3 Data Structures and Operations

Significant Learning Outcomes

After the completion of this chapter, the learner

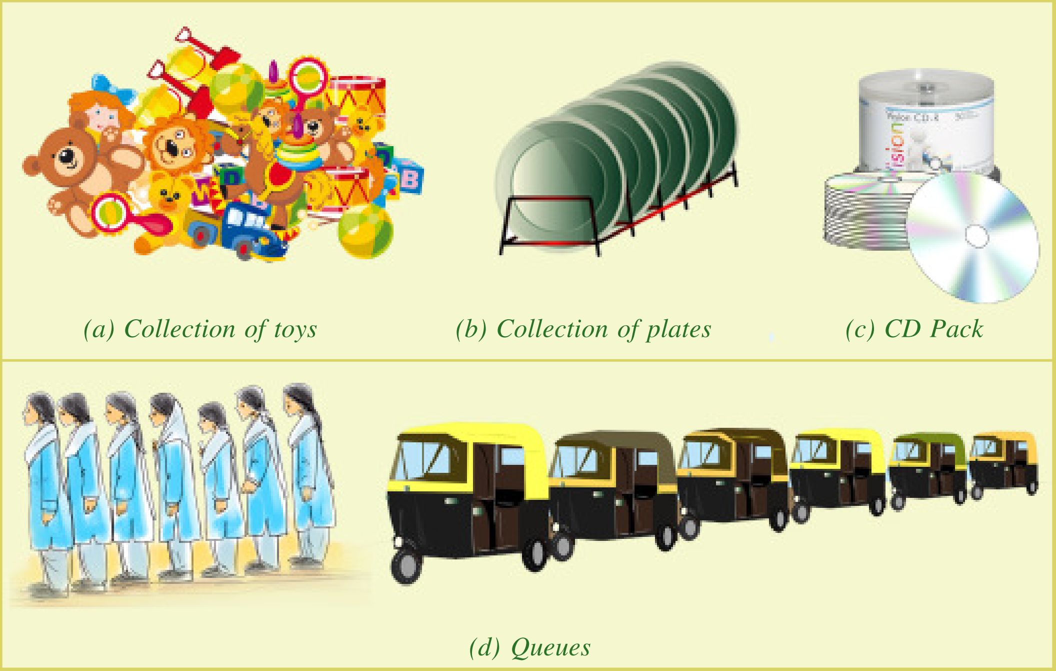
- explains the concept of data structure by citing examples.
- classifies data structures based on different criteria.
- lists different operations on data structures and explains them.
- explains the organisation of stack data structure with the help of examples.
- develops algorithms for push and pop operations in a stack.
- explains the organisation of queue data structure with the help of examples .
- develops algorithms for insertion and deletion operations in a linear queue.
- identifies the advantage of circular queue over linear queue.
- explains the concept of linked list data structure and its advantages over arrays and other static data structures.
- develops procedures to create a linked list and to perform traversal operation.

While solving problems using computers, the data may have to be processed in most of the cases. These data may be of atomic (fundamental) type or aggregate (grouped) type. We know that variables are required to refer to these data. Languages like C, C++, Java, etc. insist on declaration of variables before their use in the program. We learnt that in C++, variables for atomic type data are declared using fundamental data types like `int`, `char`, `float` and `double` or with their type modifiers. We have also seen that grouped data are referred using arrays and structures. Array is a collection of homogeneous type of data, whereas structure can be a collection of different types of data.

This chapter presents the facilities of programming languages to organise data in groups based on different principles and criteria. The amount of data that can be accommodated in a group and the operations performed on them vary depending on the organising principle followed in constituting each group of data.

3.1 Data Structure

Figure 3.1 shows some items in groups. Each group follows some kind of grouping strategy. Can you identify the principle or format followed in arranging the items in each group?



(a) Collection of toys *(b) Collection of plates* *(c) CD Pack*

(d) Queues

Fig. 3.1: Different ways of grouping

Figure 3.1(a) is a collection of toys. The toys are dumped together without any specific order or arrangement. Figure 3.1(b) is a collection of plates in a stand. The plates are placed one after the other. There is a limit for the number of plates that can be placed in a stand. A new plate can be placed at any position in the stand if there is space and any plate can be taken out from it. Figure 3.1(c) is a set of discs in a CD pack. There is a limit to the number of discs in this collection also. A new disc can be added in the collection only at the top. Similarly only the CD at the top can be removed from the collection. Figure 3.1(d) shows queues in which a new person (or a new auto rickshaw) can join the queue only at the rear end. The person (or auto rickshaw) leaves the queue only from the front end. In this collection there may not be a limit for the number of persons in the queue. But in some cases there may be a limit in the size of the queue also.

The concept of data structure is similar to the collections in figures (b), (c) and (d) in Figure 3.1. Figure 3.1(b) is a collection, which is very similar to an array that we learnt in Class XI. So we say that array is a data structure. In Computer Science, a

data structure is a particular way of organising similar or dissimilar logically related data items which can be processed as a single unit. Data structures not only allow the user to combine various types of data but also allow processing of the group as a single unit.

3.1.1 Classification of data structures

Data structures can be generally classified into simple data structures and compound data structures. Figure 3.2 shows the detailed classification.

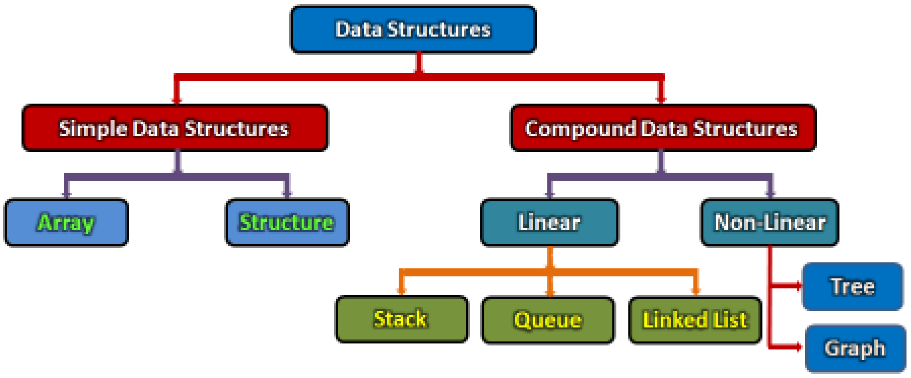


Fig. 3.2: Classification of data structures

We are familiar with the simple data structures such as array and structure. We used them in C++ programs to refer to a collection of elements. These simple data structures are combined in various ways to form compound data structures. As shown in Figure 3.2, compound data structure is further classified into linear and non-linear. A data structure is said to be linear if its elements form a sequence. The elements of a linear data structure can be represented by means of sequential memory locations. A data structure is said to be non-linear if its elements do not form any sequence in memory. In this type of data structure the elements are stored in random memory locations and they need not be accessed in sequential order. Non-linear data structures are more complex and hence you will study them in higher classes. Linear data structures such as stack, queue and linked list will be presented in detail in the coming sections.

Since data structures represent collections of data, these are closely related to computer memory, as it is the storage space for the data. The memory may be primary or secondary. Depending upon memory allocation, data structures may be classified as ***static data structures*** and ***dynamic data structures***. Static data structures are associated only with primary memory. The required memory is allocated before the execution of the program and the memory space will be fixed throughout the execution. That is, the size of the data structure is specified during the design and it cannot be changed later. Data structures designed or implemented using arrays are static in nature. But for dynamic data structures, memory is allocated during execution. Data structures implemented using linked lists are dynamic in nature. The size of the collection will not be specified in advance; rather it grows or shrinks during run-time as per user's desire. When we consider secondary memory for the storage of data, it will be in the form of files. The size of such data files increases on addition of data and reduces on deletion of data. So we can say that files are also dynamic data structures.

