distance apart, whereas linked lists do not necessarily contain consecutive memory locations. These data items can be stored anywhere in the memory in a scattered manner. To maintain the specific sequence of these data items, we need to maintain link(s) with a successor (and/or a predecessor). It is called as a *linked list* as each node is linked with its successor (and/or predecessor). Figures 6.1 and 6.2 show the realization of a linear list using a linked list.

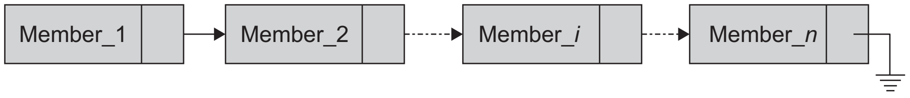


Fig. 6.1 A linked list of *n* elements

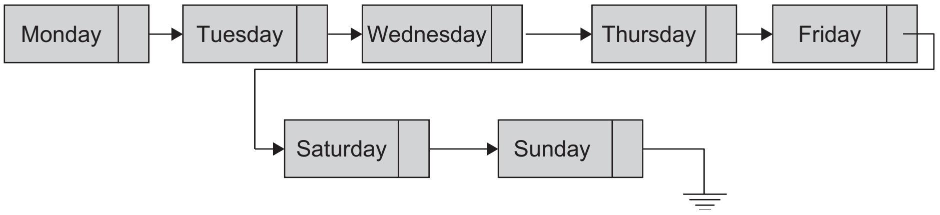


Fig. 6.2 A linked list of days in a week

The linked list, as a data structure in programming, is used quite frequently since it is very efficient. To use linked lists effectively, the concepts of pointers must be very clear to the programmer. In fact, frequent use of linked lists makes the concept of pointers very clear to the programmer. This study of the linked list will introduce us to its strengths and weaknesses. This study gives us an appreciation of the time, space, and code complexity issues. Linked list examples are a classic combination of algorithms and manipulation of pointers. Let us now learn about the linked list.

6.2 LINKED LIST

A *linked list* is an ordered collection of data in which each element (node) contains a minimum of two values, *data* and *link*(*s*) to its successor (and/or predecessor). A list with one link field using which every element is associated to its successor is known as a *singly linked list* (SLL). In a linked list, before adding any element to the list, a memory space for that node must be allocated. A link is made from each item to the next item in the list as shown in Fig. 6.3.

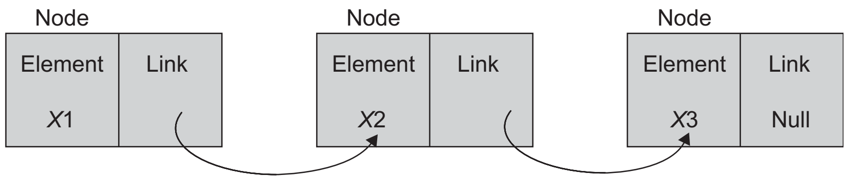


Fig. 6.3 Linked list

Each node of the linked list has at least the following two elements:

1. The data member(s) being stored in the list.
2. A pointer or link to the next element in the list.

The last node in the list contains a null pointer (or a suitable value like −1) to indicate that it is the end or tail of the list, and by suitable means we identify the first node. As elements are added to a list, memory for a node is dynamically allocated. Therefore, the number of elements that may be added to a list is limited only by the amount of memory available. To understand the linked list concept better, let us consider Examples 6.1 and 6.2.

EXAMPLE 6.1 We all are aware of the very interesting game of treasure hunt. In this game, a team member is provided the primary hint of the first locality. From the first location's hint, the participant gets the second, and so on. To reach the final target, the participant has to go through each and every location in a specific order. Even if the order of one of the locations is wrong, the participant will not obtain the clue for reaching the next location, and hence, the player will not be able to find the final destination of the treasure.

EXAMPLE 6.2 Assume that there are 10 books in a library, which form a specific sequence. This ordered set of 10 books is to be kept in a shelf. There are two ways to arrange the books. One of the arrangements is to keep all the 10 books in 10 continuous empty slots (similar to an array). The second possible arrangement is to place the books at available locations in a distributed manner (similar to a linked list) by keeping track of the various locations of the books.

Let Books = {book1, book2, book3, …, book10}

As the books form a specific sequence, both the arrangements must preserve the sequence. Let us analyse both the arrangements. Table 6.1 shows the first arrangement.

Table 6.1 Shelf and books arranged sequentially

Shelf position	S	$S + 1$	$S + 2$	$S + 3$	$S + 4$	$S + 5$	$S + 6$	$S + 7$	$S + 8$	$S + 9$
Book Number	Book1	Book2	Book3	Book4	Book5	Book6	Book7	Book8	Book9	Book10

The following are the requisites for this arrangement.

1. In a shelf, we need an empty slot that can accommodate all the 10 books.
2. We need to be aware of the position of the first book.
3. The order is maintained by keeping the books in sequence as book1, book2, and so on till book10, in successive empty locations.
4. Referring to the i^{th} location directly with respect to the first location, one can access the i^{th} book. In short, we have direct access to any i^{th} book.

Now, consider the following situation. An empty slot sufficient to accommodate 10 books is not available in the shelf, but 10 empty scattered locations are available.

Let us take a look at the second arrangement as shown in Table 6.2. Here, we need 10 empty locations available in the shelf. These 10 locations could be distributed and need not necessarily be a continuous block.

Table 6.2 Shelf and books arranged in a distributed manner

First book

Location	S	$S+1$	$S+2$	$S+3$	$S+4$	$S+5$	$S+6$	$S+7$
Book no.	Already occupied	Already occupied	Book 1	Already occupied	Book 5	Already occupied	Book 7	Book 2
Next book link			goto $S+7$		goto $S+14$		goto $S+10$	goto $S+11$

Location	$S+8$	$S+9$	$S+10$	$S+11$	$S+12$	$S+13$	$S+14$	$S+15$
Book no.	Already occupied	Book 4	Book 8	Book 3	Book 10	Book 9	Book 6	Empty
Next book link		goto $S+4$	goto $S+13$	goto $S+9$	Null	goto $S+12$	goto $S+6$	

Last book

The following steps are used for this arrangement:

1. Let us use some means to preserve the sequence. Let us keep the first book in the first free location found. Do note the location of the first book. Let us keep the second book at the second empty location in the shelf. Attach a tag as a link to the first book to remember where the second book is kept. This tag has the location ID of the second book. Put the third book in the next empty location. Attach a tag to link to the second book. The second book's link stores the location ID of the third book, and so on.
2. Remember only the first book's position.
3. We cannot refer to the third book directly. Only the link attached to the second book can indicate where the third book is. The second book's position is available at the first book's link. Hence, to get the i^{th} book, we have to go through all the books in a sequence: book1, book2, and so on till book$(i-1)$. The tag attached to the $(i-1)^{th}$ book would tell where the i^{th} book is.

The first arrangement is similar to arrays, a sequential organization. The second arrangement is a linked list, a linked organization. Now, let us compare both the arrangements. The first method needs continuous empty spaces to accommodate 10 books, whereas the second method can accommodate books in any of the 10 empty places, which may or may not be continuous. Hence, even if a continuous space to keep the 10 books is not available in the second method, the books can be accommodated.

In the first method, we have direct access to any i^{th} book in the sequence, whereas in the second method, until we traverse through the first i books sequentially, we cannot find where the $(i + 1)^{th}$ book is kept. In the first method, we must know well in advance how many books are to be kept so that we can reserve the space for the same. However, in the second method, we can keep every new book in the empty location found anywhere in the shelf; we need not reserve a location for the same.

The next point to be taken into consideration is the utilization of shelf space. In the first method, if the number of the books to be kept in a continuous space is not known in advance, this creates two problems. First, we reserve a continuous block say, m, of arbitrary size. In general, m denotes maximum size. Suppose the number of books to be kept is n, which is much smaller than m. Then, $(m - n)$ locations remain unused. The second problem is when the number of books n is greater than the reserved space m, we will not be able to accommodate the books in the continuous block.

The next aspect for comparison is with respect to the various operations on the data elements such as insertion and deletion of a book. In the first method, inserting a book at the i^{th} location needs a shifting of (i to n) books to the right side, each by one position. Similarly, taking out the i^{th} book from the shelf creates an empty space in the sequence of books. Hence, we need to shift ($i + 1$ to n) books to the left, each by one position.

The second method needs no shifting of data elements to insert or delete a data element. It only needs a few changes in the tags of the books, which are called as *links*. For the applications where the data elements to be stored are of varying sizes, that is, sequential representation, arrays are inadequate. This leads to an elegant solution, that is, linked organization.

6.2.1 Comparison of Sequential and Linked Organizations

Although linked lists are often used in computing, they are not simple to master. However, the flexibility and performance they offer is worth the pain of learning and using them. The brief features of sequential and linked organizations are described here.

Sequential organization The features of this organization are the following:

1. Successive elements of a list are stored a fixed distance apart.
2. It provides *static allocation*, which means, the space allocation done by a compiler once cannot be changed during execution, and the size has to be known in advance.
3. As individual objects are stored a fixed distance apart, we can access any element randomly.

4. Insertion and deletion of objects in between the list require a lot of data movement.
5. It is space inefficient for large objects with frequent insertions and deletions.
6. An element need not know/store and keep the address of its successive element.

Linked organization The features of this organization include the following:

1. Elements can be placed anywhere in the memory.
2. Dynamic allocation (size need not be known in advance), that is, space allocation as per need can be done during execution.
3. As objects are not placed in consecutive locations at a fixed distance apart, random access to elements is not possible.
4. Insertion and deletion of objects do not require any data shifting.
5. It is space efficient for large objects with frequent insertions and deletions.
6. Each element in general is a collection of data and a link. At least one link field is a must.
7. Every element keeps the address of its successor element in a link field.
8. The only burden is that we need additional space for the link field for each element. However, additional space is not a severe penalty when large objects are to be stored.
9. Linked organization needs the use of pointers and dynamic memory allocation.

A linked list can be implemented using arrays, dynamic memory management, and pointers. The second implementation requires dynamic memory management where one can allocate memory at run-time, that is, during the execution of a program. Linked lists are generally implemented using dynamic memory management. Each linked list has a head pointer that refers to the first node of the list and the data nodes storing data member(s). The linked list may have a *header node*, *tail pointer*, and so on.

6.2.2 Linked List Terminology

The following terms are commonly used in discussions about linked lists:

Header node A header node is a special node that is attached at the beginning of the linked list. This header node may contain special information (metadata) about the linked list as shown in Fig. 6.4.

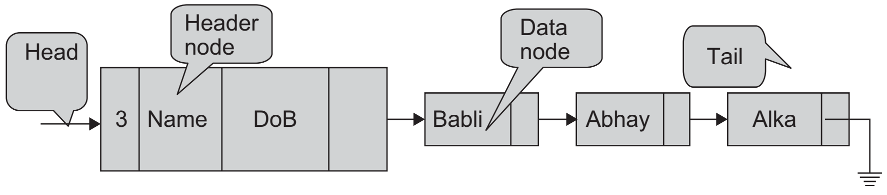


Fig. 6.4 Linked list with header node

This special information could be the total number of nodes in the list, date of creation, type, and so on. The header node may or may not be identical to the data nodes.

Data node The list contains data nodes that store the data members and link(s) to its predecessor (and/or successor).

Head pointer The variable (or handle), which represents the list, is simply a pointer to the node at the head of the list. A linked list must always have at least one pointer pointing to the first node (head) of the list. This pointer is necessary because it is the only way to access the further links in the list. This pointer is often called *head pointer*, because a linked list may contain a dummy node attached at the start position called the *header node*.

Tail pointer Similar to the head pointer that points to the first node of a linked list, we may have a pointer pointing to the last node of a linked list called the *tail pointer*.

6.2.3 Primitive Operations

The following are basic operations associated with the linked list as a data structure:

1. Creating an empty list
2. Inserting a node
3. Deleting a node
4. Traversing the list

Some more operations, which are based on the basic operations, are as follows:

5. Searching a node
6. Updating a node
7. Printing the node or list
8. Counting the length of the list
9. Reversing the list
10. Sorting the list using pointer manipulation
11. Concatenating two lists
12. Merging two sorted lists into a third sorted list

In addition, operations such as merging the second sorted list into the first sorted list and many more are possible by the use of these operations.

6.3 REALIZATION OF LINKED LISTS

In a linked organization, the data elements are not necessarily placed in continuous locations. The relationship between data elements is by means of a link. Along with each data element, the address of the next element is stored. Thus, the associated link with each data element to its successor is often referred to as a *pointer*. In general, a *node* is a collection of data and link(s). *Data* is a collection of one or more items. Each item in a node is called a *field*. A field contains either a data item or a link. Every node must contain at least one link field.

6.3.1 Realization of Linked List Using Arrays

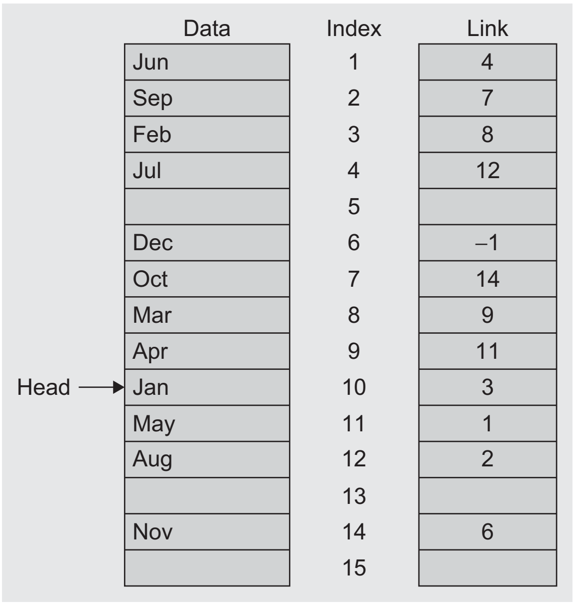
Let L be a set of names of months of the year.

$$L = \{\text{Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec}\}$$

Here, L is an ordered set. The linked organization of this list using arrays is shown in Fig. 6.5. The elements of the list are stored in the one-dimensional array, `Data`. The elements are not stored in the same order as in the set L. They are also not stored in a continuous block of locations. Note that the data elements are allowed to be stored anywhere in the array, in any order.

To maintain the sequence, the second array, `Link`, is added. The values in this array are the links to each successive element. Here, the list starts at the 10th location of the array. Let the variable `Head` denote the start of the list.

$$L = \{\text{Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec}\}$$



	Data	Index	Link
	Jun	1	4
	Sep	2	7
	Feb	3	8
	Jul	4	12
		5	
	Dec	6	–1
	Oct	7	14
	Mar	8	9
	Apr	9	11
Head →	Jan	10	3
	May	11	1
	Aug	12	2
		13	
	Nov	14	6
		15	

Fig. 6.5 Realization of linked list using 1D arrays

Here, `Head = 10` and `Data[Head] = Jan`.

Let us get the second element. The location where the second element is stored at is `Link[Head] = Link[10]`. Hence, `Data[Link[Head]] = Data[Link[10]] = Data[3] = Feb`.

Let us get the third data element through the second element. `Data[Link[3]] = Data[8] = Mar`, and so on.

Continuing in this manner, we can list all the members in the sequence. The link value of the last element is set to −1 to represent the end of the list. Figure 6.6 shows the same representation as in Fig. 6.5 but in a different manner.

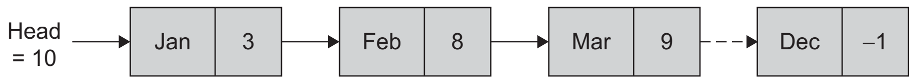


Fig. 6.6 Linked organization

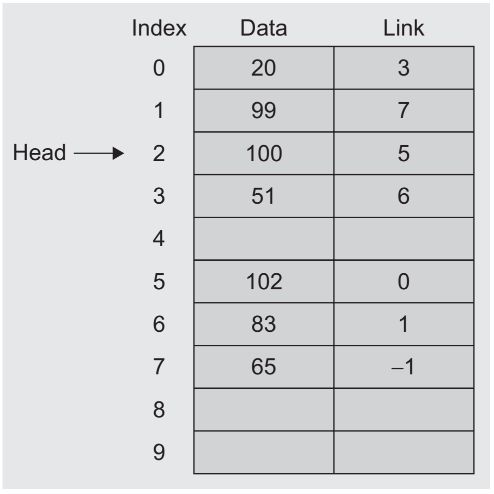
The unused locations are omitted and the list is drawn in the sequence of elements in the list *L*.

Figure 6.6 shows that the first element of the ordered list *L* is at the 10th position. The link value of the first element is 3. This indicates that the second element is at `Data[3]`. The link value of the second element is 8. This indicates that the third element is at `Data[8]`, and so on. Here, −1 is stored at `link[6]`, which indicates the end of the list.

Even though `data` and `link` are shown as two different arrays, they can be implemented using one 2D array as follows:

```
int Linked_List[max][2];
```

Figure 6.7 illustrates the realization of a linked list using a 2D array where *L* = {100, 102, 20, 51, 83, 99, 65}, `Max = 10` and `Head = 2`.



