Chapter 3  Arrays and Vectors

3.1 Arrays – Representation and basic operations

Array is a very basic and one of the simplest and most widely used data structure provided by every programming language. In all programming languages, arrays share the following characteristics:

- arrays are implemented by using contiguous block of memory,
- array elements are directly accessible, and
- in almost all programming languages, elements of an array must be of the same type\(^1\). That is, array is a collection of homogeneous data.

In C++, given a single-dimensional array \( a \), the \( i \)th element is usually accessed by using the subscript operator, \( a[i] \). In order to access this element, its address needs to be calculated. In general, if we have multi-dimensional array and we want to access some element in it, we would need some way to find out the address of that element. The following subsections discuss how to find the address of an element in an array.

3.1.1 Address of an element in a one-dimensional array

As mentioned above, the whole array is stored in a single contiguous memory block. Address of the first element is called the base address and it is also the starting address of the array. Thus the address of the \( i \)th element in a single dimensional array is given by the following formula:

\[
\text{Address}_i = \text{base address} + i \times \text{size of an element}
\]

As can be seen from the formula, given the index of an element, its address can be calculated in constant time, i.e. \( O(1) \). Resultantly, arrays provide a very efficient way of organizing and accessing data. In C++, given an array \( a \), the \( i \)th element is usually accessed by using the subscript operator, \( a[i] \). Alternatively, we can use the addressing formula and pointer arithmetic for

\(^1\) Arrays in many dynamic programming languages such as JavaScript can store heterogeneous data but that is actually an illusion (see exercise ...).
this purpose. In C++, the name of an array is a pointer pointing to the first element (its base address) in the array. Therefore address of the ith element of the array is given by simply adding i to the base address. Therefore *(a+1) would be equivalent to a[i]. This is elaborated with the help of a very simple example shown in Figure 3-1.

<table>
<thead>
<tr>
<th>float sum (float list[], int n)</th>
<th>float sum (float *list, int n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>{</td>
<td>{</td>
</tr>
<tr>
<td>float temp = 0;</td>
<td>float temp = 0;</td>
</tr>
<tr>
<td>int i;</td>
<td>int i;</td>
</tr>
<tr>
<td>for ( i = 0; i &lt; n; i++)</td>
<td>for ( i = 0; i &lt; n; i++)</td>
</tr>
<tr>
<td>temp = temp + list[i];</td>
<td>temp = temp + *(list+i);</td>
</tr>
<tr>
<td>return temp;</td>
<td>return temp;</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>

Figure 3-1 – Two versions of array processing

The two versions given in the figure are equivalent and in fact the compiler actually would convert list[i] into something similar to *(list+i). It may be noted that, for a single dimensional array, the size of the array is not used in the addressing formula and hence the dimension size is not mentioned in the function prototype.

### 3.1.2 Address of an element in a two-dimensional array

Let us now consider the code written in Figure 3-2. In this case, the second dimension of all the three arrays passed as parameters is specified. In fact, when we pass a multi-dimensional array as a parameter, size of all dimensions except the first one must be specified as constants in the function prototype. This also means that this function cannot be used to add two matrices of arbitrary size – the number of columns is fixed. The question is why is it necessary to do that and what can be done to write a more generic function that can be used to add two matrices of arbitrary size? The answer lies in the addressing formula. Let us see how.
As discussed in section 1.2.1, a two-dimensional array (in fact any multi-dimensional array) is mapped onto a one-dimensional memory space. If the mapping is achieved by storing data row by row, as shown in Figure 3-3, then it is called row major mapping. A column by column ordering is called column major ordering. Most programming languages, including C++, use row major ordering. We now need to have an addressing formula that would translate the row and column number of an element in a two-dimensional array into the offset from the base address. Let us try to develop that formula for a two dimensional array in C++.

Let us assume that we have an array a with N rows and M columns. We now want to find the offset for a[i][j].

![Figure 3-2: Passing multidimensional arrays as parameters](image)

```c
const int COLS = 100;
void add(float a[][COLS], float b[][COLS],
    float c[][COLS], int rows)
{
    float temp = 0;
    int i, j;
    for ( i = 0; i < rows; i++)
        for (j = 0; j < COLS; j++)
            c[i][j] = a[i][j] + b[i][j];
}
```

![Figure- 3-3 – Two different views of a two-dimensional array](image)
If $R_i$ is the offset of the first element of row number $i$ from the base, then offset of $a[i][j]$ will simply be given by $R_i + j$. Therefore, if we can calculate $R_i$, then the address of $a[i][j]$ can be calculated very easily. Let us now try to calculate $R_i$.

We know that the number of elements in each row is equal to the number of columns which is $M$ in this case. Since, in C++, index numbers start from 0, row number $i$ is actually the $(i+1)$th row of the matrix. That means, there are $i$ rows before row number $i$. Since each row has $M$ elements (equal to the number of columns), therefore there are $i \times M$ elements before the first element in row number $i$. Therefore $R_i = i \times M$ and hence offset for $a[i][j]$ is equal to $i \times M + j$.

It may be noted that the total number of rows is not required in order to calculate this offset but the knowledge of number of columns is essential. We can use this information to write a size independent matrix addition function as shown in Figure 3-4.

### 3.1.3 Calculating the address of an element in a multidimensional array

The same technique could be extended to calculate the offset of an element in a multi-dimensional array. We will first derive the formula for the address of an element in a three-dimensional matrix and then generalize it for a matrix with $N$ dimensions.

```c
void add(float *a, float *b, float *c, int rows, int cols) {
    int offset;
    for (int i = 0; i < rows; i++)
        for (int j = 0; j < cols; j++) {
            offset = i*cols + j;
            *(c+offset) = *(a+offset) + *(b+offset);
        }
}
```

**Figure 3-4 - A generic matrix addition function**
From a user’s perspective, as shown in Figure 3-5(a), a three-dimensional matrix takes the shape of a rectangular cuboid. As mentioned earlier, in a row major organization, the individual cells are placed in the memory row-by-row. Therefore, as depicted in Figure 3-5(b), it can be conceived as a collection of rows where each row is a two-dimensional matrix. Finally, each two-dimensional row matrix is flattened out to yield a linear organization. This organization is depicted in Figure 3-5(c). We are now ready to calculate the required offset.

Let us assume that we have a three dimensional matrix $a[D_1][D_2][D_3]$ where $D_1$, $D_2$, and $D_3$ are the sizes of its first, second, and third dimensions respectively and we want to calculate the offset for $a[i][j][k]$. In order to calculate the offset for this element, we first calculate the offset for the row that contains this element. Let us call it $row_i$. Since it is the $(i+1)$th row, we have $i$ rows before this one. We therefore have the offset for $row_i$ as given below:

$$\text{offset for } row_i = \# \text{ of elements in each row} \times i = D_2 \times D_3 \times i$$

Now we find the offset address of the column of the element within $row_i$. Once again, since it the $(j+1)$th column, we have $j$ columns before it in $row_i$. Now, its offset is given as:
offset for column $j$ within row $i = \# \text{ of elements in each column } \times j
= D_3 \times j$

As given below, we add this to the offset of row $i$ from the base to get the offset of this column from the base.

\[
\text{offset for column } j \text{ of row } i \text{ from the base } = D_2 \times D_3 \times i + D_3 \times j
= (D_2 \times i + j) \times D_3
\]

Finally, we add $k$ to it to get the offset of $a[i][j][k]$ from the base:

\[
\text{offset for } a[i][j][k] \text{ from the base } = (D_2 \times i + j) \times D_3 + k
\]

Note that the sizes of all dimensions except the first one are needed to calculate the offset of an element in a three-dimensional matrix.

In general, if we had a multidimensional array arranged in row major, we would need to know the sizes of all the dimensions except the first in order to calculate the offset of any element in the matrix. Let $A[D_0][D_1]...[D_{n-1}]$ be an n dimensional array, then the offset of $A[d_0][d_1]...[d_{n-1}]$ is given by the following formula:

\[
\text{offset of } A[d_0][d_1]...[d_{n-1}]
= (...)(((d_0 \times D_1 + d_1) \times D_2 + d_2) \times D_3 + \cdots) + d_{n-1}
\]

It may again be noted that the size of the first dimension does not figure in the formula. That is why it is necessary to specify sizes of all the dimensions except the first one when a multi-dimensional array is passed in a function as a parameter.