3.1.2 Operations on data structures

The data represented by the data structures are processed by means of certain operations. In fact, a particular data structure that one chooses for a given situation depends largely on the frequency with which specific operations are performed. The operations performed on data structures are traversing, searching, inserting, deleting, sorting and merging. Let us have some basic idea about these operations.

a. Traversing

Traversing is an operation in which each element of a data structure is visited. The travel proceeds from the first element to the last one in the data structure. Processing of the visited element can be done according to the requirements of the problem. Reading all the elements of an array is an example for traversing (Refer Chapter 8 of Computer Science Textbook for Class XI).

b. Searching

Searching, in the literal sense, is the process of finding the location of a particular element in a data structure. Searching may also be a process of finding the locations of all the elements satisfying one or more conditions. In other words, searching implies accessing the values stored in the data structure. We learned two methods for the search operation in an array in Class XI.

c. Inserting

Insertion is the operation in which a new data is added at a particular place in a data structure. In some situation, where the elements in the data structure are in a particular order, the position may need to be identified first and then the insertion is to be done.

d. Deleting

Deletion is the operation in which a particular element is removed from the data structure. The deletion is performed either by mentioning the element itself or by specifying its position.

e. Sorting

We are familiar with the sorting of an array using two methods named bubble sort and selection sort. It is the technique of arranging the elements in a specified order, i.e., either in ascending or descending order. Sorting of elements in a data structure makes searching faster.

f. Merging

Merging usually refers to the process of combining elements of two sorted data structures to form a new one. But the simplest form of merging is the joining of the elements of both the data structures into a third empty data structure. In the case of an array, first, copy all the elements of one array into a third empty array, and then append all the elements of the other array to those in the third array.

Searching, sorting and merging are three related operations which make the job of retrieval of data from the storage devices easier, faster and efficient.

In Class XI, we learned the data structure array and how the operations given above are performed. We also discussed the concept of structures and the way of operations on their elements in Chapter 1 of this book. Now, let us discuss the compound linear data structures stack, queue and linked list.

3.2 Stack

Have a close look at Figure 3.1(c) and also the collections shown in Figure 3.3. The organisation of items in these groups is the same.

Fig. 3.3: Real life examples for stack

The collection is formed by adding each item one over the other. We can say that the items are added at the top position. Similarly, we can remove only that item which is placed at last. This organising principle is known as Last-In-First-Out (LIFO). The data structure that follows LIFO principle is known as ***stack***. It is an ordered list of items in which all insertions and deletions are made at one end, usually called **Top**. Since it follows the LIFO principle, the element added at last will be the first to be removed from the stack.

Stack is a logical concept. There is no exclusive tool or facility in programming languages to create a stack. A stack can be physically implemented by an array. Such a stack inherits all the properties of arrays. The size is predetermined and hence it is static. Figure 3.4 is a stack of integers implemented by an array **STACK**. This stack can accommodate a maximum of 10 integers. The figure shows that there are six elements at present and the last element is 56 at position 5. The last position is indicated by **TOS** (to denote Top Of Stack). So, the value of **TOS** is 5. The last element in a stack is always referred to by the expression **StackName[TOS]**. In this case **STACK[TOS]** gives 56. Since we utilise arrays of C++ for implementing stacks, the first element will always be referred to by the subscript 0. Here **STACK[0]** is 25.

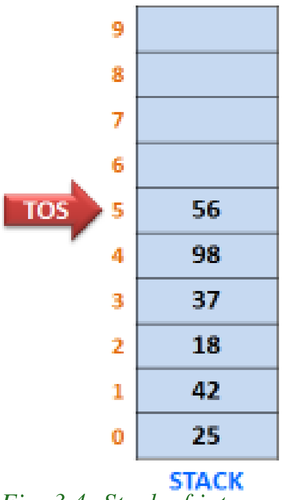


Fig. 3.4: Stack of integers

3.2.1 Implementation of stack

It is mentioned that stack can be implemented using array. In such a case, there is a limit to the number of elements that can be represented by the stack and it depends on the size of the array. Initially, the value of TOS will be set to -1 (minus one) to denote the fact that the stack is empty. Whenever an element is added (or inserted) into the stack, the value of TOS will be incremented by 1 until it reaches the highest subscript of the array. If an array STACK of N elements is used to implement a stack, the values of Top can vary from 0 to (N-1) and the elements in the stack can be referred to by the expressions STACK[0], STACK[1], STACK[2], ..., STACK[N-1].

3.2.2 Operations on stack

Though array is used to implement stack data structure, all the operations applicable to array are not performed in the same fashion. For example in an array, insertion and deletion operations can be performed at any position. But in stack, there is restriction in the position. These operations are performed only at the top position. The insertion and deletion operations on stack are known as ***push*** and ***pop*** operations respectively. Let us discuss the procedure involved in these operations.

a. Push operation

As mentioned above, ***push*** operation is the process of inserting a new data item into the stack. Even the creation of a stack itself is a repeated execution of push operations.

Let us do

Figure 3.5 shows the status of a stack during a sequence of push operations. Let us assume that we have created an array and value of TOS is set with -1. Now observe the figure and write down the procedure to perform the operation.

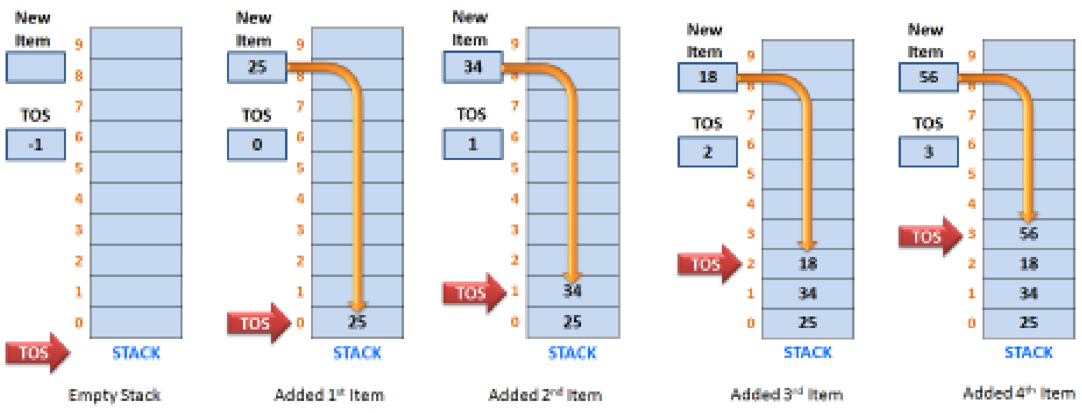


Fig. 3.5: Status of stack after a sequence of push operations

Verify whether you could derive the following steps for push operation on a stack:

Step 1: Accept the value in a variable to insert into the stack.
Step 2: Increment the value of Top by 1.
Step 3: Store the value at the TOS position.

The above steps are valid only if there is free space for insertion. The stack shown in Figure 3.5 cannot accommodate a new item when the value of TOS is 9. Once the stack is full and if we attempt to insert an item, an impossible situation arises. This is known as ***stack overflow***. Now let us write an algorithm for push operation in a stack.

Algorithm: To perform PUSH operation in a stack

Consider an array STACK[N] that implements a stack, where N is the maximum size of the array. A variable TOS keeps track of top position of the stack. A data item available in the variable VAL is to be added into the stack. The steps required for push operation are given between the Start and Stop instructions.

```
Start
    1:  If (TOS < N) Then            //Space availability checking (Overflow)
    2:      TOS = TOS + 1
    3:      STACK[TOS] = VAL
    4:  Else
    5:      Print "Stack Overflow "
    6:  End of If
Stop
```

a. Pop operation

The process of deleting an element from the top of a stack is called ***Pop*** operation. After every pop operation the value of TOS is decremented by one.

Let us try ourselves to derive the steps for deleting the element at the top of a stack. Figure 3.6 is given to you for reference. Observe the figure and write down the steps for pop operation.