	Index	Data	Link
	0	20	3
	1	99	7
Head →	2	100	5
	3	51	6
	4		
	5	102	0
	6	83	1
	7	65	–1
	8		
	9		

Fig. 6.7 Realization of linked list using 2D arrays

6.3.2 Linked List Using Dynamic Memory Management

We learnt that unlike arrays, linked lists need not be stored in adjacent locations. Individual elements can be stored anywhere in the memory. Each data element is called a *node*. Each node contains at least two fields namely *data* and *link*. Every node holds a link to the next node in the list. During run-time (execution of a program), as per the need, a node is allocated (i.e., memory is allocated for a new node). In other words, a new node of the list will be created dynamically. We just remember the pointer to the list at the end, that is, pointer to the first node. In addition, the last node's link field can be set to 0 to mark the end of the list. The 0 here represents null. A linked list thus maintains the data elements in a logical order rather than in a physical order or in other words separates the physical view from the logical view.

Empty Linked List

An empty linked list is a head pointer with the value `Null`. An empty list is also called a *null list*. The length of a null or empty list is 0.

We should note the following facts while creating and inserting a node in a linked list:

1. The nodes may not actually reside in sequential locations.
2. The locations of nodes may change during different runs of program.
3. Therefore, when we write a program that works on lists, we should never look for a specific address except when we test for 0 (i.e., null).

We need the following for the implementation of linked list:

1. A means for allocating memory for a node that has at least one link field.
2. A mechanism to verify whether the allocation is successful.
3. A mechanism to release the allocated node and add to free pool of memory, as and when needed.

These tasks can be performed using the dynamic memory management functions in C++. To verify the memory allocation process, the address returned by the memory allocation function is compared with the value `Null`. A non-null address returned indicates that the process is successful. In C++, *new* and *delete* are the operators used for the same.

6.4 DYNAMIC MEMORY MANAGEMENT

Many languages permit a programmer to specify an array's size at run-time. Such languages have the ability to calculate and assign, during execution, the memory space required by the variables in a program. The process of allocating memory at run-time is known as *dynamic memory allocation*. Let us look at the memory allocation process shown in Fig. 6.8.

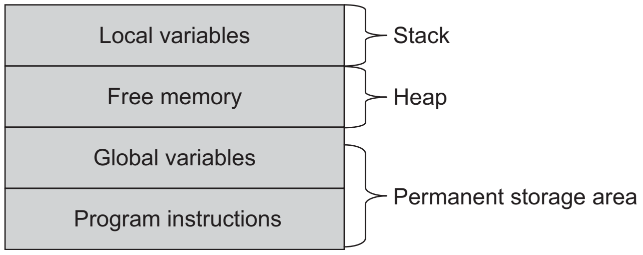


Fig. 6.8 Memory allocation process

The program instructions and global and static variables are stored in a region known as the *permanent storage area*, and the local variables are stored in another area called the *stack*. The memory space allocated between these two regions is available for dynamic allocation during the execution of the program. This free memory region is called the *heap*. The size of the heap keeps changing when a program is executed because of the creation and deletion of the variables that are local to the functions and blocks. Therefore, it is possible to encounter memory overflow during the dynamic allocation process. In such situations, the memory allocation functions as discussed in and returns a null pointer when it fails to locate enough memory requested Section 6.4.1.

6.4.1 Dynamic Memory Management in C++ with new and delete Operators

A special area of main memory, called the *heap*, is reserved for the dynamic variables. Any new dynamic variable created by a program consumes some memory in the heap. The heap is a pool of memory from which the `new` operator allocates memory. The memory allocated from the system heap using the `new` operator is de-allocated (released) using the `delete` operator. C++ enables programmers to control the allocation and de-allocation of memory in a program. The users can dynamically allocate and de-allocate memory for any built-in or user-defined data structure.

The new Operator

The `new` operator creates a new dynamic object of a specified type and returns a pointer that points to this new object (if it fails to create the desired object, it returns 0). In standard C++, a program that uses dynamic memory management should include a standard header `<new>`, which provides access to the standard version of the operator `new`. Consider the following declaration and statement:

```
MyType *ptr;
ptr = new MyType;
```

These statements create a new dynamic object of the type `MyType` of the proper size and return a pointer of the type specified to the right of the operator `new`, that is, `MyType` *.

Syntax

```
Pointer_Type_Variable = new Data_Type;
```

Note that `new` can be used to dynamically allocate any primitive type (such as `int` or `double`) or class type as follows:

1. `int *Number;`
   `Number = new int(20);`
2. `Time *timeptr; timeptr = new Time;`

3.
```
Date *B_Date_Ptr, *Today;
B_Date_ptr = new Date(20, 1, 1969);
Today = new Date(20, 1, 2005);
```

Here in example 3, `Date` is a class. If the type is a class with a constructor, the default constructor is called for the newly created dynamic variable. Initialization can be done by calling the appropriate constructor. If the program creates too many dynamic variables, it will consume all the memory in the heap. If this happens, any additional calls to `new` will fail. Hence, we should always check to see whether a call to the `new` operator is successful or not. With earlier C++ compliers, if all the memory in the heap has been used and `new` cannot create the requested dynamic variable, then it returns a special pointer named `Null`.

The Null Pointer

`Null` is a special constant pointer value that is used to give a value to a pointer variable that would not otherwise have a value. It can be assigned to a pointer variable of any type. In earlier compliers, a check was needed by the user for the successful operation of `new`. Newer compliers do not require such a check. Current compilers throw the exception `std::badalloc` and the program automatically aborts with an error message. We need no explicit check in the code. The users can 'catch' the exception.

The delete Operator

The object created exists till it is explicitly deleted, or till the function/program runs. To destroy a dynamically allocated variable/object and free the space occupied by the object, the `delete` operator is used.

```
delete ptr;
```

The `delete` operator eliminates a dynamic variable and returns the memory that it had occupied in the heap. The memory can now be reused to create new dynamic variables. After a call to delete, the value of the pointer variable, such as `ptr`, is undefined (except when the dynamic variable is an array). These undefined pointer variables are known as *dangling pointers*. One way to avoid dangling pointers is to set any such variable as null.

If we want to free a dynamically allocated array, the following is the syntax:

```
delete[] pointer_variable;
```

Such a statement will delete the entire array pointed to by `pointer_variable`. The square brackets tell C++ that a dynamic array variable is being eliminated, so the system checks the size of the array and removes that many indexed variables.

```
double* DoubleArrayPtr;
DoubleArrayPtr = new double[array_size];
```

We can use `delete` to release the dynamic array.

```
delete[] DoubleArrayPtr;
```

Similar to other operators, `new` and `delete` operators can be overloaded. Program Code 6.1 demonstrates dynamic variables, `new` and `delete` operators, and pointers.

PROGRAM CODE 6.1

```
#include<iostream.h>
int main()
{
   int *ptr1, *ptr2;
   ptr1 = new int;
   *ptr1 = 52;
   cout << "*ptr1 = " << *ptr1 << endl;
   cout << "*ptr2 = " << *ptr2 << endl;
   *ptr2 = 63;
   cout << "*ptr1 = " << *ptr1 << endl;
   cout << "*ptr2 = " << *ptr2 << endl;
   ptr1 = new int;
   *ptr1 = 98;
   cout << "*ptr1 = " << *ptr1 << endl;
   cout << "*ptr2 = " << *ptr2 << endl;
   return 0;
}
Output:
   *ptr1 = 52
   *ptr2 = Garbage
   *ptr1 = 52
   *ptr2 = 63
   *ptr1 = 98
   *ptr2 = 63
```

6.5 LINKED LIST ABSTRACT DATA TYPE

Although a linked list can be implemented in a variety of ways, the most flexible implementation is by using pointers. To implement the same in C++, we can view the entire linked list as an object of the class `LList`. Figure 6.9 shows an abstract representation of a linked list.

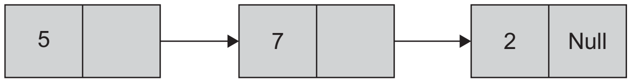


Fig. 6.9 Abstract representation of linked list

Each linked list has to have a special external link (or pointer), say, `Head`. We call it an external link because it is not stored in the list. We shall now extend the abstract notation to show the external link. Figure 6.10 illustrates the list with the external link, `Head`.

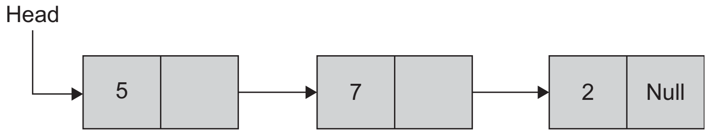


Fig. 6.10 Linked list with head pointer

To represent this linked list (Fig. 6.10), we consider it as an object of class `LList` whose definition is as follows:

```
class LList
{
   private:
      Node *Head;
   public:
      LList();
      ~LList();
      : ⎫
      : ⎬ member functions here
      : ⎭
};
```

The `LList` class has only one data member, the `Head` pointer, which points to the first node of the list, which is used to access the list. The member functions including the constructor and the destructor are used to process the list. Note that the `Head` is private and all other member functions are public. This is because particular nodes of the list are accessible to outside objects through pointers; the nodes are made inaccessible to outside objects by declaring `Head` private so that the information hiding principle is not really compromised. This is illustrated in Program Code 6.2.

PROGRAM CODE 6.2

```
class LList
{
   private:
      Node *Head;
      Node *Tail;      // optional data members
      int Size;
   public:
      LList()
      {
         Head = Tail = Null;
```

```
          Size = 0;
       }
       void Create();
       void Traverse();
       void Insert( int data, position);
       void Append(int data);
       void Delete(int position);
       void Reverse();
};
```

6.5.1 Data Structure of Node

Each node has data and link fields. The data field holds data element(s) and the link field(s) stores the address of its successor (and/or predecessor, if any). As the link field is a pointer to its successor, it should be a pointer variable, which should hold the address of its successor. The successor node is of the same type as that of the node itself. Hence, every node has one member, which points to a node of the same type as itself. As every node is a group of two (or more) data elements which are of different data types, they are logically grouped using the data type, *object*. The link field of a node is a pointer that references to a node of the same type as itself. Hence, we need a *self-referential object*.

The declaration of the data structure of a `node` is given as follows:

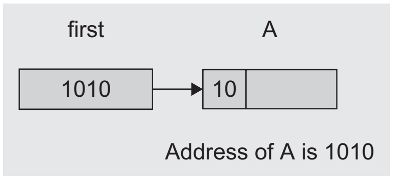
```
class Node
{
   public:
      int data;
      Node *link;
};

class List
{
   private:
      Node *Head;
      public:
      . ⎫
      . ⎬ member functions here
      . ⎭
};
```

Here, within the class, the statement `Node *link` defines the link field of a node. Here, `Node` is a data type of the pointer variable `link`.

Consider the following piece of code:

```
class Node
{
   public:
```

```
   int data;
   Node *link;
} *first, A;
first = &A;
A.data = 10;
A.link = Null;
```



Now, the statement

```
cout << first->data;
```

will print the output 10.

We discussed the node structure of the linked list. Let us now discuss the various operations on a linked list, illustrated in Program Code 6.3.

PROGRAM CODE 6.3

```
class Node
{
   public :
      int data;
      Node *link;
};

class Llist
{
   private:
      Node *Head,*Tail;
      void Recursive_Traverse(Node *tmp)
      {
         if(tmp == Null)
            return;
         cout << tmp->data << "\t";
         Recursive_Traverse(tmp->link);
      }
   public:
      Llist()
      {
         Head = Null;
      }
      void Create();
      void Display();
      Node* GetNode();
      void Append(Node* NewNode);
      void Insert_at_Pos( Node *NewNode, int position);
      void R_Traverse()
      {
         Recursive_Traverse(Head);
         cout << endl;
      }
      void DeleteNode(int del_position);
};

void Llist :: ~Llist()
{
   Node *Temp;
   while(Head != Null)
   {
      Temp = Head;
      Head = Head->link;
      delete Temp;
   }
}

void Llist :: Create()
{
   char ans;
   Node *NewNode;
   while(1)
   {
      cout << "Any more nodes to be added (Y/N)";
      cin >> ans;
      if(ans == 'n') break;
      NewNode = GetNode();
      Append(NewNode);
   }
}

void Llist :: Append(Node* NewNode)
{
   if(Head == Null)
   {
      Head = NewNode;
      Tail = NewNode;
   }
   else
   {
      Tail->link = NewNode;
      Tail = NewNode;
   }
}

Node* Llist :: GetNode()
{
   Node *Newnode;
   Newnode = new Node;
   cin >> Newnode->data;
   Newnode->link = Null;
   return(Newnode);
}

void Llist :: Display()
{
   Node *temp = Head;
   if(temp == Null)
      cout << "Empty List";
   else
   {
      while(temp != Null)
      {
         cout << temp->data << "\t";
        temp = temp->link;
      }
   }
   cout << endl;
}

void main()
{
   Llist L1;
   L1.Create();
   L1.Display();
}
```

6.5.2 Insertion of a Node

Depending on the type of list or need of the user, insertion can be made at the beginning, middle, or at the end of the list. If the list is an ordered list, the insertion should not affect the order and this may require inserting the data at proper locations so that the order is preserved. The information about where the node is to be inserted can be decided by searching through the list, obtaining the position, and then inserting the same.

Note that the symbol ⏚ shown in all figures in this chapter indicates the end of list marker representing null. We shall use the same notation throughout the book.

Insertion of a Node at a Middle Position

Assume that a node is to be inserted at some position other than the first position. Let `Prev` refer to the node after which `NewNode` node is to be inserted.

We need the following two steps:

```
NewNode->link = Prev->link;
Prev->link = NewNode;
```

The node `NewNode` is to be inserted between `Prev` and the successor of `Prev`. The link manipulation required to accomplish this is shown in Fig. 6.11 with dotted lines.

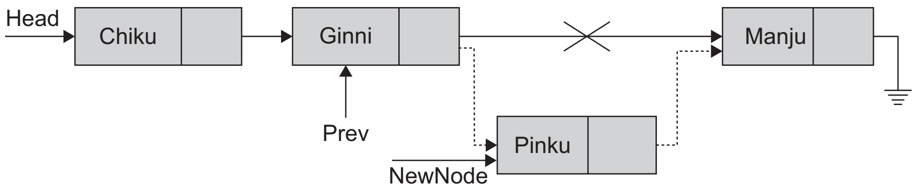


Fig. 6.11 Link manipulations for insertion of a node

The steps to perform the link manipulation are as follows:

1. `NewNode` is a node to be inserted after `Prev`. The node that is a successor of `Prev` will now become the successor of `NewNode`. Currently, `Prev->link` holds the pointer to the successor of `Prev`. Set the link field of the `NewNode` such that `Prev`'s successor node becomes the successor of `NewNode`.

   ```
   NewNode->link = Prev->link;
   ```

   In other words, `NewNode` becomes the predecessor of the node whose predecessor was `Prev`, because `NewNode` is to be placed in between `Prev` and its successor (Fig. 6.12).

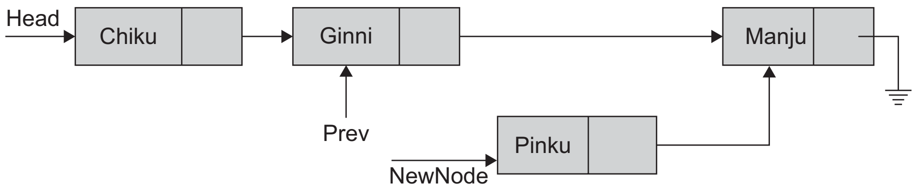


Fig. 6.12 Link manipulations for insertion of a node (Step 1)

2. Now, let us make `NewNode` the successor of `Prev`. This can be achieved by setting the link field of `Prev` to `NewNode` (Fig. 6.13).

```
Prev->link = NewNode;
```

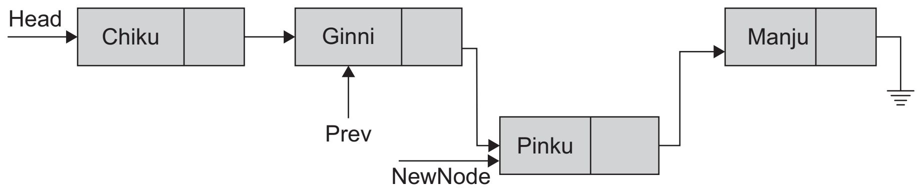


Fig. 6.13 Link manipulations for insertion of a node (Step 2)

Insertion of a Node at the First Position

Let us consider a situation when the node is to be inserted at the first position. As per the steps discussed for insertion of a node at the middle, we need `Prev`, which is a pointer to the node after which `NewNode` is to be added. To insert a node at the first position, there exists no `Prev` node.

The link manipulations needed to add a node at the first location is shown in Fig. 6.14 using dotted lines.

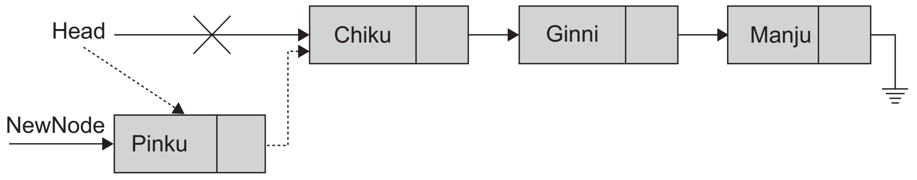


Fig. 6.14 Link manipulations for insertion of a node at the first position

`Head` is the pointer variable pointing to the starting node of the list. The insertion of `NewNode` at the first position should make `Head` point to `NewNode`, and in addition, the current node which is at the first position should become the second node of the list. Hence, the link field of `NewNode` should be set to point to the current first node, that is, the node pointed by the pointer `First`.

