The term **DATA STRUCTURE** is used to describe the way data is stored, and the term algorithm is used to describe the way data is processed. Data structures and algorithms are interrelated. Choosing a data structure affects the kind of algorithm you might use, and choosing an algorithm affects the data structures we use.

**Introduction to Data Structures:**

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

To develop a program of an algorithm we should select an appropriate data structure for that algorithm. Therefore, data structure is represented as:

**Algorithm + Data structure = Program**

A data structure is said to be *linear* if its elements form a sequence or a linear list. The linear data structures like an array, stacks, queues and linked lists organize data in linear order. A data structure is said to be *non linear* if its elements form a hierarchical classification where, data items appear at various levels. Trees and Graphs are widely used non-linear data structures. Tree and graph structures represent hierarchial relationship between individual data elements. Graphs are nothing but

trees with certain restrictions removed.

Data structures are divided into two types:

* Primitive data structures.
* Non-primitive data structures.

Classification of DataStructures

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  | | Data Structures | | | | | | | | |  |  |  |  |  |  |  |  |  |  | |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  | | |  |  |  |  |  |  |  |  |  | |  |  |  |  |  |  |  |  | | |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  | | |  |  |  |  |  |  |  |  |  | |  |  |  |  |  |  |  |  | | |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  | | |  |  |  |  |  |  |  |  |  | |  |  |  |  |  |  |  |  | | |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  | |  |  |  |  | | |  |  |  |  |  |  |  |  |  | |  |  |  |  |  |  |  |  | | |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  | Primitive Data Structures | | | | | | | | | | | |  |  |  |  |  |  |  |  | Non Primitive data structure | | | | | | | | | | | | | | | | | | | |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  | | |  |  |  |  |  |  |  |  |  | |  |  |  |  |  |  |  |  |  | |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  | |  |  |  |  |  |  |  |  |  | | | |  |  |  |  |  |  |  | | |  |  |  |  |  |  |  |  |  | |  |  |  |  |  |  |  |  | |  |  |
|  |  |  | |  |  |  |  |  |  |  |  |  | | | |  |  |  |  |  |  |  | | |  |  |  |  |  |  |  |  |  | |  |  |  |  |  |  |  |  | |  |  |
|  |  |  | |  |  |  |  |  |  |  |  |  | | | |  |  |  |  |  |  |  | |  |  |  |  |  |  |  |  |  |  | |  |  |  |  |  |  |  |  | |  |  |
|  |  | | |  |  | |  |  |  | |  |  | | | | |  |  |  |  |  |  | | |  |  |  |  |  |  |  |  |  | |  |  |  |  |  |  |  | | |  |  |
|  | int | | |  | float | |  |  | Char | |  | pointers | | | | |  |  |  |  |  | Array s | | | | |  |  |  |  | |  | lists | | | | |  |  |  | Files | | | |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | |  |  |  |  |  |  |  | | |  |  |  |  |  |  | |  |  |  | |  |  |  |  |  |  | | |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | |  |  |  |  |  |  |  | | |  |  | |  |  | | |  |  |  | |  |  | |  | |  | | |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | |  |  |  |  |  |  |  | | |  |  | |  |  | | |  |  |  | |  |  | |  | |  | | |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | |  |  |  |  |  |  |  | | |  |  | |  |  | | |  |  |  | |  |  | |  |  |  | | |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | |  |  |  |  |  |  |  | | |  | | |  |  | | |  |  |  | |  |  | |  | |  | | |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | |  |  |  |  |  | Linear | | | | | | | |  | | |  |  |  | | Non Linear | | | | | | | |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | |  |  |  |  |  | |  | |  |  |  | |  |  | | |  |  |  | |  |  | |  |  |  | | |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | |  |  |  |  | | |  | |  |  |  | |  | | | |  |  |  | | |  | |  |  |  | | | |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | |  |  |  |  | | |  | |  |  |  | |  | | | |  |  |  | | |  | |  |  |  | | | |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | |  |  |  |  | | |  | |  |  |  | |  | | | |  |  |  | | |  | |  |  |  | |  | |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | |  |  |  |  | | | | |  |  |  | | | | | |  |  |  | | | | |  |  |  | | | |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | |  |  |  | Stack | | | | |  |  | Queue | | | | | |  |  | Graphs | | | | |  |  | Trees | | | |  |
|  |  |  |  |  |  |  |  |  |  | |  | | | | | | | |  | | | | | |  |  | | | | | |  |  |  | |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  | |  | | | | | | | | | | | | | | | | | | | | | |  |  | |  |  |  |  |  |  |  |  |  |  |

**Primitive Data Structures** are the basic data structures that directly operate upon the machineinstructions. They have different representations on different computers. Integers, floating point numbers, character constants, string constants and pointers come under this category.