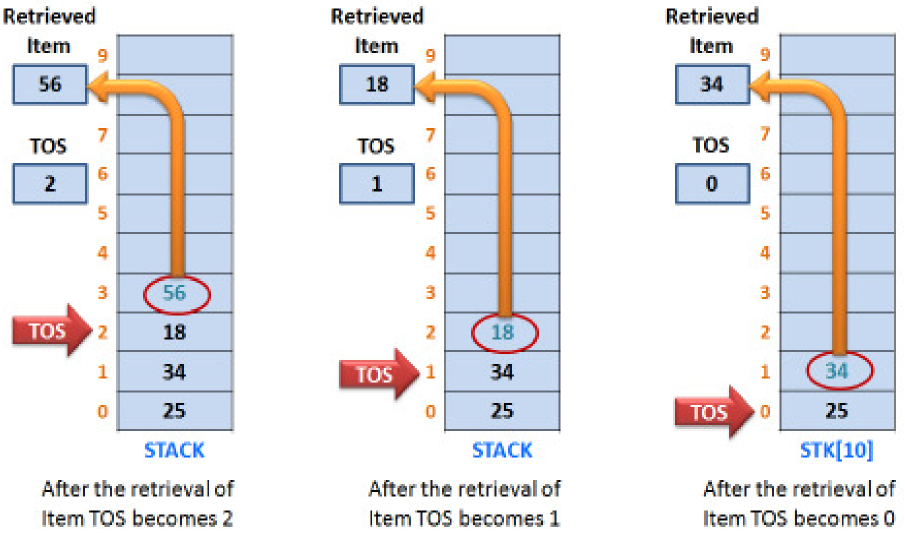

Retrieved Item
56
TOS
2
9
7
6
5
4
3
2
1
0
56
18
34
25
TOS
STACK
After the retrieval of Item TOS becomes 2
Retrieved Item
18
TOS
1
18
34
25
TOS
STACK
After the retrieval of Item TOS becomes 1
Retrieved Item
34
TOS
0
34
25
TOS
STK[10]
After the retrieval of Item TOS becomes 0

Fig. 3.6: Status of stack after a sequence of pop operations

You might have derived the following steps for pop operation:

Step 1: Retrieve the element at the TOS into a variable.

Step 2: Decrement the value of TOS by 1.

These two steps will work when there are elements in the stack. During the pop operation in an array stack, the element is not physically removed, but the access is restricted by decrementing the value of TOS. The stack shown in Figure 3.6 can be given for pop operation until the element at position 0 is deleted. After the deletion of that last element, the value of TOS becomes -1. Now the stack is empty and if we try to delete an item from the stack, an unfavourable situation arises. This situation is known as ***stack underflow***. Now let us write an algorithm for pop operation in a stack.

Algorithm: To perform POP operation in a stack

Consider an array STACK[N] that implements a stack that can store a maximum of N elements. A variable TOS keeps track of the top position of the stack. The data item retrieved from the stack may be kept in a variable, say VAL. The steps required for pop operation are given between the Start and Stop instructions.

```
Start
    1:   If  (TOS > -1) Then            //Empty status checking (Underflow)
    2:        VAL = STACK[TOS]
    3:        TOS = TOS - 1
    4:   Else
    5:        Print "Stack Underflow "
    3:   End of If
Stop
```

C++ functions for stack operations

The variables `tos` and `n` are assumed as global variables

PUSH operation

```
void push(int stack[],int val)
{
    if (tos < n)
    {
        tos++;
        stack[tos]=val;
    }
    else
        cout<<"Overflow";
}
```

POP operation

```
int pop(int stack[])
{
    int val;
    if (tos > -1)
    {
        val=stack[tos];
        tos--;
    }
    else
        cout<<"Underflow";
    return val;
}
```

Application of Stacks

Since stacks follow the LIFO principle, they are used for applications such as reversing a string, creating polish strings, evaluation of polish strings etc. Reversing a string involves the formation of a string by reversing the characters of the original string. For example, "SAD" is a string and its reversed string is "DAS". Polish string is an arithmetic expression, in which operator is placed before the operands or after the operands. For example, the arithmetic expression A+B can be converted into AB+ or +AB. The expression A+B is familiar to us and is known as infix expression. The expression AB+ or +AB is the required format for Arithmetic Logic Unit (ALU) of the computer and are known as postfix expression and prefix expression, respectively. These conversions are carried out in the computer with the help of push and pop operations. The prefix or postfix version of an arithmetic expression is also evaluated using these stack operations by ALU.

3.3 Queue

In many situations we might have become part of a queue. Figure 3.7 shows a queue in a polling station, where the voter at the front position cast his/her vote first and a new person can join the queue at the rear position. It is clear that, the first voter in the queue will come out first from the queue. This style of oraganising a group is said to follow the First-In-First-Out (FIFO) principle. So, we can say that a data structure that follows the FIFO principle is known as a ***queue***. As we can see in the figure a queue has two end points - front and rear. Inserting a new data item in a queue will be at the rear position and deleting will be at the front position. Queue is also a logical concept. It can be physically implemented by an array and such a queue is static in nature.

Fig. 3.7: Queue in a polling station

3.3.1 Implementation of queue

When a queue is implemented by an array, there is a limit for the number of elements that can be represented by it. Figure 3.8 is a queue of integers implemented by an array QUEUE, which can accommodate a maximum of 10 integers. The figure shows that there are five elements in the queue stored from positions 0 to 4. So, the value of Front is 0 and that of Rear is 4. The first element of this queue is always referred to by QUEUE[Front] and that last element by QUEUE[Rear].

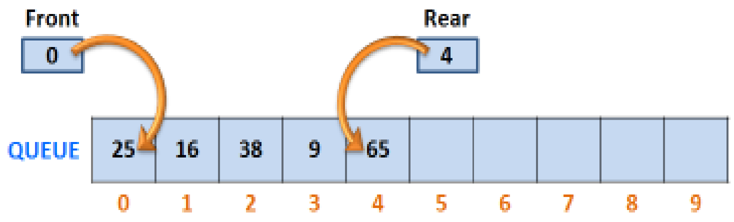


Fig. 3.8: Queue implemented by array

The highest value allowed to Rear is 9 as it is the last subscript of the array. Initially, the value of Front and Rear would have been -1, which indicates that the queue is empty. When the first element is inserted into the queue, the values of these end points will be set to 0. Thereafter, whenever an element is inserted, the value of Rear will be incremented by 1, until it reaches the highest subscript (here it is 9). Similarly on each deletion, the value of Front will be incremented by 1, until it crosses the value of Rear.

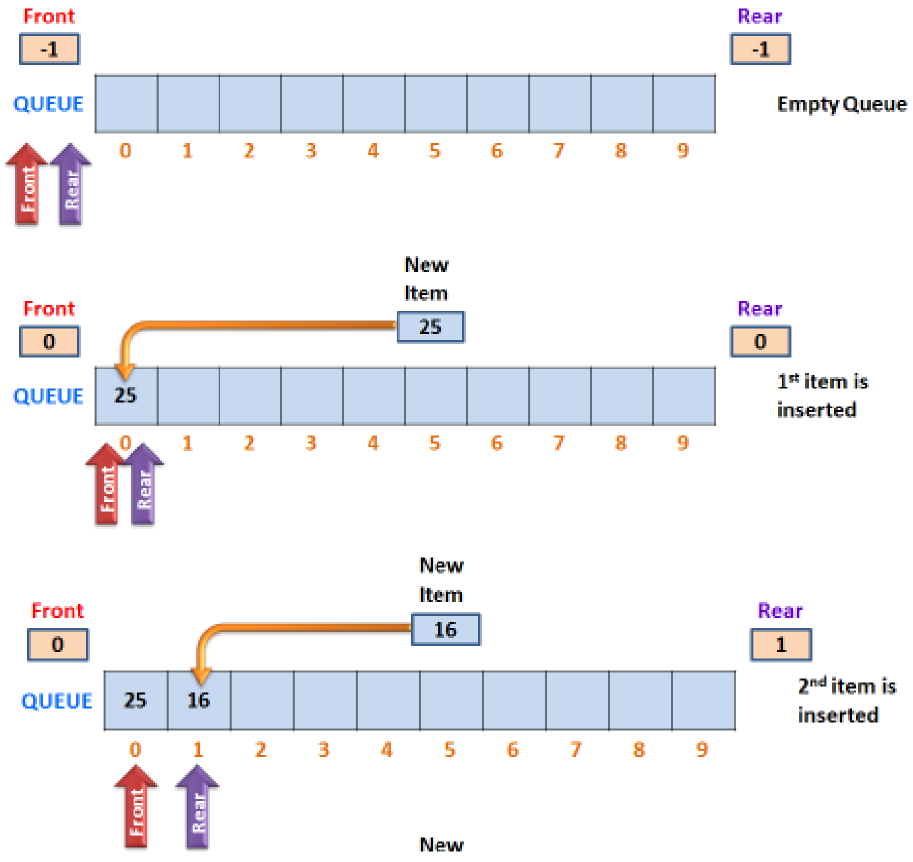
3.3.2 Operations on queue

As in the case of stacks, there are some restrictions in the insertion and deletion operations on queues. In an ordinary array, insertion and deletion operations are performed at any position and in stacks these operations are performed only at the top position. But in the case of queue, insertion and deletion operations are done at different end points.

a. Insertion operation

Insertion is the process of adding a new item into a queue at the rear end. The value of Rear is incremented first to point to the next location and the element is inserted at that position. Even the creation of a queue is accomplished by performing the insertion operation repeatedly.