The following two steps will insert `NewNode` at the beginning of the linked list.

```
NewNode->link = Head;
Head = NewNode;
```

Figure 6.15 shows `NewNode` to be inserted in the list.

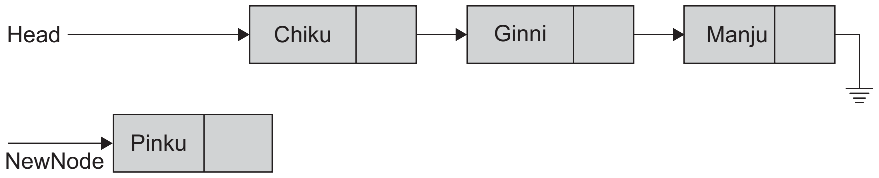


Fig. 6.15 Insertion of a node at the first position (Initial step)

Step 1 This step is represented in Fig. 6.16.

```
NewNode->link = Head;
```

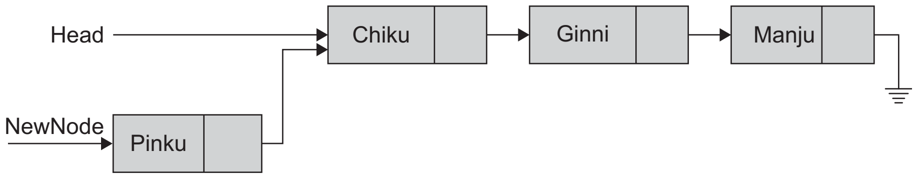


Fig. 6.16 Insertion of a node at the first position (Step 1)

Step 2 This step is represented in Fig. 6.17.

```
Head = NewNode;
```

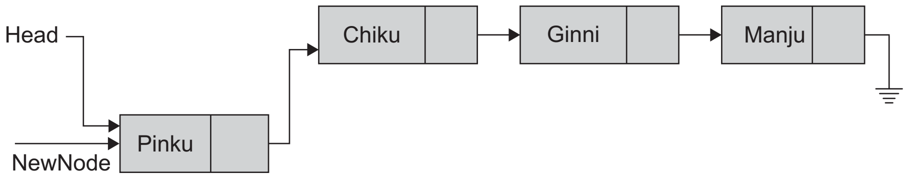


Fig. 6.17 Insertion of a node at the first position (Step 2)

Insertion of a Node at the End

The steps for inserting a node in the middle of a list also work for inserting a node at the end of the list. As the node is to be inserted after the last node, `Prev` is a pointer to the last node. Let the node to be inserted be `NewNode` as shown in Fig. 6.18.

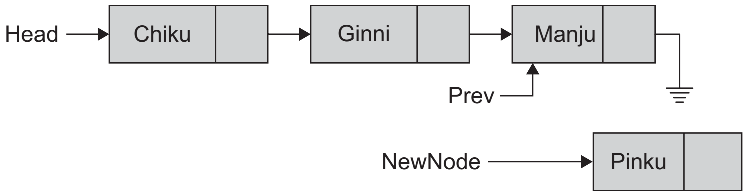


Fig. 6.18 Link manipulations for insertion of a node at the end of a list

1. `NewNode->link = Prev->link`
   As `Prev` is the last node, `Prev->link = Null`. Hence, this step can be replaced by the statement `NewNode->link = Null` if we know that `Prev` is the last node as in Fig. 6.19.

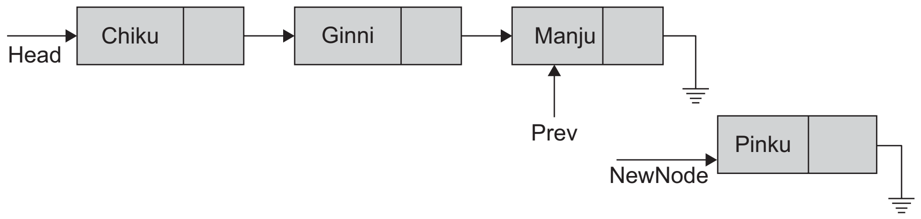


Fig. 6.19 Insertion of a node at the end of the list (Step 1)

2. `Prev->link = NewNode;`
   This will make the node `NewNode` the successor of `Prev.` This is shown in Fig. 6.20.

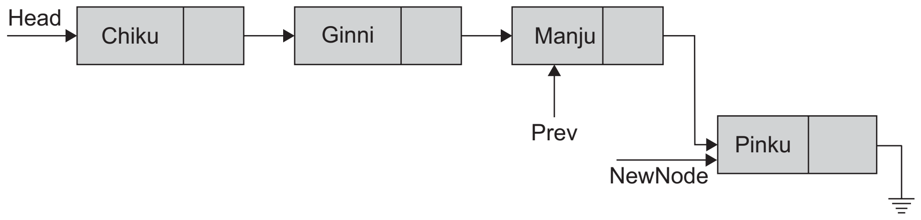


Fig. 6.20 Insertion of a node at the end of the list (Step 2)

This will insert the node `NewNode` at the last position, that is, make the node `NewNode` the last node of the list.

Generalized Insert Routine

Let us write a single insert routine which would insert a node at any random position in a list. Let us assume that the position *i* at which the node is to be inserted is known. We traverse the list till the $(i - 1)^{th}$ node to insert a new node at the i^{th} position. Now, let the $(i - 1)^{th}$ node be the previous node referenced by the pointer `Prev`. The function can be suitably modified when instead of the position, the node before or after which the new node is to be inserted is known. In that case, the proper location can be searched and then the node can be inserted. This is illustrated in Program Code 6.4.

PROGRAM CODE 6.4

```
void Llist :: Insert_at_Pos( Node *NewNode, int position)
{
   Node *temp = Head;
   int count = 1,flag = 1;
   if(position == 1)      // inserting at first position
   {
      NewNode->link = temp;
      Head = NewNode;      // update head
   }
   else
   {
      while(count != position - 1)
      {
         temp = temp->link;
         if(temp == Null)
         {
            flag = 0; break;
         }
         count ++;
      }
      if(flag == 1)
      {
         NewNode->link = temp->link;
         temp->link = NewNode;
      }
      else
         cout << "Position not found" << endl;
   }
}

void main()
{
   int pos;
   Node *NewNode;
   Llist L1;      // L1 is object of list.
   L1.Create();
   L1.Display();
   NewNode = L1.GetNode();
   cout << "Enter position where node is to be inserted"
<< endl;
   cin >> pos;
   L1.Insert_at_Pos(NewNode, pos);
   L1.Display();
}
```

Program Code 6.4 demonstrates the steps involved in inserting a node at a specified position in a linked list. A similar function can be written to insert a node before or after a specified node.

6.5.3 Linked List Traversal

List traversal is the basic operation where all elements in the list are processed sequentially, one by one. Processing could involve retrieving, searching, sorting, computing the length, and so on. List traversal requires a looping algorithm (Algorithm 6.1). To traverse the linked list, we have to start from the first node. We can access the first node through a pointer variable `Head`. Once we access the first node, through its link field, we can access the second node; through the second node's link field, we can access the third, and so on, as every node points to its successor till the last node.

ALGORITHM 6.1

```
1. Get the address of the first node, call it current; current = Head.
2. if current is Null,  goto step 6.
3. Process the data field of the current node (node pointed by current). Here, the process
   may include printing data, updating, and so on
4. Move to the next node–current  = current->link
   (Now current should point to the next node. The address of next node is in the link
   field of current. Hence, set current to the link field of current}
5. goto step 2
6. stop
```

Non-recursive Method

The non-recursive function for list traversal is shown in Program Code 6.5.

PROGRAM CODE 6.5

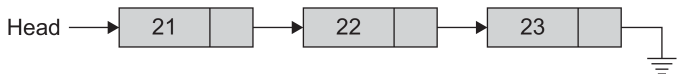
```
void Llist :: Traverse()
// just displaying the list members
{
   Node *temp = Head;
   if(temp == Null)
      cout << "Empty List";
   else
   {
      while(temp != Null)
      {
         cout << temp->data << "\t";
         temp = temp->link;
      }
   }
   cout << endl;
}
```

This function can be called by any function. The same function can also be used to print, search, update, and count length by adding a few statements.

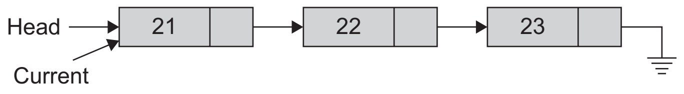
Here, the data element may not necessarily be just one. The node may hold more than one data element. Let us see output for list *L* pictorially.

$$L = \{21, 22, 23\}$$

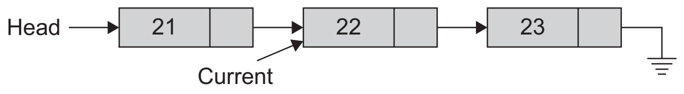
1. `Current = Head`



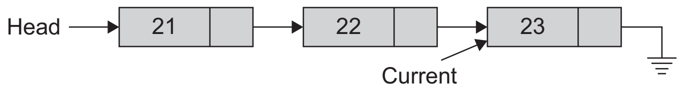
2. After execution of statement 1



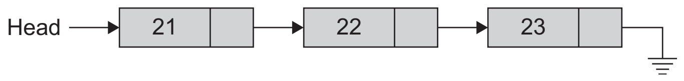
3. As `current != Null`, statements 3, 4, and 5 are executed.



4.



5.



Now `Current = Null` is true, while loop condition is false; hence stop.

```
Output
21
21   22
21   22   23
```

Recursive Traversal Method

Program Code 6.6 is the recursive code for traversing the linked list.

PROGRAM CODE 6.6

```
class Llist
{
   private:
      Node *Head, *Tail;
      void Recursive_Traverse(Node *tmp)
      {
         //Recursive traversal code
         if(tmp == Null)
            return;
         cout << tmp->data << "\t";
         Recursive_Traverse(tmp->link);
      }
   public:
      void Create();
      void Display();
      void R_Traverse()
      {
         Recursive_Traverse(Head);
         //call to recursive traversal
         cout << endl;
      }
};

void main()
{
   Llist L1;
   L1.Create();
   L1.R_Traverse();
}
Output:
21  22  23
```

Let us change the sequence of the last two statements in the recursive traverse function in Program Code 6.6.

```
void Llist :: Recursive_Traverse(Node *tmp)
{
   if(tmp == Null)
      return;
   Recursive_Traverse(tmp->link);
   cout << tmp->data << "\t";
}
```

What will be the output now? Will it be 21 22 23 or 23 22 21? Do verify.

6.5.4 Deletion of a Node

There may be nodes that are to be deleted from a list. Linear lists may very often require insertion and deletion of nodes. Linked lists are the most suitable data structures for this purpose. We discussed how to insert a node in a list. Let us learn about how to delete a node from a list.

Let us assume that the node to be deleted contains data *x*. We need the following steps to delete the same. Let $x = 13$ and let it be pointed to by the pointer `Curr`. To delete this node, the required link manipulations are shown in Fig. 6.21 with dotted lines.

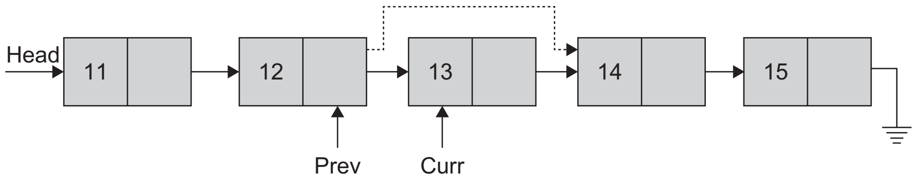


Fig. 6.21 Link manipulations for deletion of a node

To delete the node `Curr`, we need to modify the link between `Curr` and its previous node, and the link between `Curr` and its successor.

We need to modify them as shown in Fig. 6.21. The `Prev` is pointing to `Curr` as its current successor. As the `Curr` is to be deleted, the Prev's link should be modified such that it points to the successor of `Curr`. This makes the successor of `Curr` the successor of `Prev`. This deletes the node `Curr` from the linked list.

Note that we need the address of the node to be deleted as well as its predecessor to modify the links such that the node is deleted.

This can be achieved by the following steps shown in Algorithm 6.2.

ALGORITHM 6.2

```
1. Let both Curr and Prev be set to Head.
2. Traverse the list and search the node to be deleted.
3. Let Curr point to the node to be deleted and Prev be its previous node.
4. Modify the link field of Prev so that it skips Curr and points to its next.
   Prev->link = Curr->link
5. Free the memory allocated for the node Curr.
6. Stop
```

The node to be deleted can be at any position. It could be the first, middle, or last node.

Deleting the First Node

Deleting the first node is also referred to as deleting a header node. If the node at the first position is to be deleted, then we need to modify the pointer pointing to the first node (also called as the head pointer), say `Head`.

Deletion of the first node needs the link manipulations shown in Fig. 6.22 with dotted lines.

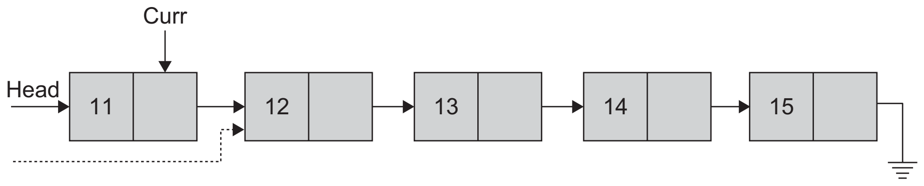


Fig. 6.22 Link manipulations for deletion of the first node

We should also release the first node using the `delete` operator. Hence, this can be accomplished in two steps as

1. Set another pointer to the first node before modifying `Head`, which is the pointer pointing to the first node. Set `Head` to point to the second node. This can be accomplished by the statements,
   ```
   Curr = Head;
   Head = Head->link;
   ```
2. Now, release the memory allocated for the first node.
   ```
   delete Curr;
   ```

These two statements will delete the first node, and `Head` will point to the second node so that the second node becomes the first node. Later, the memory allocated for the first node is freed.

Deleting a Middle Node

Let `curr` point to the node to be deleted, and `prev` be the predecessor of `curr`. Then, the following statements will delete the node `curr`.

```
prev->link = curr->link;
delete curr;
```

These two statements will also delete the last node of the list. Let us work out a function for the deletion of a node that may be at any position (Program Code 6.7).

PROGRAM CODE 6.7

```
void Llist :: DeleteNode(int pos)
{
   int count = 1, flag = 1;
   Node *curr, *temp;
```

```
    temp = Head;
    if(pos == 1)
    {
       Head = Head->link;
       delete temp;
    }
    else
    {
       while(count != pos - 1)
       {
          temp = temp->link;
          if(temp == Null)
          {
             flag = 0; break;
          }
          count++;
       }
       if(flag == 1)
       {
          curr = temp->link;
          temp->link = curr->link;
          delete curr;
       }
       else
          cout << "Position not found" << endl;
    }
}

void main()
{
    int pos,del_position;
    Llist L1;      // L1 is object of list.
    L1.Create();
    L1.Display();
    cout << "Enter position of the node to be deleted"
    << endl;
    cin >> del_position;
    L1.DeleteNode(del_position);
    L1.R_Traverse();
}
```

6.6 LINKED LIST VARIANTS

The basic idea of a linked list serves as the starting point for many useful variations. There are some variants of linked lists. In the following sections, we shall look at a few of them which have proven to be essential tools for computer scientists and software engineers.

6.6.1 Head Pointer and Header Node

A linked list must always have at least one pointer pointing to the first node of the list. This pointer is a must because otherwise, we have no way to access the linked list. This pointer is many times called a *head pointer*, because a linked list may contain a dummy node (exam) attached at the start position called *header node*. A *header node* is a special node that is attached at the front of the linked list. This header node may contain special information in data fields. The information could be the total number of nodes in the list.

Note that the header node may be of the same type as the node of the linked list or it may have a different data type with some special (additional) fields in it. A linked list with header node is called *header-linked list*.

Figure 6.23 is a header-linked list where the header node is of the same data type as that of the other nodes of the list.

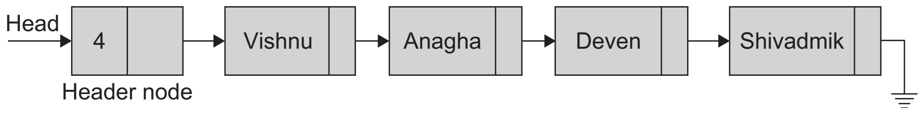


Fig. 6.23 Header-linked list

Here, the data field of the header node stores 4, which indicates that the linked list contains 4 records ahead. For example, suppose there is an application where the number of items in a list is often calculated. Usually, we need to traverse the whole list to count the length. However, if the current length is maintained in the header node, the information can be accessed easily. Figure 6.24 has a special header node whose data type is not the same as that of the other nodes of the list.

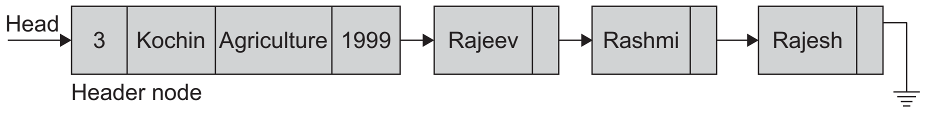


Fig. 6.24 Header-linked list with header node different from other nodes

In this list, the header node has some special fields such as length, city, department, year, and so on. Such a node will have the link field that points to the node of the linked list, as illustrated in Program Code 6.8.