### 3.1.3.1 Square Matrices and some special cases

A square matrix has the same number of rows and columns. Some special forms of square matrices include the following:

a) Diagonal Matrix: A matrix is called a diagonal matrix if the data is present on the diagonal only and all other elements are zero. That is, given a matrix $M$, $M_{i,j} = 0$ for $i \neq j$
b) Band matrix: In a band matrix, non-zero entries are present on a diagonal band only which comprises of the main diagonal and zero or more diagonals on either side. That is, $M_{i,j} = 0$ for $|i-j| < k$, where $k$ is the number of diagonals on either side of the main diagonal. The diagonal matrix is a special case of band matrix with $k = 0$.

c) Symmetric matrix: A matrix that is equal to its transpose is called a symmetric matrix. That is, $M_{i,j} = M_{j,i}$ for all $i$ and $j$.

d) Triangular matrix: A matrix where entries either below or above the main diagonal are zero is called a triangular matrix. If the entries above the main diagonal are zero then it is called the lower triangular matrix and if the entries below the main diagonal are zero then it is called the upper triangular matrix. That is, for lower triangular matrix we have $M_{i,j} = 0$ for $i < j$, and for upper triangular matrix we have $M_{i,j} = 0$ for $i > j$.

Figure 3-6 shows the general structure of the band and lower triangular matrices.

![Diagram showing general structure of band and lower triangular matrices](image)

**Figure 3-6 - General structure of band and lower triangular matrices**

Such matrices have structures that can be exploited to save space. In this section we will develop a storage scheme for symmetric matrices.
As can be easily seen, in symmetric matrices we only need to store approximately one half of the matrix as the same values are found in the other half. That is, if we had stored the data on the diagonal and either the lower or the upper triangle, we would have captured the complete information present in the matrix. Let us assume that we store the diagonal and the lower triangle of the matrix. In order to do so, we first need to find out the total number of elements to be stored. We can determine that by observing that the number of elements to be stored for the \textit{ith} row would be equal to \( i \). Since we have \( n \) rows for an \( n \times n \) matrix, the total number of elements to be stored is given as below:

\[
1 + 2 + 3 + \cdots + n = \frac{n(n + 1)}{2}
\]

If we store these elements row by row in a single dimensional array \( A \), then:

\[
M_{ij} = \begin{cases} 
M_{ji}, & \text{if } i < j \\
A + \frac{i(i + 1)}{2} + j, & \text{otherwise}
\end{cases}
\]

where \( A \) is the base address of the array and \( i \) and \( j \) go from 0 to \( n-1 \).

3.1.4 Arrays of arrays – an alternative way of representing multi-dimensional arrays

As discussed earlier, most programming languages reserve a contiguous block of memory large enough to hold all elements a multi-dimensional array. Some programming languages, including Java, take a different approach. In these languages, such an array is built as an array of arrays. That is, each dimension of the array is represented by a set of single-dimensional arrays, where each element of these arrays would be a pointer to another array representing the next dimension. The actual data is stored in the set of arrays representing the last dimension. For example, The matrix of Figure 3.3 would be stored as something similar to the one depicted in Figure 3-7

Assuming a two-dimensional array \( A \) constructed as an array of arrays, then \( A[i][j] \equiv * (* (A + i) + j) \).
Chapter 3 - Arrays and Vectors

Note that the addressing formula does not need to know the size of any dimension as each dimension is composed of a set of single dimensional arrays.

The concept of array of arrays also enables the user to have rows of different sizes. That is, as depicted in Figure 3-8, a two-dimensional array does not have to be rectangular any more. Such an array is called a “ragged” array in Java.

3.1.5 Using array to store heterogeneous data

As mentioned earlier, arrays are usually used to store homogeneous data. That is, all elements of the array have to be of the same type. Homogeneity of data is essential for calculating the address of the ith element in the array in O(1) as pointer arithmetic cannot be used with heterogeneous data.

However, some programming languages, such as JavaScript, allow an array to have elements of different types. That is, one can store an integer, a floating point number, a string, or for that matter anything, in the same array and still provide random access to the ith element in O(1). How does that happen?

As already mentioned, in order to have random access in O(1), each cell of the array has to have the same type (and hence the same size). Therefore, in order to store heterogeneous data in the array, the actual data is stored in dynamically allocated memory...
and the array is used to store their addresses only, making each cell of the array of uniform size. Figure 3-9 elaborates this concept.

### 3.2 Arrays and linear lists

According to Oxford Dictionary, a list is defined as: a number of connected items or names written or printed consecutively, typically one below the other. One of the most frequently used basic requirements is to maintain a collection as a list of items. As an abstract data type, such a list will typically support the following operations:

1. Create a new empty list.
2. Delete a list.
3. Find the number of elements in the list or length of the list.
4. Check if the list is empty.
5. Check if more elements can be added to the list. That is, check whether the list is full or not.
6. Traverse the list.
7. Get the value of \(i^{\text{th}}\) elements in the list.
8. Change the value of the \(i^{\text{th}}\) element.
9. Insert a new element in the list.
10. Delete an element.
11. Make a copy of the list
12. Determine whether an element is present in the list or not
The easiest way to represent such a list is by a one-dimensional array where we store the \( i^{th} \) element of the collection in the array element with index \( i-1 \). The ADT for a list to store a collection of items is developed in Figure 3-10.

```cpp
template<class T>
class List {
public:
    List(int size);   // constructor
    ~List () {delete [] ListArray; }   // destructor
    void insert(const T value);
    bool remove(const T value);
    void setData(int index, T value);
    T getData(int index, T value);
    int search(T key);   // linear search
    // if found, returns index of the key
    // else returns -1
    bool isFull() {return mSize == MaxSize;}
    bool isEmpty() {return mSize == 0;}
    int length() {return mSize;}
    List(const List & other);   // copy constructor
    const List & operator=(const List & rhs);   // assignment operator

private:
    bool removeAt(int index);   // remove the element at a
    // given index
    int MaxSize;   // max List size
    T *listArray;   // array to store the data
    int mSize;   // no. of elements in the List
};
```

Figure 3-10 - ADT for a linear list - specification
The List ADT has three private data members: MaxSize, listArray, and mSize. The first two (MaxSize and listArray) are similar to the ones used in the specification of SmartArry (section 1.6.2). mSize is used to maintain information about the number of items stored in the list. The data will be stored in the array from the index number 0 to mSize – 1. It will be initialized to 0 in the constructor and will be incremented or decremented every time a new item is added to list or removed from it. Accordingly, the constructor is given as follows:

\[
\text{template<class T>}
\]
\[
\text{List<T>::List(int size) \{}
\]
\[
\quad \text{if (size < 1) throw ILLEGAL_SIZE;}
\]
\[
\quad \text{else \{}\]
\[
\quad\quad \text{MaxSize = size;}
\]
\[
\quad\quad \text{listArray = new T[MaxSize];}
\]
\[
\quad\quad \text{if (listArray == NULL) throw OUT_OF_MEMORY;}
\]
\[
\quad\quad \text{mSize = 0;}
\]
\[
\quad \}\}
\]

The insert operation is trivial – we simply check if space is available and then add the new element at the end of the list and increment mSize.

\[
\text{template<class T>}
\]
\[
\text{void List<T>::insert(const T value) \{}
\]
\[
\quad \text{if (isFull()) throw OUT_OF_SPACE;}
\]
\[
\quad \text{listArray[mSize] = value;}
\]
\[
\quad \text{mSize++;}
\]
\[
\}}
\]

The remove method is a little bit more interesting. In this case we are required to remove an element present in the list. We first have to find the index of the element if it is present and then remove the element by simply pulling all the subsequent elements one place up.

\[
\text{template<class T>}
\]
\[
\text{bool List<T>::remove (const T value) \{}
\]
\[
\quad \text{int index = search(value);}
\]
\[
\quad \text{if (index == -1) // not found}
\]
\[
\quad\quad \text{return false;}
\]
\[
\quad \text{else return removeAt(index);}
\]
\[
\}}
\]
template<class T>
bool List<T>::removeAt (int index) {
    if (index < 0 || index >= mSize)
        throw ILLEGAL_INDEX;
    for (int i = index; i < mSize - 1; i++)
        listArray[i] = listArray[i+1];
    mSize--;
    return true;
}

As can be very easily seen, this version of the removeAt function has a
time complexity of O(N). We however observe that, since the order of
elements in the list is not important, we can remove the element at the
given index by simply replacing it with the last element. This results in an
algorithm with constant time complexity and is presented next.

template<class T>
bool List<T>::removeAt (int index) {
    if (index < 0 || index >= mSize)
        throw ILLEGAL_INDEX;
    listArray[index] = listArray[mSize-1];
    mSize--;
    return true;
}

The copy constructor and overloaded assignment operators are similar to
the ones discussed in chapter 1. The search function basically uses linear
search algorithm and returns the index of the key if it is found otherwise
it returns -1. getData and setData functions are trivial.
3.2.1 List Iterator

search, setData, and getData are different than the rest of the class methods as they use the index of the corresponding element as the input to the method (setData and getData) or an output of the method (search). The problem with this approach is that we are making use of the knowledge that the list is implemented using a one-dimensional array. That is, the specification of the interface for the ADT is influenced by the data structure used to implement the ADT. This will create a problem as a change in the choice of the data structure used to implement the list could require changing the interface as well. This would then result in the breakdown of the client programs.

