CHAPTER 2

Fundamentals of Data Structures

2.1 INTRODUCTION

A data structure is a particular way of organizing data so that the computer can efficiently use and manipulate the data. Data structures are also useful to implement abstract data types which specify the needed operations on the data. Different kinds of applications need the data to be stored in a particular way. Data structures provide a means to provide large amounts of data for users. Efficient data structures are the key for efficient algorithms. The implementation of data structures usually requires writing a set of procedures that create and manipulate instances of such structures.

Examples:

Array: It is a set of elements in a specific order.

Linked list: A linear collection of data elements of any type, called nodes, where each node has a value and a pointer to the next node.

Structure: An aggregate data structure which may contain different types of elements.

Low-level languages lack basic support for data structures. However, most of the high-level languages like C, C++, have special syntax and other built-in support for data structures. Modern languages feature some sort of library mechanism that allows data structure implementation to be reused by different programs.

2.2 ABSTRACT DATA TYPES

In computer science an abstract data type (ADT) is a mathematical model where a data type is defined by its semantics (behavior). From the point of view of the user of the data, some possible operations on this data type are defined. A data structure is more concerned about the representation of data from the point of view of the implementer.

Formally, an ADT may be defined as a class of objects whose logical behavior is defined by a set of values and a set of operations. ADT is a theoretical concept in computer science and does not correspond to any specific feature of a language. For example, **integer** is an ADT defined by the values …–2, –1, 0, 1, 2,… and by operations of addition, subtraction, multiplication, division *(with care)*. An ADT consists, not only of operations, but also values of the underlying data and constraints on operators.

Single Instance Style

Sometimes an ADT is defined as if only one instance existed during the execution of the algorithm and all operators are applied to that single instance. For example, we can create an instance of a stack and all operations *Push(), Pop()* are executed on that stack. On the other hand, some ADTs cannot be meaningfully defined without using multiple instances. This is the case when an operation requires multiple instances of the same ADT as parameters. For example, if we compare operation associated with the ADT, say, compare(S,T), we need two instances of the same ADT.

Functional Style Definition

Another way to define an ADT is to consider each state of the structure as a separate entity. Any operation that modifies an ADT is modeled as a mathematical function.

Example: Abstract stack (functional)

A complete functional style definition of an abstract stack could use three operations:

- **push:** takes a stack state as an argument and returns a stack state
- **top:** takes a stack state and returns a value
- **pop:** takes a stack state and returns a stack state

Instead of *create()*, the functional style assumes a special stack state, and the empty state is designated by a special symbol $\wedge$ or (), or defines a bottom operation that takes no arguments and returns a special stack state.

Advantages of Abstract Data Typing

- **Localization of change:** code that use an ADT need not be changed if the implementation of the ADT is changed
- **Flexibility:** different implementations of an ADT, having the same properties and abilities, are equivalent and may be used interchangeably in code that uses the ADT.

Other commonly used ADTs are:

LIST

The parameters are size: *N*, $\{A_0, A_1, A_2, \ldots, A_{n-1}\}$

Each element has a unique position in the list. Elements can be arbitrarily complex.

The operations on the list ADT, where X is the list and k is an element, are:

- *insert(X,k)*
- *remove(X)*
- *find(X,k)*
- *printlist(X)*

QUEUE

'*Insert*' takes place at the back but '*remove*' takes place in the front

The operations are:

- *enqueue*
- *dequeue*

2.3 ARRAYS

In computer science, an array is a data structure consisting of a collection of elements, each identified with an index. Ten 16-bit integer variables can be stored at memory addresses 1000, 10002,…,1018, so that the element with index i has the address $1000 + 2*i$. The memory address of the first element of the array is called the first address or foundation address.

The one-dimensional array is sometimes called a vector, and a two-dimensional array is called a matrix. Arrays are the most important data structures in computer science, which are used in almost every computer program. Arrays are useful, mainly because, the addresses of the array elements can be calculated at runtime. Arrays can also be multidimensional.

The declaration of a one-dimensional array in C-language is like:

```
int array_name[10]
```

The array can contain 10 elements. All the elements are integer data type. The first element is *array_name* [0] and the last element is *array_name* [9]. *array_name* itself is a pointer to the first element of the array. The element with index i is located at address *array_name* + $c*i$, where c is a fixed constant which is, sometimes called, address increment or stride. The constant c depends on the computer in use and the data type of the elements. $c = 2$ when the data type is integer and $c = 4$ when the data type is long integer.

2.3.1 Representation

The declaration for a two-dimensional array in C-language is:

```
int array_name [m][n]
```

This two-dimensional array contains mn elements of type integer. The element *array_name [i][j]* is stored at address *array_name* + $c*i + d*j$ where c and d are row and column increments.

There are two systematic layouts for representing two-dimensional arrays. For example, consider the matrix

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

In the row-major method, the elements are stored as

1 2 3 4 5 6 7 8 9

(all the elements in the first row, followed by the elements in the second row, etc.)

we have to declare another array of size greater by at least one and move the data elements to the new array. If the array is not full, we have to move $n - i + 1$ elements that follow the ith element to the right. This movement of data is an expensive operation.

2.3.3 Applications

2.3.3.1 Sparse Polynomials

Symbolic expressions such as $10x^2 + 3x - 10$ are called polynomials. We have to represent such polynomials in computers to manipulate them such as finding the sum of two polynomials, multiplying two polynomials, etc., If it is a nth degree polynomial, it will contain a maximum of $(n + 1)$ terms. Each term of the polynomial contains two parts, the coefficient and the exponent. Therefore, to represent a term of a polynomial, we need structures such as

```
typedef struct term
  {
        int coeff;
        int expon;
 }term;
```

A polynomial can then be represented as term poly[5].

For example, the polynomial $B = 2x^4 + 3x^3 + 5x^2 + 8x + 15$ can be represented as

Coefficient:	2	3	5	8	15
Exponent:	4	3	2	1	0

where the structures are (2,4), (3,3), (5,2), (8,1), (15,0). However, if we have polynomials such as $A = 10x^{999} + 100$, where most of the terms have coefficients zero, the above representation is not suitable since a lot of memory space is wasted for storing terms whose coefficients are zero. A better way would be to store only the non-zero terms, indicating the first term and the last term. In this way, we can store more than one polynomial in the same array of terms. The polynomials, *A* and *B* can be represented as

Coefficient:	10	100	2	3	5	8	15
Exponent:	999	0	4	3	2	1	0
	af	*al*	*bf*				*bl*

where *af* and *al* specify the first term and last term of the polynomial *A* and *bf* and *bl* specify the first and last terms of the polynomial *B*. It can easily be seen that a polynomial *E* which has *n* non-zero terms will have $el = ef + n - 1$, where *ef* and *el* are the indices in the global array.

Polynomial Addition

In the normal representation, addition of polynomials is easy. If the degrees of the polynomials are also the same, a simple for loop can be used to add two polynomials.

Let $A = \sum_i^n a_i x^i$ and $B = \sum_i^n b_i x^i$. if $C = A + B$ and $C = \sum_i^n c_i x^i$, then $c_i = a_i + b_i$

Let us represent A and B and C as array of structures of type term as
termA[n]; termB[n]; termC[n].
The sum of the polynomials A and B is stored in C using the loop

```
for ( i=0; i<n; i++)
  {
      termC[i].coeff = termA[i].coeff + termB[i].coeff;
      termC[i].expon = termA[i].expon;
   }
```

However, if the polynomials are of different degrees, we can extend the smaller polynomial to the size of the bigger polynomial with zero coefficients and add them.