**Non-primitive data structures** are more complicated data structures and are derived fromprimitive data structures. They emphasize on grouping same or different data items with relationship between each data item. Arrays, lists and files come under this category.

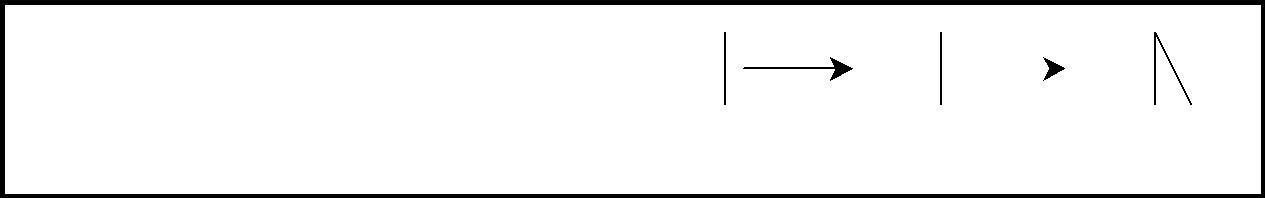


**Data structures: Organization of data**

The collections of data you work with in a program have some kind of structure or organization. No matter how complex your data structures are they can be broken down into two fundamental types:

* Contiguous
* Non-Contiguous.

In contiguous structures, terms of data are kept together in memory (either RAM or in a file). An array is an example of a contiguous structure. Since each element in the array is located next to one or two other elements. In contrast, items in a non-contiguous structure and scattered in memory, but we linked to each other in some way. A linked list is an example of a non-contiguous data structure. Here, the nodes of the list are linked together using pointers stored in each node



|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 |  | 2 |  | 3 |  | 1 |  | 2 |  |  |  | 3 |
|  |  |
|  |  |  |  |  |  |  |  |  | |  |  |  |
|  | (a) Contiguous | | |  |  |  | (b) non-contiguous | | | |  |  |

Contiguous and Non-contiguous structures compared

**Abstract Data Type (ADT):**

The design of a data structure involves more than just its organization. You also need to plan for the way the data will be accessed and processed – that is, how the data will be interpreted actually, non-contiguous structures – including lists, tree and graphs – can be implemented either contiguously or non- contiguously like wise, the structures that are normally treated as contiguously - arrays and structures – can also be implemented non-contiguously.

The notion of a data structure in the abstract needs to be treated differently from what ever is used to implement the structure. The abstract notion of a data structure is defined in terms of the operations we plan to perform on the data.

Considering both the organization of data and the expected operations on the data, leads to the notion of an abstract data type. An *abstract data type* in a theoretical construct that consists of data as well as the operations to be performed on the data while hiding implementation.

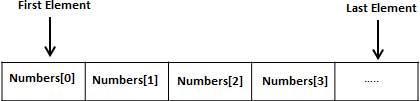
For example, a stack is a typical abstract data type. Items stored in a stack can only be added and removed in certain order – the last item added is the first item removed. We call these operations, pushing and popping. In this definition, we haven‟t specified have items are stored on the stack, or how the items are pushed and popped. We have only specified the valid operations that can be performed.To be made useful, an abstract data type (such as stack) has to be implemented and this is where data structure comes into ply. For instance, we might choose the simple data structure of an array to represent the stack, and then define the appropriate indexing operations to perform pushing and popping.

**ARRAY**

Arrays a kind of data structure that can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

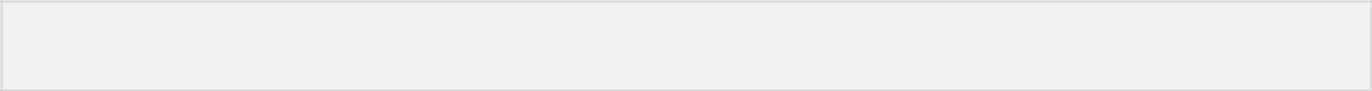
Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



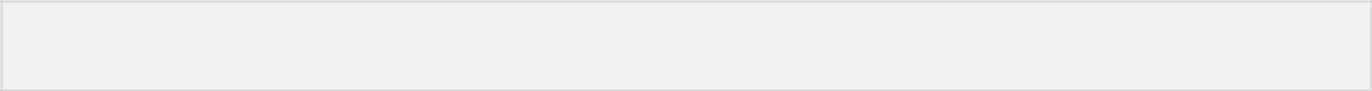
**Declaring Arrays**

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows −



type arrayName [ arraySize ];

This is called a *single-dimensional* array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C data type. For example, to declare a 10-element array called **balance** of type double, use this statement −

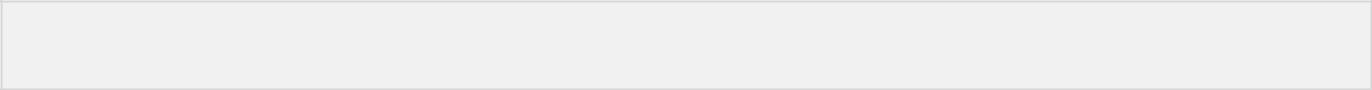


double balance[10];

Here *balance* is a variable array which is sufficient to hold up to 10 double numbers.

