UNIT 12 LISTS

Structure **Page No.**

12.1	Introduction Objectives	23
12.2	Basic Terminology	24
12.3	Static Implementation of Lists	24
12.4	Pointer Implementation of Lists Storage of Sparse Arrays using Linked List	27
12.5	Doubly Linked Lists	36
12.6	Circular Linked List	39
12.7	Storage Allocation	40
12.8	Storage Pools	41
12.9	Garbage Collection	41
12.10	Fragmentation, Relocation and Compaction	42
12.11	Summary	43
12.12	Solutions/Answers	43

12.1 INTRODUCTION

In Unit 11 of this Block we discussed a basic data structure, arrays. Arrays, although available in almost all the programming languages, have certain limitations on structuring and accessing data. In this Unit we turn our attention to another data structure called the **List**.

Lists, like arrays are used to store ordered data. A List is a linear sequence of data objects of the same type. Real-life events such as people waiting to be served at a bank counter or at a railway reservation counter, may be implemented using List structures. In computer science, Lists are extensively used in data base management systems, process management, operating systems, editors etc.

In Sec. 12.2, we introduce basic terminology related to Lists. In Sec. 12.3, we discuss static implementation of Lists using arrays. In Sec. 12.4, we discuss dynamic implementation of lists using pointers. We also discuss various operations that can be performed on a List like insertion of an element, deletion of an element etc. We had already seen how to store sparse matrices using arrays in Unit 11. Here, we will discuss storage of sparse arrays using linked Lists. In Sec. 12.5, and Sec. 12.6, we will discuss some variants of linked Lists called doubly linked Lists and circular linked Lists, respectively. In the last two sections, we will discuss some applications of lists to garbage collection and storage allocation.

Objectives

After studying this unit, you should be able to

- store data structures in computer memory in two different ways viz. sequential allocation and linked allocation;
- differentiate between a linear list and an array;
- implement linear lists in terms of built-in data types in C;
- code following algorithms using a linked list represented as an array of records;
    - creating a linked list
    - inserting an element at a specified location in a linked list
    - deleting an element from a linked list.

12.2 BASIC TERMINOLOGY

In this section we will introduce you to basic terminology. We begin by defining a List.

Definition 1: A **linear List** is an ordered set consisting of a variable number of elements to which addition and deletions can be made. A linear list displays the relationship of physical adjacency.

The first element of a List is called **head** of List and the last element is called the **tail** of List.

Every element of List, unless it is the head has a **predecessor** and every element of the List, unless it is the tail of the List, has a **successor**.

The elements in a List are tied together by their successor-predecessor relationship.

Following are some of the basic operations that may be performed on a List:

- Create a List
- Check for an empty List
- Search for an element in a List
- Search for a predecessor or a successor of an element of a List
- Delete an element from a List
- Add an element at a specified location of a List
- Retrieve an element from a List
- Update an element of a List
- Sort a List
- Print a List
- Determine the size or number of elements in a List
- Delete a List

More complex operations may be performed on a List. However a complex operation would generally turn out to be a combination of two or more of the above basic operations.

A List can be implemented statically or dynamically using an array index or pointers respectively. We will discuss static implementation of Lists in the next section.

12.3 STATIC IMPLEMENTATION OF LISTS

Static implementation is the simplest implementation. Its size is fixed and allocated at compilation time. A List can be implemented as an array as follows:

Let our List elements be names of all the colours, say BLUE, RED, YELLOW, GREEN and ORANGE.

We may have an array List declared as `List[LIST_SIZE]`, and fix its size as 8. Therefore, we have something like in Fig. 1 on the next page. The elements are sequentially stored in LIST[0], LIST[1], …, LIST[4]. LIST[5] through LIST[7] are allocated but not used.

	LIST	
LIST(0)	BLUE	
LIST(1)	RED	
LIST(2)	YELLOW	
LIST(3)	GREEN	
LIST(4)	ORANGE	
LIST(5)		Allocated, but not used
LIST(6)		
LIST(7)		

Fig. 1: A list declared as an array.

The predecessor of LIST[0] (or head) is NIL; LIST[4] is the tail of the List and has no successor. We may also tabulate the predecessors and successors of the other elements of the List as in Table 1.

Table 1: Predecessors and successors.