A C-program which adds two sparse polynomials represented in a global array is as follows:

```
1.  #include<stdio.h>
2.  #include<stdlib.h>
3.  #include<conio.h>
4.  typedef struct term
5.  { int coeff;
6.  int exp;
7.  }term;
8.  term terms[100];
9.  int af,bf,cf,al,bl,cl,p,q,ant,bnt,c
    ,i, coef,ex;
10. void newterm(int,int);
11. void main()
12. { printf("number of terms in first
    polynomial");
13. scanf("%d",&ant);
14. af=0;
15. al=ant-1;
16. bf=ant;
17. printf("enter number of terms in
    second polynomial");
18. scanf("%d",&bnt);
19. bl=bf+bnt;
20. cf=bl+1;
21. printf("enter first polynomial
    terms");
22. for(i=af;i<=al;i++)
23. {
24. scanf("%d",&coef);
25. scanf("%d",&ex);
26. terms[i].coeff=coef;
27. terms[i].exp=ex;
28. };
29. printf("enter second polynomial
    terms");
30. for(i=bf;i<bl;i++)
31. {
32. scanf("%d",&coef);
33. scanf("%d",&ex);
34. terms[i].coeff=coef;terms[i].
    exp=ex;
```

```
35. }
36. p=af;
37. q=bf;
38. while(p<=al&&q<=bl)
39. { if(terms[p].exp==terms[q].exp)
40. {
41. c=terms[p].coeff+terms[q].coeff;
42. if(c!=0)
43. newterm(c,terms[p].exp);
44. p=p+1;q=q+1;
45. }
46. else
47. if(terms[p].exp<terms[q].exp)
48. {
49. newterm(terms[q].coeff,terms[q].
    exp);
50. q=q+1;
51. }
52. else
53. {
54. newterm(terms[p].coeff,terms[p].
    exp);
55. p=p+1;
56. }
57. }
58. while(p<=al)
59. {
60. newterm(terms[p].coeff,terms[p].
    exp);
61. p=p+1;
62. };
63. while(q<=bl)
64. {
65. newterm(terms[q].coeff,terms[q].
    exp);
66. q=q+1;
67. };
68. printf("complete polynomial is ");
69. for(i=0;i<cl-1;i++)
70. printf("%dx^%d+",terms[i].
    coeff,terms[i].exp);
71. printf("\b ");
72. }
73. void newterm(int c, int e)
74. {
75. terms[cl].coeff=c;
76. terms[cl].exp=e;
77. cl++;
78. }
```

Program: 2.1

Output of the above program is

Let the input polynomial be

$A =$

Coeff	2	1
expo	1000	0

	1	2	3
A[0]	4	11	13
A[1]	1	1	2
A[2]	1	7	8
A[3]	1	8	9
A[4]	2	1	3
A[5]	2	2	5
A[6]	2	7	4
A[7]	3	5	1
A[8]	3	6	9
A[9]	3	7	7
A[10]	4	1	1
A[11]	4	2	3
A[12]	4	3	4
A[13]	4	11	6

The first column gives the row numbers.

In the above matrix we have 14 rows and 3 columns. There will be as many rows as there are non-zero elements +1. The first row contains information about the sparse matrix. The first element in the first row is the number of rows, the second element in the first row is the number of columns, and the third element is the value of non-zero element. Some of the operations we wish to perform on these matrices is *transpose*, *addition*, and *multiplication*.

1. Transpose: As we know, the transpose of a matrix is the matrix obtained by interchanging the rows and columns. In other words, the element in the ith row, j^{th} column will be in the jth row and i^{th} column in the transposed matrix. The transpose of the above matrix is

	1	2	3
B[0]	11	4	13
B[1]	1	1	2
B[2]	1	2	3
B[3]	1	4	1
B[4]	2	2	5
B[5]	2	4	3
B[6]	3	4	4
B[7]	5	3	1
B[8]	6	3	9
B[9]	7	1	8
B[10]	7	2	4

B[11]	7	3	7
B[12]	8	1	9
B[13]	11	4	6

One way of obtaining the transpose is to interchange columns 1 and 2. However, this will not be in the form of representation, we had chosen, in the increasing row number and within the same row number, in increasing column number and it requires moving the rows and columns to bring it to the required form. To avoid this movement of data, we may proceed in the increasing order of column number:

```
For all elements in column j // j ranges from 1 to column number
Place element (i,j,val) at (j,i,val)
```

This says, place all elements of column 1 and store them in row 1, find all elements of column 2 and store them in row 2, etc. Since the rows are originally in order, we will locate the elements in the correct column order.

A procedure for the transpose of a sparse matrix can be as follows:

```
1.  procedure transpose(sparsematrix A)
2.  {
3.  sparsematrix B;
4.  int m,n,p,q,t,col;
5.  m=A[0,1]; n=A[0,2];t=A[0,3];
6.  B[0,1]=n;B[0,2]=m;B[0,3]=t;
7.  if(t>0)
8.   {
9.   q=1;
10.  for (col=1;col<=n;col++)
11.  for(p=1;p<=t;p++)
12. if(A[p,2]=col)
13. {
14. B[q,1]=A[p,2];
15. B[q,2]=A[p,1];
16. B[q,3]=A[p,3];
17. q=q+1;
18. }
19. }
20. }
```

Program: 2.2

The conversion of a matrix with large number of zeros can be achieved through the following program:

```
1. #include<stdio.h>
2. #include<conio.h>
3. main()
4. {
5. int
   i,j,k,l,m,n,a[10][10],sb[10]
   [3],str[10][3],count =0;
6. printf("enter rows and coloumns of
   the matrix\n");
7. scanf("%d%d",&m,&n);
8. printf("enter matrix elements\n");
9. for(i=0;i<n;i++)
10. {
11. for(j=0;j<n;j++)
12. {
13. scanf("%d",&a[i][j]);
14. if(a[i][j] != 0)
15. count++;
16. }
17. }
18. //convert normal matrix to sparse
    matrix
19. sb[0][0] = m;
20. sb[0][1] = n;
21. sb[0][2] = count;
22. k = 1;
23. for(i=0;i<m;i++)
24. for(j=0;j<n;j++)
25. if(a[i][j] != 0)
26. {
27. sb[k][0] = i+1;
28. sb[k][1] = j+1;
29. sb[k][2]= a[i][j];
30. k++;
31. }
32. k=0;
33. for(i=0;i<=count;i++)
34. {
35. str[i][0]=sb[i][1];
36. str[i][1]=sb[i][0];
37. str[i][2]=sb[i][2];
38. {
39. printf("given sparse matrix is \n");
40.  for(i=0;i<count+1;i++)
41. {
42.     for(j=0;j<3;j++)
43.          printf("%3d",sb[i][j]);
44.     printf("/n");
45. }
46. printf("Transpose of given sparse
    matrix is \n");
47. for(i=0;i<count+1;i++)
48. {
49.    for(j=0;j<3;j++)
50.          printf("%3d",str[i][j]);
51.    printf("/n");
52. }
53. }
```