PROGRAM CODE 6.8

```
class Head_Node
{
   public:
      int count;
      char City[15];
      char Dept[30];
      int Est_Year;
      .
      .
      .
      Node *link;
      // header node links to first node of the list
};

class Node
{
   public:
      emp_name[20];
      Node *link;
      // every node links to its successor of the same
      type
};
```

The most popular convention is to call the pointer that points to the first node of the list as head pointer no matter whether the header node is present or not.

6.6.2 Types of Linked List

We studied that in a linked list, every node must have at least one linked field. Thus, each node provides information about its predecessor and/or successor in the list. It may also have the knowledge about where the previous node lies in the memory. Thus, linked lists can be classified broadly as follows:

1. Singly linked list
2. Doubly linked list

The list and operations we discussed so far had only one link pointing to its successor and is called as singly linked list.

Singly Linked List

A linked list in which every node has one link field, to provide information about where the next node of the list is, is called as *singly linked list* (SLL). It has no knowledge about where the previous node lies in the memory. In SLL, we can traverse only in one direction. We have no way to go to the i^{th} node from $(i + 1)^{th}$ node, unless the list is traversed again from the first node (Fig. 6.25).

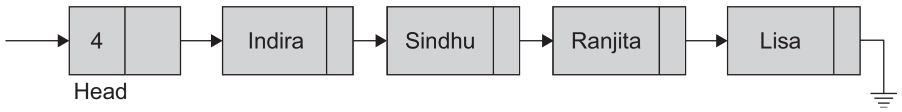


Fig. 6.25 Singly linked list

Often SLL is just referred to as a linked list.

Doubly Linked List

In a doubly linked list (DLL), each node has two link fields to store information about the one to the next and also about the one ahead of the node. Hence, each node has knowledge of its successor and also its predecessor. In DLL, from every node, the list can be traversed in both the directions (Fig. 6.26).

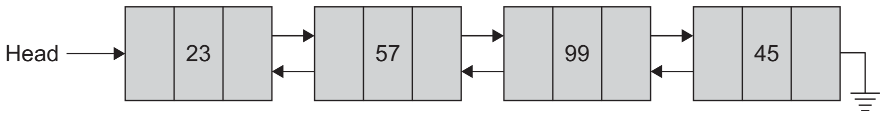


Fig. 6.26 Doubly linked list

Both SSL and DLL may or may not contain a header node. The one with a header node is explicitly mentioned in the title as a header-SLL and a header-DLL.These are also called as singly linked list with header node and doubly linked list with header node.

6.6.3 Linear and Circular Linked Lists

The other classification of linked lists based on their method of traversal is as follows:

1. Linear linked list
2. Circular linked list

Linear Linked List

The linked lists that we have seen so for are known as *linear linked lists*. All elements of such a linked list can be accessed by traversing a list from the first node of the list.

Circular Linked List

Although a linear linked list is a useful and popular data structure, it has some shortcomings. For example, consider an SLL. Given a pointer *A* to a node in a linear list, we cannot reach any of the nodes that precede the node to which *A* is pointing. This disadvantage can be overcome by making a small change. This change is without any additional data structure. The link field of the last node is set to `Null` in a linear list to mark the end of the list. This link field of the last node can be set to point to the first node rather than `Null`. Such a linked list is called a *circular linked list* (Fig. 6.27).

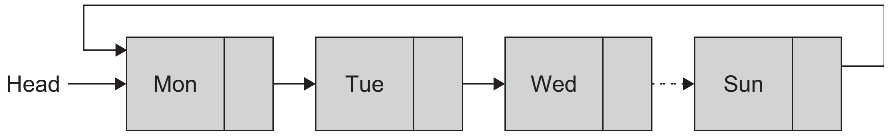


Fig. 6.27 Circular linked list

From any node in such a list, it is possible to reach any other node in the list. A circular list could be singly circular or doubly circular list and with or without a header node. Circular lists have many applications. We shall study those in further topics.

Linear lists are also called *non-circular* or *grounded lists*. The last node's link field of a linear list is set to `Null`. It is pictorially denoted using the 'ground' symbol used in electronic circuits. Let us discuss the DLL and its operations.

6.7 DOUBLY LINKED LIST

In SLL, each node provides information about where the next node is. It has no knowledge about where the previous node is. For example, if we are at the *i*th node in the list currently, then to access the (*i* – 1)th node or (*i* – 2)th node, we have to traverse the list right from the first node. In addition, it is not possible to delete the *i*th node given only a pointer to the *i*th node. It is also not possible to insert a node before the *i*th node given only a pointer to the *i*th node (there are other ways that are without link manipulations such as using data exchange).

For handling such difficulties, we can use DLLs where each node contains two links, one to its predecessor and other to its successor (Fig. 6.28).

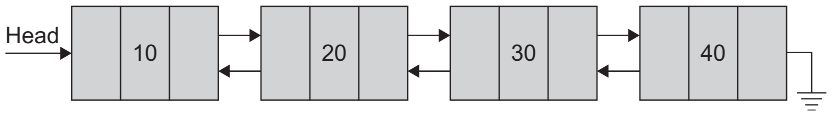


Fig. 6.28 Doubly linked list of four nodes

Each node of a DLL has three fields in general but must have at least two link fields (Fig. 6.29).

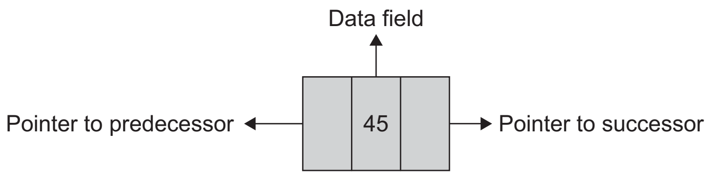


Fig. 6.29 Node structure of doubly linked list

Program Code 6.9 shows the class of a doubly linked list node.

PROGRAM CODE 6.9

```
class DLL_Node
{
   Public:
      int Data;
      DLL_Node *Prev, *Next;
      DLL_Node()
      {
         Prev = Next = Null;
      }
};
```

A DLL may either be linear or circular and it may or may not contain a header node. DLLs are also called *two-way lists*.

6.7.1 Creation of Doubly Linked List

Creation of DLL has the same procedure as that of SLL, as shown in Program Code 6.10. The only difference is that each node must be linked to both its predecessor and successor.

PROGRAM CODE 6.10

```
class DLL_Node
{
   public:
      int Data;
      DLL_Node *Prev, *Next;
      DLL_Node()
      {
         Prev = Next = Null;
      }
};

class DList
{
   private:
      DLL_Node  *Head, *Tail;
   public:
      DList()
      {
         Head = Tail = Null;
      }
      void Create();
      DLL_Node* GetNode();
      void Append(DLL_Node* NewNode);
      void Traverse();
      void DeleteNode(int val);
      void Delete_Pos(int pos);
      void Insert_Before(int val);
      void Insert_After(int val);
      void Insert_Pos(DLL_Node *NewNode, int pos);
};

DLL_Node* DList :: GetNode()
{
   DLL_Node *Newnode;
   Newnode = new DLL_Node;
   cout << "Enter Data";
   cin >> Newnode->Data;
   Newnode->Next = Newnode->Prev = Null;
   return(Newnode);
}

void DList :: Append(DLL_Node* NewNode)
{
   if(Head == Null)
   {
      Head = NewNode;
      Tail = NewNode;
   }
   else
   {
      Tail->Next = NewNode;      //Attach to last node
      NewNode->Prev = Tail;
      Tail = NewNode;
   }
}

void DList :: Create()
{
   char ans;
   DLL_Node *NewNode;
   while(1)
   {
      cout << "Any more nodes to be added (Y/N)";
      cin >> ans;
      if(ans == 'n') break;
      NewNode = GetNode();
      Append(NewNode);
   }
}

void DList :: Traverse()
{
   DLL_Node *Curr;
   Curr = Head;
   if(Curr == Null)
      cout << "The list is empty \n";
   else
      while(Curr != Null)
      {
         cout << Curr->Data << "\t";
         Curr = Curr->Next;
      }
      cout << endl;
}

void main()
{
   DList L2;
   L2.Create();
   L2.Traverse();
```

6.7.2 Deletion of a Node from a Doubly Linked List

Deleting from a DLL needs the deleted node's predecessor, if any, to be pointed to the deleted node's successor. In addition, the successor, if any, should be set to point to the predecessor node as shown in Fig. 6.30.

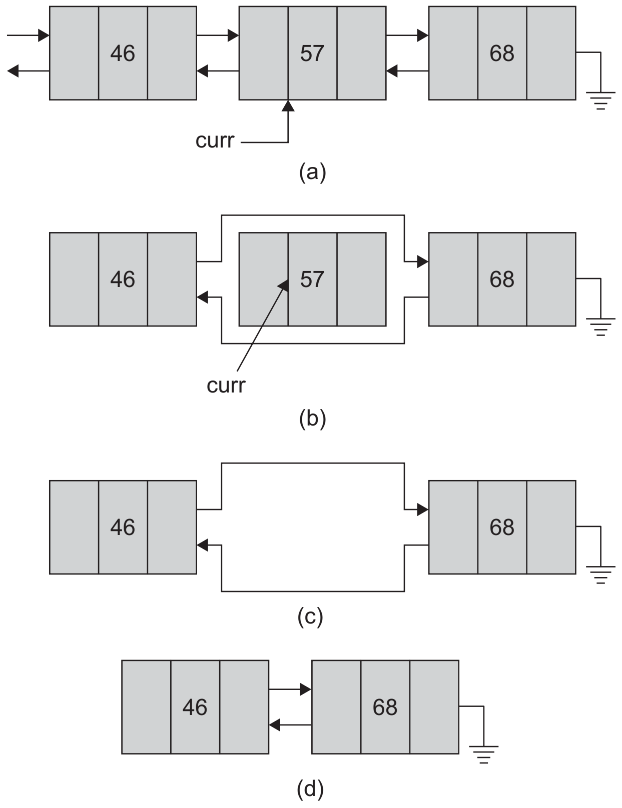


Fig. 6.30 Deletion node in doubly linked list (a) Links modified on deletion of node (b) Memory of the deleted node freed (c) Realignment of nodes (d) After node deletion

The core steps involved in this process are the following:

```
(curr->Prev)->Next = curr->Next;
(curr->Next)->Prev = curr->Prev;
delete curr;
```

The C++ code for the same is as shown in Program Code 6.11.

PROGRAM CODE 6.11

```
void DList :: DeleteNode(int val)
{
   DLL_Node *curr, *temp;
```

```
    curr = Head;
    while(curr!=Null)
    {
       if(curr->Data == val)
          break;
       // curr is pointing to the node to be deleted
       curr = curr->Next;
    }
    if(curr != Null)
    {
       if(curr == Head)      // delete first node
       {
          Head = Head->Next;
          Head->Prev = Null;
          delete curr;
       }
       else
       {
          if(temp == Tail)      // delete last node
          {
             Tail = temp->Prev;
             (temp->Prev)->Next = Null;
             delete temp;
          }
          else
          {
             (curr->Prev)->Next = curr->Next;
             (curr->Next)->Prev = curr->Prev;
             delete curr;
          }
       }
       if(Head == Null)
       {
          Tail = Null;
       }
    }
    else
       cout << "Node to be deleted is not found \n";
}
void DList :: Delete_Pos(int pos)
{
    DLL_Node *temp = Head;
```

```
   {
      if(pos == 1)      // delete header node
      {
         Head = Head->Next;
         Head->Prev = Null;
         delete temp;
      }
      else
      {
         while(count != pos)
         {
            temp = temp->Next;
            if(temp != Null)
               count++;
            else
            break;
         }
         if(count == pos)
         {
            if(temp == Tail)      // delete last node
            {
               Tail = temp->Prev;
               (temp->Prev)->Next = Null;
               delete temp;
            }
            else
            {
               (temp->Prev)->Next = temp->Next;
               (temp->Next)->Prev = temp->Prev;
               delete temp;
            }
         }
         else
            cout << "The node to be deleted is not
            found" << endl;
      }
   }
}

void main()
  int count = 1;
   if(Head != Null)
```

```
{
   int val,pos;
   DList L2;
   L2.Create();
   L2.Traverse();
   cout << "Enter Node Data to be deleted-->";
   cin >> val;
   L2.DeleteNode( val);
   L2.Traverse();
   cout << "Enter Node position to be deleted-->";
   cin >> pos;
   L2.Delete_Pos(pos);
   L2.Traverse();
}
```

6.7.3 Insertion of a Node in a Doubly Linked List

Now, let us discuss inserting a node in DLL. To insert a node, say `Current`, we have to modify four links as each node points to its predecessor as well as successor. Let us assume that the node `Current` is to be inserted in between the two nodes say `node1` and `node2`. We have to modify the following links:

`node1->Next`, `node2->Prev`, `Current->Prev`, and `Current->Next`

When the `Current` node is inserted in between `node1` and `node2`, `node1`'s successor node changes. Hence, we need to modify `node1->Next`. For the node `node2`, its predecessor changes. Therefore, we need to modify `node2->Prev` This is shown in Fig. 6.31.

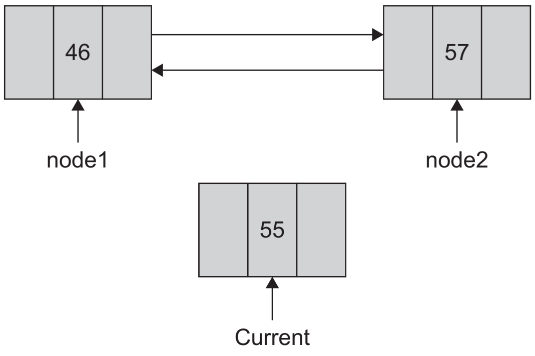


Fig. 6.31 Inserting a node current

`Current` is a new node to be inserted. We need to set both its predecessor and successor by setting the links as `Current->Prev` and `Current->Next`

After the insertion of `Current`, the resultant modified links should be shown as in Fig. 6.32.

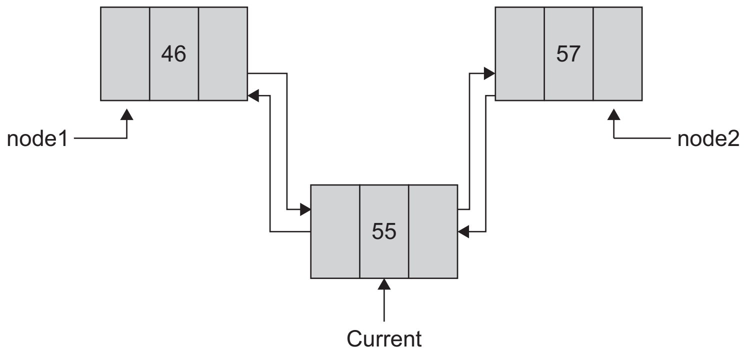


Fig. 6.32 Link modification for insertion of a node in a DLL

Hence, to modify the links, the statements would be

1. To modify `node1->Next` we use the operation
   ```
   node1->Next = Current;
   ```
2. To modify `node2->Prev` we use the operation
   ```
   node2->Prev = Current;
   ```
3. To set `curr->Next`, we use the operation
   ```
   Current->Next = node2;
   ```
4. To set `curr->Prev`, we use the operation
   ```
   Current->Prev = node1;
   ```

In brief, the statements to insert a node in between `node1` and `node2` are as follows:

```
node1->Next = Current;
node2->Prev = Current;
Current->Next = node2;
Current->Prev = node1;
```

These statements are with respect to Fig. 6.32, where we considered that the node is to be inserted in between `node1` and `node2`.

Though the new node is to be inserted between `node1` and `node2`, we need to know only about `node1`. The `node2` is the successor of `node1`, which can be accessed through `node1->Next`. Practically, the node can be inserted in DLL given only one node after which (or before which) the node is to be inserted.

Let us consider the insertion of a node given one node before or after which the node is to be inserted, say before `node2`. Then, the four statements could be

```
(node2->Prev)->Next = Current;
Current->Prev = node2->Prev;
```

```
Current->Next = node2;
node2->Prev = Current;
```

In brief, a node can be inserted anywhere in the DLL given a node after/before which it is to be inserted. The function can be written by passing to it either a node after/before which to insert or the position where to insert. One of the parameters would be the node to be inserted. Let us see how to insert a node at the first position. We are given a pointer to the DLL say `Head`.

We have to modify the links as shown in Fig. 6.33.

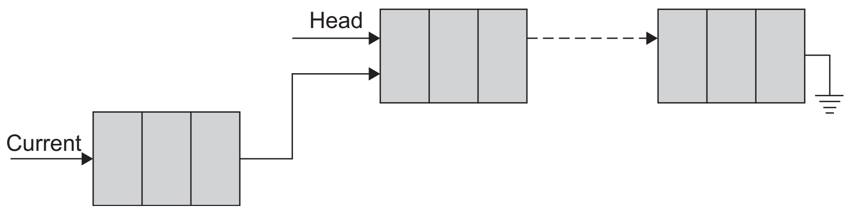


Fig. 6.33 Inserting a node before first node

This is represented by the following statements:

```
Current->Next = Head;
Head->Prev = Current;
Head = Current;
Current->Prev = Null;
```

6.7.4 Traversal of DLL

Given a head pointer to the DLL; traversal is the same as that of an SLL. The advantage of DLL over SLL is, given a pointer *P* pointing to any of the nodes of list, the list can be traversed only in one (forward) direction in SLL, whereas the list can be traversed in both (forward and backward) directions in DLL. Again, if we have a circular DLL, it has more advantages. It helps us keep the traversal procedure an unending one. Program Code 6.12 shows the traversal of a DLL.