**Initializing Arrays**

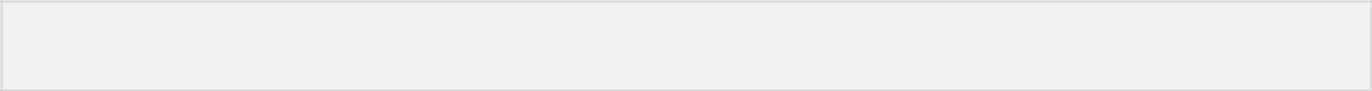
You can initialize an array in C either one by one or using a single statement as follows −



double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};

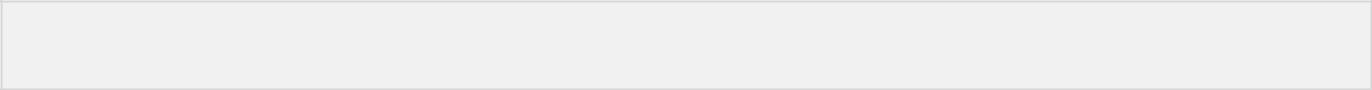
The number of values between braces { } cannot be larger than the number of elements that we declare for the array between square brackets [ ].

If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write −



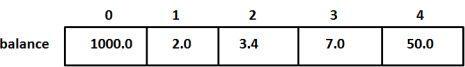
double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};

You will create exactly the same array as you did in the previous example. Following is an example to assign a single element of the array −



balance[4] = 50.0;

The above statement assigns the 5th element in the array with a value of 50.0. All arrays have 0 as the index of their first element which is also called the base index and the last index of an array will be total size of the array minus 1. Shown below is the pictorial representation of the array we discussed above



**Types of Arrays**

1. .Single Dimensional Array :



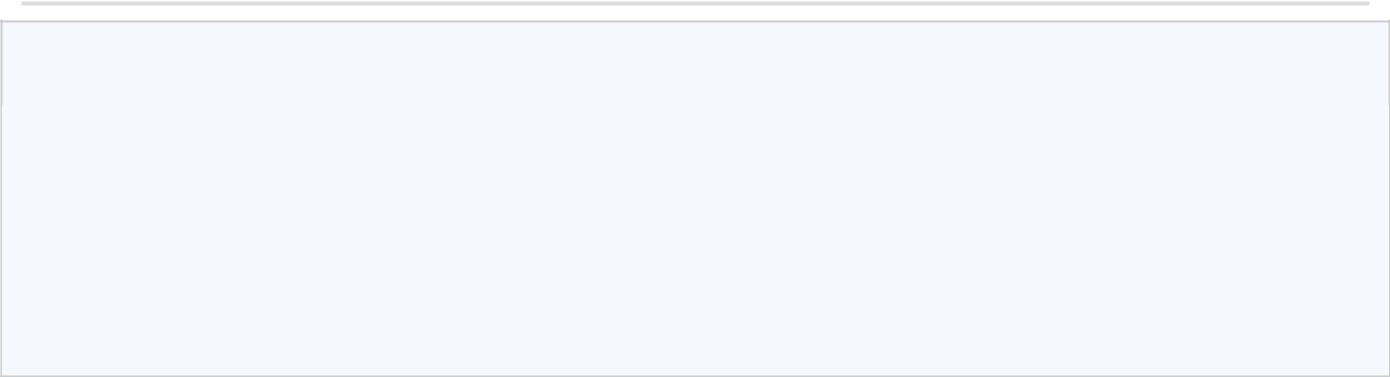
1. Single or One Dimensional array is used to represent and store data in a linear form.
2. Array having only one subscript variable is called One-Dimensional array
3. It is also called as Single Dimensional Array or Linear Array

**Syntax :**

****

<data-type> <array\_name> [size];

**Example** of Single Dimensional Array :



int iarr[3] = {2, 3, 4};

char carr[20] = "c4learn" ;

float farr[3] = {12.5,13.5,14.5} ;

1. Multi Dimensional Array :
   1. Array having more than one subscript variable is called [Multi-Dimensional array.](http://www.c4learn.com/c-programming/c-multidimensional-array/)
   2. Multi Dimensional Array is also called as Matrix.



