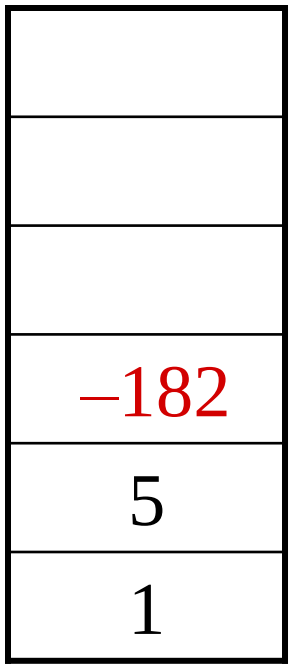
Push 7 onto the stack

1 2 3 + 4 5 6 × – 7 × + – 8 9 × +

| |
|---|
| |
| |
| 7 |
| –26 |
| 5 |
| 1 |

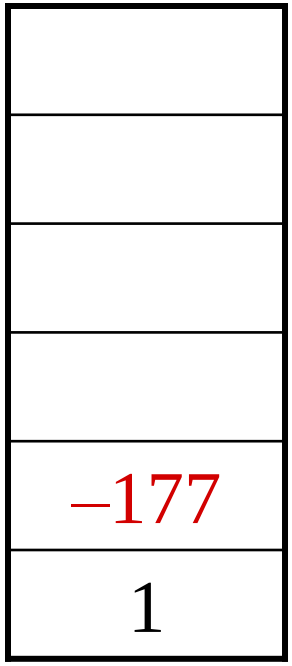
Pop 7 and –26 and push –26 × 7 = –182

1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



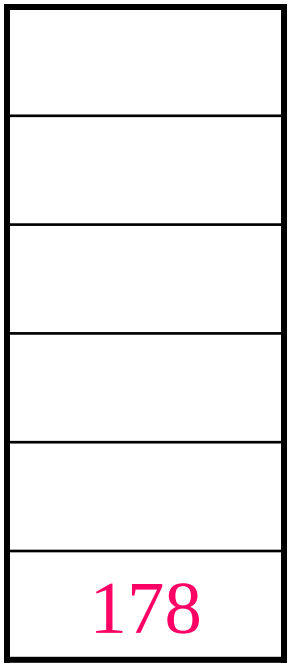
Pop –182 and 5 and push –182 + 5 = –177

1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



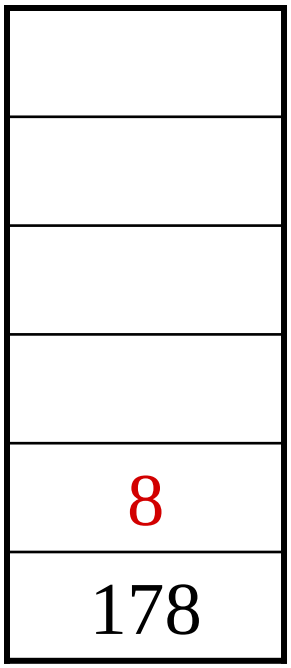
Pop –177 and 1 and push 1 – (–177) = 178