The solution to this problem lies in finding out a way to specify the desired location without breaking the abstraction. That is, we need to find an implementation independent mechanism to specify a particular location in the underlying data structure. In general, we need a mechanism that would allow us to iterate over the elements of a container class and give us access to the data stored in there so that we can process data according to our own specific needs without exposing the internal implementation details of the underlying collection class – iterators are precisely meant for this purpose.\(^2\)

An iterator in its simplest form would provide at least the following five basic operations for that purpose: (i) attach an instance of the iterator to an instance of the collection/container class for which the iterator is written, (ii) position it to the first element in the container class, (iii) move to the next element in the collection, (iv) give access to the data at the current position, and (v) test if the collection class has more data that needs to be processed. Such an iterator is called a forward iterator. A bi-directional operator would also provide the facility to go back to the previous element in the container class.

\(^2\) Iterators are important components of the C++ Standard Template Library but take a different shape and form than what is presented here.
In order for an iterator to work, it needs access to the private data members of the corresponding collection class. One way to achieve this in C++ is by making the iterator class a friend of the collection class. All public methods of the collection class that need to take or give information about the location of a specific element would now be implemented through the iterator class. It may be noted that the compiler will not allow default assignment operator in this case and hence it must be declared as private without any body. The modified version of the list class as well as its forward iterator is shown in Figure.

---

3 list is a reference type data member. A reference is bound in the constructor, and cannot be reassigned. Therefore we cannot create the default assignment operator as it would require member-wise assignment which is not possible in this case.

4 In C++ STL, find has a different interface and is part of standard algorithms.
We can now use this iterator to implement the desired functionality as shown in Figure...

```
List<int> intList(100);
// add data to the list
ListIterator<int> myIterator(intList); // create iterator and
// attach list to it
int temp = 0;
myIterator.begin();
while(!myIterator.isDone()) {
    temp = temp + myIterator.getData();
    myIterator.next();
}
// temp now has sum of all elements
```
3.2.2 Ordered List

One of the most common data object is a linear ordered list. An ordered list has elements with definite ordering. The simplest example of an ordered list is an ordered pair which has only two elements. Following are some examples of ordered lists:

1. Months of the years (Jan, Feb, Mar, Apr, May, ..., Dec)
2. English Alphabets (A, B, C, ..., Z)
3. Words in a sentence ("This is a book on data structures.")
4. Names of the students in a class stored in the order of their roll numbers.

The elements in an ordered-list must maintain a specified order. For example, changing the orders of words in a sentence will change the sentence. That is, it will no longer be the same sentence. A very common application of ordered list is a collection in which the items are maintained in ascending or descending order. In this section we will be looking at an ordered list to maintain a list of items in ascending order.

As an abstract data type, an ordered-list will typically support the same set of operations defined for the simple or unordered list discussed in the previous section. Once again, the easiest way to represent an ordered list is by a one-dimensional array where we store the \( i \)th element of the list in the array element with index \( i \). So, our OrderedList will have a specification identical to the one shown in Figure

The restriction on the order requires a different implementation. We will have to rewrite the insert, remove, and setData methods (we can assume ascending or descending order). Also, instead of linear search, we can now use binary search. The rest remains the same. Interestingly, there will be no change in the iterator class (except for the list type associated with it).

The corresponding insert function is given next (first version of remove can be used here, find and setData are left as an exercise).
template<class T>
void OrderedList<T>::insert(T value) {
    if (isFull()) throw OUT_OF_SPACE;
    int i = mSize;
    while (i > 0 && value < ListArray[i-1]) {
        ListArray[i] = ListArray[i -1];
        i--;
    }
    ListArray[i] = value;
    mSize++;
}

3.3 Stacks and queues

Stacks and queues are two very important and very commonly used data structures that belong to the class of ordered lists. The following subsections are dedicated to the discussion of these two data structures.

3.3.1 Stacks

Stack is a data structure that can be used to store data which can later be retrieved in the reverse or last in first out (LIFO) order. Stack is an ordered-list in which all the insertions and deletions are made at the same end to maintain the LIFO order. In order to understand stacks, we can consider the example of batteries in a flashlight. Batteries inserted in the flashlight are removed from it in the reverse or LIFO order.

In computer science, stack is one of the most important and most useful data structure. In fact, without stacks it would not even be possible to make nested subprogram calls and hence it would not be possible to write any meaningful programs.

The operations defined on a stack ADT are:

- Stack: create an empty stack
- Push: Store an element onto the stack
- Pop: retrieve the last element stored on the stack
- Top: examine the top element in the stack
- isEmpty: check if the stack is empty
- isFull: check if the stack is full
3.3.1.1 Stacks – Array Implementation

A stack can be very easily implemented using arrays by maintaining a pointer called the stack pointer. If a stack is implemented using arrays, the following two conventions can be used:

1. A stack can grow from index 0 to the maximum index (growing upwards), or it can grow from the maximum index to index 0 (growing downwards).
2. Stack pointer can point to the last element inserted into the stack or it can point to the next available position where is next element is to be inserted.

That gives rise to the four possible implementations that are enumerated and depicted in Fig.
Figure ... shows array based stack ADT and its implementation for configuration (a). Note that all operations on a stack are performed in O(1).
```cpp
template <class T>
class Stack {
public:
    Stack(int size);        // constructor
    ~Stack();              // destructor
    void push (const T &); // add an element to the stack
    T pop();               // remove an element from stack
    bool isFull();          // check if the stack is full
    bool isEmpty();         // check if the stack is empty
private:
    const int MaxSize;     // maximum storage capacity
    int stkptr;            // stack pointer
    T *stackArray;         // array used to implement the stack
    Stack(const Stack &other); // disallow copy
    Stack& operator ==(const Stack & other); // disallow assignment
};

template <class T>
Stack<T>::Stack(int s): MaxSize(s > 0 ? s :
    throw ILLEGAL_ARRAY_SIZE_EXCEPTION),
    stackArray(new T[size])
{
    stkptr = 0;
}

template <class T>
Stack<T>::~Stack()
{
    delete []stackArray;
}

template <class T>
bool Stack<T>::isEmpty()
{
    return stkptr == 0;
}

template <class T>
bool Stack<T>::isFull()
{
    return stkptr == MaxSize;
}

template <class T>
void Stack<T>::push(const T &elem)
{
    if (!isFull()) {
        stackArray[stkptr] = elem;
        stkptr++;
    } else
        throw STACK_FULL;
}

template <class T>
T Stack<T>::pop()
{
    if (!isEmpty()) {
        stkptr--;
        return stackArray[stkptr];
    } else
        throw STACK_EMPTY;
}
```
3.3.1.2 Applications of stacks – use of stack in evaluation of expressions

Evaluation of fully parenthesized expressions

Normally, arithmetic expressions are written with operators coming in-between their operands. This notation is called the infix notation. \( a + b \ast (e - g) + h - f \ast i \) is an example of such an expression.

Evaluation of expression written in the infix notation is a challenging task because in this case the expression is not strictly evaluated from left to right as the order of operator precedence needs to be preserved. For example, \( a + b \ast c/d \) would require first multiplying \( b \) with \( c \), the result is then divided by \( d \) and then \( a \) is added to the result to get the final answer. The problem can be simplified by fully parenthesizing the expression. Doing so will convert \( a + b \ast c/d \) into \( (a + ((b \ast c)/d)) \). A fully parenthesized expression can be evaluated easily with the help of a stack. Algorithm given in Fig., can be used for this purpose.