PROGRAM CODE 6.12

```
void DList :: Insert_Pos(DLL_Node* NewNode, int pos)
{
   DLL_Node *temp = Head;
   int count = 1;
   if(Head == Null)
      Head = Tail = NewNode;
```

PROGRAM CODE 6.12

```
void DList :: Insert_Pos(DLL_Node* NewNode, int pos)
{
   DLL_Node *temp = Head;
   int count = 1;
   if(Head == Null)
      Head = Tail = NewNode;
   else if(pos == 1)      // insert before head
   {
      NewNode->Next = Head;
      Head->Prev = NewNode;
      Head = NewNode;
   }
   else
   {
      while(count != pos)
      {
         temp = temp->Next;
         if(temp != Null)
            count++;
         else
            break;
      }
      if(count == pos)
      {
         (temp->Prev)->Next = NewNode;
         NewNode->Prev = temp->Prev;
         temp->Prev = NewNode;
      }
      else
       cout << "The node position is not found" << endl;
   }
}
```

6.8 CIRCULAR LINKED LIST

The linked lists that we have seen so far are known as linear linked lists. All elements of such a linked list can be accessed by first setting up a pointer pointing to the first node in the list and then traversing the entire list. Although a linear linked list is a useful data structure, it has some drawbacks. For example, consider an SLL. Given a pointer `Current` to a node in an SLL, we cannot reach any of the nodes that precede the Current node (this is not the case with DLL as DLL has two, one backward and one forward, links). This drawback can be overcome by making a small change, and this change is without any additional data structure. In a singly linear list, the last nodes link field is set to `Null`. Instead of that, store the address of the first node of the list in that link field. This change will make the last node point to the first node of the list. Such a linked list is called *circular linked list* , shown in Fig. 6.34.

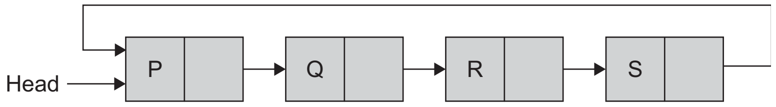


Fig. 6.34 Circular linked list

From any node in such a list, it is possible to reach to any other node in the list. We need not traverse the list again right from the first node. Circular linked list is used in many applications. Circular linked list is used to keep track of free space (unused nodes) in memory. In a circular list, traversal can be continued from `current` node. It helps us to keep the traversal procedure an unending one. The two primary applications of circular list is time slicing and memory management.

We can have a circular SLL or DLL. Both alternatives are possible. Similarly, circular linked lists could be with or without header nodes.

6.8.1 Singly Circular Linked List

Let us consider an SLL without a header node as shown in Fig. 6.35.

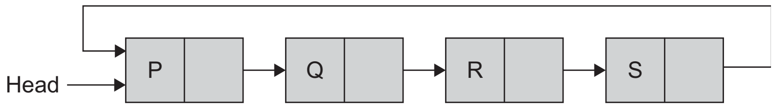


Fig. 6.35 Singly circular linked list

In a singly circular list, the pointer head points to the first node of the list. From the last node, we can access the first node. Remember that we cannot access the last node through the header node; we have access to only the first node. We need to traverse the whole list to reach to the last node. An elegant solution to this is set the pointer `Head` to point to the last node instead of the first node. This is illustrated in Fig. 6.36.

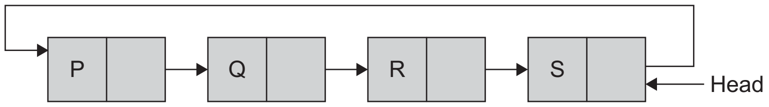


Fig. 6.36 Singly circular linked list

Now, through `Head` we have access to the last node, and it also (`Head->next`) gives us the address of the first node.

6.8.2 Circular Linked List with Header Node

Consider a circular list with a single node in the list (Fig. 6.37).

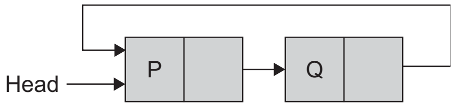


Fig. 6.37 Singly circular linked list with two nodes

Circular list with a single node has a problem of checking end of traversal as

```
(while(x->link != Head));
```

This would enter an infinite loop.

So, we can use a circular linked list with header node as shown in Fig. 6.38.

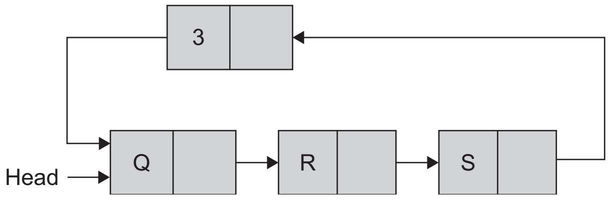


Fig. 6.38 Singly circular linked list with header node

The circular list with header node drawn in Fig. 6.38 can be redrawn as in Fig. 6.39.

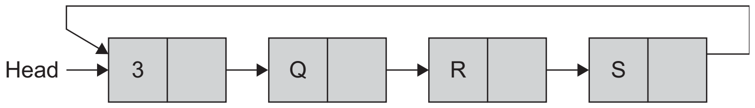


Fig. 6.39 Singly circular linked list with header node—representation 2

Suppose we want to insert a new node at the front of this list. We have to change the link field of the last node. In addition, we have to traverse the whole list to reach till the last node as the link field of the last node is also to be updated. Hence, it is convenient if the head pointer points to the last node rather than the header node, which is the first node of the list.

If the singly headed circular linked list has a head pointer as shown in Fig. 6.40, then a node can easily be inserted at the front and also at the rear of the list.

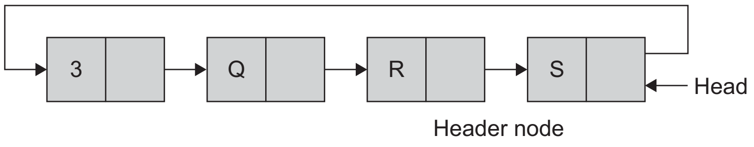


Fig. 6.40 Singly headed circular linked list with head pointing to last node

This procedure will have constant time complexity for both insert at front and at rear.

6.8.3 Doubly Circular Linked List

In doubly circular linked list, the last node's next link is set to the first node of the list and the first node's previous link is set to the last node of the list. This gives access to the last node directly from the first node (Fig. 6.41).

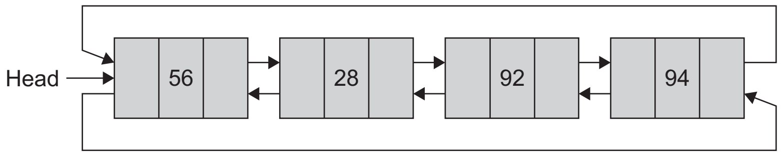


Fig. 6.41 Doubly circular list

Figure 6.41 represents the doubly circular linked list without a header node. Figure 6.42 is the doubly circular linked list with header node. Header node may store some relevant information of the list.

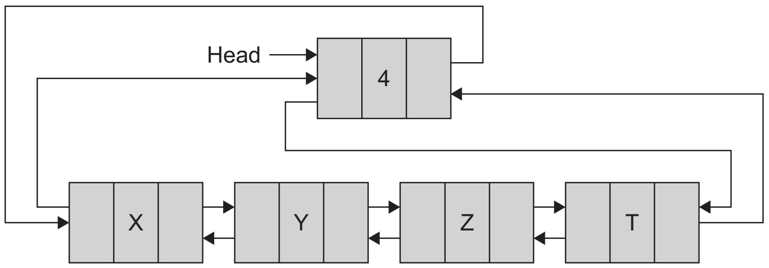


Fig. 6.42 Headed doubly circular list

The operations on circular linked list—`insert`, `delete`, `create` and `traverse`—follow the same method as that of linear list except for a few changes. We can redraw the circular list with header node as in Fig. 6.43.

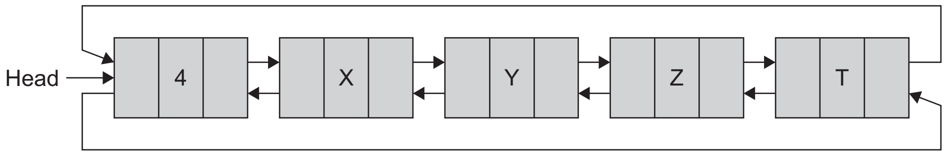


Fig. 6.43 Headed doubly circular list—representation 2

6.9 POLYNOMIAL MANIPULATIONS

We have already studied the representation and operations of polynomials using arrays. Let us now learn the representation of single variable polynomials using linked list. The manipulation of symbolic polynomials is a good application of list processing. Let the polynomial we want to represent using a linked list be $A(x)$. It is expanded as,

$$A(x) = k_1x^m + \dots + k_{n-1}x^2 + k_nx^1$$

where k_i is a non-zero coefficient with exponent m such that $m > m - 1 > \dots > 2 > 1 \geq 0$. A node of the linked list will represent each term. A node will have 3 fields, which represent the coefficient and exponent of a term and a pointer to the next term (Fig. 6.44).

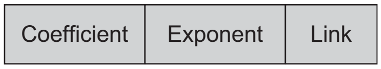


Fig. 6.44 Polynomial node

For instance, the polynomial, say $A = 6x^7 + 3x^5 + 4x^3 + 12$ would be stored as in Fig. 6.45.

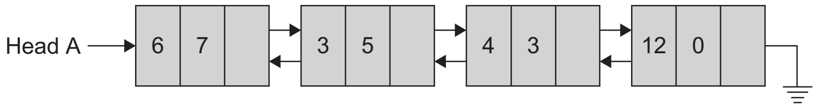


Fig. 6.45 Polynomial $A = 6x^7 + 3x^5 + 4x^3 + 12$

The polynomial $B = 8x^5 + 9x^4 - 2x^2 - 10$ would be stored as in Fig. 6.46.

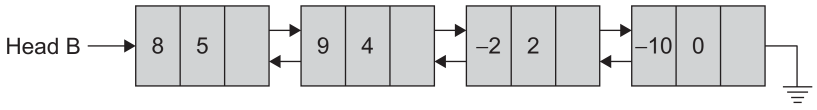


Fig. 6.46 Polynomial $B = 8x^5 + 9x^4 - 2x^2 - 10$

The function for the creation of a polynomial can be written as follows. Here, as the polynomial is stored in the SLL, the `create` procedure remains the same as that of the linked list we studied before. The difference is the data field we used earlier had single integer data fields, whereas here, we have two data fields and one linked field. The two data fields are the exponent and the coefficient of each term of the polynomial. Program Code 6.13 shows the creation of a polynomial.

PROGRAM CODE 6.13

```
class PolyNode
{
  public:
     int coef;
    int exp;
```

```
      PolyNode *link;
};

class Poly
{
   private:
      PolyNode *Head, *Tail;
   public:
      Poly() {Head = Tail = Null;}      // constructor
      void Create();
      PolyNode *GetNode();
      void Append(PolyNode* NewNode);
      void Display();
      Poly PolyMult(Poly A);
      Poly PolyAdd(Poly A);
      void Insert(PolyNode*);
      int Evaluate(int val );
};

void Poly :: Create()
{
   char ans;
  PolyNode *NewNode;
   while(1)
   {
      cout << "Any term to be added? (Y/N)\n";
     cin >> ans;
     if(ans == 'N'|| ans == 'n')
         break;
      NewNode = GetNode();
      if(Head == Null)
      {
         Head = NewNode;
         Tail = NewNode;
      }
      else
         Append(NewNode);
   }
}

void Poly :: Append(PolyNode* NewNode)
{
   if(Tail == Null)
      Head = Tail = NewNode;
   else
   {
      Tail->link = NewNode;
      Tail = NewNode;
   }
}

PolyNode* Poly :: GetNode()
{
   PolyNode *NewNode;
   NewNode = new PolyNode;
   if(NewNode == Null)
   {
      cout << "Error in memory allocation \ n";
      // exit(0);
   }
   cout << "Enter coefficient and exponent";
   cin >> NewNode->coef;
   cin >> NewNode->exp;
   NewNode->link = Null;
   return(NewNode);
}
```

6.9.1 Polynomial Evaluation

The function traversal of SLL can be used with a few modifications for polynomial evaluation. Given a value of *x*, we have to evaluate the polynomial as shown in Program Code 6.14.

PROGRAM CODE 6.14

```
int Poly :: Evaluate(int val)
{
   int j, result = 0,Power;
   PolyNode *tmp = Head;
   while(tmp != Null)
   {
      Power = 1;
      for(j = 1; j <= tmp->exp; j++)
         Power = Power * val;
      result += (tmp->coef) *Power;
      tmp = tmp->link;
   }
   return result;
}
```

6.9.2 Polynomial Addition

Let two polynomials *A* and *B* be

$$A = 4x^9 + 3x^6 + 5x^2 + 1$$
$$B = 3x^6 + x^2 - 2x$$

The polynomial *A* and *B* are to be added to yield the polynomial *C*. The assumption here is the two polynomials are stored in linked list with descending order of exponents.

The two polynomials *A* and *B* are stored in two linked lists with pointers `ptr1` and `ptr2` pointing to the first node of each polynomial, respectively. To add these two polynomials, let us use the paper–pencil method. Let us use these two pointers `ptr1` and `ptr2` to move along the terms of *A* and *B*.

Paper–Pencil Method

If the exponents of the two terms are equal, then their coefficients are added and a new term is created for the resultant polynomial *C*. If the exponent of the current term in *A* is less than the exponent of the current term of *B*, then a duplicate of the term in *B* is created and attached to *C*. The pointer `ptr2` is advanced to the next term. Similar action is taken on *A* if the exponent of the current term of *A* is greater than the exponent of the current term of *B*.

Each time a new node is generated, its exponent and coefficient fields are set accordingly, and the resultant term is attached to the end of the resultant term *C*. For polynomial *C*, we have `ptr3` to move along the resultant polynomial *C*. It always points to the newly appended term, that is, points to the last term of *C*. This avoids traversal of list *C* to append to the node each time. Attaching a node to a polynomial is the same as that of inserting a node at the end of a list. Only when the first node is added, the appropriate steps are carried out to initialize `ptr3`.

An algorithm to attach the term `NewTerm` to a polynomial, say *C*, with pointer `ptr3` is as follows:

```
1. if(c_ptr = Null)
      then c_ptr = NewTerm;
   else
      c_ptr->link = NewTerm;
   c_ptr = NewTerm;
2. stop
```

Polynomial Addition Algorithm

The following are the steps to add two polynomials *A* and *B* to yield the polynomial *C*.

```
1. Let A_ptr and B_ptr be pointers to polynomials A and B, respectively
2. Let C_ptr = Null, be a pointer to C
3. while(A_ptr != Null and B_ptr != Null)
   begin
      allocate node say NewTerm
      NewTerm->link = Null
      if(A_ptr->exponent = B_ptr->exponent)
      then
      begin
         NewTerm->exponent = A_ptr->exponent
         NewTerm->coefficient = A_ptr->coefficient + B_ptr->coefficient
         A_ptr = A_ptr->link
         B_ptr = B_ptr->link
      end
      else if(A_ptr->exponent > B_ptr->exponent)
      begin
         NewTerm->exponent = A_ptr->exponent
         NewTerm->coefficient = A_ptr->coefficient
         A_ptr = A_ptr->link
      end
      else
      begin
         NewTerm->exponent = B_ptr->exponent
         NewTerm->coefficient = B_ptr->coefficient
         B_ptr = B_ptr->link
      end
   attach NewTerm to C
4. while(A_ptr != Null)
   begin
      allocate new node
      NewTerm->link = Null
      NewTerm->exponent = A_ptr->exponent
      NewTerm->coefficient = A_ptr->coefficient
      A_ptr = A_ptr->link
      Attach NewTerm to C
end
5. while(B_ptr != Null)
   begin
      allocate new node
      NewTerm->link = Null
      NewTerm->exponent = B_ptr->exponent
      NewTerm->coefficient = B_ptr->coefficient
      B_ptr = B_ptr->link
      Attach NewTerm to C
   end
6. stop
```

Program Code 6.15 illustrates the code for polynomial addition.