1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



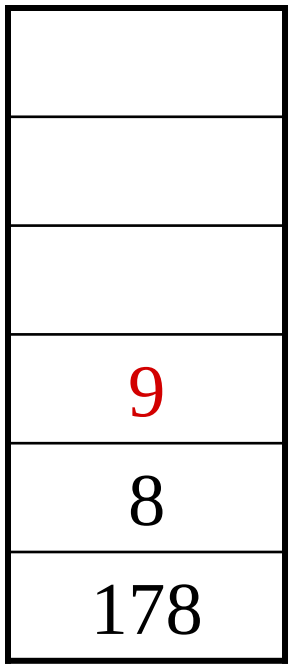
Push 8 onto the stack

1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



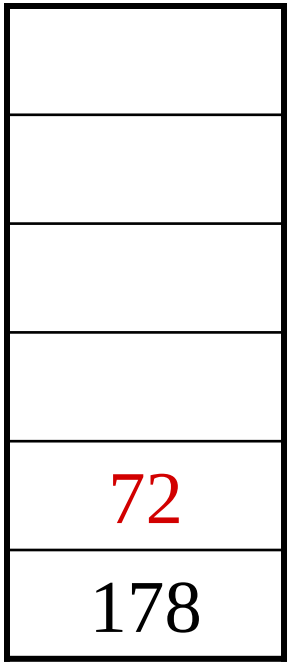
Push 1 onto the stack

1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



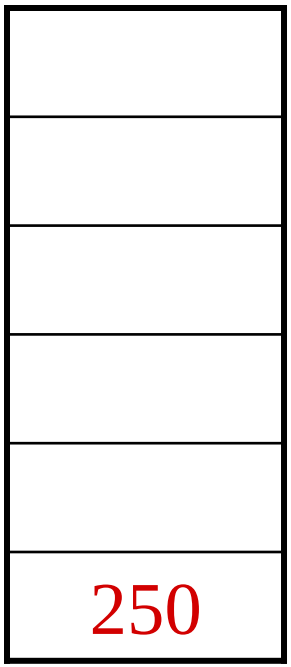
Pop 9 and 8 and push 8 × 9 = 72

1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Pop 72 and 178 and push 178 + 72 = 250

1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Thus

$$1\ 2\ 3\ +\ 4\ 5\ 6\ \times\ -\ 7\ \times\ +\ -\ 8\ 9\ \times\ +$$

evaluates to the value on the top: 250

The equivalent in-fix notation is

$$((1 - ((2 + 3) + ((4 - (5 \times 6)) \times 7))) + (8 \times 9))$$

We reduce the parentheses using order-of-operations:

$$1 - (2 + 3 + (4 - 5 \times 6) \times 7) + 8 \times 9$$

Incidentally,

$$1 - 2 + 3 + 4 - 5 \times 6 \times 7 + 8 \times 9 = -132$$

which has the reverse-Polish notation of

$$1\ 2\ -\ 3\ +\ 4\ +\ 5\ 6\ 7\ \times\ \times\ -\ 8\ 9\ \times\ +$$

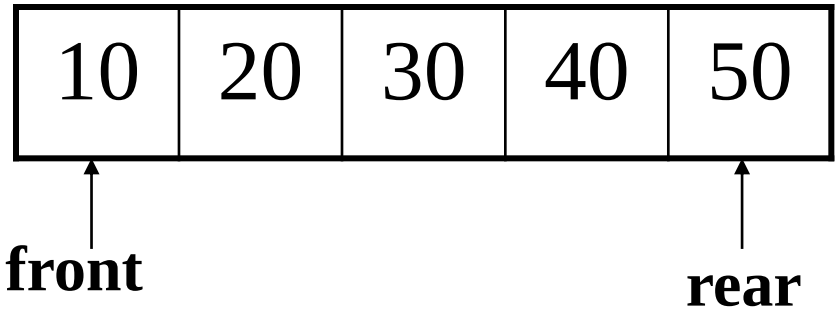
For comparison, the calculated expression was

$$1\ 2\ 3\ +\ 4\ 5\ 6\ \times\ -\ 7\ \times\ +\ -\ 8\ 9\ \times\ +$$

# Queue

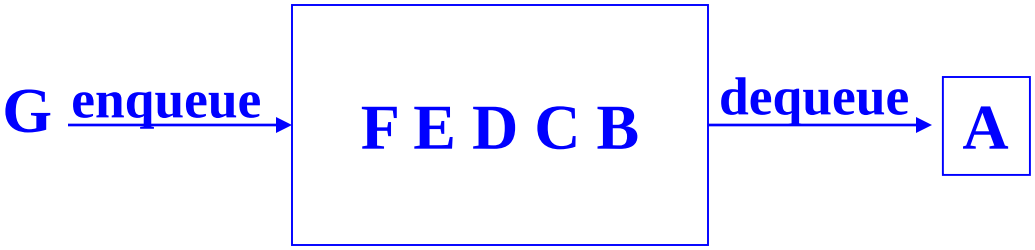
- Queue are first in first out type of data structure (i.e. FIFO)
- In a queue new elements are added to the queue from one end called REAR end and the element are always removed from other end called the FRONT end.
- The people standing in a railway reservation row are an example of queue.