Figure 3.9 shows the status of a queue during a sequence of insertion operations. Let us assume that we have created an array and value of Front and Rear are set to -1. Now, observe the figure and write down the procedure to perform the operation.



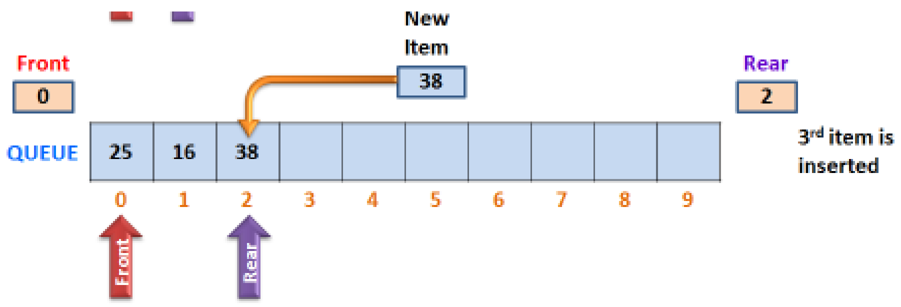


Fig. 3.9: Status of a queue after a sequence of insertion operations

Now check whether you could derive the following steps for the insertion operation

Step 1: Accept the value in a variable for inserting into the queue.

Step 2: Increment the value of Rear by 1.

Step 3: Store the value at the Rear position.

Note that during the first insertion operation in the empty queue, the values of both Front and Rear are incremented, i.e., they are set to 0. Thereafter, only the Rear is incremented. This can be continued until Rear becomes 9 (the last subscript of the array). The attempt of next insertion causes ***queue overflow*** as in the case of stack. Now, let us write an algorithm for insertion operation in a queue.

Algorithm: To perform INSERTION operation in a queue

Consider an array QUEUE[N] that implements a queue, where N is the maximum size of the array. The variables FRONT and REAR keep track of the front and rear positions of the queue. A data item available in the variable VAL is to be inserted into the stack. The steps required for insertion operation are given between the Start and Stop instructions.

```
Start
     1:   If (REAR == –1) Then          //Empty status checking
     2:        FRONT = REAR = 0
     3:        Q[REAR] = VAL
     4:   Else If (REAR < N) Then       //Space availability checking
     5:        REAR = REAR + 1
     6:        Q[REAR] = VAL
     7:   Else
     8:        Print "Queue Overflow "
     9:   End of If
Stop
```

b. Deletion operation

Deletion operation is the removal of the item at the front end. After the deletion, the value of Front is incremented by 1. In the case of array queue, the item is not physically removed, rather its access is denied by incrementing the value of Front.

Assume that we have a queue implemented by an array. Figure 3.10 shows the status of this queue during a sequence of deletion operations. Let us try to derive the steps for the operation ourselves.

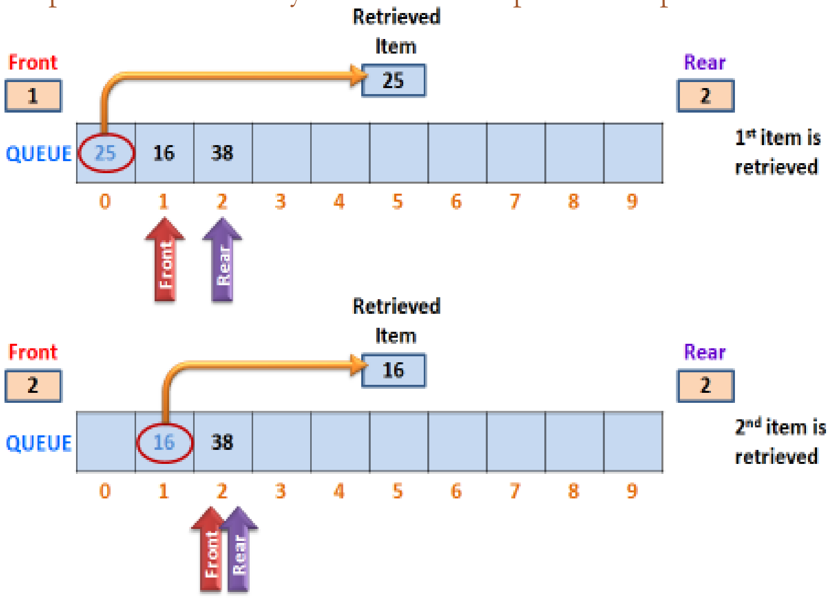


Fig. 3.10: Status of a queue after a sequence of dequeue operations

Could you derive the following steps for deletion operation?

Step 1: Retrieve the element at the Front position into a variable.

Step 2: Increment the value of Front by 1.

According to Figure 3.10, there is no shifting of elements towards the front as you may expect. The queues in real life situations may conflict with this concept. But in queue data structure, deletion operation does not cause a shift of elements; rather the value of Front is incremented by 1. The two steps derived will be enough as long as there is element in the queue. Look at the status of the queue after the deletion of 2nd element. Front and Rear point to the same element, which is at subscript 2. Assume that, one more deletion is carried out in the queue. According to the procedure the value of Front becomes 3, which is greater than that of Rear. We know that it is not reasonable. We can also see that the queue is now empty. Remember that, the value of Front and Rear is -1 when the queue is empty. So we have to include a step to reset the values of Front and Rear as -1 when the last element is deleted from the queue. In this state, further deletion cannot be allowed.

If an attempt is made to delete an item from an empty queue, the situation is called ***queue underflow***. Now let us develop a complete algorithm for deletion operation on a queue.

Algorithm: To perform DELETION operation in a queue

Consider an array QUEUE[N] that implements a queue with a maximum of N elements. The variables FRONT and REAR keep track of the front and rear positions of the queue. The data item removed from the queue will be stored in the variable VAL. The steps required for deletion operation are given between the Start and Stop instructions.

```
Start
    1:   If (FRONT > –1 AND FRONT < REAR) Then     // Empty status checking
    2:           VAL = Q[FRONT]
    3:           FRONT = FRONT + 1
    4:   Else
    5:           Print "Queue Underflow "
    6:   End of If
    7:   If (FRONT > REAR) Then            // Checking the deletion of last element
    8:           FRONT = REAR = -1
    9:   End of If
Stop
```

C++ functions for queue operations

The variables `n`, `front` and `rear` are assumed as global variables

INSERTION operation

```
void ins_q(int queue[],int val)
{
    if (rear == -1)
    {
        front=0;
        rear=0;
        q[rear]=val;
    {
    else (if rear < n)
    {
        rear++;
        q[rear]=val;
    }
    else
        cout<<"Overflow";
}
```

DELETION operation

```
int del_q(int queue[])
{
    int val;
    if (front > -1)
    {
        val=q[front];
        front++;
    }
    else
        cout<<"Underflow";
    if (front > rear)
    {
        front= -1;
        rear= -1
    }
    return val;
}
```

Application of Queues

Mostly the queue concept in computer applications occurs in job scheduling. The computer operating systems use this queue concept in scheduling the memory, processor and the files. Print queue is an example of job scheduling. Since the printer is slow compared to the processor, the print jobs submitted by the user are put in the print buffer. The jobs are organised in the buffer following the FIFO principle and thus the print buffer becomes a queue.

3.3.3 Circular queue

Queues we discussed so far are known as linear queues. The elements are arranged in a row or line. The two ends of such queues never meet at any point. There is a drawback in linear queue. Consider the queue shown in Figure 3.11 in which six deletion operations are done. At present, there are only three elements. Obviously, the value of Front is 6 and that of Rear is 8.