1. while (not end of expression) do
   1. get next input symbol from the expression
   2. if input symbol is not “)”
      1. push it onto the stack
   3. else
      1. repeat
         1. pop the symbol from the stack
         2. until you get “(“ 
      3. apply operators on the operands
      4. push the result back into stack
2. end while

The value at top of stack is the answer

Fig., shows evaluation of \((a+(b/c))\) using this algorithm with \( a = 2 \), \( b = 6 \), and \( c = 3 \).
Evaluation of Fully Parenthesized Expression

(a+(b/c))
Assuming a=2, b=6, c=3

<table>
<thead>
<tr>
<th>Input Symbol</th>
<th>Stack</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>(</td>
<td>Push</td>
</tr>
<tr>
<td>a</td>
<td>(a)</td>
<td>push</td>
</tr>
<tr>
<td>+</td>
<td>(a+)</td>
<td>push</td>
</tr>
<tr>
<td>(</td>
<td>(a+()</td>
<td>push</td>
</tr>
<tr>
<td>b</td>
<td>(a+b)</td>
<td>push</td>
</tr>
<tr>
<td>/</td>
<td>(a+b/)</td>
<td>push</td>
</tr>
<tr>
<td>c</td>
<td>(a+b/c)</td>
<td>Push</td>
</tr>
<tr>
<td>)</td>
<td>(a+2)</td>
<td>Pop&quot;(b/c&quot; and evaluate and push the result back</td>
</tr>
<tr>
<td>)</td>
<td>4</td>
<td>Pop&quot;(a+2&quot; and evaluate and push the result back</td>
</tr>
</tbody>
</table>

Polish and Reverse Polish Notations and evaluation of expressions in RPN

Reverse Polish Notation (RPN) or postfix notation provides an alternative solution to the same problem. In RPN or postfix notation, instead of being inserted between the associated operands, the operators follow (to the right of) the operands, hence the name postfix. The notation was developed by the Australian computer scientist Charles Hamblin in the mid-1950s. He was inspired by the work of the Polish mathematician Jan Łukasiewicz, who had invented the Polish notation in 1920s. The Polish notation is also called prefix notation as it places operators before (to the left of) their operands. What make these notations special is that both

Prefix notation is very commonly used in mathematics as well as in programming as the syntax of functions uses the prefix notation – the operation (name of the function) comes before the operands (the parameters). Arithmetic expressions are nonetheless usually written in infix notation. However, some programming languages, including LISP, the entire syntax is based upon prefix notation where even the arithmetic expressions are also written in prefix.
these notations are parentheses-free. That is, expressions written in prefix (Polish notation) or postfix (Reverse Polish Notation or RPN) can be evaluated from left to right without having the need to use any parentheses.

The following steps can be used to convert an infix expression into its equivalent prefix or postfix expression.

1. Fully parenthesize the expression.
2. Take the operator from inside the corresponding pair of parentheses to outside – before the corresponding left parenthesis or after the corresponding right parenthesis for prefix or postfix expression respectively.
3. Remove all parentheses to get the required expression.

The following example illustrates the process.

<table>
<thead>
<tr>
<th>The expression: ( a + b \ast c/d )</th>
<th>Fully parenthesized version: ((a + ((b \ast c)/d)))</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>For prefix</strong></td>
<td><strong>For postfix</strong></td>
</tr>
<tr>
<td>Bring the operator from inside the corresponding pair of parentheses to before the corresponding left parenthesis:</td>
<td>Bring the operator from inside the corresponding pair of parentheses to after the corresponding right parenthesis:</td>
</tr>
<tr>
<td>(+ (a /((b \ast c) \ast d)))</td>
<td>((a ((b \ast c) \ast d))/ +)</td>
</tr>
<tr>
<td>Remove all parentheses:</td>
<td>Remove all parentheses:</td>
</tr>
<tr>
<td>(+ a / \ast b c d)</td>
<td>(a b c \ast d / +)</td>
</tr>
</tbody>
</table>

It may be noted that associativity of + and * operators make it possible to parenthesize the same expression differently. We may therefore get different prefix and postfix equivalents for the same infix expression. For example \( a \ast b \ast c \) could be parenthesized as \((a * (b \ast c))\) or \(((a \ast b) \ast c)\) and hence could be written as \(a b c * *\) or \(a b \ast c *\) in RPN.

Figure .. shows some examples of expressions in infix and their corresponding pre and post-fix expressions.
### 3.3.2 Evaluation of expressions in RPN

Because Postfix operators use values to their left, any values involving computations will already have been calculated as we go from left-to-right. We can use this property to evaluate expressions written in RPN. Like fully parenthesized expressions, expression written in RPN can be evaluated easily with the help of stacks. Algorithm in Fig..

```plaintext
while (not end of expression) do
    1. get next input symbol
    2. if input symbol is an operand then
       a) push it into the stack
    3. else if it is an operator then
       a) pop the operand(s) from the stack
       b) apply operator to the operand(s)
       c) push the result back onto the stack
    End while

the top of stack is answer.
```

This algorithm is demonstrated in an example of in Fig…
Chapter 3 - Arrays and Vectors

Algorithm to Evaluate Expressions in RPN

\[(a+b)^{(c+d)} \rightarrow ab+cd+^\ast\]

Assuming a=2, b=6, c=3, d=-1

<table>
<thead>
<tr>
<th>Input Symbol</th>
<th>Stack</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a</td>
<td>Push</td>
</tr>
<tr>
<td>b</td>
<td>a b</td>
<td>Push</td>
</tr>
<tr>
<td>+</td>
<td>8</td>
<td>Pop a and b from the stack, add, and push the result back</td>
</tr>
<tr>
<td>c</td>
<td>8 c</td>
<td>Push</td>
</tr>
<tr>
<td>d</td>
<td>8 c d</td>
<td>Push</td>
</tr>
<tr>
<td>+</td>
<td>8 2</td>
<td>Pop c and d from the stack, add, and push the result back</td>
</tr>
<tr>
<td>*</td>
<td>16</td>
<td>Pop 8 and 2 from the stack, multiply, and push the result back. Since this is end of the expression, hence it is the final result.</td>
</tr>
</tbody>
</table>

Now the big question is how to convert an expression written in infix notation into postfix. Once again we can use the stack for this purpose.

What we need to do is find a way to apply a higher precedence operator before a lower precedence operator. That is, given two consecutive operators, we need to reverse the order in which these operators are to be applied if the second operator has higher precedence than the first one, otherwise the operators would be applied in the order of their appearance in the infix expression. For example, the expression \(a + b \cdot c\) would be transformed in to RPN as \(a b c + \cdot\). On the other hand \(a \cdot b + c\) would be expressed as \(a b c + \cdot\). That is, in the first case the order of operators is reversed whereas, in the second case, it is retained.

This can be achieved very easily with the help of a stack. In order to understand the basic idea, let us consider an infix expression with two operators with different precedence. When we scan the expression and find the first operator, it is pushed onto the stack and when we get the next operator we compare it with the operator on top of the stack. If the
first one has a higher precedence then it is ought to be applied before the
second one therefore it is popped from the stack and used and the second
operator is pushed onto the stack. If however the second one has higher
precedence than the first one then it is pushed on to the stack so that
when the two operators are popped eventually, the one with the higher
precedence is popped first and hence it is applied before the operator
which has a lower precedence.

The general algorithm for converting an expression in infix to postfix is
outlined in Figure …

Algorithm for Infix to RPN Conversion
1. Initialize an empty stack of operators
2. While not end of expression do
   a. Get the next input token
   b. If token is
      i. “(“ push
      ii. “)” pop and display stack element until a
           left parenthesis is encountered, but do
           not display it.
      iii. An operator:
           if
               stack is empty or token has higher
               precedence than the element at the top of stack, push
               Note: “(” has the lowest precedence
           else
               pop and display the top stack element
               and repeat step # iii.
      iv. An Operand: Display
3. Until the stack is empty, pop and display

This algorithm is elaborated with the help of an example given in Figure
a+b*c/(d+e) → a b c * d e + / +

<table>
<thead>
<tr>
<th>Input Symbol</th>
<th>Stack</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>+</td>
<td>Operand – display – RPN → a</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>Push as stack is empty</td>
</tr>
<tr>
<td>b</td>
<td>+</td>
<td>Operand – display – RPN → a b</td>
</tr>
<tr>
<td>*</td>
<td>+ *</td>
<td>Push as * has higher precedence than +</td>
</tr>
<tr>
<td>c</td>
<td>+ *</td>
<td>Operand – display – RPN → a b c</td>
</tr>
<tr>
<td>/</td>
<td>+ /</td>
<td>Pop * and push / as * and / have the same precedence but / has higher precedence than + – RPN → a b c *</td>
</tr>
<tr>
<td>(</td>
<td>+ / (</td>
<td>Push</td>
</tr>
<tr>
<td>d</td>
<td>+ / (</td>
<td>Operand – display – RPN → a b c * d</td>
</tr>
<tr>
<td>+</td>
<td>+ / ( +</td>
<td>Push as + has higher precedence than (</td>
</tr>
<tr>
<td>e</td>
<td>+ / ( +</td>
<td>Operand – display – RPN → a b c * d e</td>
</tr>
<tr>
<td>)</td>
<td>+ /</td>
<td>Pop till “(” is found – RPN → a b c * d e +</td>
</tr>
</tbody>
</table>

**3.3.3 Applications of stacks – Use of Stack in the Implementation of Subprograms**

In a typical program, the subprogram calls are nested. Address of the next instruction in the calling program must be saved in order to resume the execution from the point of subprogram call. Since the subprogram calls are nested to an arbitrary depth, use of stack is a natural choice to preserve the return address. This information is saved in a data structure called the activation record.