- Each new person comes and stands at the end of the row and person getting their reservation confirmed get out of the row from the front end.
- The bellow show figure how the operations take place on a stack:



# Queue ADT

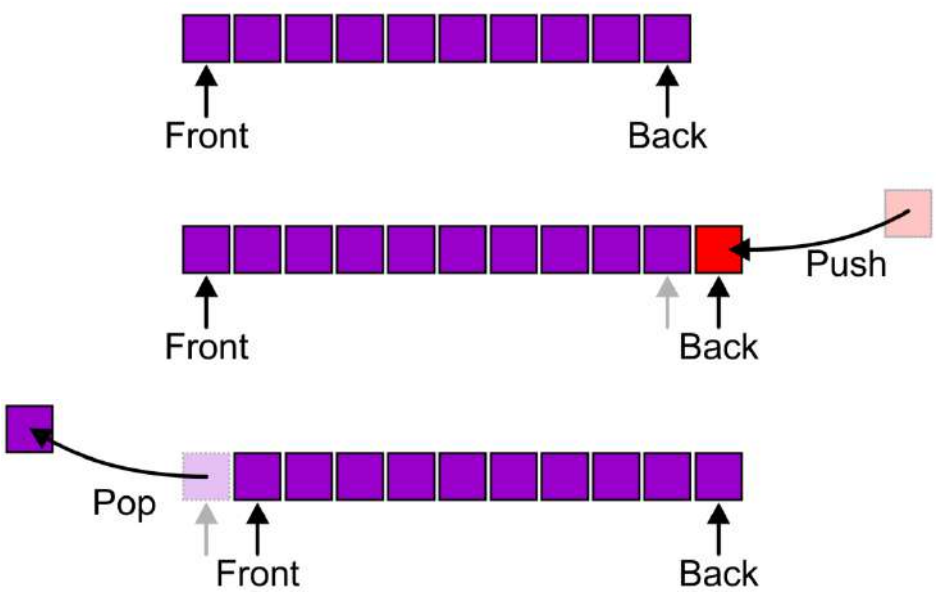
- Queue operations
  - create
  - destroy
  - enqueue
  - dequeue
  - is_empty
- Queue property: if x is enQed before y is enQed, then x will be deQed before y is deQed



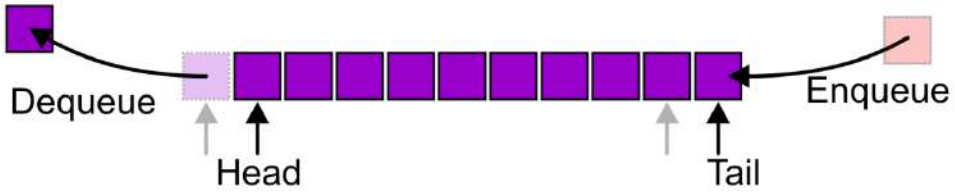
# Abstract Queue

Also called

- Graphical



Alternative terms may be used for the four operations on a queue, including:




There are two exceptions associated with this abstract data structure:

It is an undefined operation to call either pop or front on an empty queue

# Implementations

We will look at two implementations of queues:

- Singly linked lists
- Circular arrays

Requirements:

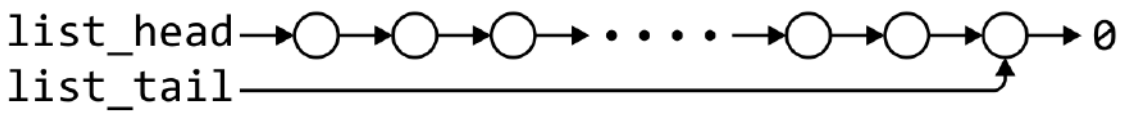
- All queue operations must run in $\Theta(1)$ time

# Linked-List Implementation

Removal is only possible at the front with $\Theta(1)$ run time

| | Front/1st | Back/$n^{th}$ |
|---|---|---|
| Find | $\Theta(1)$ | $\Theta(1)$ |
| Insert | $\Theta(1)$ | $\Theta(1)$ |
| Erase | $\Theta(1)$ | $\Theta(n)$ |

The desired behaviour of an Abstract Queue may be reproduced by performing insertions at the back