**Syntax :**

****

<data-type> <array\_name> [row\_subscript][column-subscript];

**Example :** Two Dimensional Array



int a[3][3] = { 1,2,3

5,6,7

8,9,0 }

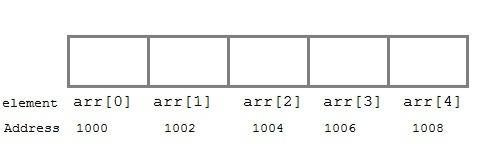
3. Pointer Arrays

When an array is declared, compiler allocates sufficient amount of memory to contain all the elements of the array. Base address which gives location of the first element is also allocated by the compiler.

Suppose we declare an array **arr**,

int arr[5]={ 1, 2, 3, 4, 5 };

Assuming that the base address of **arr** is 1000 and each integer requires two byte, the five element will be stored as follows



Here variable **arr** will give the base address, which is a constant pointer pointing to the element, **arr[0]**. Therefore **arr** is containing the address of **arr[0]** i.e 1000.

**arr** *is equal to* **&arr[0]** // by default



We can declare a pointer of type int to point to the array **arr**.

int \*p;

p = arr;

or p = &arr[0]; //both the statements are equivalent.

Now we can access every element of array **arr** using **p++** to move from one element to another.



**NOTE :** You cannot decrement a pointer once incremented.p--won't work.

*Pointer to Array*

As studied above, we can use a pointer to point to an Array, and then we can use that pointer

to access the array. Lets have an example,

int i;

int a[5] = {1, 2, 3, 4, 5};

int \*p = a; *// same as int\*p = &a[0]*

for (i=0; i<5; i++)

{

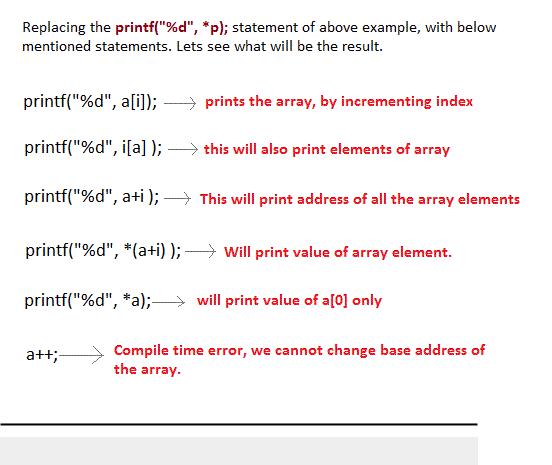
printf("%d", \*p);

p++;

}

In the aboce program, the pointer \*p will print all the values stored in the array one by one.

We can also use the Base address (a in above case) to act as pointer and print all the values.



*Pointer to Multidimensional Array*

**

A multidimensional array is of form, a[i][j] . Lets see how we can make a pointer point to such an array. As we know now, name of the array gives its base address. In a[i][j] , a will give the base address of this array, even a+0+0 will also give the base address, that is the address of a[0][0] element.



Here is the generalized form for using pointer with multidimensional arrays. \*(\*(ptr + i) + j) *is same as* a[i][j]

**STACK**

A stack is a list of elements in which an element may be inserted or deleted only at one end, called the top of the stack. Stacks are sometimes known as LIFO (last in, first out) lists.

As the items can be added or removed only from the top i.e. the last item to be added to a stack is the first item to be removed.

The two basic operations associated with stacks are:

* *Push*: is the term used to insert an element into a stack.
* *Pop*: is the term used to delete an element from a stack.

“Push” is the term used to insert an element into a stack. “Pop” is the term used to delete an element from the stack.

All insertions and deletions take place at the same end, so the last element added to the stack will be the first element removed from the stack. When a stack is created, the stack base remains fixed while the stack top changes as elements are added and removed. The most accessible element is the top and the least accessible element is the bottom of the stack.

**Representation of Stack:**

Let us consider a stack with 6 elements capacity. This is called as the size of the stack. The number of elements to be added should not exceed the maximum size of the stack. If we attempt to add new element beyond the maximum size, we will encounter a *stack overflow* condition. Similarly, you cannot remove elements beyond the base of thestack. If such is the case, we will reach a *stack underflow* condition. When an element is added to a stack, the operation is performed by push().