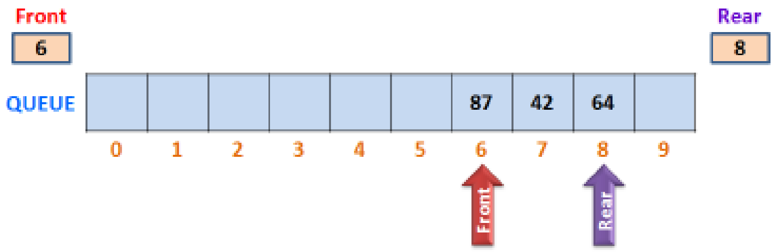


Fig. 3.11: Queue with three elements after a sequence of six deletion operations

Even though the first six locations are free, we can insert only one element according to the insertion algorithm. Imagine the worst situation that there is only one element at the last position of the queue (say at 9). If we try an insertion operation in this queue, there will be overflow situation. This limitation of linear queue can be overcome by circular queue. It is a queue in which the two end points meet as shown in Figure 3.12.

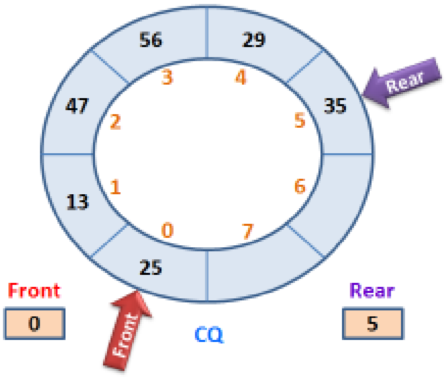


Fig.3.12: Circular queue

Assume that two deletions are performed at this stage. So the value of Front will be 2 and now we have four free locations as shown in Figure 3.13(a). The subscripts of these position are 6, 7 and then 0, 1. If we insert two elements one by one, the value of Rear becomes 7, but still we have two free locations with subscripts 0 and 1,

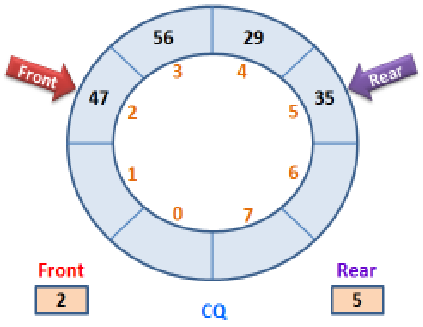


Fig.3.13(a): Circular queue after deletion

as shown in Figure 3.13(b). So insertion operation can be performed again. This time the value of Rear should be set with 0 first and then insertion is to be carried out. Figure 3.13(c) shows this status of the queue.

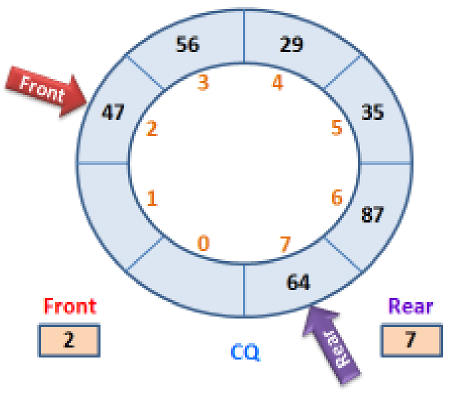


Fig.3.13(b): Rear gets its highest value

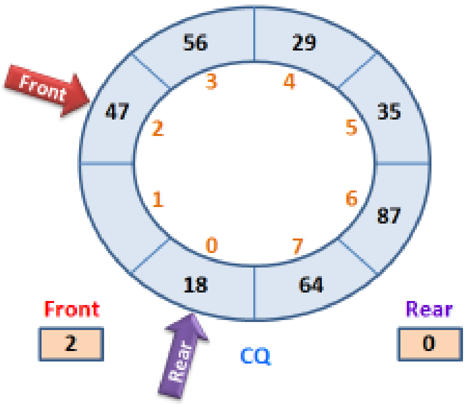


Fig.3.13(c): Rear takes the lower bound value

Know your progress

1. Define data structure.
2. Stack follows __________ principle for oragising data.
3. Name the data structure that follows FIFO principle.
4. What is meant by underflow?
5. Which element of stack can be deleted (First or Last)?

3.4 Linked list

Linked list is a collection of data items where there is no limit in the number of items. Stacks and queues discussed in the previous sections are merely logical concepts, which are physically implemented using arrays. So these implementations are static. But linked list is a dynamic data structure. It grows as and when new data items are added in the list and shrinks whenever any data is removed from the list. Memory will not be allocated in advance for the entire list. The memory allocation takes place during the execution time just when a new item is about to insert in the list. That is why it is considered as an example for dynamic data structure. Another difference compared to array based data structures is that the elements in the linked list are scattered in the memory. But they are linked with pointers. As we learnt in Chapter 1, pointer is a variable that contains the address of a memory location. So it is clear that an element in a linked list consists of data and an address. Such an element is known as ***node***. The address contained in the node is known as ***link***.

So, a ***linked list*** is a collection of nodes, where each node consists of a ***data*** and a ***link*** - a pointer to the next node in the list. That is, the first node in the list contains the first data item and the address of the second node; the second node contains the second data and the address of the third node; and so on. The last node at a particular stage contains the last data in the list and a *null pointer*, i.e., a pointer that points to nowhere. Now, a question arises. Where will the address of the first node be? There is a special pointer associated with each linked list, known as Start or Header and it contains the address of the first node. Figure 3.14 shows a linked list of five numbers.

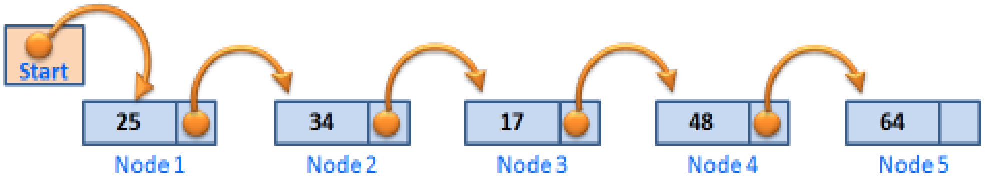


Fig. 3.14: Linked list with five nodes

The figure shows that each node in the linked list consists of a number as its data and a pointer as the link that points to the next node. The size of all the nodes will be the same. That is, memory space allocated for each node will be of the same size.

3.4.1 Implementation of linked list

We have seen that linked list is a collection of nodes and each node is allocated memory space. The memory location for a node consists of at least two types of data, one is the actual data item and the other is a pointer to the memory location for the next node. In Chapter 1, we learnt that structure is a user-defined data type consisting of different types of data. The element of a structure can be a pointer and it may even be a pointer to the same structure. We call such a structure a self referential structure. So, linked list is created with the help of self referential structures. The nodes in the linked list in Figure 3.14 can be designed using the following self referential structure:

```
struct Node
{
    int data;
    Node *link;
};
```

As we can see, the name of the structure is Node and the second element is a pointer to the same structure type **`Node`**. The special pointer **`Start`** that points to the first node can be created using the following statement:

```
struct Node * Start;        or        Node *Start;
```

Figure 3.15 shows a linked list of strings. The data of the nodes will be filled with strings and link with addresses of other nodes. Note that the addresses are only assumed numbers. The following structure can represent these nodes:

```
struct Node
{
    char data[10];
    Node *link;
};
```

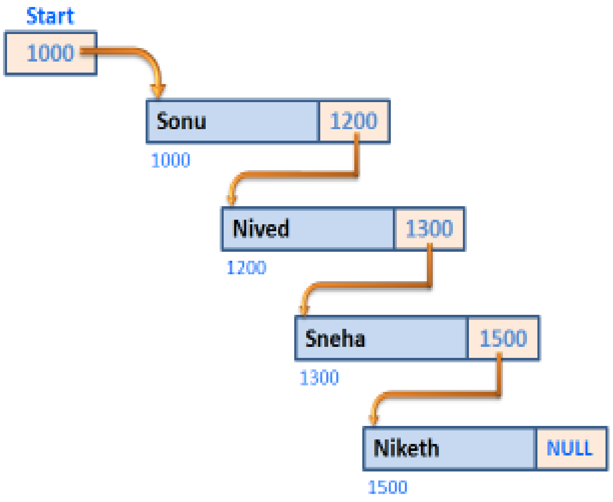


Fig.3.15: Implementation of a linked list

It is assumed that memory location at address 1000 is allocated during run-time for the first node and hence the pointer `Start` contains 1000. The data part of this node is filled with a name "Sonu". The second node is given memory space at location 1200 and its data part is filled with another name "Nived". Being the second node, its address is stored in the link part of the first node. As seen in the figure, the last node contains null pointer.