# Single_list Definition

The definition of single list class from Project 1 is:

```
template <typename Type>
class Single_list {
        public:
                int size() const;
                bool empty() const;
                Type front() const;
                Type back() const;
                Single_node<Type> *head() const;
                Single_node<Type> *tail() const;
                int count( Type const & ) const;

                void push_front( Type const & );
                void push_back( Type const & );
                Type pop_front();
                int erase( Type const & );
};
```

The queue class using a singly linked list has a single private member variable:  a singly linked list

```
template <typename Type>
class Queue{
    private:
        Single_list<Type> list;
    public:
        bool empty() const;
        Type front() const;
        void push( Type const & );
        Type pop();
};
```

# Array Implementation

A one-ended array does not allow all operations to occur in $\Theta(1)$ time

| | Front/1st | Back/$n^{th}$ |
|---|---|---|
| Find | $\Theta(1)$ | $\Theta(1)$ |
| Insert | $\Theta(n)$ | $\Theta(1)$ |
| Erase | $\Theta(n)$ | $\Theta(1)$ |

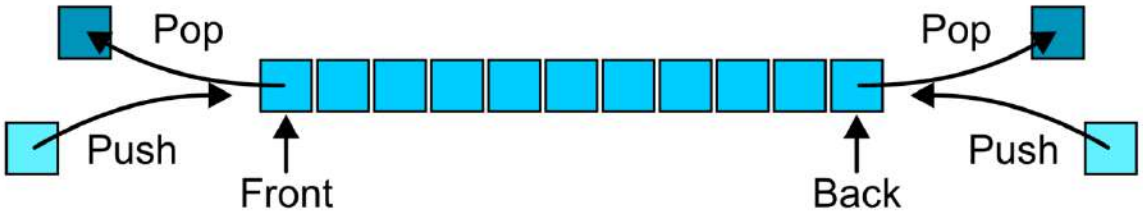
Using a two-ended array, $\Theta(1)$ are possible by pushing at the back and popping from the front


| | Front/$1^{st}$ | Back/$n^{th}$ |
|---|---|---|
| Find | $\Theta(1)$ | $\Theta(1)$ |
| Insert | $\Theta(1)$ | $\Theta(1)$ |
| Remove | $\Theta(1)$ | $\Theta(1)$ |

# Abstract Deque

An Abstract Deque (Deque ADT) is an abstract data structure which emphasizes specific operations:

- Uses a explicit linear ordering
- Insertions and removals are performed individually
- Allows insertions at both the front and back of the deque



The operations will be called

| | |
|---|---|
| `front` | `back` |
| `push_front` | `push_back` |
| `pop_front` | `pop_back` |

There are four errors associated with this abstract data type:

- It is an undefined operation to access or pop from an empty deque

# Applications

Useful as a general-purpose tool:

- Can be used as either a queue or a stack

Problem solving:

- Consider solving a maze by adding or removing a constructed path at the front
- Once the solution is found, iterate from the back for the solution

# Linked List

- A linked list is a data structure where each object is stored in a *node*
- As well as storing data, the node must also contain a reference/pointer to the node containing the next item of data
- We must dynamically create the nodes in a linked list
- Thus, because `new` returns a pointer, the logical manner in which to track a linked lists is through a pointer
- A `Node` class must store the data and a reference to the next node (also a pointer)

# Node Class

The node must store data and a pointer:

```
class Node {
    public:
        Node( int = 0, Node * = nullptr );

        int value() const;
        Node *next() const;

    private:
        int node_value;
        Node *next_node;
};
```

# Node Constructor

The constructor assigns the two member variables based on the arguments

```
List::Node::Node( int e, Node *n ):
node_value( e ),
next_node( n ) {
    // empty constructor
}
```

The default values are given in the class definition:

```
class Node {
    public:
        Node( int = 0, Node * = nullptr );
        int value() const;
        Node *next() const;
    private:
        int node_value;
        Node *next_node;
};
```

# Accessors

The two member functions are accessors which simply return the **`node_value`** and the **`next_node`** member variables, respectively

```
int List::Node::value() const {
    return node_value;
}

List::Node *List::Node::next() const {
    return next_node;
}
```

Member functions that do not change the object acted upon are variously called *accessors*, *readonly functions*, *inspectors*, and, when it involves simply returning a member variable, *getters*