**3.3.3.1 Activation Records**

An activation record is a data structure which keeps important information about a sub program. In modern languages, whenever a subprogram is called, a new activation record corresponding to the subprogram call is created, and pushed into the
stack. As depicted in Figure ..., among other things, the information stored in an activation record includes the return address of the next instruction in the calling program to be executed, and current value of all the local variables and parameters. i.e. the context of a subprogram is stored in the activation record. When the subprogram finishes its execution and returns back to the calling function, its activation record is popped from the stack and destroyed—restoring the context of the calling function.

Use of activation records and stacks is best explained with the help of recursive functions which are discussed next.

### 3.3.3.2 Recursive Functions

Recursion is a very powerful algorithm design tool. A subprogram is called recursive if it directly or indirectly calls itself. A recursive program has two parts: (1) The end condition(s), and (2) the recursive step(s).

An end condition specifies where to stop and with each recursive step we should come closer to the end condition. In other words, with a recursive step, we apply the same algorithm on a scaled down problem and this process is repeated until the end of condition is reached. Following are some examples of recursive programs.

**Recursion – Some Examples**

The first example is a recursive version of our linear search program. In this case we have two end conditions and one recursive step. As mentioned above, the end conditions indicate when the program should terminate and the recursive steps takes us closer to the end condition. In this case the end conditions are trivial—we stop when either key is found, or from > to. Otherwise we take the recursive step by reducing the search space by one and the repeating the process again.

---

6 As mentioned, this is a simplified depiction of an activation record. In addition, the order in which parameters, local variable, and return address are arranged is also different in actual activation records.
int linear_search (int a[], int from, int to, int key) {
    if (from <= to) {
        if (key == a[from]) //end condition
            return from;
        else
            return linear_search(a, from+1, to, key); //recursive step
    }
    return -1; //end condition
}

The second example is of binary search. In this case we have two end conditions and two recursive steps. The two end conditions are: (a) when key is found, and (b) when low > high. Otherwise we take the recursive step. Accordingly, the two recursive steps are: (a) process is repeated by passing the high index to be equal to mid-1 when k < a[mid], and (b) calling the function again by setting the low index to be equal to mid+1 otherwise.

int binary_search (int a[], int high, int low, int key) {
    int mid = (high + low)/2;
    if (high >= low) {
        if (key == a[mid]) //end condition
            return mid;
        else if (key < a[mid])
            return binary_search(a, mid-1, low, key); //recursive step
        else
            return binary_search(a, high, mid + 1, key); //recursive step
    }
    return -1; // end condition
}

It is important to note that the test for the end condition must be made before the recursive step. Failing to do so would result in infinite recursion. Let us now try to understand the role of stacks in the implementation of recursion.
How Does Recursion Work - Some More Recursive Functions and Their Simulation

As mentioned earlier, whenever a subprogram call is made, a corresponding activation record is created and is pushed on to the stack. The information stored in the activation record includes parameters, local variables, and return address. When the subprogram terminates, its activation record is popped from the stack and the return address is used to go back to the calling subprogram and resume its execution from the point from where the previous subprogram call was made. Values of parameters and local variables of the calling subprogram are retrieved from its activation record which is now on top of the stack. The process continues until the entire program is terminated. This concept is elaborated with the help of a few recursive programs.

We first look at the example of factorial function shown below. Line numbers are added to keep track of the return address.

```c
int factorial (int n) {
    int temp;
    /* 1. */ if (n<0) throw ILLEGAL_INPUT; //end condition
    /* 2. */ else if (n <=1) return 1; //end condition
    /* 3. */ else {
    /* 4. */ temp = factorial(n-1); //recursive step
    /* 5. */ return n*temp;
    }
}
```

We now call this function with n = 4. The function will recursively calculate factorial of 4. Let us now trace the execution of this recursive function:

1. Activation record for factorial(4) is created
2. Recursive call from statement #4 is made and activation record for factorial(3) is created and put on top of the stack.
3. Recursive call from statement #4 is made and activation record for factorial(2) is created and put on top of the stack.
4. Recursive call from statement #4 is made and activation record for factorial(1) is created and put on top of the stack.
5. The function returns 1. Activation record for factorial(1) is removed from the stack. Control goes back to line#4 in the previous activation where temp is assigned the return value which is 1.

6. Execution of factorial(2) is resumed and statement #5 is executed – temp is multiplied by n (which is 2) giving a result of 2 – the function returns 2. Activation record for factorial(2) is removed from the stack. Control goes back to line#4 in the previous activation where temp is assigned the return value which is 2.

7. Execution of factorial(3) is resumed and statement #5 is executed – temp is multiplied by n (which is 3) giving a result of 6 – the function returns 6. Activation record for factorial(3) is removed from the stack. Control goes back to line#4 in the previous activation where temp is assigned the return value which is 6.

8. Execution of factorial(4) is resumed and statement #5 is executed – temp is multiplied by n (which is 4) giving a result of 24 – the function returns 24. Activation record for factorial(4) is removed from the stack. Since there are no more activation records on the stack for factorial, therefore 24 is the answer of factorial(4).

This sequence of steps and the corresponding situation of the stack is depicted in Figure …
Activation record for factorial(4)

1

n = 4
temp = undefined
caller’s address

Activation record for factorial(4)

2

n = 4
temp = undefined
caller’s address

Activation record for factorial(4)

3

n = 3
temp = undefined
line#4

Activation record for factorial(3)

4

n = 1
temp = undefined
line#4

Activation record for factorial(1)

5

n = 2
temp = undefined
line#4

Activation record for factorial(2)

6

n = 3
temp = undefined
line#4

Activation record for factorial(3)

7

n = 4
temp = 6
caller’s address

Activation record for factorial(4)
Let us now consider a more interesting example. In this example, we recursively compute the nth number in the Fibonacci sequence. The Fibonacci sequence, named after the great Italian Mathematician of the 13\textsuperscript{th} century who defined it for the first time, is defined as follows:

$$Fibonacci(n) = \begin{cases} 1 & \text{if } n = 1 \text{ or } n = 2 \\ Fibonacci(n - 1) + Fibonacci(n - 2), & \text{otherwise} \end{cases}$$

That is, for all values of $n > 2$, any number in the sequence can be found by adding the last two numbers in the sequence. Hence the first 7 numbers in the sequence are: 1, 1, 2, 3, 5, 8, and 13.

Although it is not difficult to write a non-recursive solution, we can get a recursive solution by simply rewriting the definition of Fibonacci sequence in C++ as shown in Figure ... Once again line numbers are added just to keep track of the execution steps.

```cpp
int fibonacci (int n)
{
    int temp1, temp2;
    if (n<=0) throw ILLEGAL_INPUT;
    /* 1. */
    else if (n<=2)
        /* 2. */
        return 1; //end condition
    /* 3. */
    else{
        /* 4. */
        temp1 = fibonacci(n - 1); //recursive step 1
        /* 5. */
        temp2 = fibonacci(n - 2); //recursive step 2
        /* 6. */
        return temp1 + temp2;
    }
}
```

The code is simple and straightforward and the only thing to note is that it involves two recursive calls. The code in the else part is broken into three statements just to demonstrate how recursion works. Let us now trace the execution of fibonacci(4).

1. Activation record for Fibonacci(4) is created. For this activation $n = 4$, temp1 and temp2 are initially unassigned. Recursive call from line# 4 is made for Fibonacci(3).
2. Activation record for Fibonacci(3) is created. For this activation n = 3, temp1 and temp2 are initially unassigned. Recursive call from line# 4 is made for Fibonacci(2).

3. Activation record for Fibonacci(2) is created. For this n = 2, temp1 and temp2 are unassigned.

4. The function returns 1. Activation record for fibonacci(2) is removed from the stack.. Control goes back to line#4 in the previous activation where temp1 is assigned the return value which is 1.

5. Recursive call from line#5 is made for Fibonacci(1). Activation record for Fibonacci(1) is created. For this n = 1, temp1 and temp2 are unassigned.

6. The function returns 1. Activation record for fibonacci(1) is removed from the stack.. Control goes back to line#5 in the previous activation where temp2 is assigned the return value which is 1.

7. The function returns 2. Activation record for fibonacci(3) is removed from the stack.. Control goes back to line#4 in the previous activation where temp1 is assigned the return value which is 2.

8. Recursive call from line#5 is made for Fibonacci(2). Activation record for Fibonacci(2) is created. For this n = 2, temp1 and temp2 are unassigned.

9. The function returns 1. Activation record for fibonacci(2) is removed from the stack.. Control goes back to line#5 in the previous activation where temp2 is assigned the return value which is 1.

10. The function returns 3. Activation record for fibonacci(4) is removed from the stack.. Control goes back to the calling program. Since there are no more activation records on the stack for fibonacci, therefore 3 is the answer of fibonacci(4).