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  | 4 |  |  |  | 4 |  |  |  | 4 |  |  |  | 4 |
|  |  |  | 3 |  |  |  | 3 |  |  |  | 3 |  |  |  | 3 |
|  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  | TOP | |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  | 33 |  |
|  |  |  | 2 |  |  |  | 2 | TOP | |  | 2 |  |  | 2 |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  | 22 |  |  |  | 22 |  |
|  |  |  | 1 | TOP | |  | 1 |  |  | 1 |  |  | 1 |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  | 11 |  |  |  | 11 |  |  |  | 11 |  |
| TOP | |  | 0 |  |  | 0 |  |  | 0 |  |  | 0 |
|  |  |  |  |  |  |  |  |  |  |
|  |  | Empty | |  |  | Insert |  |  |  | Insert |  |  |  | Insert |  |
|  |  | Stack | |  |  | 11 |  |  |  | 22 |  |  |  | 33 |  |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  | 4 |  |  |  | 4 |  |  |  | 4 |  |  |  | 4 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| TOP | |  | 3 |  |  |  | 3 |  |  |  | 3 |  |  |  | 3 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  | 33 | 2 | TOP | |  | 2 |  |  |  | 2 |  |  |  | 2 |
|  |  |  |  |  |  |  |  |  |  |
|  |  | 22 |  |  |  | 22 |  |  |  |  |  |  |  |  |  |
|  |  | 1 |  |  | 1 | TOP | |  | 1 |  |  |  | 1 |
|  |  |  |  |  |  |  |  |  |  |
|  |  | 11 |  |  |  | 11 |  |  |  | 11 |  |  |  |  |  |
|  |  | 0 |  |  | 0 |  |  | 0 | TOP | |  | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  | Initial |  |  |  | POP |  |  |  | POP |  |  |  | POP | |
|  |  | Stack |  |  |  |  |  |  |  |  |  |  |  | Empty | |

**Algebraic Expressions:**

An algebraic expression is a legal combination of operators and operands. Operand is the quantity on which a mathematical operation is performed. Operand may be a variable like x, y, z or a constant like 5, 4, 6 etc. Operator is a symbol which signifies a mathematical or logical operation between the operands. Examples of familiar operators include +, -, \*, /, ^ etc.

An algebraic expression can be represented using three different notations. They are infix, postfix and prefix notations:

**Infix:** It is the form of an arithmetic expression in which we fix (place) the arithmetic operator in between the two operands.

Example: (A + B) \* (C - D)

**Prefix:** It is the form of an arithmetic notation in which we fix (place) the arithmetic operator before (pre) its two operands.

The prefix notation is called as polish notation Example: \* + A B – C D

**Postfix:** It is the form of an arithmetic expression in which we fix (place) the arithmetic operator after (post) its two operands. The postfix notation is called as *suffix* *notation* and is also referred to *reverse polish notation*.

Example: A B + C D - \*

The three important features of postfix expression are:

1. The operands maintain the same order as in the equivalent infix expression.
2. The parentheses are not needed to designate the expression un-ambiguously.
3. While evaluating the postfix expression the priority of the operators is no longer

relevant.

We consider five binary operations: +, -, \*, / and $ or ↑ (exponentiation). For these binary operations, the following in the order of precedence (highest to lowest):



|  |  |  |  |
| --- | --- | --- | --- |
| OPERATOR | PRECEDENCE | VALUE |  |
|  |  |  |  |
| Exponentiation ($ or ↑ or ^) | Highest | 3 |  |
|  |  |  |  |
| \*, / | Next highest | 2 |  |
|  |  |  |  |
| +, - | Lowest | 1 |  |
|  |  |  |  |
|  |  |  |  |



**Converting expressions using Stack:**

Let us convert the expressions from one type to another. These can be done as follows:

1. Infix to postfix
2. Infix to prefix
3. Postfix to infix
4. Postfix to prefix
5. Prefix to infix
6. Prefix to postfix

**Conversion from infix to postfix:**

Procedure to convert from infix expression to postfix expression is as follows:

1. Scan the infix expression from left to right.
2. a) If the scanned symbol is left parenthesis, push it onto the stack.
   1. If the scanned symbol is an operand, then place directly in the postfix expression (output).
3. If the symbol scanned is a right parenthesis, then go on popping all the items from the stack and place them in the postfix expression till we get the matching left parenthesis.
4. If the scanned symbol is an operator, then go on removing all the operators from the stack and place them in the postfix expression, if and only if the precedence of the operator which is on the top of the stack is greater than (*or greater than or equal*) to the precedence of the scanned operator and push the scanned operator onto the stack otherwise, push the scanned operator onto the stack.