# Structure

To begin, let us look at the internal representation of a linked list

Suppose we want a linked list to store the values

**42 95 70 81**

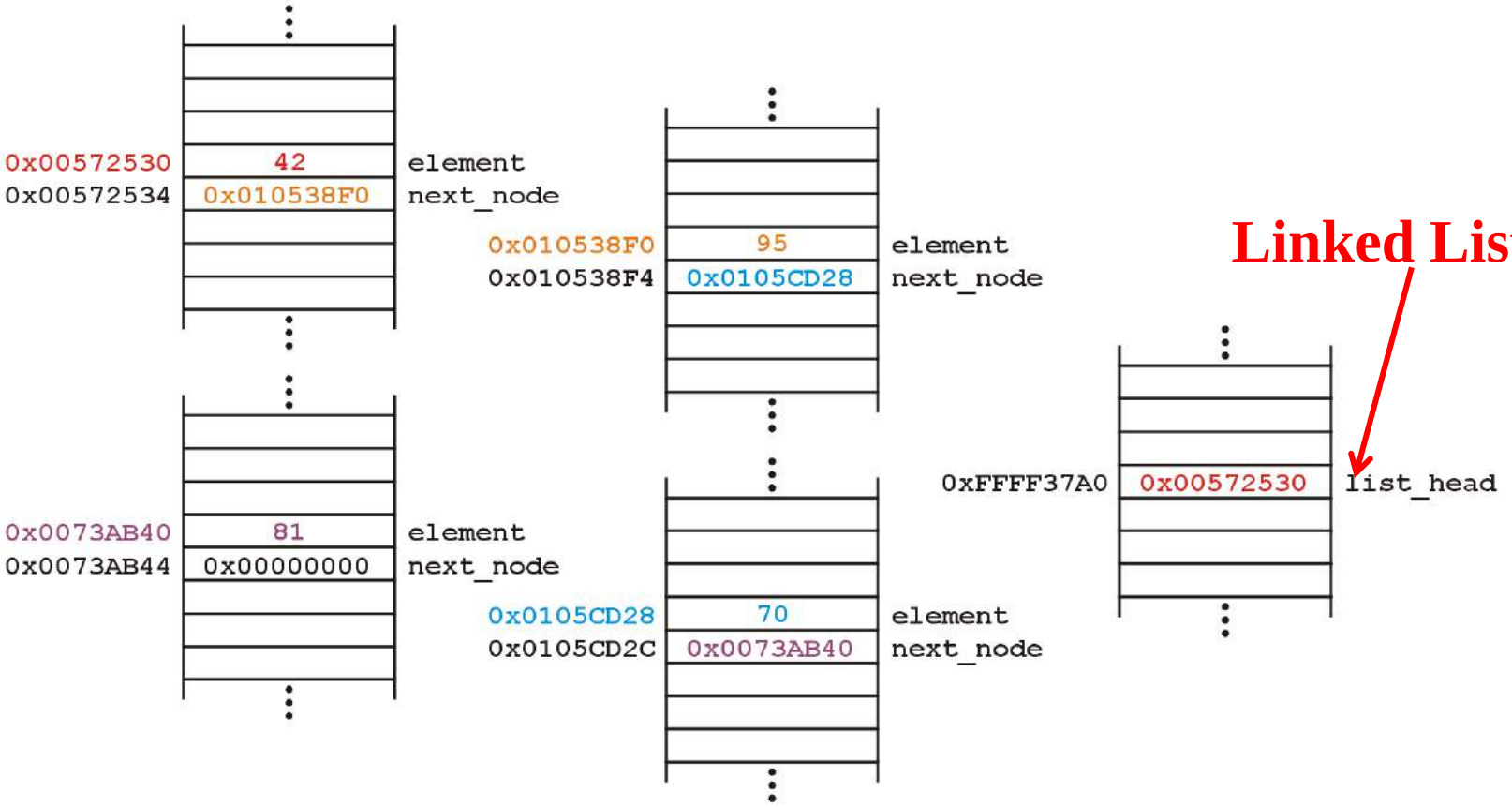
in this order

A linked list uses linked allocation, and therefore each node may appear anywhere in memory