	Data	Array Index	Predecessor Index	Successor Index
	BLUE	0	NIL	1
	RED	1	0	2
	YELLOW	2	1	3
	GREEN	3	2	4
	ORANGE	4	3	NIL
Unused locations	–	–	–	–
	–	–	–	–
	–	–	–	–

Since the elements are sequentially stored, we don't need to store the predecessor and successor indices. Any element in the List can be accessed through its index.

Now let us see an Insert operation. If we want to insert an element at Kth position i.e. after LIST[K-1].

To do this we must shift elements LIST[K] through LIST [Last] to respectively LIST [K+1] through LIST [Last+1]. At the same time we must also check that Last + 1 does not exceed the value of SIZE.

LIST[Last] is nothing but the tail of the List.

Let us see the operations to be performed in an algorithmic form.

```
/*Check if Last+1 is less than or equal to Size*/
if (Last + 1 > List_size - 1)
   error; /*overflow*/
else{
/*Shifting elements from K + 1 through Last + 1*/
for(i = Last; i > K; i--)
   List[i] = List[i-1];
/*Insert element in the Kth position*/
List[K]=element
Last=Last+1;
```

In the delete operation we need to shift element in upwards direction and also decrement the value of tail by 1.

Example 2: Let us now write a C program that creates a list that can hold 10 strings. We will set the sets the first five elements to BLUE, RED, YELLOW, GREEN and ORANGE. Then we

will insert CYAN in the third position.

```
/*Program-12.1.Examp of List implementation using array.
File name:unit12-listarray.c*/
#include<stdio.h>
#include<string.h>
#define MAX_L 10
#define MAX_WD 20
int last=0;
int ins_element(char array[MAX_L][MAX_WD], int pos, char *text);
int del_element(char array[MAX_L][MAX_WD], int pos);
int print_array(char array[MAX_L][MAX_WD]);
void Error(char *message);
int main()
{
    char colours[MAX_L][MAX_WD];
    int i;
    for (i = 0; i < MAX_L; i++)
        strcpy(colours[i], "-");
    ins_element(colours, 0, "BLUE");
    ins_element(colours, 1, "RED");
    ins_element(colours, 2, "YELLOW");
    ins_element(colours, 3, "GREEN");
    ins_element(colours, 4, "ORANGE");
    print_array(colours);
    ins_element(colours, 2, "CYAN");
    print_array(colours);
    del_element(colours, 3);
    print_array(colours);
    return 0;
}
```

Here is the function that inserts an element in a list.

```
int ins_element(char array[MAX_L][MAX_WD], int pos, char *text)
{
    int j = MAX_L - 1;
    if(last >= j){
        Error("Error! Overflow!!");
    }
    else{
        for (; j > pos; j--)
                strcpy(array[j], array[j - 1]);
        strcpy(array[pos], text);
        last=last+1;
    }
    return 0;
}
```

We use the `Error()` function that we used in the answer to exercise 1 of Unit 11 to print an error message if the list is already full. Here is the function that deletes a particular element from the list.

```
int del_element(char array[MAX_L][MAX_WD], int pos)
{
    int i;
    for (i = pos; i < MAX_L - 1; i++)
        strcpy(array[i - 1], array[i]);
    strcpy(array[MAX_L - 1], "-");
    last = last-1;
    return 0;
}
```


You may recall that `list[i]` is a pointer to the ith row of the array. Also, note the use of

`strcpy()` here. This is necessary because C language does not allow assigning one array to another; we have to copy one array to another element by element.

* * *

E1) Write C functions that:
   i) Print the array in our example.
   ii) Return the ith element of the array.
   iii) Return the number of elements in the array.

You must have noticed that array implementations of a List has certain drawbacks. These are:

1) Memory storage space is wasted; very often the List is much shorter than the array size declared.
2) List cannot grow in its size beyond the size of the declared array if required during program execution.
3) Operations like insertion and deletion at a specified location in a List required a lot of movement of data, therefore, leading to inefficient and time consuming algorithms.

Some of these drawbacks can be avoided if we implement lists using pointers. We will do so in the next section.

12.4 POINTER IMPLEMENTATION OF LISTS

In this section, we will discuss lists implemented through pointers. We will also compare this with the static implementation of lists through arrays. Let us now go back to the example we used in the earlier section. Before we actually implement the list, let us discuss the concept through diagrams.