Program: 2.3

Lines 35 to 37 create the kth row of the sparse matrix.
Lines 19 to 21 create the first row of the sparse matrix.
Lines 11 to 16 find non-zero elements.

```
19. sa[0][0] = m;
20. sa[0][1] = n;
21. sa[0][2] = count;
22. k = 1;l = 0;
23. for(i=0;i<m;i++)
24. for(j=0;j<n;j++)
25. if(a[i][j] != 0)
26. {
27. sa[k][l] = i+1;
28. sa[k][l+1] = j+1;
29. sa[k][l+2]= a[i][j];
30. k++;
31. }
32. printf("sparse matrix of a is \n");
33. for(i=0;i<=count;i++)
34. {
35. for(j=0;j<3;j++)
36. printf("%4d",sa[i][j]);
37. printf("\n");
38. }
39. count1 =0;
40. printf("enter second sparse matrix
    elements\n");
41. for(i=0;i<n;i++)
42. {
43. for(j=0;j<n;j++)
44. {
45. scanf("%d",&b[i][j]);
46. if(b[i][j] != 0)
47. count1++;
48. }
49. }
50. sb[0][0] = m;
51. sb[0][1] = n;
52. sb[0][2] = count1;
53. k = 1;l = 0;
54. for(i=0;i<m;i++)
55. for(j=0;j<n;j++)
56. if(b[i][j] != 0)
57. {
58. sb[k][l] = i+1;
59. sb[k][l+1] = j+1;
60. sb[k][l+2]= b[i][j];
61. k++;
62. }
63. printf("sparse matrix of b is\n");
64. for(i=0;i<=count1;i++)
65. {
66. for(j=0;j<3;j++)
67. printf("%4d",sb[i][j]);
68. printf("\n");
69. }
70. /*sparse matrix addition is as
    follows*/
71. i=1;j=1;k=1;
72. while((i<=count) &&(j<=count1))
73. {
74. if((sa[i][0]==sb[j][0]) && (sa[i]
    [0]==sb[j][0]))
75. {
76. sc[k][0]=sa[i][0];
77. sc[k][1]=sa[i][1];
78. sc[k][2]=sa[i][2]+sb[j][2];
79. i++;j++;
80. }
81. else
82. if(sa[i][0] > sb[j][0])
83. {
84. if(sa[i][1] < sb[j][1])
85. {
```

```
86. sc[k][0]=sa[i][0];
87. sc[k][1]=sa[i][1];
88. sc[k][2]=sa[i][2];
89. i++;
90. }
91. else
92. {
93. sc[k][0]=sb[j][0];
94. sc[k][1]=sb[j][1];
95. sc[k][2]=sb[j][2];
96. j++;
97. }
98. }
99. else
100. if(sa[i][0]>sb[j][0])
101. {
102. sc[k][0]=sa[i][0];
103. sc[k][1]=sa[i][1];
104. sc[k][2]=sa[i][2];
105. i++;
106. }
107. else
108. {
109. sc[k][0]=sb[j][0];
110. sc[k][1]=sb[j][1];
111. sc[k][2]=sb[j][2];
112. j++;
113. }
114. k++;
115. }
116. while(i<=count)
117. {
118. sc[k][0]=sa[i][0];
119. sc[k][1]=sa[i][1];
120. sc[k][2]=sa[i][2];
121. i++;
122. k++;
123. }
124. while(j<=count1)
125. {
126. sc[k][0]=sb[j][0];
127. sc[k][1]=sb[j][1];
128. sc[k][2]=sb[j][2];
129. j++;
130. k++;
131. }
132. sc[0][0]=m;sc[0][1]=n;sc[0][2]=k-1;
133. for(i=0;i<k;i++)
134. {
135. for(j=0;j<3;j++)
136. printf("%3d",sc[i][j]);
137. printf("\n");
138. }
139. getch();
140. }
```

Program: 2.4

```
Output:
Enter rows and columns of the matrices
5
5
Enter first matrix elements
1 0 0 0 1
0 0 0 0 0
0 0 0 1 0
0 0 0 0 1
1 0 0 0 0

Enter second matrix elements
0 0 0 0 1
0 0 2 0 0
0 0 0 0 4
0 0 0 0 1
1 0 0 0 0

First sparse matrix is
5 5 5
0 0 1
0 4 1
2 3 1
3 4 1
4 0 1

Second sparse matrix is
5 5 5
0 4 1
1 2 2
2 4 4
3 4 1
4 0 1

Resultant sparse matrix is
5 5 7
0 0 1
0 4 2
1 2 2
2 3 1
2 4 4
3 4 2
4 0 2
```

```
1.  #include<stdio.h>
2.  #include<stdlib.h>
3.  #include<conio.h>
4.  typedef struct node
5.  {
6.  int data;
7.  struct node *next;
8.  }node;
9.  int main(void)
10. {
11. node *head, *temp, *p, *q;
12. int x, y=1;
13. scanf("%d", &x);
14. head=(struct node*) malloc (sizeof  (struct node));
15. head->data=x;
16. head->next=NULL;
17. p=head;
18. while(y!=-1)
19. {
20. scanf("%d",&y);
21. temp=head;
22. while(temp->next!=NULL)
23. temp=temp->next
24. temp->data=y;
25. temp->next=NULL;
26. }
27.  printf("Head->");
28.  while(p->next!=NULL)
29. {
30. printf("%d",p->data)
31. printf("->");
32. };
33. printf("NULL");
34. return 0;
35. }
```

Program 2.5

Input: 2 3 4 5 6 –1

```
Output:
Head ->1->2->3->4->5->6->-1->NULL
```

Head is the pointer to the first node containing data 1 and the other nodes are linked sequentially and the last node is null.

Operations that need to be performed on the linked-lists are:

1. Insertion
2. Deletion
3. Traversal

2.4.1.2 Insertion

There are four positions where you can insert a node:

1. In the front
2. At the end
3. After a given node
4. Before a given node

We have already created a linked-list in the previous program with data elements 1, 2, 3, 4, 5, 6.

2.4.1.2.1 Insertion at the Front

Let us insert a node with data element 100 in the front. The C program which does this is as follows:

```
1.   void insertFront(node head)
2.   {
3.   node *temp,*p;
4.   temp=(struct node*)malloc(sizeof(struct node));
5.   temp->data=100;
6.   temp->next=head;
7.   head=temp;
8.   p=head;
9.   printf("Head->");
10.  while(p->next!=null)
11.  {
12.  printf("%d",p->data);
13.  printf("->");
```

```
14.   p=p->next;
15.   }
16.   printf("null");
17.   return 0;
18.   };
```

Program 2.6

```
Output:
Head->100->1->2->3->4->5->6->-1->null
```

2.4.1.2.2 Insertion at the end

Insertion at the end can be similarly executed. We start at the head and reach the last node and replace the null pointer in the last node with a pointer to the *new_node*. The function to do this can be expressed in the C program as follows:

```
 1.  void insertEnd(node head, int x)
 2.  {
 3.  node *new_node, *temp, *p;
 4.  x=7;
 5.  new_node=(struct node*)malloc(sizeof(struct node));
 6.  new_node->data=x;
 7.  new_node->next=NULL;
 8.  p=head;
 9.  while(p->next!=null)
10.  p=p->next;
11.  p->next=new_node;
12.  temp=head;
13.  printf("head");
14.  while(temp->next!=null)
15.  {
16.  printf("%d->", temp->data)
17.  temp=temp->next
18.  };
19.  return 0;
20.  }
```

Program 2.7

Input: The list created previously ; x=3, y=15

```
Output:
Head->1->2->3->15->4->5->6->7->null
```

2.4.1.3 Deletion

Deletions from a linked-list are comparatively easier than insertions. Once again there are three types of deletion.

1. Deletion of the node in the front
2. Deletion of the node at the end
3. Deletion of the node in the middle

The program segment for the deletion of node in the front is a single statement.

```
new_head=head->next;
```

and we may free the previous head:

```
free(head);
```

Deletion of the node at the end can be achieved with the following statements.

```
temp=head;
while(temp->next!=null)
{
   p=temp;
   temp=temp->next;
}
```

Program 2.9

When we come out of the loop, *temp* is the last node and *p* is its previous node.