3.4.2 Operations on linked list

All operations specified in Section 3.1.2 can be performed in linked lists without any restrictions. But we will discuss only the creation, traversal, insertion and deletion operations only. You will learn the remaining operations during higher studies.

a. Creation of linked list

We have to define a suitable self referential structure in the beginning. A pointer variable say **Start** or **Header** is then declared and initialised with **NULL** value. Now we start creating a linked list by dynamically allocating memory for the nodes according to the requirements. In Chapter 1, we learnt that when memory is dynamically allocated, an address is be returned. This address is stored in a pointer variable and using this variable the location can be accessed.

Let us do

Figure 3.16 shows the status of a linked list during its creation. Let us assume that we have defined a self referential structure named **Node** and initialised a **Node** type pointer **Start** with **NULL**. Now observe the figure and write down the procedure to create a linked list.

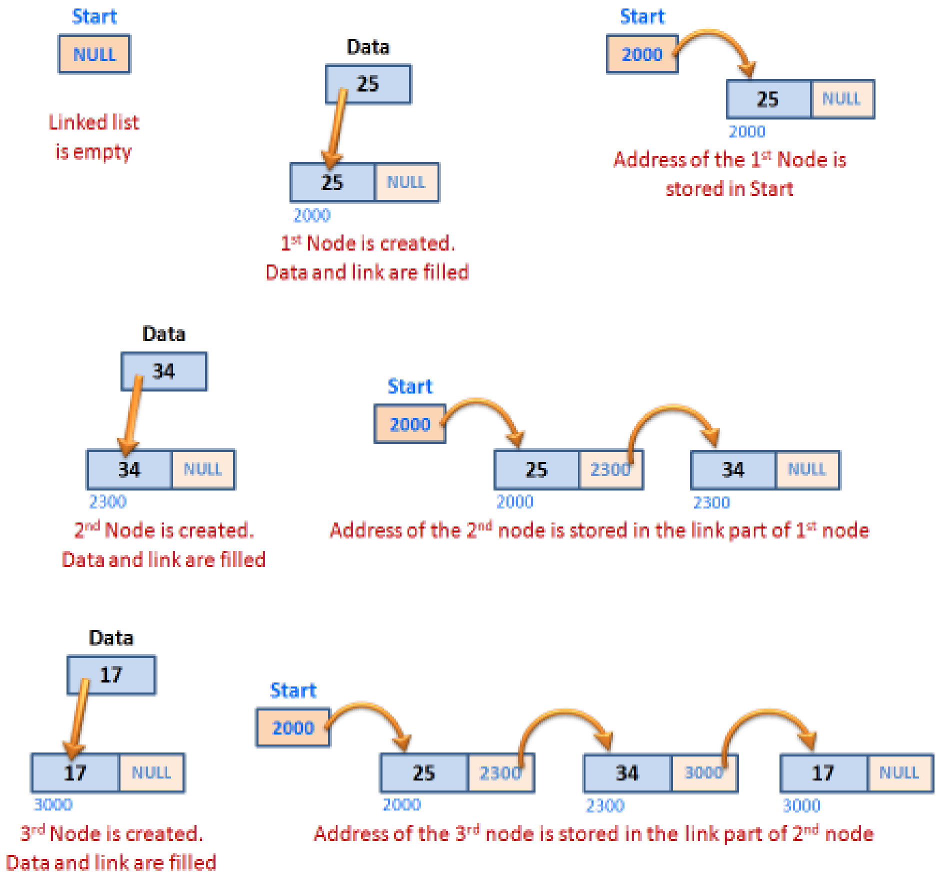

Start
NULL
Linked list is empty
Data
25
25 NULL
2000
1st Node is created. Data and link are filled
Start
2000
25 NULL
2000
Address of the 1st Node is stored in Start
Data
34
34 NULL
2300
2nd Node is created. Data and link are filled
Start
2000
25 2300
34 NULL
2000
2300
Address of the 2nd node is stored in the link part of 1st node
Data
17
17 NULL
3000
3rd Node is created. Data and link are filled
Start
2000
25 2300
34 3000
17 NULL
2000
2300
3000
Address of the 3rd node is stored in the link part of 2nd node

Fig. 3.16: Sequence of operations to create a linked list

Could you develop the following steps?

Step 1: Create a node and obtain its address.
Step 2: Store data and NULL in the node.
Step 3: If it is the first node, store its address in Start.
Step 4: If it is not the first node, store its address in the link part of the previous node.
Step 5: Repeat the steps 1 to 4 as long as the user wants.

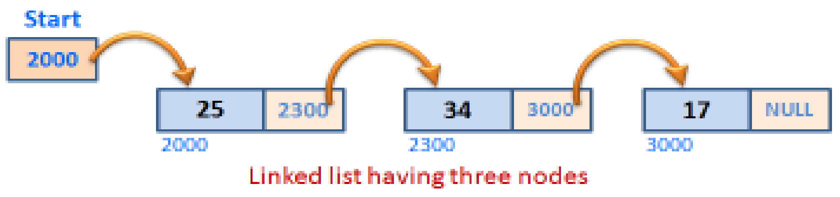
Actually the creation of linked list can be viewed as repeated insertion operations at the end of a linked list. Insertion of the second node onwards requires a traversal operation in the list to get the address of the last node in the current list. So let us discuss the traversal operation.

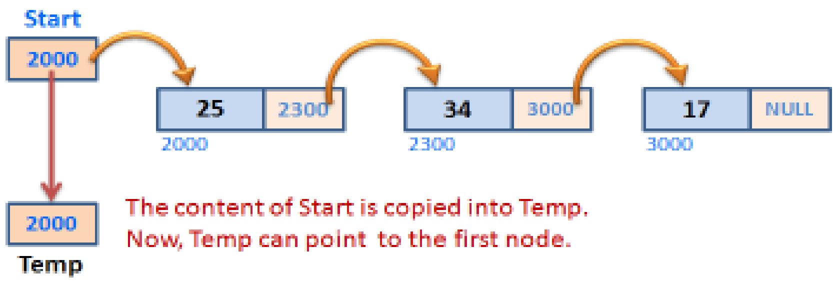
b. Traversing a linked list

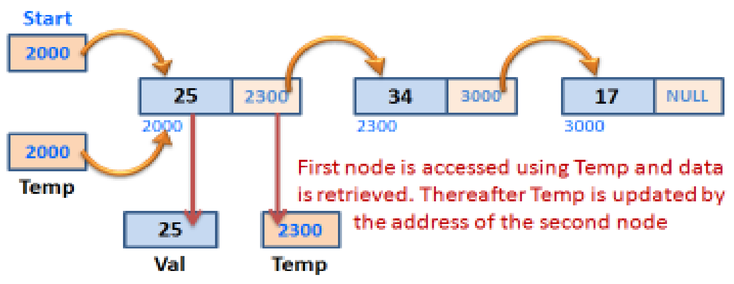
We know that traversal means visiting all the elements in a data structure. In the case of a linked list, we begin traversal from the first node. The pointer **Start** gives the address of the first node, so that we can access the data part using *arrow* (**->**) *operator*. Then we access the link part of the first node, which is the address of the second node. Using this address we can access the data and link of the second node. This process is continued until we found NULL pointer in the link of a node.