Fig. .. shows the steps involved in calculating Fibonacci(4).
Chapter 3 - Arrays and Vectors

$$n = 4$$
$$\text{temp1} = \text{undef}$$
$$\text{temp2} = \text{undef}$$

**caller's address**

Activation record for `fibonacci(4)`

1

$$n = 4$$
$$\text{temp1} = \text{undef}$$
$$\text{temp2} = \text{undef}$$

**caller's address**

Activation record for `fibonacci(4)`

2

$$n = 3$$
$$\text{temp1} = \text{undefined}$$
$$\text{temp2} = \text{undefined}$$

**line#4**

Activation record for `fibonacci(3)`

3

$$n = 3$$
$$\text{temp1} = 1$$
$$\text{temp2} = \text{undefined}$$

**line#5**

Activation record for `fibonacci(3)`

4

$$n = 2$$
$$\text{temp1} = \text{undefined}$$
$$\text{temp2} = \text{undefined}$$

**line#5**

Activation record for `fibonacci(2)`

5

$$n = 4$$
$$\text{temp1} = 2$$
$$\text{temp2} = \text{undefined}$$

**line#5**

Activation record for `fibonacci(4)`

6

$$n = 4$$
$$\text{temp1} = 2$$
$$\text{temp2} = \text{undefined}$$

**caller's address**

Activation record for `fibonacci(4)`

7

$$n = 2$$
$$\text{temp1} = \text{undefined}$$
$$\text{temp2} = \text{undefined}$$

**line#5**

Activation record for `fibonacci(2)`

8

$$n = 4$$
$$\text{temp1} = 2$$
$$\text{temp2} = \text{undefined}$$

**caller's address**

Activation record for `fibonacci(4)`

9
Tower of Hanoi

This problem, known as Tower of Hanoi, was invented by French mathematician Lucas in 1880s.

It says that in the town of Hanoi, monks are playing a game with 3 diamond needles fixed on a brass plate. One needle contains 64 pure gold disks of different diameters. Plates are put on top of each other in the order of their diameters with the largest plate lying at the bottom on the brass plate. Priests are supposed to transfer all the disks from one needle to the other such that: (a) only one disk can be moved at a time, and (b) at no time a disk of larger diameter should be put on a disk of smaller diameter. According to the legend, end of this game will mark the end of time!

Our task is to write a computer program to play the game with n disks.

A straightforward recursive algorithm is outlined next and is depicted in Figure ....

1. Base condition: When there is nothing to move, stop.
2. Recursive Step:
   a. Move n-1 disks from the origin to the auxiliary pin using the target pin as the temporary holding place.
   b. Move the nth disk from the origin to the target
   c. Move the n-1 disks moved to the auxiliary pin in step (a) to the target pin using the origin pin as the temporary holding place.

At that time all disks would have moved from the origin pin to the target pin.
(a) initial condition

(b) move n-1 disks from origin to auxiliary

(c) move the last disk from origin to target

(d) move n-1 disks from auxiliary to target

(a) initial condition

origin auxiliary target

(b) move n-1 disks from origin to auxiliary

origin auxiliary target

(c) move the last disk from origin to target

origin auxiliary target

(d) move n-1 disks from auxiliary to target

origin auxiliary target
Converting the algorithm to computer program is now a simple task and is given in Figure...

```c
void TOH (int from, int to, int using, int n)
{
    /* 1. */ if (n > 0)     // end condition - stop when there
        // is nothing to be moved
    /* 2. */ {
    /* 3. */    TOH (from, using, to, n-1);    // recursive step
                // move n-1 plates from the starting
                // disk to the auxiliary disk
    /* 4. */    move (from, to);                // move the nth disk to the destination
    /* 5. */    TOH (using, to, from, n-1);    // recursive step
                // move n-1 plates from the auxiliary
                // disk to the destination disk
    }
}
```

Figure ... shows the activity on the stack for the recursive function calls of TOH for transferring 3 disks from pin number 1 to pin number 3 using pin number 2 as the auxiliary pin.
Arrays and Vectors

f = 1, t = 2, u = 3
n = 3

activation record for move
f = 1, t = 1
return to line#3

f = 1, t = 3, u = 2
n = 2
return to line#2

f = 1, t = 2, u = 3
n = 3

activation record for move
f = 2, t = 1
return to line#3

f = 2, t = 3, u = 1
n = 1
return to line#4

f = 1, t = 3, u = 2
n = 2
return to line#2

activation record for move
f = 1, t = 2
return to line#3

f = 1, t = 3, u = 2
n = 2
return to line#2

f = 1, t = 2, u = 3
n = 3

activation record for move
f = 2, t = 3
return to line#3

f = 2, t = 3, u = 1
n = 1
return to line#4

f = 1, t = 3, u = 2
n = 2
return to line#2

f = 1, t = 2, u = 3
n = 3

activation record for move
f = 1, t = 2
return to line#3

f = 1, t = 3, u = 2
n = 2
return to line#2

f = 1, t = 2, u = 3
n = 3

activation record for move
f = 1, t = 2
return to line#3

f = 1, t = 3, u = 2
n = 2
return to line#2

f = 1, t = 2, u = 3
n = 3

activation record for move
f = 1, t = 2
return to line#3

1 2 3 4 5
6 7 8 9 10
11 12 13 14 15
16 17 18 19 20
Simulation of TOH with 3 disks last move not correct???
3.3.4 Complexity analysis of recursive algorithms.

Complexity analysis of recursive algorithms requires estimating the depth of recursion as well as the number of times the recursive call is made. The time complexity of such algorithms can be described in the form of recurrence relations. As mentioned in chapter 2, a recurrence relation describes a function in terms of itself with smaller inputs and is very closely related with mathematical induction. A recurrence relation for a recursive program can be written easily by just following the structure of the recursive program.

Following are some examples of recursive programs and their recurrence relation.

Example 1 (a).

```c
int fool(int n) {
    if (n <= 1)
        return n;
    else
        return n+fool(n-1);
}
```

There is only one recursive call where the input size is reduced by 1 and the function is called again recursively. Hence the recurrence relation is:

\[ T(n) = T(n - 1) + c \]

Where \( T(n) \) denotes the time for completing the function with input \( n \) which is equal to time for completing the function with input \( n - 1 \) plus some constant \( c \).
Example 1 (b).

```c
int foo2(int n) {
    if (n <= 1)
        return n;
    else if (n%2 == 0)
        return n + foo2(n-4);
    else
        return n + foo2(n-2);
}
```

The two recursive calls are mutually exclusive and for the worst case analysis we take the one with smaller step size and hence the recurrence relation for this function is:

\[ T(n) = T(n-2) + c \]

Example 1 (c)

```c
void foo3(int n, int m) {
    if (n <= 1)
        cout << n << endl;
    else {
        foo3(n-2, m+n);
        cout << n+m << endl;
        foo3(n-4, m-n);
        cout << n-m << endl;
    }
}
```

There are two recursive calls with different step sizes and hence the recurrence relation is:

\[ T(n) = T(n-2) + T(n-4) + c \]
The following table shows the recurrence relations for the recursive programs discussed earlier:

<table>
<thead>
<tr>
<th>Program</th>
<th>Recurrence Relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear search</td>
<td>$T(n) = T(n-1) + c$, and $T(1) = k$</td>
</tr>
<tr>
<td>Binary search</td>
<td>$T(n) = T(\text{ceiling}(n/2)) + c$, and $T(1) = k$</td>
</tr>
<tr>
<td>Fibonacci number</td>
<td>$T(n) = T(n-1) + T(n-2) + c$, $T(1) = k$, and $T(2) = k$</td>
</tr>
<tr>
<td>Tower of Hanoi</td>
<td>$T(n) = 2T(n-1) + c$, and $T(1) = k$</td>
</tr>
</tbody>
</table>

3.3.4.1 Solving recurrence relations

In this book we shall limit our discussion of solving the recurrence relations to the recurrence relations of the form:

$$T(n) = cT(m) + f(n), \text{ and } T(b) = g(n).$$

Where $T(b)$ denotes the base condition.

Such relations can be solved easily by repeatedly substituting the bigger input by the smaller input until a base condition is reached. At that point the problem is solved.

Let us see how it works by solving a few examples.

Example 1

$$T(n) = T(n-1) + c$$

$$T(1) = c$$

$$T(n) = T(n-1) + c$$

$$= [T(n-2)+c] + c = T(n-2) + 2c$$

$$= [T(n-3)+c] + 2c1 = T(n-3) + 3c$$

$$...$$

$$=T(n-(n-1))+ (n-1)c1$$

$$=T(1) + (n-1)c1$$

$$= c2 + (n-1)c1 = O(n)$$

Example 2

$$T(n) = T(\text{ceiling}(n/2)) + c$$

$$T(1) = c2$$
For complexity analysis of the cases that involves ceiling a floor functions such as \( \left\lceil \frac{n}{b} \right\rceil \) or \( \left\lfloor \frac{n}{b} \right\rfloor \), it is useful to assume that \( n \) can be written as a power of \( b \).