The following two statements delete the last node:

```
p->next=null;
free(temp);
```

Deleting a node from the middle is a little more complicated since we have to first traverse the list up to the node we wish to delete. Let the node to be deleted contain the data *x*. The following statements delete the node containing *x*. We assume such a node exists in the list.

```
temp=head;
while (temp->data!=x)
{
 p=temp;
 temp=temp->next;
}
```

Program 2.10

When we come out of the loop, *temp* is the node to be removed, since it contains data *x* and *p* is its previous node. We link *p* with the node next to *temp* and free *temp*.

```
p->next = temp->next;
free(temp);
```

Program 2.11

We can find the length of linked-list by executing the following C program

```
int Length (node *head)
{
  struct node  *current=head;
  int count=0;
  while (current != NULL)
  {
    count++;
    current=current->next;
  }
return count;
}
```

In some implementations, an extra dummy node, usually called a 'sentinel' node is added before the first data record and after the last data record. Every list, therefore, has two sentinel nodes—the first one and the last one, even when the list is empty. Since the reference to the first node gives access to the whole list, it is often called the 'address' or 'pointer', or handle to the list.

2.4.2 Doubly Linked-lists

In doubly linked-lists each node, besides containing a pointer to the next node also contains a pointer to the previous node.

A doubly linked-list contains an extra pointer, usually called *prev* (previous) together with the next pointer and data:

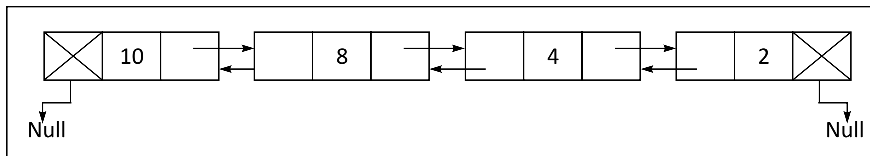


Figure 2.1

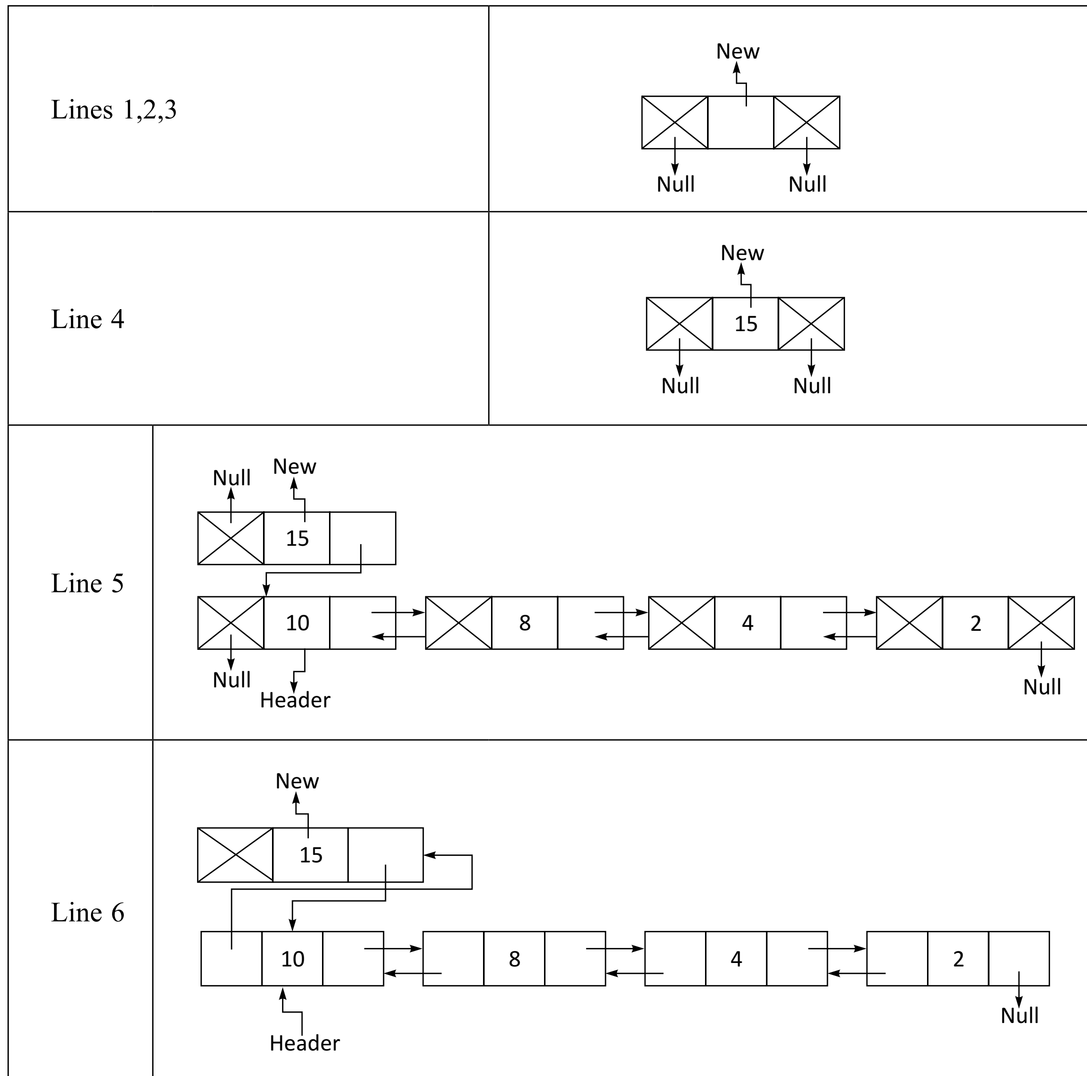


Figure 2.2

2.4.2.1.2 Insert a node after a given node in a doubly linked-list

This requires 6 steps as detailed below. Let *prev_node* be the given node.

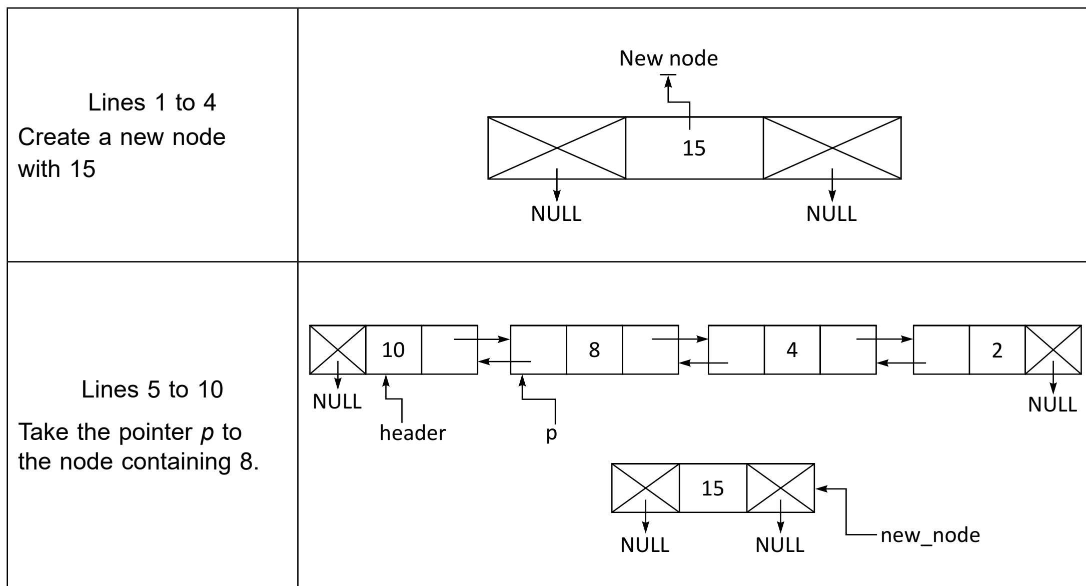
1. Create a new node
2. Put in the data
3. Set prev of *new_node* as *prev_node*
4. Set next of *new_node* as next of *prev_node*
5. Set prev of next of *prev_node* as *new_node*

These six steps can be executed through the function *insertAfter()* function shown below.

```
1.  struct node* new_node= (struct node*)malloc(sizeof(struct node));
2.  new_node->data = new_data;
3.  new_node->next = NULL;
4.  new_node->prev = NULL;
5.  p=header;
6.  q=p;
7.  while(p->data != 8)
8.  {
9.  q=p;
10. p=p->next
11. }
12. new_node->prev = p;
13. new_node ->next = p->next;
14. p->next->prev = new_node;
15. p->next = new_node;
```

Program 2.13

If we insert a node with data 15 after the node containing data 8, execution of the above program leads to the following sequence of steps.