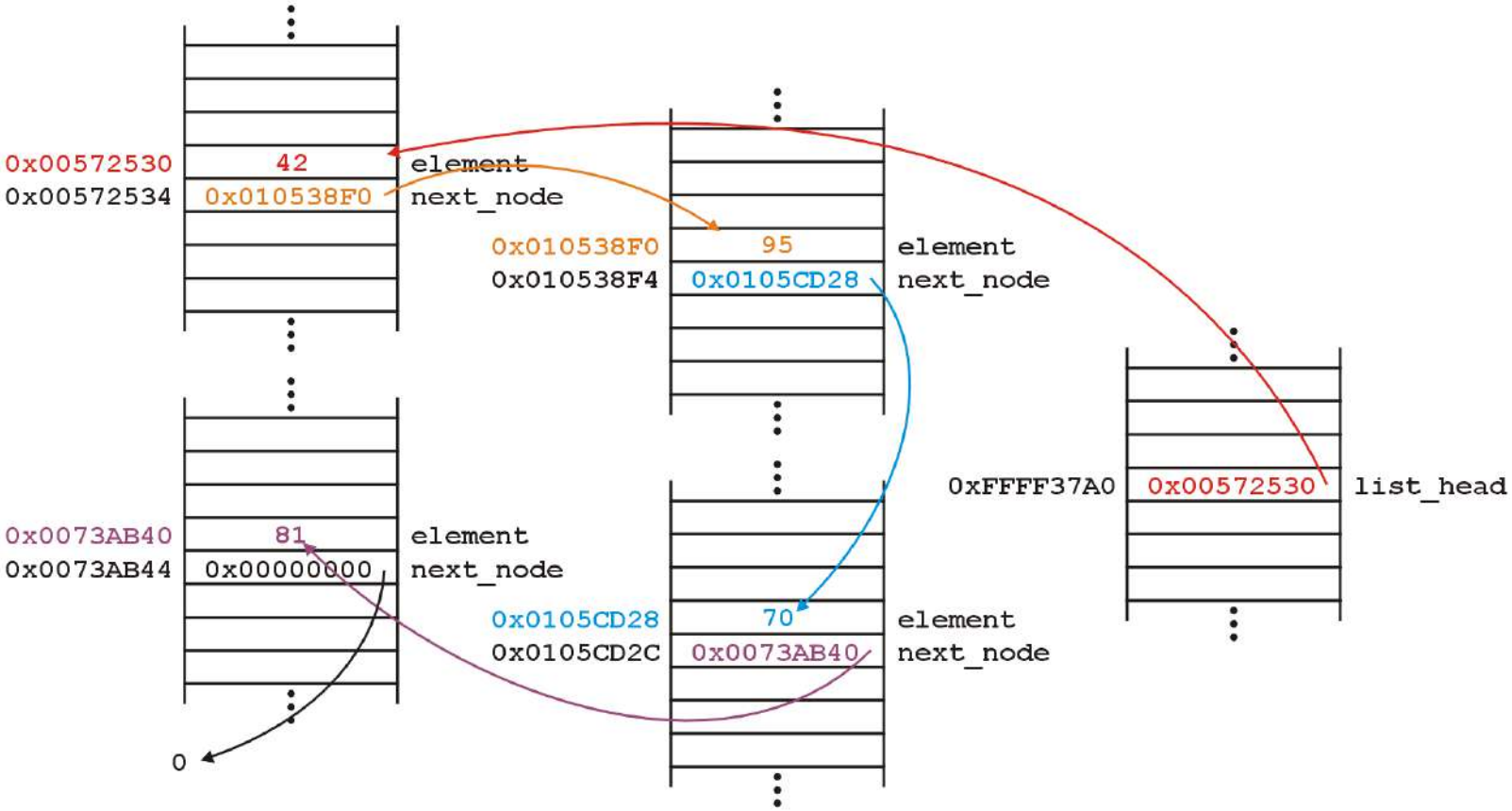
Also the memory required for each node equals the memory required by the member variables