That is \( n = b^k \) or \( k = \log_b n \), where \( k \) is a whole number. This allows us to write the relation without the ceiling or the floor function.

Let us now solve the following recurrence relation:

\[
T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + c1
\]

\[
T(1) = c2
\]

Assuming \( n = 2^k \), we get:

\[
T(n) = T\left(\frac{n}{2}\right) + c1
\]

We are now ready to solve it by repeated substitution.

\[
T(n) = T(n/2) + c1 \\
= [T(n/4) + c1] + c1 = T(n/4) + 2c1 \\
= [T(n/8) + c1] + 2c1 = T(n/8) + 3c1 = T(n/2^3) + 3c1 \\
... \\
= T(n/2^k) + kc1 \\
... \\
= T(n/2^k) + kc1 \\
= T(n/n) + kc1 \\
= T(1) + c1 \log(n) \\
= c2 + c1 \cdot \log(n) \\
= O(\log n)
\]

Example 3.

\[
T(n) = 2T(n-1) + C1 \\
= 2[2T(n-2)+c1] + c1 = 4T(n-2) + 3c1 \\
= 2[4T(n-3)+c1]+3c1 = 8T(n-3) + 5c1 = 2^3T(n-3) + (2.3-1)c1 \\
...
\]
\[2^i \cdot T(n-i) + (2i-1)c1\]

\[
\ldots
\]

\[= 2^{n-1} \cdot T(n-(n-1)) + (2(n-1)-1)c1\]

\[= 2^{n-1} \cdot T(1) + 2n.c1 - 3c1\]

\[= 2^{n-1} \cdot c2 + 2n.c1 - 3c1\]

\[= O(2^n) + O(n) + O(1)\]

\[= O(2^n)\]
3.4 **Queues**

Just like a queue of people at a ticket window where service is delivered on the first-come-first-served basis, the queue data structure can be used to store data which can later be retrieved in the first-in-first-out (FIFO) order or first-come-first-served (FCFS) basis.

As in the case of the queue of people at the ticket window where newcomers join the queue at the tail or rear end of the queue and service is provided to the one at the front or head of the queue, in the queue data structure insertions and deletions are made at two different ends (compare it with stack where these two operations are performed at the same end) to maintain the FIFO order.

The Queue ADT supports the following operations:

- **Create:** Create an empty queue
- **Add:** Store an element onto a Queue
- **remove:** retrieve (delete) from Queue
- **Is_empty:** check if the Queue is empty
- **Is_Full:** check if the Queue is full

As can be easily seen, the list of operations supported by a queue is pretty similar to the ones supported by the Stack ADT.

As a queue is essentially a linear list and, just like a stack, it can therefore be very easily implemented by an array. An array based specification for the queue is given in Figure...
As compared to Stack, implementation of Queue ADT is however a little bit more difficult. To start with, we need to maintain two separate indices, front, and rear, to maintain information about the front and rear end of the queue and we also maintain the size of the queue. Fig. presents a simplistic implementation.

```cpp
template <class T>
class Queue {
public:
    Queue(int); // constructor
    ~Queue(); // destructor
    void add (const T &); // add an element to the Queue
    T remove(); // remove an element from Queue
    bool isFull(); // check if the Queue is full
    bool isEmpty(); // check if the Queue is empty
private:
    const int MaxSize; // maximum storage capacity
    int size; // no of elements in the queue
    int front; // index of the front element
    int rear; // index of the place where the new element is to be added
    T *queueArray; // array used to implement the Queue
    Queue& operator =(const Queue& other); // disallow assignment
    Queue(const Queue &other); // disallow copy
};
```

```cpp
template <class T>
Queue<T>::Queue(int s):
    MaxSize(s > 0 ? s :
    throw ILLEGAL_ARRAY_SIZE_EXCEPTION),
    queueArray(new T[size])
{
    rear = front = 0;
}

template <class T>
bool Queue<T>::isEmpty()
{
    return size == 0;
}
```

```cpp
template <class T>
Queue<T>::~Queue()
{
    delete []queueArray;
}
```

```cpp
template <class T>
bool Queue<T>::isFull()
{
    return size == MaxSize;
}
```
template <class T>
void Queue<T>::add(const T &elem) {
    if (!isFull()) {
        queueArray[rear] = elem;
        rear++;
        size++;
    } else
        throw QUEUE FULL;
}

template <class T>
T Queue<T>::remove() {
    T temp;
    if (!isEmpty()) {
        temp = queueArray[front];
        front++;
        size--;
        return temp;
    } else
        throw QUEUE EMPTY;
}

It looks very similar to the implementation of a Stack, except that insertions and deletions are performed at two different ends of the queue.

This solution however suffers from some very serious problems which are elaborated with the help of the following example.

Let us assume that we have a Queue of MaxSize 8. The Queue is initially empty and hence front, rear, and size are initialized to 0. After 8 insertions and 3 deletions we have front, rear, and size to be equal to 3, 8, and 5 respectively. This scenario is depicted in Figu..
(a) Is the Queue full?
This is obviously not true as there are only five elements in the queue whereas it can hold up to 8 elements.
(b) Can we add another element to the Queue?
This is a tricky question. Although the Queue is not full but the rear has a value equal to 8 which is not a valid index. The add operation however does not check for this condition and hence is logically incorrect. We can add that condition and disallow new additions to the queue once the rear pointer has reached MaxSize. This nevertheless would not allow us to add new elements whereas there would still be space available in the Queue. So, what can be done?
There can be many different solutions. One way is to avoid this situation by modifying the remove operation as described next: After removing the element from the Queue, all elements are pushed back one place so that front is always equal to 0 (in this particular case there is in fact no need to maintain information about the front). The rear is also adjusted accordingly. As all the remaining elements need to be shifted one place back, the time complexity of this operation will be O(n). All the other operations will be O(1).

We can achieve removal of elements in O(1) if we treat the array as a circular object instead of a linear one. As depicted in Figu..., this can be achieved by logically joining the first and last element and forcing the indexes (front and rear) to go to 0 once they have reached MaxSize.

Figure ... shows a queue of maxSize size 8 in its different states: (a) initial state, (b) after 8 additions and 3 deletions, and (c) after one more addition.
3.4.1 Priority queues

In many cases a priority is associated with elements stored on to a queue and it is required that, instead of the FIFO order, elements are removed from the queue in the order of their priority. Such a queue is called a priority queue. One way to implement such a queue is by implementing it as an ordered list where the relative order (ascending or descending) among elements is maintained. This would ensure that the element with the highest priority is always at the front and hence would be the first to be removed from the queue. To avoid unnecessary movement of data at the time of removal, the queue is maintained as a circular structure. It is
easy to see that in this case removal and insertion would take $O(1)$ and $O(n)$, respectively.

Many operating systems use priority queues to allocate the CPU to the process with the highest priority. There are usually a fixed number of priority levels and a separate FIFO queue is associated with each priority level so that any 2 processes with the same priority are handled in FIFO order. New jobs are inserted into their corresponding queues and jobs are removed from the highest priority non-empty queue.

We shall revisit the topic of priority queues in Chapter …
3.5 **exercise**

1. Write a function to add two 3-dimensional matrices of arbitrary sizes and store the result in a third matrix.
2. Develop addressing formula for a two dimensional matrix with column major organization. What would be the general formula for multi-dimensional matrices with column major organization?
3. Many programming languages allow the programmers to specify the first index of an array/matrix instead of using 0 as the first index of the array/matrix. For example, in Pascal, we can have the following definition of an array of integers which can hold 6 elements and the indices start from 5 and go up to 10.

   ```pascal
   myArray : Array[5..10] of Integer;
   ```

   Develop the addressing formula for such languages. Assume row major organization.
4. Develop storage scheme and the corresponding addressing formula for lower and upper triangular matrices.
5. Develop storage scheme and the corresponding addressing formula for band matrices.
6. Write code to create a lower triangular matrix using the concept of ragged arrays.
7. Convert the following into Polish as well as Reverse Polish Notation:
   (a) \( \frac{a}{b} + c \cdot d - e \)
   (b) \( \frac{a}{b + c \cdot (d - e)} \)
   (c) \((a-b) \cdot (c+d)-e\)
8. Assuming \(a=2, b=1, c=3, d=2\) and \(e=6\), use stack to evaluate the following expressions in RPN:
   (a) \(a \ b \ / \ c \ d \ * \ + \ e \ -\)
   (b) \(a \ b \ c \ d \ e \ - \ * \ + \ /
   (c) \(a \ b \ - \ c \ d \ + \ * \ e \ -\)
9. Exercises for recursion:

10. Ackermann’s function $A(m, n)$ is defined as:

$$A(m, n) = \begin{cases} 
(n + 1, & \text{if } m = 0 \\
A(m - 1, 1), & \text{if } n = 0 \\
A(m - 1, A(m, n - 1)), & \text{otherwise}
\end{cases}$$

Write a recursive function to compute $A(m, n)$ for some $m$ and $n$.