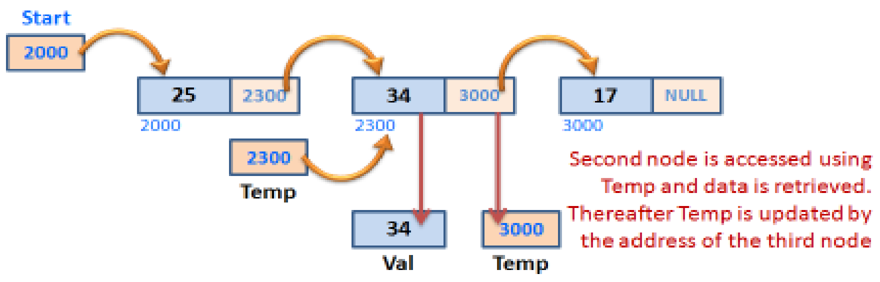
Let us do

Let us observe Figure 3.17 and derive the steps required for traversal operation in a linked list. It is assumed that Temp is a pointer of Node type and Val is a variable to store the data read from a node.




Start
2000
25
2300
2000
34
3000
2300
17
NULL
3000
2000
Temp
The content of Start is copied into Temp.
Now, Temp can point to the first node.


Start
2000
25
2300
2000
34
3000
2300
17
NULL
3000
2000
Temp
First node is accessed using Temp and data is retrieved. Thereafter Temp is updated by the address of the second node
25
Val
2300
Temp


Start
2000
25
2300
2000
34
3000
2300
17
NULL
3000
2300
Temp
Second node is accessed using Temp and data is retrieved. Thereafter Temp is updated by the address of the third node
34
Val
3000
Temp

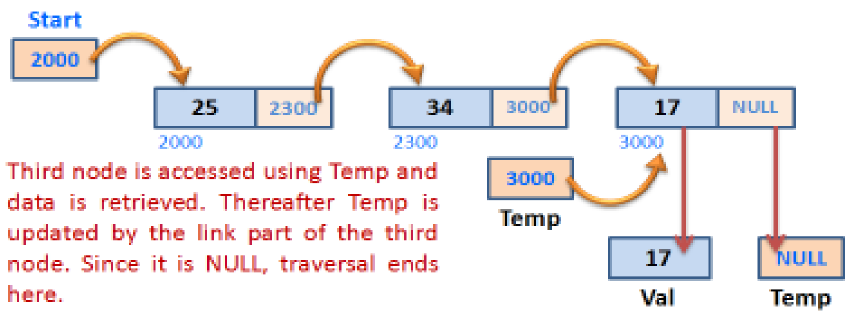

Start
2000
25
2300
2000
34
3000
2300
17
NULL
3000
Third node is accessed using Temp and data is retrieved. Thereafter Temp is updated by the link part of the third node. Since it is NULL, traversal ends here.
3000
Temp
17
Val
NULL
Temp

Fig. 3.17: Traversal operation in a linked list

Check whether you could derive the following steps:

Step 1: Get the address of the first node from Start and store it in Temp.

Step 2: Using the address in Temp, get the data of the first node and store in Val.

Step 3: Also get the content of the link part of this node (i.e., the address of the next node) and store it in Temp.

Step 4: If the content of Temp is not NULL, go to step 2; otherwise stop.

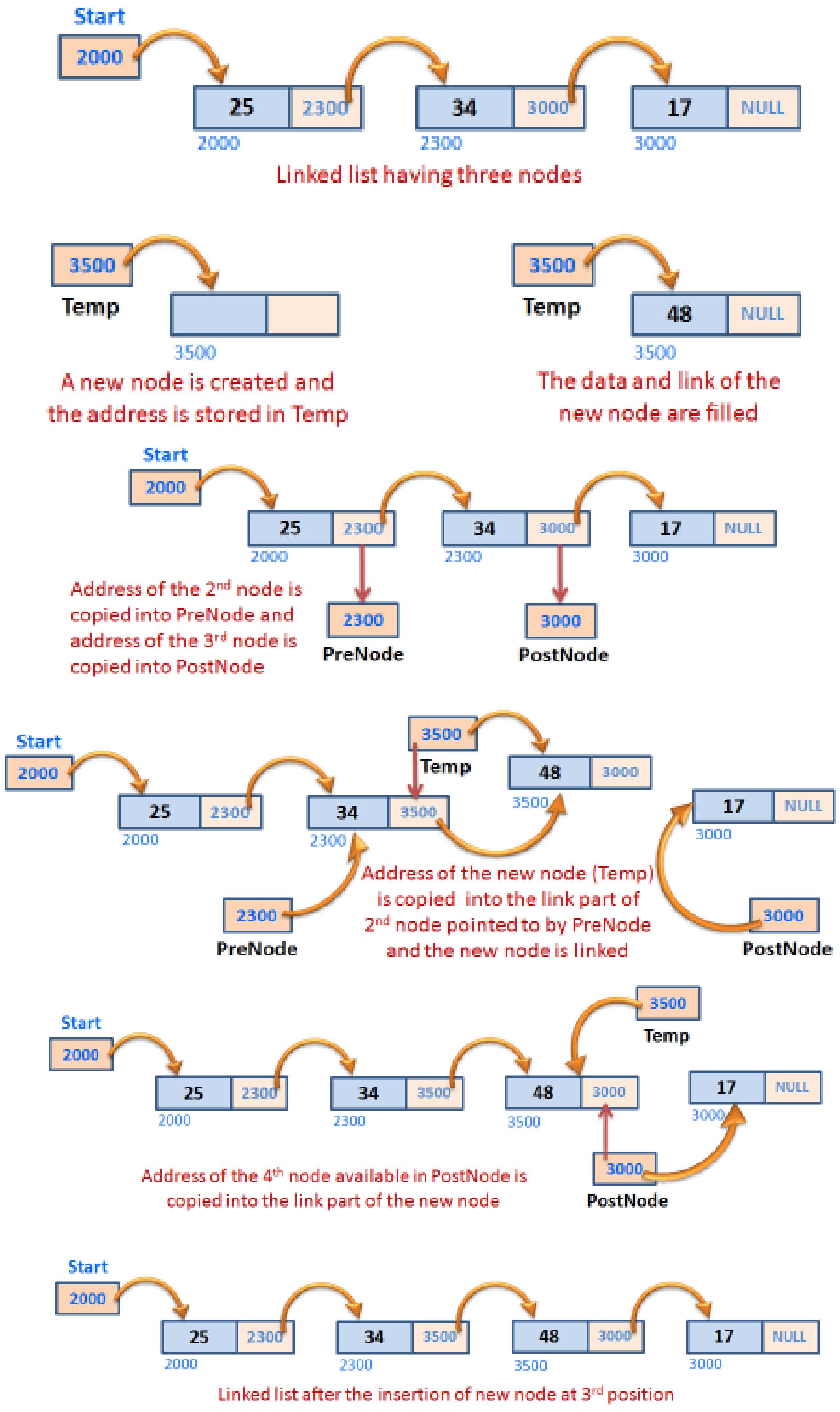
The above steps are required in the creation of a linked list in case Start is not NULL. In such a case, we have to find the address of the last node for storing the address of the new node in its link. The traversal operation begins from the first node by procuring its address from Start. This address will be copied into a temporary pointer variable (Temp in the figure) and it will be updated by copying the content of the link of the node pointed to by Temp. The visit will be continued until the link of a node pointed by Temp shows NULL value.

c. Insertion in a linked list

Insertion of an item in a linked list is the process of placing the node containing the item in a particular position. As in the case of arrays, a node can be inserted anywhere in a linked list - at the beginning, at the end or in between any two nodes. Once a new node is created, it can be inserted at the beginning by copying the content of Start into the link part of the new node and the address of the new node into Start. Similarly to insert a node at the end, we have to copy the address of the new node into the link part of the last node. Let us try to inert a node at a specified position.

Let us do

Figure 3.18 shows a series of operations to be performed to insert a node at the 3rd position in a linked list that has three nodes initially. Observe the figure and let us derive the steps required for insertion operation in a linked list. It is assumed that Temp, PreNode and PostNode are pointers of Node type structure. Let POS be a variable that contains the position value where the node is to be inserted.


Start
2000
25
2300
2000
34
3000
2300
17
NULL
3000
Linked list having three nodes
3500
Temp
3500
A new node is created and the address is stored in Temp
3500
Temp
48
NULL
3500
The data and link of the new node are filled
Start
2000
Address of the 2nd node is copied into PreNode and address of the 3rd node is copied into PostNode
2300
PreNode
3000
PostNode
Address of the new node (Temp) is copied into the link part of 2nd node pointed to by PreNode and the new node is linked
Address of the 4th node available in PostNode is copied into the link part of the new node
Linked list after the insertion of new node at 3rd position
Fig. 3.18: Insertion operation in a linked list

The following steps can be developed for the insertion operation:

Step 1: Create a node and store its address in Temp.