PROGRAM CODE 6.15

```
poly Poly :: PolyAdd(Poly P2)
{
   PolyNode *Aptr = Head;
   PolyNode *Bptr = P2.Head;
```

```
Poly C;
PolyNode *NewTerm;
while(Aptr != Null && Bptr != Null)
{
   NewTerm = new PolyNode;
   NewTerm->link = Null;
   if(Aptr->exp == Bptr->exp)
   {
      NewTerm->coef = Aptr->coef + Bptr->coef;
      NewTerm->exp = Aptr->exp;
      C.Append(NewTerm);
      Aptr = Aptr->link;
      Bptr = Bptr->link;
   }
   else if(Aptr->exp > Bptr->exp)
   {
      NewTerm->coef = Aptr->coef;
      NewTerm->exp = Aptr->exp;
      C.Append(NewTerm);
      Aptr = Aptr -> link;
   }
   else
   {
      NewTerm->coef = Bptr->coef;
      NewTerm->exp = Bptr->exp;
      C.Append(NewTerm);
      Bptr = Bptr -> link;
   }
}       // end of while
while(Aptr != Null)
{
   NewTerm = new PolyNode;
   NewTerm->link = Null;
   NewTerm->coef = Aptr->coef;
   NewTerm->exp = Aptr->exp;
   C.Append(NewTerm);
   Aptr = Aptr->link;
}
while(Bptr != Null)
{
   NewTerm = new PolyNode;
```

```
        NewTerm->link = Null;
        NewTerm->coef = Bptr->coef;
        NewTerm->exp = Bptr->exp;
        C.Append(NewTerm);
        Bptr = Bptr->link;
    }
    return C;
}
```

6.9.3 Polynomial Multiplication

Let $A = 4x^9 + 3x^6 + 5x^3 + 1$ and $B = 3x^6 + x^2 - 2x$ be the two polynomials to be multiplied and the resultant polynomial be C. Let us revise the paper–pencil method. Polynomial A is multiplied by each term of B. We get n partial products if B has n terms in it. Finally, we add all these partial products to get the result.

This method generates partial products each of length m, where m is the length of the polynomial A. Such n partial products are generated, stored, and finally added to get the resultant polynomial. Here, m and n are input-dependent. Let us devise a better approach where we need not generate, store, and then add all partial products. Hence, a better solution is to pick up a term from the polynomial B and multiply it with each term of the polynomial A. One term of B and one term of A when multiplied yield one resultant term. This term can be immediately added to the resultant polynomial C, and this process is repeated.

To add a resultant term to polynomial C, it is compared with each term of the resultant polynomial C to insert the new term at the appropriate location in polynomial C. If the new term with equal exponent is found, then the term is added, else it is inserted in the resultant polynomial at the appropriate position. This process is repeated for each term of B with each term of A. The major steps can be listed as follows:

1. Let A and B be two polynomials.
2. Let the number of terms in A be M and number of terms in B be N.
3. Let C be the resultant polynomial to be computed as $C = A \times B$
4. Let us denote the i^{th} term of the polynomial B as tB_i. For each term tB_i of the polynomial B, repeat steps 5 to 7 where $i = 1$ to N.
5. Let us denote the j^{th} term of the polynomial A as tA_j. For each term of tA_j of the polynomial A, repeat steps 6 to 7 where $j = 1$ to M.
6. Multiply tA_j and tB_i. Let the new term be $tC_k = tA_j \times tB_i$.
7. Compare tCk with each term of the polynomial C. If a term with equal exponent is found, then add the new term tCk to that term of the polynomial C, else search for the appropriate position for the term tCk and insert the same in the polynomial C.
8. Stop.

Program Code 6.16 shows the multiplication of two polynomials.

PROGRAM CODE 6.16

```
poly Poly :: PolyMult(Poly P2)
{
   PolyNode *Aptr = Head;
   PolyNode *Bptr = P2.Head;
   Poly C;
   PolyNode *NewTerm;
   while(Bptr != Null)
   {
      Aptr = Head;
      while(Aptr != Null)
      {
         NewTerm = new PolyNode;
         NewTerm->link = Null;
         NewTerm->coef = Aptr->coef * Bptr->coef;
         NewTerm->exp = Aptr->exp + Bptr->exp;
         C.Insert(NewTerm);
         Aptr = Aptr->link;
         cout << "\n C \n";
         C.Display();
      }
      Bptr = Bptr->link;
   }
   return C;
}

void Poly :: Insert(PolyNode *NewTerm)
{
   PolyNode *prev = Head, *Curr = Head;
   if(Head == Null)       // if 1
      Head = Tail = NewTerm;
   else
   {
      Curr = Head;
      while(Curr != Null)
      {
         if(Curr->exp == NewTerm->exp)      //if 2
         {
            Curr->coef += NewTerm->coef;
            break;
         }
         else       // else2
         {
            if(Curr->exp < NewTerm->exp)      //if 3
            {
               if(Curr == Head)      //if 4
               {
                  NewTerm->link = Head;
                  Head = NewTerm;
                  break;
               }
               else      // else 4
               {
                  prev->link = NewTerm;
                  NewTerm->link = Curr;
                  break;
               }
            }      // end if 3
         }      // end else 2
        prev = Curr;
        Curr = Curr->link;
      }      // end of while
      if(Curr == Null)      // add at end
      {
         prev->link = NewTerm;
         Tail = NewTerm;
      }
   }      // end of else
}      // end of function

void main()
{
   Poly P1, P2, P3;
   P1.Create();
   P1.Display();
   P2.Create();
   P2.Display();
   P3 = P1.PolyMult(P2);
   P3.Display();
   getch();
}
```

6.10 REPRESENTATION OF SPARSE MATRIX USING LINKED LIST

We have studied the sparse matrix representation using arrays, which is a sequential allocation scheme. Representing a sparse matrix sequentially allows faster execution of matrix operations, and it is more storage efficient than linked allocation schemes. However, it has many shortcomings. The insertion and deletion of elements need the movement of many other elements. In applications with frequent insertions and deletions, a linked representation can be adopted. A basic node structure as shown in Fig. 6.47 is required to represent each matrix element.

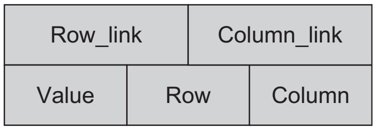


Fig. 6.47 Node structure for linked sparse matrix

The value, row, and column fields contain the value, row, and column indices, respectively, of one matrix element. The fields row_link and column_link are pointers to the next element in a circular list containing matrix elements for row and column, respectively.

Here, row_link points to the next node in the same row and column_link points to the next node in the same column. The principle is that all the nodes, particularly in a row (or column), are circularly linked with each other; each row and column contains a header node. Thus, for a sparse matrix of order $m \times n$, we have to maintain m header nodes for all rows and n header nodes for all columns, plus one extra node, the header node.

Header nodes for each row and column are used such that more efficient insertion and deletion algorithms can be implemented. The header node of each row contains 0 in the column field, and that of each column contains 0 in the row field. During the implementation in any programming language, 0 can be replaced by any other suitable value such as –1. *Header* is one additional header node that points to the starting address of the sparse matrix.

Header Nodes

1. Row field contains the number of rows.
2. Column field contains the total number of non-zero entries.
3. Row_link field contains pointer to the header node of the first row.
4. Column_link field contains pointer to the header node of the first column.

We may have arrays of pointers A Column[] and A Row[] that contain pointers to the header nodes of each column and row, respectively. In Fig. 6.48, both the header nodes pointing to the first header node of row and column and the array pointers are shown. The header node can provide the pointer to the header nodes linked list of both rows and columns, but it is through sequential traversal. However, arrays of pointers A Column and A Row can provide direct access to each row header node and column header node. Further element access will be obviously through sequential traversal. Hence, we may implement both or either of A Row/A Column and header node.

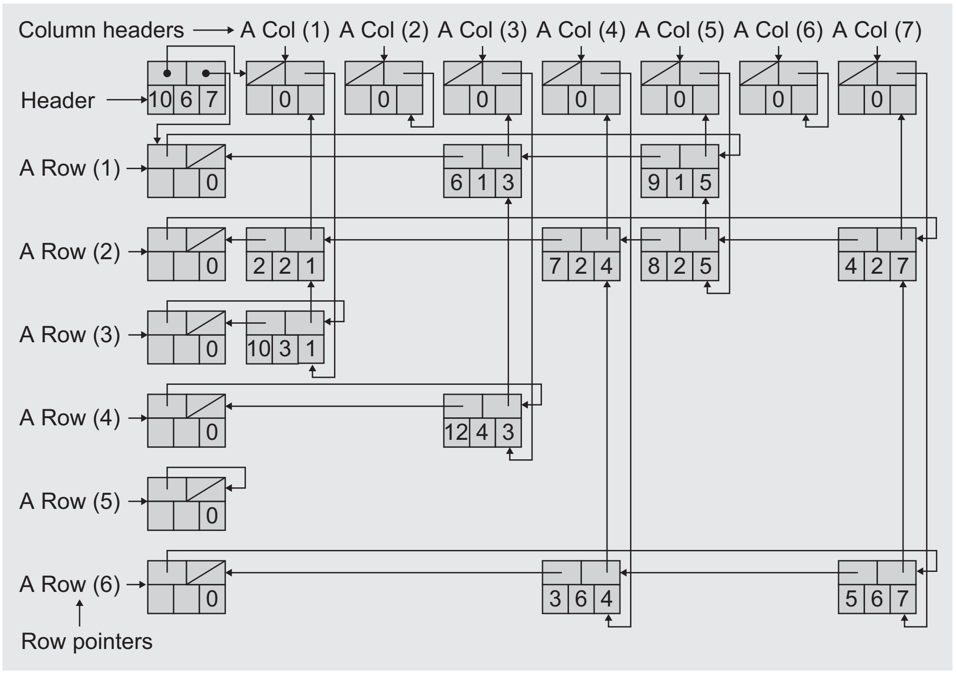


Fig. 6.48 Multilinked sparse matrix

6.11 LINKED STACK

In Chapter 3, we have implemented stacks using arrays. However, an array implementation has certain limitations. One of the limitations is that such a stack cannot grow or shrink dynamically. This drawback can be overcome by using linked implementation. We have studied linked list implementation of a linear list. Let us study the same linked list with restriction on addition and deletion of a node to use it as a stack. A stack implemented using a linked list is also called *linked stack*.

Each element of the stack will be represented as a node of the list. The addition and deletion of a node will be only at one end. The first node is considered to be at the top of the stack, and it will be pointed to by a pointer called `top`. The last node is the bottom of the stack, and its link field is set to `Null`. An empty stack will have `Top = Null`. A linked stack with elements (*X*, *Y*, *Z*) in order (*X* on top) may be represented as in Fig. 6.49.

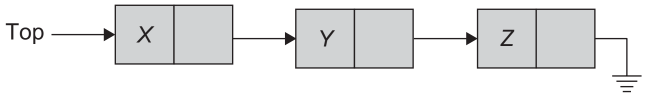


Fig. 6.49 Linked stack of elements (*X*, *Y*, Z)

Figure 6.49 shows a pictorial representation of the stack *S* containing three elements (*X*, *Y*, Z). Here, `top` is a pointer pointing to the top element of the stack. *X* is at the top of the stack and *Z* is at the bottom of the stack. SLL is suitable to implement stack using linked organization as we operate at one end of the list only.

6.11.1 Class for Linked Stack

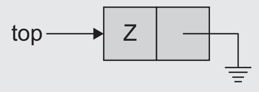
The node of the list structure is defined in Program Code 6.17.

PROGRAM CODE 6.17

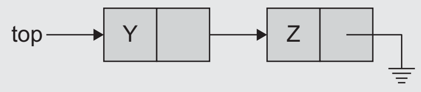
```
class Stack_Node
{
   public:
      int data;
      Stack_Node *link;
};

class Stack
{
   private:
      Stack_Node *Top;
      int Size;
      int IsEmpty();
   public:
      Stack()
      {
         Top = Null;
         Size = 0;
      }
      int GetTop();
      int Pop();
      void Push( int Element);
      int CurrSize();
};
```

Here, the stack can have any data type such as `int`, `char`, `float`, `struct`, and so on for the data field. The link field is a pointer pointing to the node below (next to) it. The `Top` serves the purpose of the variable associated with the data structure `stack` here. Similar to array implementation, an empty stack can be created by initializing the `Top`. This is going to hold the address of a node. It is a pointer rather than an integer as in contiguous stack. Hence to represent an empty stack, `Top` is initialized to `Null`. Every `insert` and `delete` of a node will be only at the end pointed by the pointer variable `Top`. Figure 6.50 represents the insertion of data in a linked stack considering the following sequence of instruction:

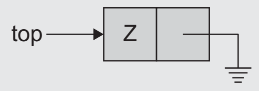
```
S.Create(), S.Push(Z), S.Push(Y), S.Pop(), S.Push(X)
1. Create S
   Top = Null
2. S.Push(Z)
```



```
3. S.Push(Y)
```



```
4. S.Pop()
```



```
5. S.Push(X)
```

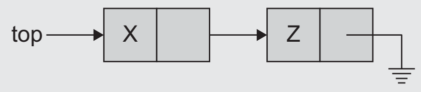


Fig. 6.50 Insertion of data in linked stack

Here, the stack grows and also shrinks at `Top`. Let us see the functions required to implement a stack using a linked list.

6.11.2 Operations on Linked Stack

The memory for each node is dynamically allocated on the heap. So when an item is pushed, a node for it is created, and when an item is popped, its node is freed (using `delete`). The only difference is that the capacity of a linked stack is generally greater than that of a contiguous stack since a linked stack will not become full until the dynamic memory is exhausted Program Code 6.18 shows operations on a linked stack. Figure 6.51 shows a logical view of the linked stack.

PROGRAM CODE 6.18

```
class Stack_Node
{
   public:
      int data;
      Stack_Node *link;
};

class Stack
{
   private:
      Stack_Node *Top;
      int Size;
      int IsEmpty();
   public:
      Stack()
      {
         Top = Null;
         Size = 0;
      }
      int GetTop();
      int Pop();
      void Push(int Element);
      int CurrSize();
};

int Stack :: IsEmpty()
{
   if(Top == Null)
      return 1;
   else
      return 0;
}

int Stack :: GetTop()
{
   if(!IsEmpty())
      return(Top->data);
}
```

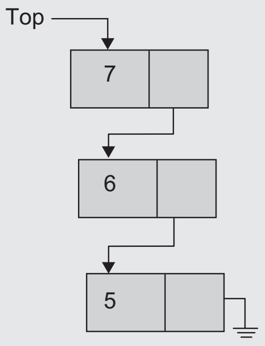


Fig. 6.51 Logical view of a linked stack

```
void Stack :: Push(int value)
{
   Stack_Node* NewNode;
   NewNode = new Stack_Node;
   NewNode->data = value;
   NewNode->link = Null;
   NewNode->link = Top;
   Top = NewNode;
}

int Stack :: Pop()
{
   Stack_Node* tmp = Top;
   int data = Top->data;
   if(!IsEmpty())
   {
      Top = Top->link;
      delete tmp;
      return(data);
   }
}
```

We have designed the functions for operations on stack, where the stack is implemented using linked organization. The `Top` is initialized to `Null` to indicate empty stack. The `Push()` function dynamically creates a new node. After creating a new node, the pointer variable `Top` should point to the newly added node in the stack.

```
void main()
{
   Stack S;
   S.Push(5);
   S.Push(6);
   cout << S.GetTop()<<endl;
   cout << S.Pop()<<endl;
   S.Push(7);
   cout << S.Pop()<<endl;
   cout << S.Pop()<<endl;
}
```

```
Output
   6
   6
   7
   5
```

6.12 LINKED QUEUE

We studied about how to represent queues using sequential organization in Chapter 5. Such a representation is efficient if we have a circular queue of fixed size. However, there are many drawbacks of implementing queues using arrays. The fixed sizes do not give flexibility to the user to dynamically exceed the maximum size. The declaration of arbitrarily maximum size leads to poor utilization of memory. In addition, the major drawback is the updating of `front` and `rear`. For correctness of the said implementation, the shifting of the queue to the left is necessary and to be done frequently. Here is a good solution to this problem which uses linked list. We need two pointers, `front` and `rear`. Figure 6.52 shows a linked queue which is easy to handle.

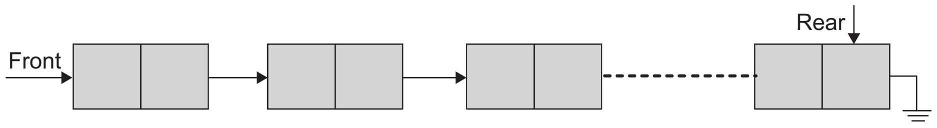


Fig. 6.52 The linked queue

Notice that the direction link for nodes is to facilitate easy insertion and deletion of nodes. One can easily add a node at the `rear` and delete a node from the `front`.

One of the node structures could be as in Program Code 6.19.

PROGRAM CODE 6.19

```
class Student
{
   public:
      int Roll_No;
      char Name[30];
      int Year;
      char Branch[8];
      Student *link;
};
class Queue
{
   Student *front, *rear;
   public:
      Queue()
      {
         front = rear = Null;
      }
};
```

Let us consider the following node structure for studying the linked queue and operations:

```
class QNode
{
   public:
      int data;
      QNode *link;
};

class Queue
{
   QNode *Front, *Rear;
   int IsEmpty();
   public:
      Queue()
      {
         Front = Rear = Null;
      }
      void Add( int Element);
      int Delete();
      int FrontElement();
      ~Queue();
};

int Queue :: IsEmpty()
{
   if(Front == Null)
      return 1;
   else
      return 0;
}
```

The queue element is declared using the class `QNode`. Each node contains the data declaration and the link pointer to the next element in the queue. This declaration creates an empty queue and initializes the pointers `front` and `rear` to `Null`. Here, `front` always points to the first node of queue and `rear` points to the last node of queue.