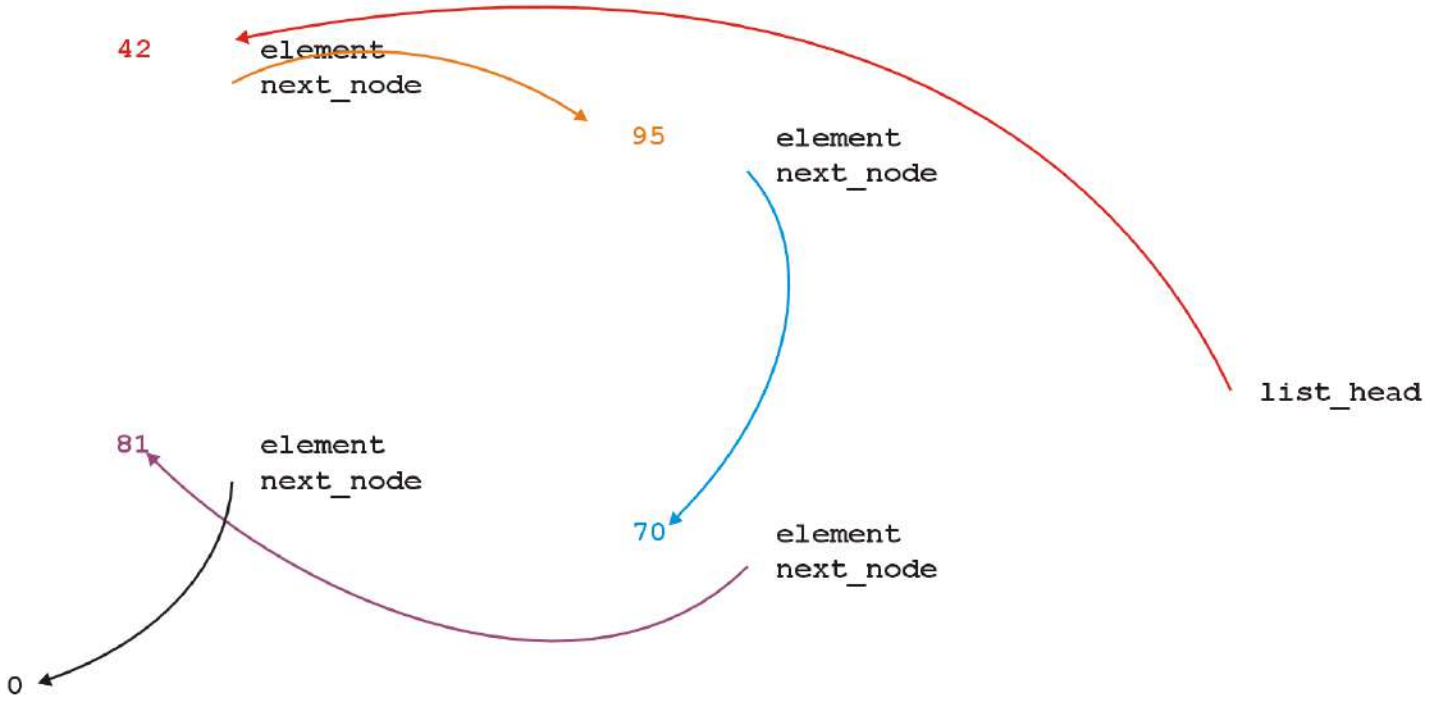
- 4 bytes for the linked list (a pointer)
- 8 bytes for each node (an **int** and a pointer)
- We are assuming a 32-bit machine

Such a list could occupy memory as follows:



The **next_node** pointers store the addresses of the next node in the list




42
element
next_node
95
element
next_node
list_head
81
element
next_node
70
element
next_node
0

We will clean up the representation as follows:


We do not specify the addresses because they are arbitrary and:

- The contents of the circle is the value
- The `next_node` pointer is represented by an arrow

# Operations

First, we want to create a linked list

We also want to be able to:

- insert into,
- access, and
- erase from

the values stored in the linked list

We can do them with the following operations:

- Adding, retrieving, or removing the value at the front of the linked list

```
void push_front( int );
int front() const;
void pop_front();
```

- We may also want to access the head of the linked list

```
Node *begin() const;
```

All these operations relate to the first node of the linked list

We may want to perform operations on an arbitrary node of the linked list, for example:

- Find the number of instances of an integer in the list:

```
int count( int ) const;
```