Step 2: Store the data and link part of this node using Temp.

Step 3: Obtain the addresses of the nodes at positions (POS-1) and (POS+1) in the pointers PreNode and PostNode respectively, with the help of a traversal operation.

Step 4: Copy the content of Temp (address of the new node) into the link part of node at position (POS-1), which can be accessed using PreNode.

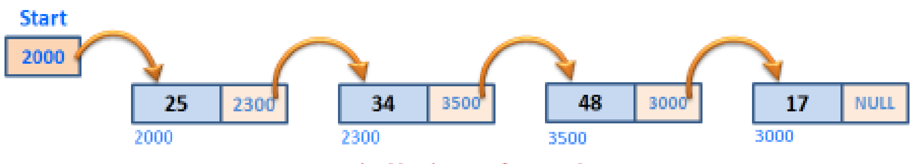
Step 5: Copy the content of PostNode (address of the node at position POS+1) into the link part of the new node that is pointed to by Temp.

d. Deletion from a linked list

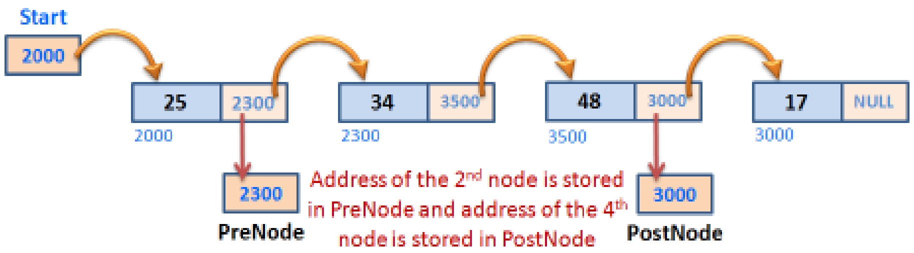
Deletion of an item from a linked list is the process of removing a node from the list. The position of the node to be removed will be given. Instead of this, if the data item is given, the node containing that item is to be searched and its position is to be noted. Then we apply the steps for removal operation. Any node can be removed from a linked list. To remove the first node, we have to copy the content in the link part of the first node into Start. The last node can be removed by assigning the NULL value to the link part of the second last node. Let us discuss the procedure to remove a node from a given position.

Let us do

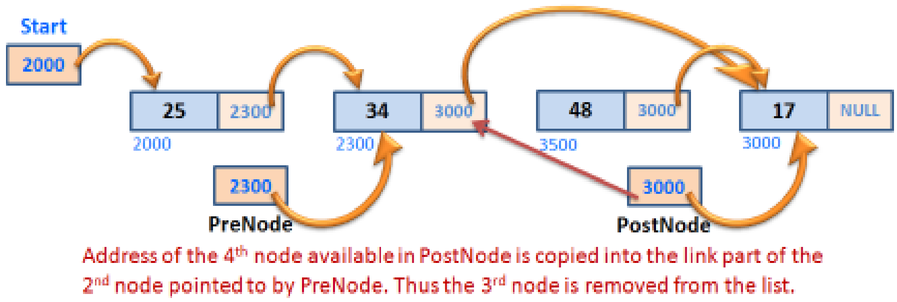
Figure 3.19 illustrates the procedure for the removal of 3rd node from the linked list having four nodes initially. Observe the figure and try to develop the steps required for deletion operation. It is assumed that PreNode and PostNode are pointers of Node type structure. Let POS be a variable that contains the position of the node to be removed.



Linked list having four nodes



Start
2000
25
2300
2000
34
3500
2300
48
3000
3500
17
NULL
3000
2300
PreNode
3000
PostNode
Address of the 2nd node is stored in PreNode and address of the 4th node is stored in PostNode


Start
2000
25
2300
2000
34
3000
2300
48
3000
3500
17
NULL
3000
2300
PreNode
3000
PostNode
Address of the 4th node available in PostNode is copied into the link part of the 2nd node pointed to by PreNode. Thus the 3rd node is removed from the list.

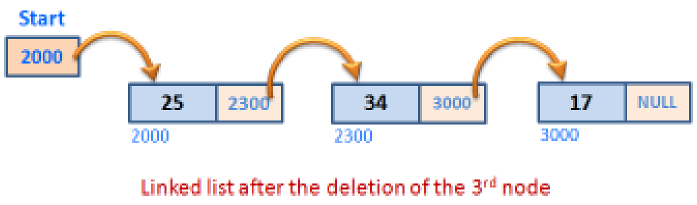


Fig. 3.19: Deletion operation in a linked list

The figure may help you derive the following steps for the deletion operation:

Step 1: Obtain the addresses of the nodes at positions (POS-1) and (POS+1) in the pointers PreNode and PostNode respectively, with the help of a traversal operation.

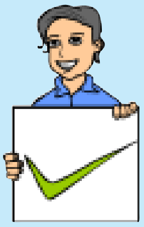
Step 2: Copy the content of PostNode (address of the node at position POS+1) into the link part of the node at position (POS-1), which can be accessed using PreNode.

Step 3: Free the node at position POS.

If we apply the first two steps in the linked list illustrated in Figure 3.19, even after the delinking of the 3rd node from the 2nd one, the presence of the 3rd node will be still there in the memory, pointing to the 4th node. So it should be freed using memory de-allocation facility provided by the programming language.

While implementing operations in linked lists, the temporary pointers like TEMP, PreNode, PostNode, etc. should also be freed after the operations.

Know your progress



1. Name an example for dynamic data structure.
2. What is linked list?
3. A node of a linked list consists of ______ and ______.
4. Which is the facility of programming language used to define the node of a linked list?
5. What is the content of Start or Header in a linked list?

Stacks and queues can be implemented using linked list too, which results into dynamic stacks and queues. The linked lists we have discussed are singly linked list, in the sense that a node can point to the next node only. But there are doubly linked lists, in which each node points to the next node as well as the previous node. It is made possible by including two pointers in the self referential structure. Complex data structures like tree are created using doubly linked lists.

Let us conclude

We are familiarised with the concept of data structures and the operations performed on them. There is a variety of data structures, so that any kind of data can be represented using a suitable one. The operations also vary depending on the oraganising principle followed in grouping the elements. Though some of the data structures are logical concepts, we have discussed their physical implementations. The difference between the array and linked list implementations has been mentioned during the discussion. The concepts covered in this chapter are the essential and important basics for your higher studies in Computer Science.

Let us assess

1. Read the following statements:
   (i) A collection of data processed as a single unit.
   (ii) All data structures are implemented using arrays.
   (iii) Stacks and queues are logical concepts and implemented using arrays and linked lists.
   (iv) Overflow occurs in the case of static data structures.

   Which of the above statements are true? Choose the correct options from the following:

   a. Statements (i) and (iii) only b. Statements (i), (ii) and (iii) only
   c. Statements (i), (iii) and (iv) only d. Statements (i), (ii) and (iv) only
2. Data structures may be static or dynamic.
   a. Give two examples for static data structures.
   b. Static data structures may face overflow situation. Why?
   c. Linked list is a dynamic data structure. Justify this statement.
3. Write an algorithm to insert an element into a stack.
4. What is meant by push and pop operations?

5. Write an algorithm to delete an element from a stack.
6. Write algorithms to perform insertion and deletion operations in linear queues.
7. How does circular queue overcome the limitation of linear queue?
8. Some of the operations performed on data structures are given below:

   i. Insertion ii. Deletion iii. Searching iv. Sorting

   a. Which of these operations may face underflow situation?

   b. Explain this situation in the context of an appropriate data structure.
9. Match the following:

A	B	C
a. Array	i. Start	1. Insertion and deletion at different ends
b. Stack	ii. Subscript	2. Insertion and deletion at the same end
c. Queue	iii. Rear	3. Self referential structure is utilized
d. Linked list	iv. Top	4. Elements are accessed by specifying its position

10. Explain why linked lists do not face overflow situation as in the case of array based data structures.