`Queue empty` condition is simply checked by comparing the `front` with `Null`. The function `IsEmptyQ` returns 1 (i.e., true) if the queue is empty and returns 0 (i.e., false), otherwise.

```
int Queue :: IsEmpty()
{
   if(Front == Null)
      return 1;
   else
      return 0;
}
```

`FrontElement()` returns the data element at the front of the queue. Here, the `front` is not updated. `FrontElement()` just reads what is at front.

```
int Queue :: GetFront()
{
   if(!IsEmpty())
      return(Front->data);
}
```

Note that if the `NewNode` is a node getting added in an empty queue, then along with the `rear`, the `front` should also be set to point to the newly added node, which is at the front of the queue. Hence, as both the `front` and the `rear` may get updated. Program Code 6.20 shows the addition of an element to a linked queue.

PROGRAM CODE 6.20

```
void Queue :: Add(int x)
{
   QNode *NewNode;
   NewNode = new QNode;
   NewNode->data = x;
   NewNode->link = Null;
   // if the new is a node getting added in empty queue
   //then front should be set so as to point to new
   if(Rear == Null)
   {
      Front = NewNode;
      Rear = NewNode;
   }
   else
   {
      Rear->link = NewNode;
      Rear = NewNode;
   }
}
```

`Delete()` function first verifies if there is any data element in the queue. If there is an element, `Delete()` gets and returns the data at the front of the queue to the caller function. Then, the `front` is set to point to the new queue front node, which is next to the node being deleted. If the last node is being deleted, then the deleted node's next pointer is guaranteed to be `Null`. Note that if the current deletion of a node results in `queue empty` state, then along with the `front`, the `rear` should also be set to `Null`.

```
int Queue :: Delete()
{
   int temp;
   QNode *current = Null;
   if(!IsEmpty())
   {
      temp = Front->data;
      current = Front;
      Front = Front->link;
      delete current;
      if(Front == Null)
         Rear = Null;
      return(temp);
   }
}

int Queue :: FrontElement()
{
   if(!IsEmpty())
      return(Front->data);
}

void main()
{
   Queue Q;
   Q.Add(11);
   Q.Add(12);
   Q.Add(13);
   cout << Q.Delete() << endl;
   Q.Add(14);
   cout << Q.Delete() << endl;
   cout << Q.Delete() << endl;
   cout << Q.Delete() << endl;
   Q.Add(15);
   Q.Add(16);
   cout << Q.Delete() << endl;
   cout << Q.Delete() << endl;
}
```

```
Output
11
12      // due to FrontElement
12
13
14
15
16
```

6.12.1 Erasing a Linked Queue

The following function in Program Code 6.21 traverses through the whole queue and also releases the memory allocated for each node. This task is handled by a destructor.

PROGRAM CODE 6.21

```
void Queue :: ~Queue()
{
   QNode *temp;
   while(Front! = Null)
   {
      temp = Front;
      Front = Front->link;
      delete temp;
   }
   Front = Rear = Null;
}
```

The linked queue may have the first node on a queue as a header node where the data field may hold some relevant information. In such a list, the first node, that is, the header node, is ignored (i.e. skipped) during `Delete()` operation. Similarly, the `Add()` function and `queue empty` condition will be changed accordingly.

6.13 GENERALIZED LINKED LIST

We have defined and represented linear list, which contains series of data elements, all of which had the same data type. In this topic, we shall extend the notion of list even further. We shall study generalized lists, which may be a list of lists.

Generalized lists are defined recursively as lists whose members may be single data elements or other generalized lists. Generalized lists are the most flexible and useful structures. We can use such lists to represent virtually all of the data structures. In addition, generalized lists provide the key data structure for several programming languages, such as LISP. Other languages, such as T and Miranda, include generalized lists and their operations as built-in capabilities. This widespread inclusion of generalized lists in many languages and environments attests the value of such lists in many applications.

6.13.1 Definition

A generalized list is a linear list (non-indexed) of zero or more data elements or generalized lists. In other words, a generalized list is a finite sequence of $n \geq 0$ elements, α_1, α_2, ... α_n, which we write as list $A = (\alpha_1, \alpha_2, ..., \alpha_n)$, where a_i is either an atom or the list. The elements of α_i, where $1 \leq i \leq n$, which are not atoms are said to be the sub-lists of the list. Here A is the name of generalized list and n is its length.

Thus, a generalized list may be made up of a number of components, some of which are data elements (atoms) and others are generalized lists.

Let us use the common terms being referred to with respect to the generalized list, Head and Tail. These terms refer to parts of the generalized list, that is, Head is the first component in the generalized list, and Tail is the list with the first component removed. If $n \leq 1$, then a_1 is the *head of list* whereas $(\alpha_2, \ldots \alpha_n)$ is the *tail of list*.

Some examples of generalized lists are the following:

1. $A = ()$	The empty (or null) list.
2. $B = (a, (b, c), d)$	List of three elements—the first element is a, the second element is list (b, c), and the third element is d.
3. $C = (B, B, A)$	List of length 3 with the first and the second element as list B and the third element as list A, which is a null list.
4. $D = (a, b, D)$	List of length 3 which is recursive as it includes itself as one of the elements. It can also be written as

$$D = (a, b, (a, b, (a, b, \ldots)\ldots$$

In example 2, A is a list made up of three components. The first component is an atom, the second component is a list made up of two atoms, and the third component is the atom d.

One of the better approaches to visualize the generalized lists is using a header node. In this approach, each generalized list has a header node labelled Head. Figure 6.53 shows the pictorial representation of list B.

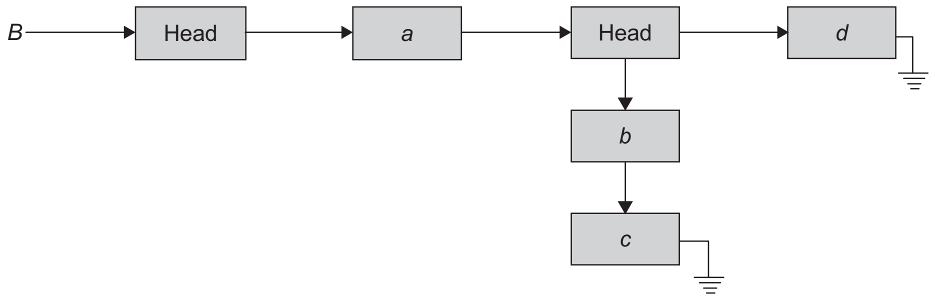


Fig. 6.53 Representation of $B = (a, (b, c), d)$

In example 3, the list C has three components: the first component is list B, the second component is again list B, and the third component is list A. This can be pictorially viewed as in Fig. 6.54.

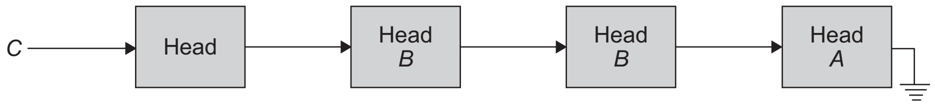


Fig. 6.54 Representation of $C = (B, B, A)$

The list *D* in example 4 can be viewed as Fig. 6.55.

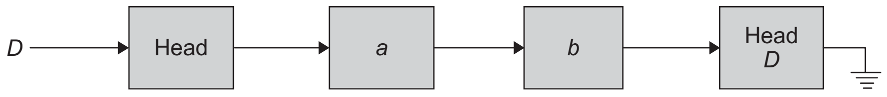


Fig. 6.55 Representation of *D* = (*a*, *b*, *D*)

These cases represent the categories of generalized lists in the order of implementation complexity. The lists could be one of the following categories:

1. Lists with no shared references—The components of one list are not members of any other list. In example 2, *B* is a list with no shared references.
2. Lists with shared references—The components of one list can be the members of another list. The logical interpretation of lists leads to two categories as the following:
   (a) *Static interpretation*—The current status of the referenced list is anticipated. The referenced list is copied into the referencing list.
   (b) *Dynamic interpretation*—The list itself is anticipated. Any future changes in the referenced list should be reflected in the referencing list.
3. Recursive list—A recursive list is the one that directly or indirectly references itself. Here *D* is a recursive list.

Here, the referenced list is the one that is a member of the other list, and the referencing list is the one being created.

6.13.2 Applications

The generalized list is the most flexible data structure that can be used for almost every data structure that is linear or non-linear. Let us represent the set and the polynomial using a generalized list to learn why generalized list is said to be the supreme data structure. For simplicity, we shall learn the implementation of generalized list with no shared references and no recursive lists. Such list has members that are not shared references, that is, members of list would not reference to other list and the list would not have the member that directly or indirectly refers itself. The popular implementation of such lists uses the linked list with a header node as in Figs. 6.56–6.58. Let us consider three lists *L*1, *L*2, and *L*3 as *L*1 = (*a*, *b*, *c*, *d*), *L*2 = (*a*, (*b*), (*c*,*d*), *e*), and *L*3 = (*a*, ((*b*)), *c*). The pictorial representation of these lists using header node is shown in Figs 6.56–6.58, respectively.

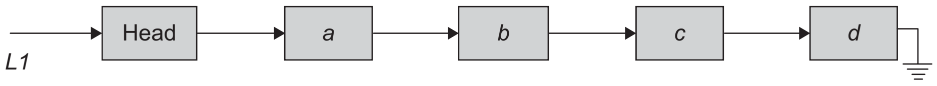


Fig. 6.56 GLL with header nodes for *L*1 = (*a*, *b*, *c*, *d*)

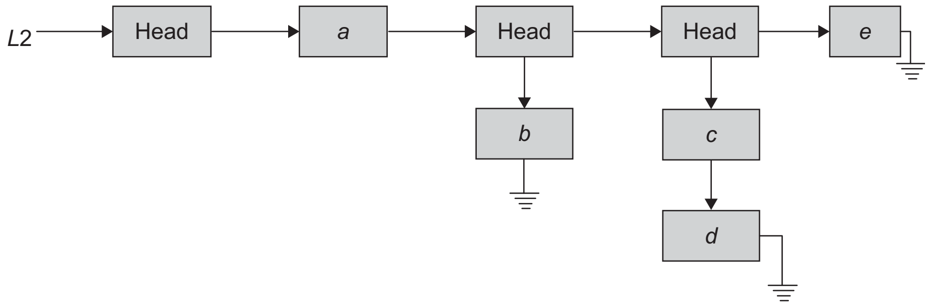


Fig. 6.57 GLL with header nodes for *L*2 = (*a*, (*b*), (*c*,*d*), *e*)

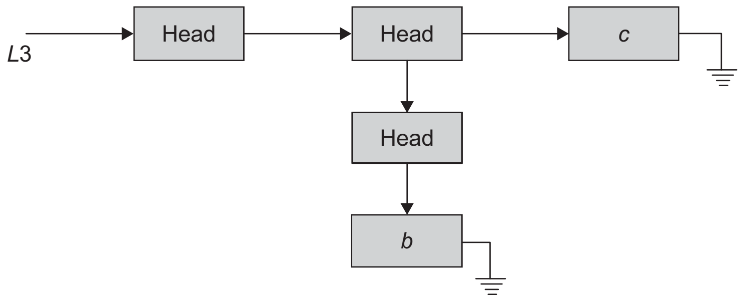


Fig. 6.58 GLL with header nodes for *L*3 = (*a*, ((*b*)), *c*)

Here, *L*1 has four members which are atoms, *L*2 too has four members but two of them are lists, and *L*3 has three members in it one of which is a list that has the list as a member again. The pictorial representation very clearly reveals it. Now, we need to reflect this data type and implement the code for the generalized linked list. We need to clearly distinguish between a member that is an atom and a member that is a list. In a linked representation of the generalized list, each node has fields as either

1. Data and `nLink`—the data field(s) would store data and the `nLink` field refers to the next member node which could be an atom or a list or
2. Header node that has two links `dlink` and `nLink`, where `dlink` is used to refer to the first node of the list member (which could be an atom or a list) and the `nLink` refers to the next member node (which could be an atom or a list) (Fig. 6.59).

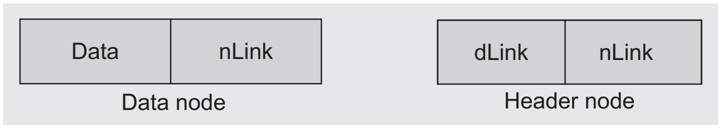


Fig. 6.59 Node structure of a generalized list

In general, it indicates that `nLink` is the field that holds the address of the next node that represents the member, which could be an atom or a header node of the list. The first field of each node is either `data` or `dLink` in case of Header node. Hence, we need to differentiate the first field clearly. One of the solutions is to add an additional filed, say `tag`, to indicate whether the first field is `data` or `dLink` that would clearly differentiate between the `data` and Header node (Fig. 6.60).

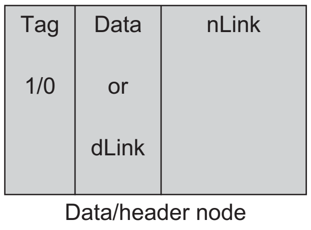


Fig. 6.60 Tag for differentiating between data node and header node

Here, when `Tag = 1,` it indicates that the second field is data, and `Tag = 0` indicates that it is the header node where the second field holds the address of the first node of the list member.

Further, we notice that the second field at any instant holds either `data` or `dLink` but not both. Two of these are of different data types. Hence, it would be efficient to share memory location. This leads to the use of union (also known as *variant records*) of programming language. The node structure now can be defined as in Program Code 6.22.

PROGRAM CODE 6.22

```
class GNode
{
   int Tag;
   union
   {
      <data type> Data;
      GNode *dLink;
   }
   GNode * nLink;
};

class GLL
{
   private:
      GNode * Head;
   public:
      GLL() {Head = Null;}
      void InsertNode();
      void PrintGLL();
};
```

Let us now see how we can use the generalized linked list to efficiently represent multi-variable polynomials and sets.

6.13.3 Representation of Polynomials Using Generalized Linked List

We have learned the use of linked list for the representation and operations of polynomial with a single variable. In practice, we often need to process a polynomial with more than one variable. Consider the following polynomial P with three variables x ,y, and z. Consider the two-variable polynomial Q of x and y.

$$Q(x, y) = 5x^4y^3 + 6x^6y^5 + 3x^5y^2 + xy$$

Now, similar to a single variable polynomial, we can represent this polynomial $Q(x,y)$ as a sequential organization with four fields: coefficient, Exp_X, Exp_Y, and nLink as in Fig. 6.61.

Coefficient	Exp_X	Exp_Y	nLink

Fig. 6.61 Two-variable polynomial

Similarly, for $P(x, y, z) = 9x^8y^2z + 4x^4y^3z^3 + x^6y^5z^4 + 8x^5y^2z + 7x^4y^6z + 4xyz + 3xz$ we can represent this polynomial $P(x, y, z)$ as a sequential organization with five fields: coefficient, Exp_X, Exp_Y, Exp_Z, and nLink as in Fig. 6.62.

Coefficient	Exp_X	Exp_Y	Exp_Z	nLink

Fig. 6.62 Three-variable polynomial

However, such representations denote that the polynomials in different number of variables would need a different number of fields. These nodes would have to differ in size depending on the number of variables. Such representations would lead to complexity in storage management for the polynomials with two, three, or more variables. We need to devise an efficient representation of multiple variable polynomials. An elegant solution is to go for a generalized list with fixed size nodes, which would represent the polynomial with any number of variables. Let us see how can we achieve it.

Consider the following polynomial:

$$P(x,y,z) = 5x^9y^4z^3 + 6x^7y^4z^3 + 3x^8y^2z^3 + 3x^5y^3z + 8x^3y^3z + 2y^2z$$

This polynomial can be rewritten as

$$((5x^9 + 6x^7)y^4 + (3x^8)y^2)z^3 + ((3x^5 + 8x^3)y^3 + 2y^2)z$$

We can write such a polynomial as one with a single variable whose each term node would be as in Fig. 6.63.

Tag	Coefficient or dLink	Variable	Exponent	nLink

Fig. 6.63 Representation of multi-variable polynomial as single variable polynomial

For example, the term as $9z^2$ would be represented as in Fig. 6.64.

Tag = 1	9	Z	2	nLink

Fig. 6.64 Representation of the term $9z^2$

The term as $(2y^3 + 3x^2)z^2$ would be represented as in Fig. 6.65.

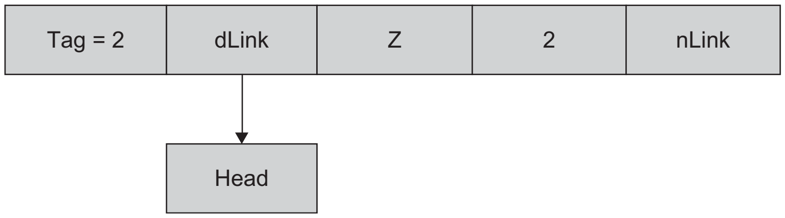


Fig. 6.65 Representation of the term $(2y^3 +3x^2)z^2$

We notice that for a polynomial of z with 10 terms, the third field of all nodes would be set to z for all term nodes. Can we avoid storing z for all terms of a polynomial? This is possible by storing it only once using the header node. For the header node, the fields `Tag`, `nLink`, and `dLink` are used, and the remaining two fields remain unused and it can be used for storing the variable.

Now, the node structure becomes as in Fig. 6.66.

Tag = 0/1/2	Variable, coefficient, or dLink	Exponent	nLink

Fig. 6.66 Representation of multi-variable polynomial as single variable polynomial

The multi-variable polynomial that is represented as a single variable polynomial whose coefficient is either constant or another polynomial, can now be very well stored using a linked list with such a node structure.