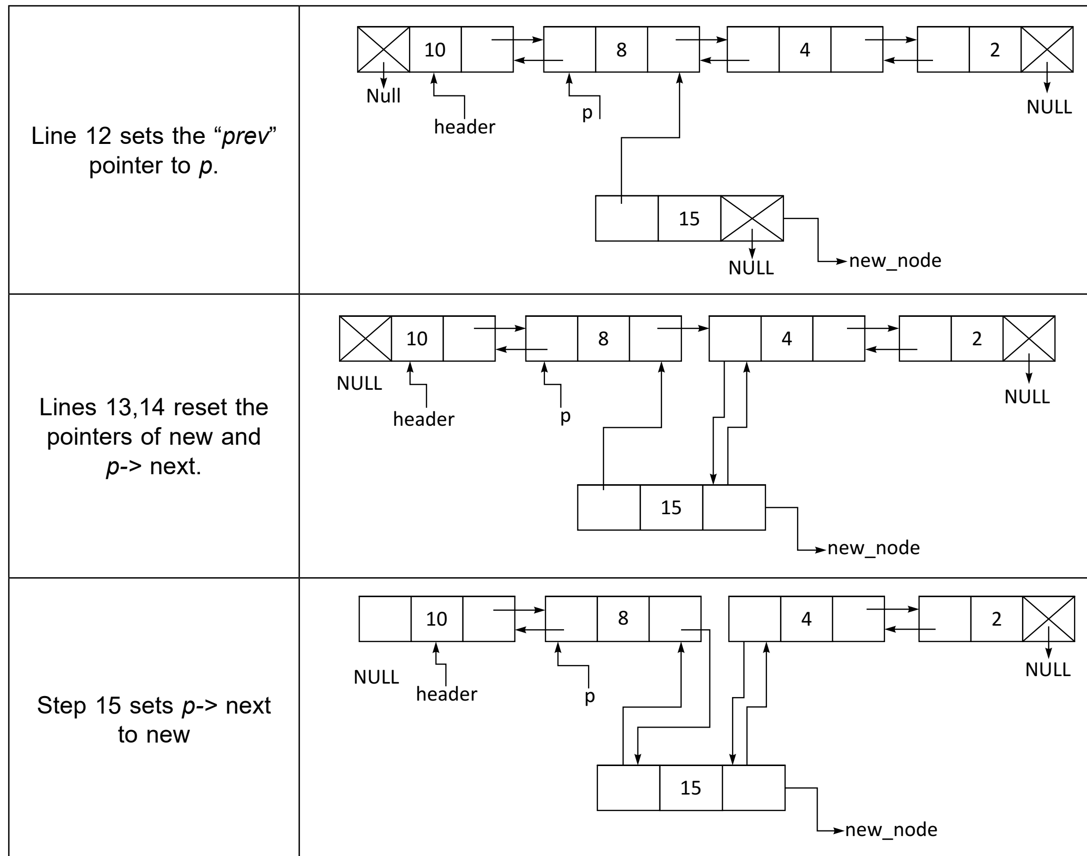


Figure 2.3

2.4.2.1.3 Insert a node at the end

Let us call the function which does this as append.

```
 1.  void append(struct node *head, int new_data);
 2.  {
 3.  struct node *new_node =( struct node*)malloc(sizeof(struct node));
 4.  new_node->data = new_data;
 5.  new_node->next = NULL;
 6.  new_node->prev = NULL;
 7.  temp = header;
 8.  while(temp->next != NULL)
 9.  temp = temp->next;
10.  temp->next = new_node;
11.  new_node->prev = temp
12.  }
```

Program 2.14

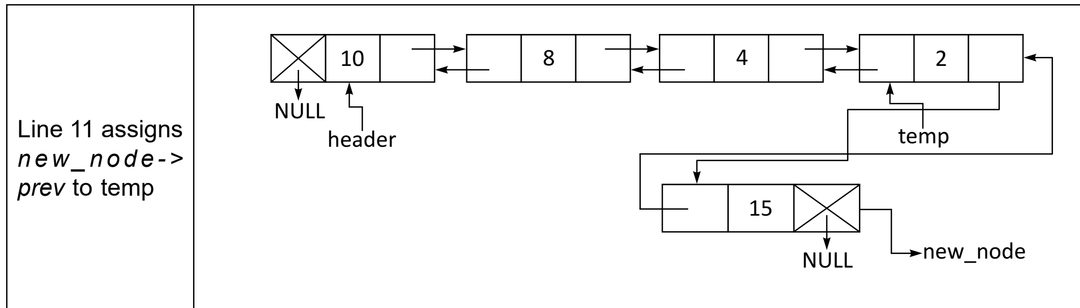


Figure 2.4

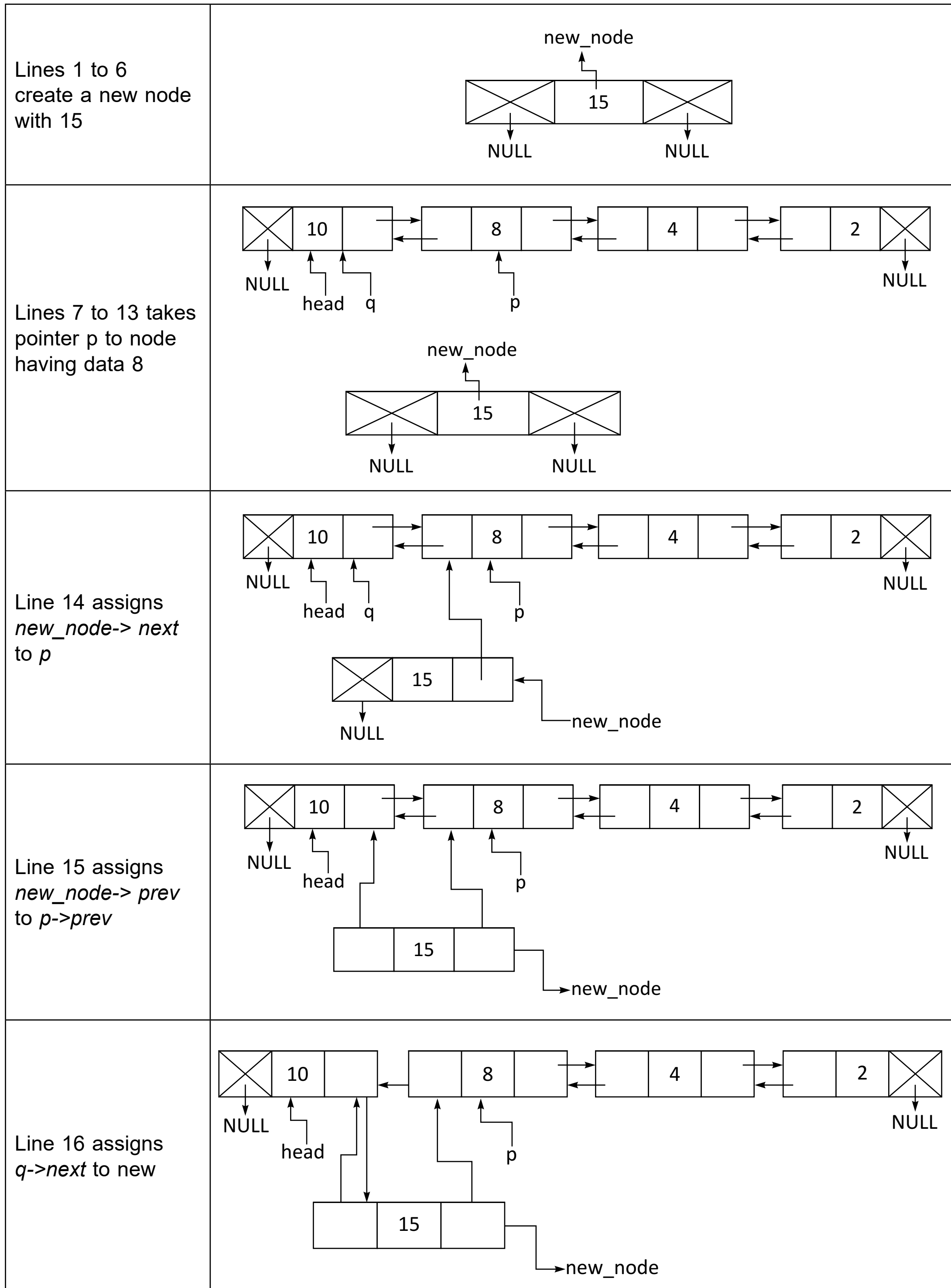
2.4.2.1.4 Insert a node before a given node