**Example 1:**

Convert ((A – (B + C)) \* D) ↑ (E + F) infix expression to postfix form:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  | SYMBOL | POSTFIX STRING | STACK | REMARKS |
|  |  |  |  |  |
|  | ( |  | ( |  |
|  |  |  |  |  |
|  | ( |  | ( ( |  |
|  |  |  |  |  |
|  | A | A | ( ( |  |
|  |  |  |  |  |
|  | - | A | ( ( - |  |
|  |  |  |  |  |
|  | ( | A | ( ( - ( |  |
|  |  |  |  |  |
|  | B | A B | ( ( - ( |  |
|  |  |  |  |  |
|  | + | A B | ( ( - ( + |  |
|  |  |  |  |  |
|  | C | A B C | ( ( - ( + |  |
|  |  |  |  |  |
|  | ) | A B C + | ( ( - |  |
|  |  |  |  |  |
|  | ) | A B C + - | ( |  |
|  |  |  |  |  |
|  | \* | A B C + - | ( \* |  |
|  |  |  |  |  |
|  | D | A B C + - D | ( \* |  |
|  |  |  |  |  |
|  | ) | A B C + - D \* |  |  |
|  |  |  |  |  |
|  | ↑ | A B C + - D \* | ↑ |  |
|  |  |  |  |  |
|  | ( | A B C + - D \* | ↑ ( |  |
|  |  |  |  |  |
|  | E | A B C + - D \* E | ↑ ( |  |
|  |  |  |  |  |
|  | + | A B C + - D \* E | ↑ ( + |  |
|  |  |  |  |  |
|  | F | A B C + - D \* E F | ↑ ( + |  |
|  |  |  |  |  |
|  | ) | A B C + - D \* E F + | ↑ |  |
|  |  |  |  |  |
|  | End of |  | The input is | now empty. Pop the output symbols |
|  | string | A B C + - D \* E F + ↑ | from the stack until it is empty. | |
|  |  |  |  |  |

**Conversion from infix to prefix:**

The precedence rules for converting an expression from infix to prefix are identical. The only change from postfix conversion is that traverse the expression from right to left and the operator is placed before the operands rather than after them. The prefix form of a complex expression is not the mirror image of the postfix form.

**Example 1:**

Convert the infix expression A + B - C into prefix expression.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
|  | SYMBOL | PREFIX | STACK |  | REMARKS |
|  | STRING |  |
|  |  |  |  |  |
|  | C | C |  |  |  |
|  |  |  |  |  |  |
|  | - | C | - |  |  |
|  |  |  |  |  |  |
|  | B | B C | - |  |  |
|  |  |  |  |  |  |
|  | + | B C | - + |  |  |
|  |  |  |  |  |  |
|  | A | A B C | - + |  |  |
|  |  |  |  |  | |
|  | End of | - + A B C | The input is now empty. Pop the output symbols from the | | |
|  | string |  | stack until it is empty. | |  |
|  |  |  |  |  |  |

**Conversion from postfix to infix:**

Procedure to convert postfix expression to infix expression is as follows:

1. Scan the postfix expression from left to right.
2. If the scanned symbol is an operand, then push it onto the stack.
3. If the scanned symbol is an operator, pop two symbols from the stack and create it as a string by placing the operator in between the operands and push it onto the stack.
4. Repeat steps 2 and 3 till the end of the expression.

**Conversion from postfix to prefix:**

Procedure to convert postfix expression to prefix expression is as follows:

1. Scan the postfix expression from left to right.
2. If the scanned symbol is an operand, then push it onto the stack.
3. If the scanned symbol is an operator, pop two symbols from the stack and create it as a string by placing the operator in front of the operands and push it onto the stack.

4.Repeat steps 2 and 3 till the end of the expression.

**Conversion from prefix to infix:**

Procedure to convert prefix expression to infix expression is as follows:

1. Scan the prefix expression from right to left (reverse order).
2. If the scanned symbol is an operand, then push it onto the stack.
3. If the scanned symbol is an operator, pop two symbols from the stack and create it as a string by placing the operator in between the operands and push it onto the stack.
4. Repeat steps 2 and 3 till the end of the expression.

**Conversion from prefix to postfix:**

Procedure to convert prefix expression to postfix expression is as follows:

1. Scan the prefix expression from right to left (reverse order).
2. If the scanned symbol is an operand, then push it onto the stack.
3. If the scanned symbol is an operator, pop two symbols from the stack and create it as a string by placing the operator after the operands and push it onto the stack.
4. Repeat steps 2 and 3 till the end of the expression.

**Applications of stacks:**

1. Stack is used by compilers to check for balancing of parentheses, brackets and braces.
2. Stack is used to evaluate a postfix expression.
3. Stack is used to convert an infix expression into postfix/prefix form.
4. In recursion, all intermediate arguments and return values are stored on the processor‟s stack.
5. During a function call the return address and arguments are pushed onto a stack and on return they are popped off.

**Queues:**

A queue is another special kind of list, where items are inserted at one end called the rear and deleted at the other end called the front. Another name for a queue is a “FIFO” or “First-in-first-out” list.

The operations for a queue are analogues to those for a stack, the difference is that the insertions go at the end of the list, rather than the beginning. We shall use the following operations on queues:

* *enqueue*: which inserts an element at the end of the queue.
* *dequeue*: which deletes an element at the start of the queue.

**Representation of Queue:**

Let us consider a queue, which can hold maximum of five elements. Initially the queue is empty.