11. The binomial coefficient is defined as:

$$\binom{n}{m} = \begin{cases} 
1, & \text{if } m = n \text{ or } m = 0 \\
\binom{n - 1}{m - 1} + \binom{n - 1}{m}, & \text{otherwise}
\end{cases}$$

Write a recursive function to compute the binomial coefficient for some $m$ and $n$.

12. Write a recursive function to print the data stored in an array in reverse order.

13. Write a recursive function to return the maximum value in an array.

14. Write a recursive function that determines whether a string is a palindrome.

15. Write a recursive function to compute sum of elements in an array.

16. min and max values

17. Write a recursive function to convert an integer into its binary equivalent.

18. Determine what following program does:

```c
int mystery (int n) {
    if (n <= 0) return n;
    else return mystery(n-1) + n;
}
```

19. n choose k = (n-1 choose k-1) + (n-1 choose k), for n, k >=1

20. write a recursive function to translate prefix to postfix.

21. develop an algorithm to evaluate an expression in Polish notation without converting to RPN.

22. write a recursive program to find the kth smallest element of an array.

23. the eight queen problem is to place eight queens on a chessboard such that no queen is attacking another. Write a recursive program to determine the positions of the eight queen such that the condition is satisfied.

24. maze

25. Trace the execution of the following function for:
int array[] = {1, 2, 3};
traceMe(array, 1, 3);

void traceMe(int *a, int m, int n) {
    if (m == n - 1) {
        for (int i = 0; i < n; i++)
            cout << a[i] << " ";
        cout << endl;
    }
    else
        for (int i = m; i < n; i++) {
            swap(a[i], a[m]);
            traceMe(a, m + 1, n);
            swap(a[i], a[m]);
        }
}

26. Write a recursive function to generate the powerset of a given set. Assume that the
    elements of a set are stored in an array.
    (hint: carefully study traceMe function of Q. 7 and then devise your solution)
27. Indirect recursion – calculate sin and cos
28. Make a copy of a stack using only push and pop functions
29. Develop a program to implement two stacks in a single array.
30. Devise an algorithm using stack(s) to convert an expression in prefix to postfix.
31. Convert a postfix expression into a fully parenthesized infix expression.
32. Given the following recursive definition of an algebraic expression, write a
    recursive program to determine whether an expression is a valid algebraic
    expression:

    (i) Expression → Term + Term | Term
    (ii) Term → Factor * Factor | Factor
    (iii) Factor → Letter | ( Expression )

33. A priority queue can be implemented by using an array of queues – linux
34. A double-ended-queue (deque) is a queue in which items can be inserted at or
    deleted from either end.
35. Implement a stack using a deque
36. Devise a data structure that can be used to implement undo operation that can
    be used undo up to last 10 operations.
37. Show how to implement a queue using two stacks. Insert and remove operations are not necessarily $O(1)$.

38. The array-based stack implementation introduced in Program □ uses a fixed length array. As a result, it is possible for the stack to become full.

1. Rewrite the push method so that it doubles the length of the array when the array is full.
2. Rewrite the pop method so that it halves the length of the array when the array is less than half full.
3. Show that the average time for both push and pop operations is $O(1)$. **Hint**: Consider the running time required to push $n = 2^k$ items onto an empty stack, where $k \geq 0$.

39. Use the C++ STL class vector to implement a stack.

40. Reverse the contents of a given queue using another queue. Cannot use a stack or any other data structure.

41. Reverse the order of a queue using a stack.

42. Sort the elements of a stack using one additional stack and some non-array variables.

43. Sort the elements of a queue using one additional queue and some non-array variables.

44. Develop a template based Stack class to store heterogeneous data.

45. Develop a template based Queue class to store heterogeneous data.

46. The array-based queue implementation introduced in Program □ uses a fixed length array. As a result, it is possible for the queue to become full.

   a. Rewrite the enqueue method so that it doubles the length of the array when the array is full.
   b. Rewrite the dequeue method so that it halves the length of the array when the array is less than half full.
   c. Show that the average time for both enqueue and dequeue operations is $O(1)$. 
47. Suppose we add a new operation to the stack ADT called $\textit{findMinimum}$ that returns a reference to the smallest element in the stack. Show that it is possible to provide an implementation for $\textit{findMinimum}$ that has a worst case running time of $O(1)$.

48. Suppose that a client performs an intermixed sequence of (stack) push and pop operations. The push operations put the integers 0 through 9 in order on to the stack; the pop operations print out the return value. Which of the following sequence(s) could not occur?

(a) 4 3 2 1 0 9 8 7 6 5
(b) 4 6 8 7 5 3 2 9 0 1
(c) 2 5 6 7 4 8 9 3 1 0
(d) 4 3 2 1 0 5 6 7 8 9
(e) 1 2 3 4 5 6 9 8 7 0
(f) 0 4 6 5 3 8 1 7 2 9
(g) 1 4 7 9 8 6 5 3 0 2
(h) 2 1 4 3 6 5 8 7 9 0

49. Write a stack client $\textit{Reverse.java}$ that reads in strings from standard input and prints them in reverse order.

50. Write a stack client $\textit{Parentheses.java}$ that reads in a text stream from standard input and uses a stack to determine whether its parentheses are properly balanced. For example, your program should print $\texttt{true}$ for $\text{[()]{}{}{{[()()]}()}}$ and false for $\text{[()]}$. Hint: Use a stack.

51. What does the following code fragment print when $N$ is 50? Give a high-level description of what the code fragment does when presented with a positive integer $N$.

```java
Stack stack = new Stack();
while (N > 0) {
    stack.push(N % 2);
    N = N / 2;
}
while (!stack.isEmpty())
    StdOut.print(stack.pop());
StdOut.println();
```

$\textit{Answer:}$ prints the binary representation of $N$ ($110010$ when $N$ is 50).

52. What does the following code fragment do to the queue $q$?
Stack stack = new Stack();
while (!q.isEmpty())
    stack.push(q.dequeue());
while (!stack.isEmpty())
    q.enqueue(stack.pop());

53. Suppose that a client performs an intermixed sequence of (queue) 
enqueue and dequeue operations. The enqueue operations put the
integers 0 through 9 in order on to the queue; the dequeue
operations print out the return value. Which of the following
sequence(s) could not occur?

(a)  0 1 2 3 4 5 6 7 8 9
(b)  4 6 8 7 5 3 2 9 0 1
(c)  2 5 6 7 4 8 9 3 1 0
(d)  4 3 2 1 0 5 6 7 8 9

54. Write an iterable Stack client that has a static methods copy() that
takes a stack of strings as argument and returns a copy of the stack.
Note: This ability is a prime example of the value of having an iterator,
because it allows development of such functionality without changing
the basic API.

55. Josephus problem Program Josephus.java uses a queue to solve the
Josephus problem.

56. Dynamic shrinking. With the array implementations of stack and queue,
we doubled the size of the array when it wasn't big enough to store the
next element. If we perform a number of doubling operations, and then
delete alot of elements, we might end up with an array that is much
bigger than necessary. Implement the following strategy: whenever the
array is 1/4 full or less, shrink it to half the size. Explain why we don't
shrink it to half the size when it is 1/2 full or less.

57. Queue with two stacks. Show how to implement a queue using two
stacks. Hint: If you push elements onto a stack and then pop them all,
they appear in reverse order. If you repeat this process, they're now
back in order.
58. **Text editor buffer.** Implement an ADT for a buffer in a text editor. It should support the following operations:

- `insert(c)`: insert character c at cursor
- `delete()`: delete and return the character at the cursor
- `left()`: move the cursor one position to the left
- `right()`: move the cursor one position to the right
- `get(i)`: return the ith character in the buffer

*Hint:* use two stacks.

59. Suppose you have a single array of size N and want to implement two stacks so that you won't get overflow until the total number of elements on both stacks is N+1. How would you proceed?

*Hint:* the two stacks are initialized to start from the opposite ends.

60. C uses row-major organization for multidimensional arrays whereas FORTRAN uses column major. We have been given a two-dimensional array that was created in FORTRAN. We want to convert it to row-major so that we can use it in C. Write the code for the following function to achieve that task:

```c
void convertCol3RowMajor(int *fromArray, int *toArray, int rows, int columns);
```