Let us call this function *addBefore()*

```
1.   void addBefore(struct node *next_node, int new_data)
2.   {
3.   struct node *new= (struct node*)malloc(sizeof(struct node));
4.   new->data = new_data;
5.   new->prev = NULL;
6.   new->next = NULL;
7.   p=head;
8.   q=p;
9.   while(p->data!=8)
10.  {
11.  q=p;
12.  p=p->next
13.  }
14.  new ->next = p;
15.  new ->prev = q->prev;
16.  q-> next = new;
17.  p->prev = new;
18.  }
```

Program 2.15

If we insert a node with data 15 before the node containing data 8, execution of the above program leads to the following sequence of steps.


Lines 1 to 6 create a new node with 15
new_node
15
NULL
NULL
Lines 7 to 13 takes pointer p to node having data 8
10
8
4
2
NULL
head
q
p
NULL
new_node
15
NULL
NULL
Line 14 assigns *new_node-> next* to *p*
10
8
4
2
NULL
head
q
p
NULL
15
NULL
new_node
Line 15 assigns *new_node-> prev* to *p->prev*
10
8
4
2
NULL
head
p
NULL
15
new_node
Line 16 assigns *q->next* to new
10
8
4
2
NULL
head
p
NULL
15
new_node

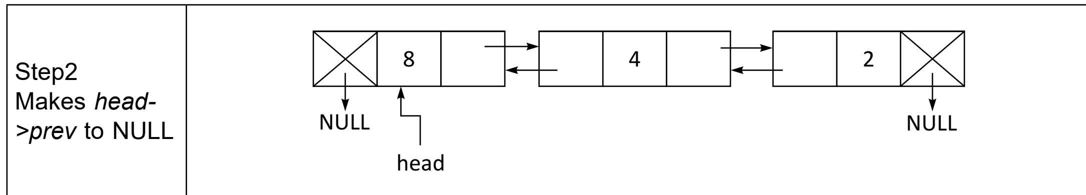


Figure 2.7

2. Let us say, we have to remove the last node. The execution of the following instructions will do this

```
3.  temp=head;
4.  while(temp->next != NULL)
5.  temp = temp->next;
6.  temp->prev->next = NULL;
```

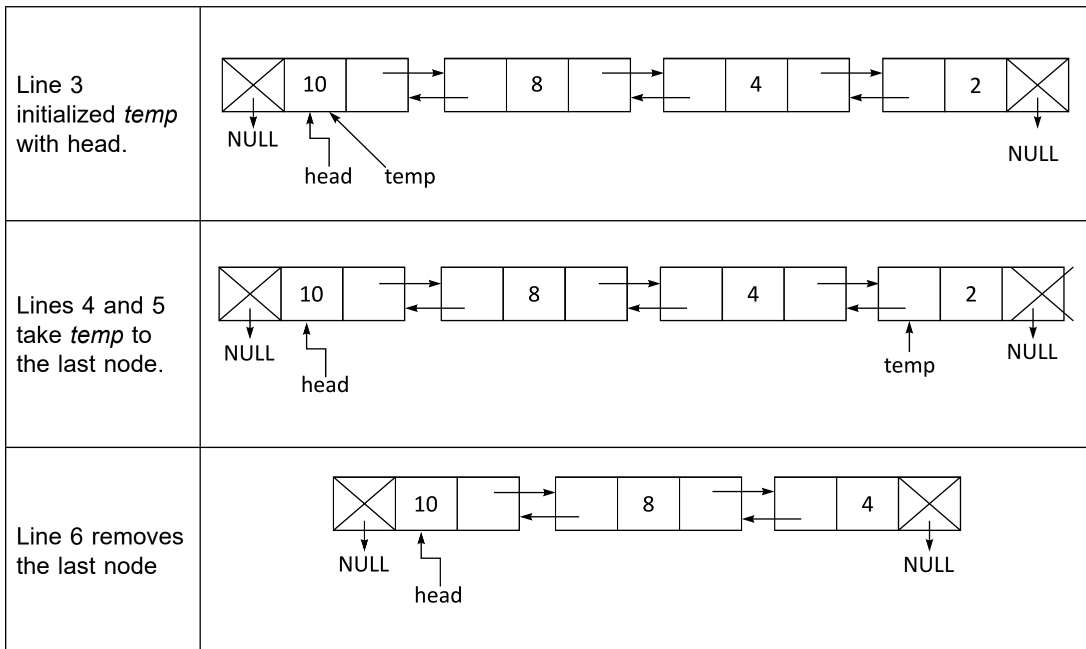


Figure 2.8

3. If the node to be removed is neither the first nor the last node and is the node whose data is *x*, we can execute the following instructions:

```
 7.    temp = head;
 8.    while(temp->data != x)
 9.    temp = temp->next;
10.    temp->prev->next=temp->next;
11.    temp->next->prev=temp->prev;
12.    free(temp);
```