For example, the three-variable polynomial $P(x,y,z)$ can be represented factoring out a variable z, followed by the second variable y.

$$\text{Let } P(x, y, z) \text{ be } 5x^9y^4z^3 + 6x^7y^4z^3 + 3x^8y^2z^3 + 3x^5y^3z^2 + 8x^3y^3z^2 + 6y^2z$$

This polynomial can be rewritten as

$$(5x^9y^4 + 6x^7y^4 + 3x^8y^2)z^3 + (3x^5y^3 + 8x^3y^3)z^2 + 6y^2z$$

On observation of $P(x, y, z)$, we can notice that there are two terms in the variable z, $BZ^i + CZ^j + DZ^k$, where B, C, and D are polynomials themselves of variables x and y.

Now, the polynomial can further be rewritten as

$$((5x^9 + 6x^7)y^4 + (3x^8)y^2)z^3 + ((3x^5 + 8x^3)y^3)z^2 + ((6x^0)y^2\ )z.$$

Now, $C(x, y)$, $B(x, y)$, and $D(x, y)$ are of the form $Ey^m + \ldots + Fy^n$, where E and F are polynomials of x. Continuing in this way, we see that every polynomial consists of a variable plus coefficient and exponent pairs, and the coefficient itself could be a polynomial.

Each node would be one of the three—the header node (Fig. 6.67), data node with constant coefficient (Fig. 6.68), and the data node whose coefficient is a polynomial (Fig. 6.69). These can be pictorially viewed as follows:

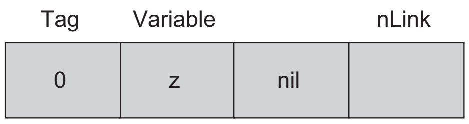


Fig. 6.67 Representation of header node

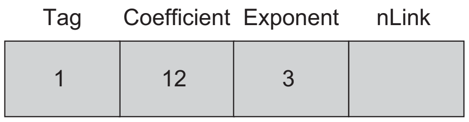


Fig. 6.68 Representation of data node with constant coefficient

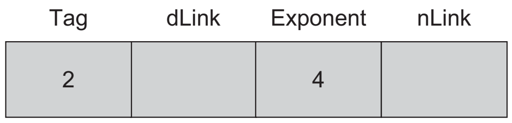


Fig. 6.69 Representation of data node with polynomial coefficient

Thus, every polynomial, regardless of the number of variables in it, can be represented using nodes. This is presented in Program Code 6.23:

PROGRAM CODE 6.23

```
class GLLPolyNode
{
    int Tag;
    union
    {
        char variable;
        float coefficient;
        GLLPolyNode *dLink;
    };
    int exponent;
    GLLPolyNode *nLink;
};

class GLLPoly
{
    private:
        GLLPolyNode *Head;
    public:
        GLLpoly() {Head = Null;}
        void InsertNode();
        void PrintGLL();
};
```

Pictorially, this can be viewed as in Fig. 6.70. Here, `dLink` is the downlink and `nLink` is the next link.

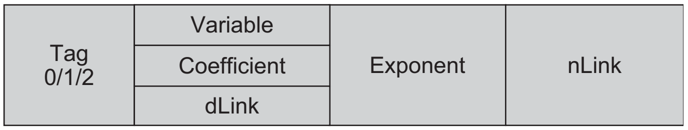


Fig. 6.70 The GLL node for polynomial

The following are a few examples to elucidate this concept:

1. $P(x,y) = 9x^2y^2 + 6xy^2 + y + x^2$
   This polynomial of two variables can be rewritten as

$$P = y^2(9x^2 + 6x) + y + x^2y^0$$

This is represented in Figs 6.71(a) and (b).

2. $Q = 8x^3y^3z^3 + 3x^3y^2z^3 + y^2z^2 + xy^2z^2 + 8x + 9y$

This can be rewritten as $z^3(x^3(8y^3 + 3y^2)) + z^2(y^2\ (1 + x)) + 8xz^0 + 9yz^0$. The pictorial representation of the Q is shown in Fig. 6.72.

Note that only three fields of the nodes are shown for convenience and the unused one is omitted.

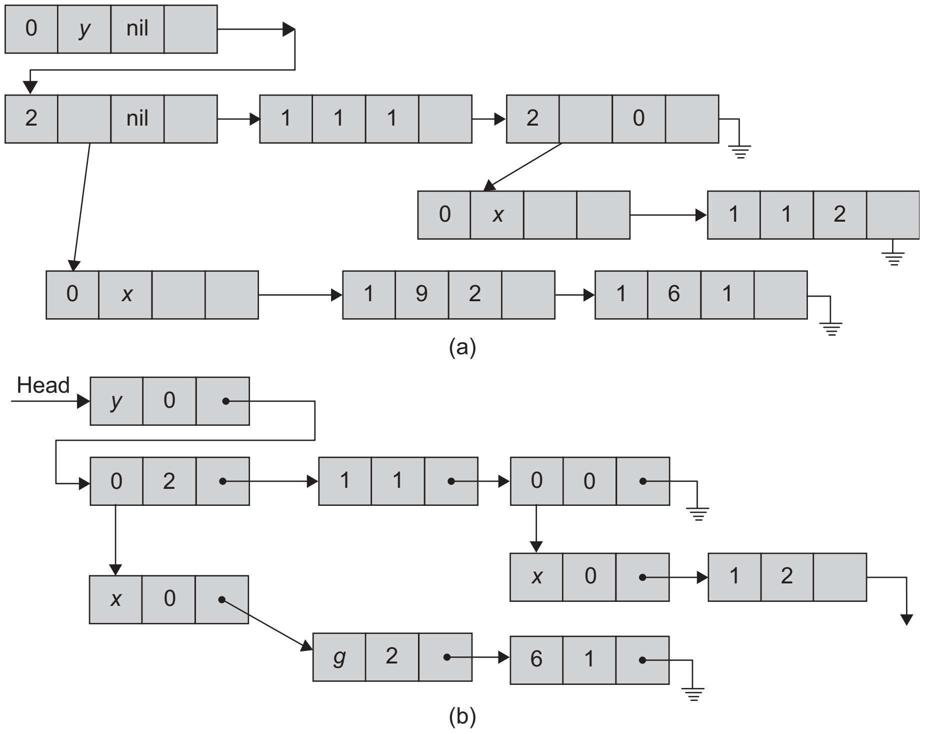


Fig. 6.71 Polynomial representation (a) The GLL for $9x^2y^2 + 6xy^2 + y + x^2$ (b) The GLL with three fields, omitting unused field

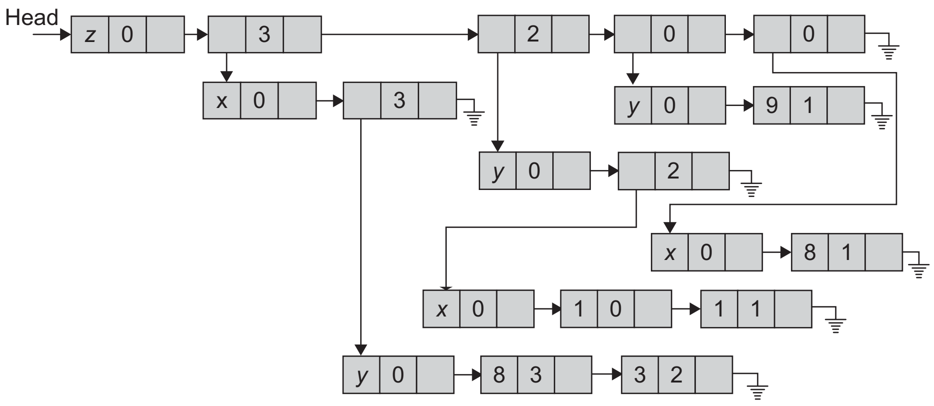


Fig. 6.72 The GLL for $8x^3y^3z^3 + 3x^3y^2z^3 + y^2z^2 + xy^2z^2 + 8x + 9y$

6.13.4 Representation of Sets Using Generalized Linked List

Let A be a set, $A = \{a, b, \{c, d, \{\ \}\}, \{e, f\}, g\}$. Here, A consists of elements that are either atoms or sets. Hence, we need a GLL node to convey whether the member of set is an atom or a set. The generalized list can be represented using the node structure as Fig. 6.73.

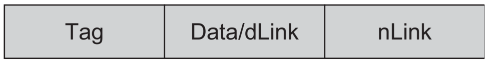


Fig. 6.73 Generalized list representation using node structure

Here, the `tag` field is set to 0 if the member is an atom and is set to 1 if it is another list. Accordingly, the second field would represent the data or downlink, respectively.

Figure 6.74 shows the GLL representation for the following sets:

1. $B = (a, (b, c), d)$
2. $C = (B, B, ())$
3. $D = (a, D)$

Figure 6.75 shows the GLL representation for the set $A = \{\{a, b\}, \{\{c, d\}, e\}\}$.

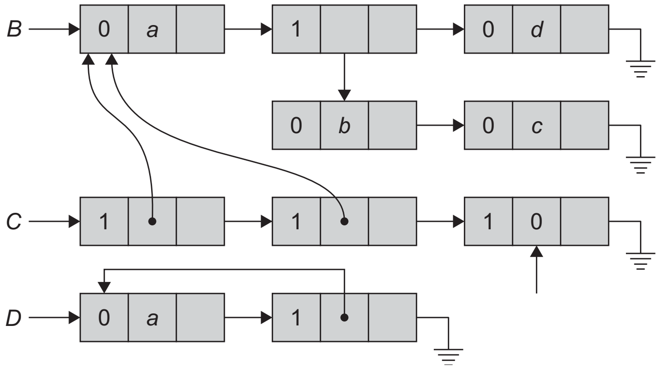


Fig. 6.74 The GLL representation for *B*, *C,* and *D*

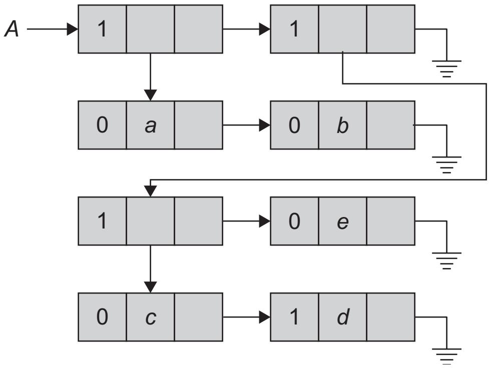


Fig. 6.75 The GLL representation for *A*

The set $X = \{L, M, \{N, \{O, P\}\}, \{Q, \{R, \{S, T\}\}, A, \{B, C\}\}$ is pictorially represented using a generalized linked list in Fig. 6.76.

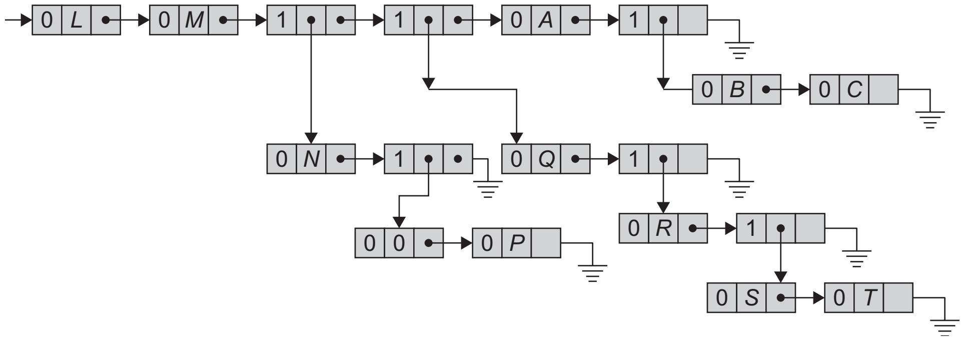


Fig. 6.76 The GLL for set *X*

We have represented the polynomials and sets using generalized linked list. Let us write the functions for traversing and printing the generalized linked list.

Printing Generalized Linked Lists

Program Code 6.24 gives the code for printing a GLL.

PROGRAM CODE **6.24**

```
void GLL :: PrintGLL()
{
   Stack S;
   GLLNode *curr = Null;
   S.Push(Head);
   curr = Head;
   while(1)
   {
      if(curr == Null)
      {
         if !S.IsEmpty()
            curr = S.Pop();
         if(curr→tag == 1)
            cout << curr→data;
         curr = curr→nlink;
         else if(curr→nlink != Null)
            S.push(curr→nlink)
         curr = curr→dlink
      }       //end if
   }       // end while
}       // end print
```

6.14 MORE ON LINKED LISTS

The function traversal of SLL can be used with a few modifications for polynomial evaluation. Given a value of x, we have to evaluate the polynomial as shown in Program Code 6.14.

6.14.1 Copying a Linked List

Consider the `Copy_List()` function, shown in Program Code 6.25, that takes a list and returns a complete copy of that list. One pointer can iterate over the original list in the usual way. Two other pointers can keep track of the new list: one head pointer and one tail pointer, which always points to the last node in the new list. The first node is done as a special case, and then the tail pointer is used in the standard way for the others.

PROGRAM CODE 6.25

```
Node *Llist :: CopyList()
{
   Node *current = Head;
   Node *newList = Null;
   Node *Tail = Null;
   while(current != Null)
   {
      if(newList == Null)
      {
         newList = new Node;
         newList->Data = current->Data;
         newList->link = Null;
         Tail = newList;
      }
      else
      {
         Tail->link = new Node;
         Tail = Tail->link;
         Tail->Data = current->Data;
         Tail->link = Null;
      }
      current = current->link;
   }
   return(newList);
}
```

16.4.2 Computing the Length of a Linked List

The `Length()` function, in Program Code 6.26, takes a linked list and computes the number of elements in the list.

`Length()` is a simple list function, but it demonstrates several concepts, which will be used later in more complex list functions.

PROGRAM CODE 6.26

```
int Llist :: Length()
{
   Node *current = Head;
   int count = 0;
   while(current != Null)
   {
      count++;
      current = current->link;
   }
   return count;
}
```

Calling Length()

Program Code 6.27 is a typical code that calls `Length()`. It first calls `create()` to make a list and store the head pointer in a local variable. It then calls `Length()` on the list and catches the `int` result in a local variable.

PROGRAM CODE 6.27

```
void LengthTest()
{
   Llist myList;
   mylist.Create();
   int len = mylist.Length();
}
```

6.14.3 Reversing Singly Linked List Without Temporary Storage

The procedure for reversing a singly linked list without temporary storage is illustrated by Program Code 6.28.

PROGRAM CODE 6.28

```
void Llist :: Reverse()
{
   Node *curr, *prev, *next;
   prev = Head;
   curr = Head->link;
   prev->link = Null
   while(temp != Null)
   {
      next = temp->link;
      temp->link = prev;
      prev = temp;
      temp = next;
   }
   head = prev;
}
```

6.14.4 Concatenating Two Linked Lists

Concatenation of two linked lists is illustrated by Program Code 6.29.

PROGRAM CODE 6.29

```
void Llist :: concatanate(Llist A)
{
   Node *X, *Y;
   X = Head;
   Y = A.Head;
   while(X->link != Null)
   {
      X = X->link;
   }
   X->link = Y;
   Head = X;
}

//A call to concatenate:
{
   Llist L1, L2;
   L1.Create(); L2.Create();
   L1.Concatanate(L2);
}
```

Here, *X* and *Y* are concatenated, and *X* is the pointer to the first node of the resultant list.

6.14.5 Erasing the Linked List

The procedure for erasing a linked list and returning all nodes to the free pool of memory is illustrated by Program Code 6.30.

PROGRAM CODE 6.30

```
void Llist :: ~Llist()
{
   Node *temp;
   while(Head != Null)
   {
      temp = Head;
     Head = Head->link;
      delete temp;
   }
}
```

6.15 APPLICATION OF LINKED LIST—GARBAGE COLLECTION

Memory is just an array of words. After a series of memory allocations and de-allocations, there are blocks of free memory scattered throughout the available heap space. To be able to reuse this memory, the memory allocator will usually link the freed blocks together in a free list by writing pointers to the next free block in the block itself. An external free list pointer points to the first block in the free list. When a new block of memory is requested, the allocator will generally scan the free list looking for a free block of suitable size and delete it from the free list (relinking the free list around the deleted block).

One of the components of an operating system is the memory management module. This module maintains a list, which consists of unused memory cells. This list very often requires the operations to be performed on the list, such as `insert`, `delete`, and `search` (traversal). Such a list implemented as a linked organization is called the *list of available space*, *free storage list*, or the *free pool*.

Suppose some memory block is freed by the program. The space available can be used for future use. One way to do so is to add the blocks in the free pool. For good memory utilization, the operating system periodically collects all the free blocks and inserts into the free pool. Any technique that does this collection is called *garbage collection*. Garbage collection usually takes place in two phases. First, the process runs through all the lists, tagging those cells, which are currently in use. In the second phase, the process runs through memory, collecting all untagged blocks and inserting the same in free pool. In general, garbage collection takes place when either overflow or underflow occurs. In addition, when the CPU is idle, the garbage collection starts. Note that the garbage collection is invisible to the programmer.

Overflow Sometimes, a new data node is to be inserted into data structure, but there is no available space, that is, free pool is empty. This situation is called *overflow*.