0 1 2 3 4

Que u e E mpt y



F RO NT = REA R = 0

F R

Now, insert 11 to the queue. Then queue status will be:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |  |
|  |  |  |  |  | REA R = REA R + 1 = 1 |
| 11 |  |  |  |  |
|  |  |  |  | F RO NT = 0 |
|  |  |  |  |  |
|  |  |  |  |  |  |

Now, insert 22 to the queue

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |  |
|  |  |  |  |  | REA R = REA R + 1 = 1 |
| 11 |  |  |  |  |
|  |  |  |  | F RO NT = 0 |
|  |  |  |  |  |
|  |  |  |  |  |  |

Now, insert 33 to the queue

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
|  |  |  |  |  |
| 11 | 22 | 33 |  | REA R = REA R + 1 = 1 |
|  | F RO NT = 0 |
|  |  |  |  |
|  |  |  |  |  |

**Circular Queue:**

A more efficient queue representation is obtained by regarding the array Q[MAX] as circular. Any number of items could be placed on the queue. This implementation of a queue is called a circular queue because it uses its storage array as if it were a circle instead of a linear list.

There are two problems associated with linear queue. They are:

• Time consuming: linear time to be spent in shifting the elements to the

beginning of the queue.

* Signaling queue full: even if the queue is having vacant position. For example, let us consider a linear queue status as follows:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |  |
|  |  |  |  |  | REA R = 5 |
|  |  | 33 | 44 | 55 |
|  |  | F RO NT = 2 |
|  |  |  |  |  |
|  |  |  |  |  |  |



F R

Next insert another element, say 66 to the queue. We cannot insert 66 to the queue as the rear crossed the maximum size of the queue (i.e., 5). There will be queue full signal. The queue status is as follows:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |  |
|  |  |  |  |  | REA R = 5 |
|  |  | 33 | 44 | 55 |
|  |  | F RO NT = 2 |
|  |  |  |  |  |
|  |  |  |  |  |  |



F R

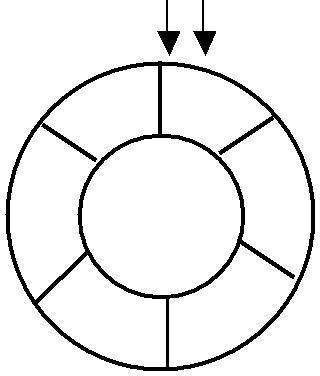
This difficulty can be overcome if we treat queue position with index zero as a position that comes after position with index four then we treat the queue as a **circular queue.**

In circular queue if we reach the end for inserting elements to it, it is possible to insert new elements if the slots at the beginning of the circular queue are empty.

**Representation of Circular Queue:**

Let us consider a circular queue, which can hold maximum (MAX) of six elements. Initially the queue is empty.

F R



|  |  |  |  |
| --- | --- | --- | --- |
| 5 | 0 | |  |
|  | |  |
|  | 1 | | Que u e E mpt y |
| 4 | | M A X = 6 | | |
|  | |
|  | | F RO NT = REA R = 0 | | |
|  | | CO U NT = 0 | | |

1. 2 Circ ular Que ue

Now, insert 11 to the circular queue. Then circular queue status will be:

F

5  0

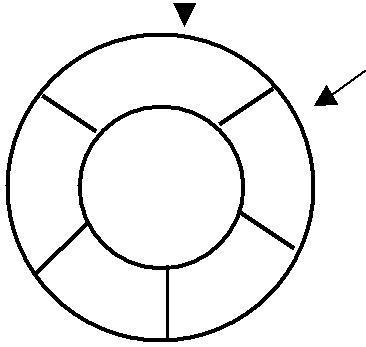


11



R

F RO NT = 0

1

4

1. 2 Circ ular Que ue

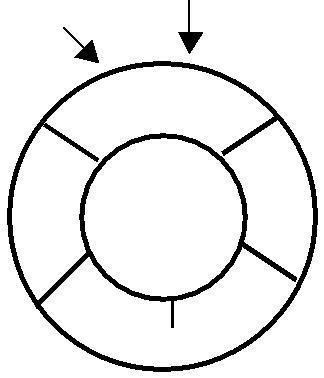
REA R = ( REA R + 1) % 6 = 1

CO U NT = 1

Insert new elements 22, 33, 44 and 55 into the circular queue. The circular queue status is:

F

R



|  |  |  |  |
| --- | --- | --- | --- |
|  | 5 | 0 |  |
|  | 11 |  |
|  |  |  |
|  |  |  |  |
| 4 | 55 | 22 1 | FRONT = 0 |
|  | REAR = (REAR + 1) % 6 = 5 |
|  |  |  |