Let us illustrate the procedure of deleting the node containing data 8

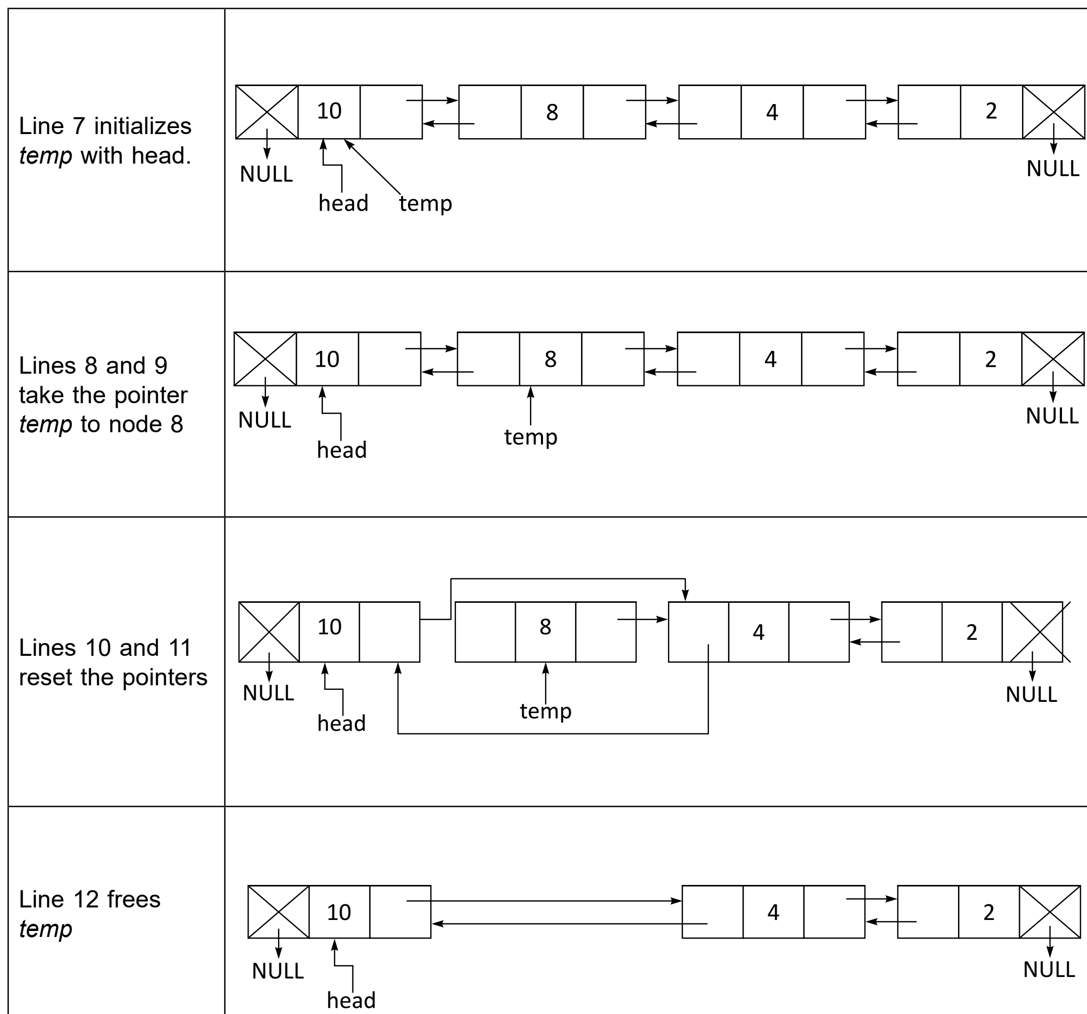


Figure 2.9

The doubly linked-list can be traversed in both directions. The following program traverses the DLL in the forward direction and prints the data.

```
12.  printlist(struct node *head);
13.  }
14.  void printlist(struct node *head)
15.  {
16.  temp =head;
17.  while(temp->next!=NULL)
18.  temp=pemp->next
19.  last=temp;
20.  temp = last;
21.  while(temp->prev!=NULL)
22.  {
23.  printf("%d\n", temp->data);
24.  temp=temp->prev;
25.  }
```

Program 2.17

Advantages of doubly linked-list

1. The doubly linked-list can be traversed in both directions.
2. We can access all the nodes starting from any node, not necessarily starting from the head.

Disadvantages doubly linked-list

1. Every node in the doubly linked-list needs extra space.
2. All operations need to maintain two pointers.

2.4.3 Circularly Linked-lists

In the last node of a singly linked-list, we have a NULL reference to indicate the lack of further nodes. If we make it point to the first node (or the head node) of the list we have a circularly linked-list. In other words, the last element contains the address of the first element.

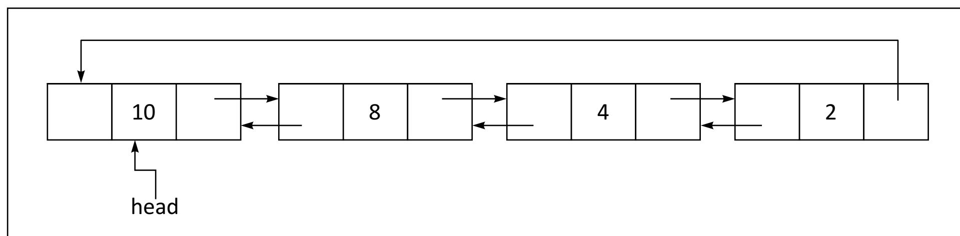


Figure 2.10

Applications

- In our personal computers all the running programs are kept in a circularly linked-list. The operating system fixes time slots for all the running programs in the circular list.
- Circular linked-list can also be used to create circular queues.

As usual there are three operations that can be performed on a circular list.

- Look-up
- Insertion
- Deletion

2.4.3.1 Look-up

Let us say, we are looking for data item *x* in the circular list. We execute the following steps

1. Start at the head node. (The head node can be any one of the nodes.)
2. Follow the links from the head node until we reach the node containing the desired value, if it exists.
3. If we reach back to the head node, report failure.

Let us first create a circular list. If we are creating the first node in the list, let us call it head. We define the node as a structure:

```
struct node
 {
     int data;
     struct node *next;
 }
```

```
Typedef struct node node;
Now node is a datatype of struct node
```

Declare head as a node and initialize it

```
head=(struct node*)malloc(sizeof(struct node))
```

Head is a pointer to a node. We initialize it as:

```
head->data =x;
head->next = head;
```

This will be circular list containing only one node

```
Enter node value
3
Enter next node value otherwise enter -1
4
Enter next node value otherwise enter -1
5
Enter next node value otherwise enter -1
6
Enter next node value otherwise enter -1
-1
Circular list nodes are
3 4 5 6
```

With data elements 3, 4, 5, 6, the circular list looks like:

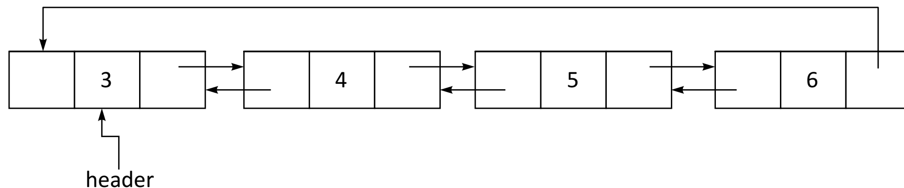


Figure 2.12

Let us say we are searching data item *z* in the list. We start at the head node and follow the links (the next field) until we reach the required node, in which case we report success, or the last node (points to the head node), in which case we report failure. The program segment that does this is as follows:

```
p=head;
while((p->data !=z ) || (p->next!=head))
  p=p->next;
if(p->next==head)
 printf("unsuceessful search:")
else
 printf("search successful);
```

2. If the node to be deleted is the middle node the procedure is the same as in the case of a singly linked-list.
3. If the node to be deleted is the last node:

```
p=head;
while(p->next!=head)
{
q=p;
p=p->next;
}
```

When we come out of the loop, *p->next = head* and hence *p* is the last node and *q* is its previous node.

```
q->next=head;
```

deletes the last node.

Advantages of Linked Lists

- Linked lists are dynamic data structures which can grow and be pruned allocating and de-allocating memory while the program is running
- Insertion and deletion operations are easily implemented
- Dynamic data structures such as stacks and queues can be implemented using a linked list

Disadvantages of Linked Lists

- Different amount of times are needed to access different elements
- Access is sequential
- It is not easy to sort elements stored in a linked list
- We cannot traverse a singly linked-list from the end to the beginning

2.4.4 Cursor-based Linked-list

Some languages do not support pointers. If linked lists are required, then an alternative to pointer representation is needed. This is known as cursor implementation.