- Remove all instances of an integer from the list:

```
int erase( int );
```

# Linked Lists

Additionally, we may wish to check the state:

- Is the linked list empty?

```
bool empty() const;
```

- How many objects are in the list?

```
int size() const;
```

The list is empty when the `list_head` pointer is set to `nullptr`

Consider this simple (but incomplete) linked list class:

```
class List {
    public:
     class Node {...};
        List();
        // Accessors
        bool empty() const;
     int size() const;
        int front() const;
        Node *begin() const;
        Node *end() const;
        int count( int ) const;
        // Mutators
        void push_front( int );
        int pop_front();
        int erase( int );

    private:
        Node *list_head;
};
```

The constructor initializes the linked list

We do not count how may objects are in this list, thus:

- we must rely on the last pointer in the linked list to point to a special value
- in C++, that standard value is **`nullptr`**

# Allocation

The constructor is called whenever an object is created, either:

Statically

The statement `List ls;` defines `ls` to be a linked list and the compiler deals with memory allocation

Dynamically

The statement

```
List *pls = new List();
```

requests sufficient memory from the OS to store an instance of the class

- In both cases, the memory is allocated and then the constructor is called

# Static Allocation

Example:

```
int f() {
    List ls;
// ls is declared as a local variable on the stack
    ls.push_front( 3 );
    cout << ls.front() << endl;
// The return value is evaluated
// The compiler then calls the destructor for local variables
    // The memory allocated for 'ls' is deallocated

    return 0;
}
```

# Dynamic Allocation

Example:

```
List *f( int n ) {
    List *pls = new List();  // pls is allocated memory by the OS

    pls->push_front( n );
    cout << pls->front() << endl;

    // The address of the linked list is the return value
    // After this, the 4 bytes for the pointer 'pls' is deallocated
    // The memory allocated for the linked list is still there

    return pls;
}
```

# Static Allocation

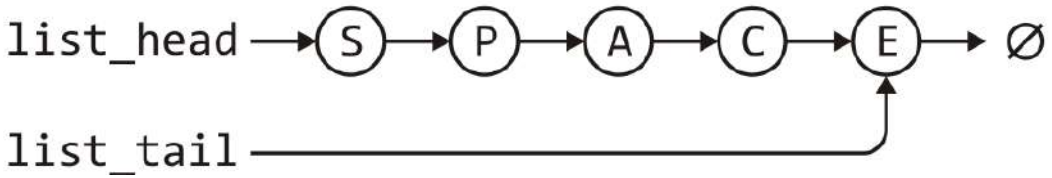
```
List *f() {
  List ls;   // ls is declared as a local variable on the stack
  ls.push_front( 3 );
  cout << ls.front() << endl;

  // The return value is evaluated
  // The compiler then calls the destructor for local variables
  // The memory allocated for 'ls' is deallocated

  return &ls;
}
```

# Using an array?

Suppose we store this linked list in an array?




```
list_head = 5;
list_tail = 2;
```

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| A | | E | P | | S | C | |
| 6 | | -1 | 0 | | 3 | 2 | |

Rather than using, `-1`, use a constant assigned that value

- This makes reading your code easier

```
list_head = 5;
list_tail = 2;
```

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| A | | E | P | | S | C | |
| 6 | | NULLPTR | 0 | | 3 | 2 | |

To achieve this, we must create an array of objects that:

- Store the value
- Store the array index where the next entry is stored

```
template <typename Type>
class Single_node {
    private:
        Type node_value;
        next_node int;
    public:
        Type value() const;
        int next() const;
};
```

Now, memory allocation is done once in the constructor:

```
template <typename Type>
class Single_list {
    private:
        int list_capacity;
        int list_head;
        int list_tail;
        int list_size;
        Single_node<Type> *node_pool;

        static const int NULL;
    public:
        Single_list( int = 16 );
        // member functions
};

const int Single_list::NULLPTR = -1;
```

```
template <typename Type>
Single_list<Type>::Single_list( int n ):
list_capacity( std::max( 1, n ) ),
list_head( NULLPTR ),
list_tail( NULLPTR ),
list_size( 0 ),
node_pool( new Single_node<Type>[n] ) {
    // Empty constructor
}
```


THANK YOU