COUNT = 5

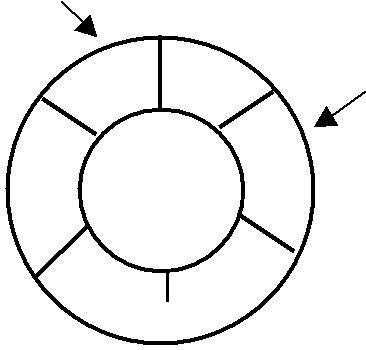
1. 33

32

Circular Queue

Now, delete an element. The element deleted is the element at the front of the circular queue. So, 11 is deleted. The circular queue status is as follows:

R



|  |  |  |  |
| --- | --- | --- | --- |
|  | 5 | 0 |  |
|  |  |  |
|  |  |  | F |
| 4 | 55 | 22 | 1 |
|  |  |

44 33

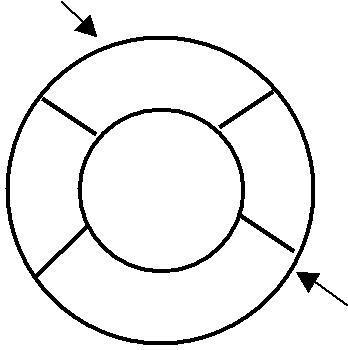
1. 2 Circ ular Que ue

F RO NT = (F R O NT + 1) % 6 = 1 REA R = 5

CO U NT = CO U NT - 1 = 4

Again, delete an element. The element to be deleted is always pointed to by the FRONT pointer. So, 22 is deleted. The circular queue status is as follows:

R

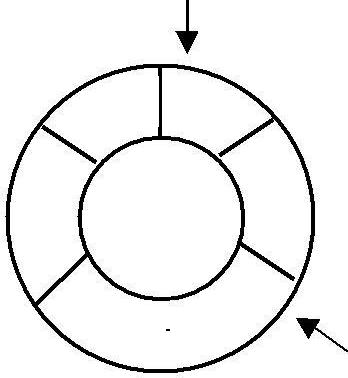


|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 5 |  | 0 | | |  |
|  |  |
|  |  |  |  |  |
|  |  |  |  |  |  |
| 4 55 | 1 | | | | F RO NT = (F R O NT + 1) % 6 = 2 |
|  |  |  |  | REA R = 5 |
|  |  |  |  |  | CO U NT = CO U NT - 1 = 3 |
| 44 |  | | | | F |
|  | |  | 33 |
| 3 |  |  |  | 2 |
|  | |  |  |
|  |  |

Circ ular Que ue

Again, insert another element 66 to the circular queue. The status of the circular queue is:

R



1. 0

66

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 4 | 55 |  |  | 1 |
|  |  | F RO NT = 2 |
|  |  |  |  |
|  |  |  |  | REA R = ( REA R + 1) % 6 = 0 |
|  |  |  |  | CO U NT = CO U NT + 1 = 4 |
|  | 44 |  | |  |
|  |  | 33 |  |
|  | 3 |  | 2 | F |
|  |  |
|  | Circ ular Que ue | | |  |

**Deque:**

In the preceding section we saw that a queue in which we insert items at one end and from which we remove items at the other end. In this section we examine an extension of the queue, which provides a means to insert and remove items at both ends of the queue. This data structure is a *deque*. The word *deque* is an acronym derived from *double-ended* *queue*. Figure 4.5 shows the representation of a deque.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Deletion | |  |  | 36 | | 16 | 56 | 62 | 19 | |  |  | Insertion |
|  |  |  |  |
|  |  |  | |  |  |  |
| Insertion | |  |  |  |  |  |  |  |  |  |  |  | Deletion |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  | front | |  |  |  | rear | | |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |



Figure 4.5. Representation of a deque.

A deque provides four operations. Figure 4.6 shows the basic operations on a deque.

* enqueue\_front: insert an element at front.
* dequeue\_front: delete an element at front.
* enqueue\_rear: insert element at rear.
* dequeue\_rear: delete element at rear. There are two variations of deque. They are:
* Input restricted deque (IRD)
* Output restricted deque (ORD)

An Input restricted deque is a deque, which allows insertions at one end but allows deletions at both ends of the list.

**Priority Queue:**

A priority queue is a collection of elements such that each element has been assigned a priority and such that the order in which elements are deleted and processed comes from the following rules:

1. An element of higher priority is processed before any element of lower priority.
2. two elements with same priority are processed according to the order in which

they were added to the queue.

A prototype of a priority queue is time sharing system: programs of high priority are processed first, and programs with the same priority form a standard queue. An efficient implementation for the Priority Queue is to use heap, which in turn can be used for sorting purpose called heap sort.

An output restricted deque is a deque, which allows deletions at one end but allows insertions at both ends of the list.