Two important features that exist in a pointer representation are:

1. The data is stored in a collection of structures.
2. A new structure can be obtained by a call to *malloc()* and released by a call to *free()*.

The cursor implementation must be able to simulate these features. One way to simulate feature 1 is to have a global array of structures. Array index can be used in place of addresses or pointers.

We must now simulate feature 2 by allowing the equivalent of *malloc()* and *free()* for cells in the *CURSOR_SPACE* array. To do this we keep a free list of cells that are not in any list. The list will have cell 0 as the header. The initial configuration of the *CURSOR_SPACE* array is like:

Table 2.1

Slot	*Element*	*Next*
0	-	1
1	-	2
3	-	4
4	-	5
5	-	6
6	-	7
7	-	8
8	-	9
9	-	10
10	-	0

Initialized *CURSOR_SPACE*

A value 0 for next is the equivalent of a NULL pointer. To perform *malloc()* the first element after the head in the free list is removed from the free list and appropriate action taken. Let us define the global array of structures as follows:

```
typedef unsigned int node_ptr;
struct node
 {
  element_type element;
  node_ptr next;
}
```

Note that next has an integer value.

```
typedef node_ptr LIST;
typedef node_ptr POSITION; // LIST and POSITION are also integers.
struct node CURSOR_SPACE[SPACE_SIZE];
```

Program 2.19

CURSOR_SPACE[] is an global array of nodes of size SPACE_SIZE.
The allocation and freeing of cells can be done using the following sub-routines:

```
void CURSOR_ALLOC(void)
  {
   position p;
   p = CURSOR_SPACE[0].next;//CURSOR_SPACE[0] is the header for freelist
 }
```

The third statement declares *p* as an integer and the fourth statement stores the first index of the list in *p*. Line 5 checks if the index is positive (if *p* = 0, then it is the end of the list) and if the element at index *p* is the required element, i.e., *x*. If it is not, it checks the next element.

To insert an element we execute the following function:

Line 3 declares *temp_cell* as an integer and line 4 stores the index returned by *CURSOR_ALLOC()* in *temp_cell*. Line 5 checks if the *temp_cell* is equal to zero (0 indicates the end of the list). If *temp_cell* is not equal to zero, line 12 is executed assigning the element *x* to *temp_cell* in the *CURSOR_SPACE*. Then lines 13 and 14 reassign the indices.

```
 1.  void insert (element_type x, list L, position p) // insert after p in L.
 2.  {
 3.  position temp_cell;
 4.  temp_cell = CURSOR_ALLOC();
 5.  if(temp_cell!=0)
 6.   {
 7.     printf( " out of space");
 8.     return;
 9.    }
10.  else
11.   {
12.   CURSOR_SPACE[temp_cell].element = x;
13.   CURSOR_SPACE[temp_cell].next = CURSOR_SPACE[p].next;
14.   CURSOR_SPACE[p].next = temp_cell;
15.   }
16.  }
```

Program 2.21

It is also easy to delete an element from a list in *CURSOR_SPACE*.

```
 1.  void delete (element_type x, list L)
 2.   {
 3.  position p, temp_cell;
 4.  p = find_previous(x, L);
 5.  If(!is_Last(p, L)
 6.   {
 7.   temp_cell = CURSOR_SPACE[p].next;
 8.   CURSOR_SPACE[p].next = CURSOR_SPACE[temp_cel].next;
 9.   CURSOR_FREE(temp_cell);
10.   }
11.  }
```

Program 2.22

The free list represents an interesting data structure in its own right. The cell that is removed from the free list is the one that is most recently placed in the free list using the free routine. Thus, the last cell placed in the free list is the first cell that is taken off. The data structure that has this property is known as STACK.

As in the case of other data structures we have to test if the CURSOR_SPACE is full before inserting, and test the *CURSOR_SPACE* is empty before deleting. The two functions that do this are:

```
int is_Last(position p, list L)
  {
   return (CURSOR_SPACE[p].next == 0);
 }
```

Obviously, if p.next = 0, p is the last element in the list.

```
int is_Empty( list L)
 {
  return( CURSOR_SPACE[L].next==0);
 }
```

L is the head of the list and if *L.next* = 0, the list is empty.

2.4.5 Application of Linked-lists

The linked list is a useful data structure to represent and manipulate polynomials. Most of the operating systems maintain the free list of memory blocks in the form of a linked list.

REVIEW QUESTIONS

1. What is an algorithm? What are the essential features of an algorithm?
2. Write a program to delete duplicate elements from a single-dimensional array.
3. Define a singly linked-list and write an algorithm to reverse it.
4. How do you convert a matrix with large number of zeros into a sparse matrix using singly linked-list?
5. How do you represent a polynomial for a computer. Write a program to add the following polynomials.

$$4x^4 + 2x^3 + 3x + 5 \qquad 5x^5 + 3x^2 + 8$$

(i) Insertion at the front of the linked list
(ii) Insertion at the end of the linked list
(iii) Deletion of the front node of the linked list
(iv) Deletion of the last node of the linked list

(a) (i) and (ii)
(b) (i) and (iii)
(c) (i), (ii) and (iii)
(d) (i), (ii) and (iv)

8. Consider an implementation of unsorted doubly linked-list. Suppose it has its representation with a head pointer only. Given this representation, which of the following operation can be implemented in O(1) time?
   (i) Insertion at the front of the linked list
   (ii) Insertion at the end of the linked list
   (iii) Deletion of the front node of the linked list
   (iv) Deletion of the end node of the linked list
   (a) (i) and (ii)
   (b) (i) and (iii)
   (c) (i), (ii) and (iii)
   (d) (i), (ii), (iii) and (iv)
9. In a linked-list each node contains minimum two fields. One field is data field to store the data and the second field is?
   (a) Pointer to character
   (b) Pointer to integer
   (c) Address of next node
   (d) None of the above
10. What would be the asymptotic time complexity to add an element in the linked list?
   (a) $O(1)$
   (b) $O(n)$
   (c) $O(n^2)$
   (d) None
11. The concatenation of two lists can be performed in O(1) time. Which of the following variation of linked list should be used?
   (a) Singly linked-list
   (b) Doubly linked-list
   (c) Circular doubly linked-list
   (d) Array implementation of list
12. Consider the following declaration in C programming language

```
struct node
{
int data;
struct node * next;
}
typedef struct node NODE;
NODE *ptr;
```

Which of the following c code is used to create new node?

(c) *head=p; p->next=q; q->next=NULL;*
(d) *q->next=NULL; p->next=head; head=p;*

16. Which of the following data structure can't store the non-homogeneous data elements?
    (a) Arrays
    (b) Records
    (c) Pointers
    (d) Stacks
17. Which of the following is not the part of ADT description?
    (a) Data
    (b) Operations
    (c) Both of the above
    (d) None of the above
18. …………… are not the components of a data structure.
    (a) Operations
    (b) Storage structures
    (c) Algorithms
    (d) All of the above
19. Which of the following is true about the characteristics of abstract data types?
    (i) It exports a type.
    (ii) It exports a set of operations
    (a) True, False
    (b) False, True
    (c) True, True
    (d) False, False
20. Arrays are best data structures
    (a) for relatively permanent collections of data
    (b) for the size of the structure and the data in the structure are constantly changing
    (c) for both of above situation
    (d) for none of above situation

Answers

1. (c)	2. (a)	3. (a)	4. (b)	5. (d)
6. (a)	7. (c)	8. (b)	9. (c)	10. (b)
11. (c)	12. (a)	13. (d)	14. (c)	15. (d)
16. (a)	17. (d)	18. (d)	19. (c)	20. (a)