Each element of a linked list is called a **node**. For the List elements BLUE, RED, YELLOW, GREEN and ORANGE, we can form a linked List using pointers. The last node in the List points to NULL. The structure of such a List may be schematically shown in Fig. 2. We have indicated that the pointer in the last node points to the NULL pointer by a \. This is a singly

Fig. 2: A linked list.

linked List structure i.e. each of its elements have

- data and
- a pointer pointing to next element of the List

For an empty List the Head points to NULL.

The primary advantage of Linked Lists over arrays is that Linked Lists can grow and shrink in size during their lifetime. In particular, their maximum size need not be known in advance. In practical applications, this often makes it possible to have several data structures share the same space, without paying particular attention to their relative size at any time.

A second advantage of Linked Lists is that they provide flexibility in allowing the items to be rearranged efficiently. This flexibility is gained at the expense of quick access to any arbitrary

item in the List. This will become more apparent below, after we have examined some of the basic properties of Linked Lists and some of the fundamental operations we perform on them.

Now, this explicit representation of the ordering allows certain operations to be performed much more efficiently than would be possible for arrays. For example, suppose that we want to move the GREEN to the beginning of the List. In an array, we would have to move every item to make room for the new item at the beginning; in a linked List, we just change three links.

Let us now to write a C program that create a linked list with the colours BLUE, RED, YELLOW and GREEN. We will do this in stages.

Example 3: We start by defining a **self referential** structure called node. This is called a self referential structure because the second component of the structure is a pointer to another structure of the same type. Instead of referring to this again and again as **struct** node, we use the **typedef** statement to create a new type called Node. Here is a small program that creates a linked list containing one node. Let us examine this program.

```
1  /*Program 12.2. Example node creation in
2  Linked lists. File name:unit12-myfirstll.c*/
3  #include<stdio.h>
4  #include<string.h>
5  #include<stdlib.h>
6  #define MAX_WD 20
7  struct node{
8          char colour[MAX_WD];
9          struct node *next;};
10 typedef struct node Node;
11 Node *CreateNode(char *colour);
12 void Error(char *message);
13 int main()
14 {
15     Node *head = NULL;
16     head=CreateNode("RED");
17     printf("%s", head->colour);
18         return 0;
19         }
20 Node *CreateNode(char *Colour)
21 {
22     Node *ptr;
23     if((ptr = malloc(sizeof(Node)))){
24         ptr->next= NULL;
25         strcpy(ptr->colour,Colour);
26         return(ptr);
27     }
28     else{
29         Error("Unable to create node!");
30         return(ptr);
31     }
32 }
33 void Error(char *message)
34 {
35     fprintf(stderr,"Error! %s\n",message);
36 }
```

In lines 7 to 9 of this program we define a self referential structure called node. The first component of this structure is a character array of size MAX_WD which we have **#define**d to be 20 earlier in line 6.

Line 20 calls the function CreateNode() and assigns the value returned by it to the pointer head. As we will see, CreateNode() returns a pointer to a newly created node and head will also point to this node. The function CreateNode() creates a node with the colour

passed to it as argument. The function calls `malloc()` to allocate memory and assigns the pointer returned by `malloc()` to the local variable `ptr` which is a pointer to `Node`. If the `malloc()` is unable to allocate memory it will return NULL pointer. In this case, the value of the expression `ptr=malloc(sizeof(Node))` will be 0 and so the **else** part part of the **if** statement will print an error message. Otherwise, the line `ptr->next=NULL` initialises the node to NULL and copies the name of the colour, which is a string, on to `colour` part of the `Node`. Line 26 returns the pointer to the main program. The `printf()` statement in line 17 checks if the node has been created successfully. The situation after creation of the node is given in Fig. 3. Throughout this example, we will say 'the node RED', 'the node BLUE' etc

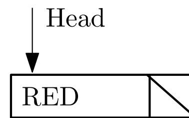


Fig. 3: A linked list with one node.

instead of saying 'the node containing RED', 'the node containing BLUE' etc.

We will now see how to insert a node at the beginning of the list. We will now insert a new node containing the colour BLUE at the beginning of the list. Here are the steps involved(We have shown the steps in Fig. 4 on the next page.):

a) Create a new node using `CreateNode()` function and make `new` point to the same location as the pointer returned by `CreateNode()`

```
new = CreateNode("BLUE");
```

b) Make the pointer in the the `new` point to where `head` points, i.e. the node RED.

```
new->next = head;
```

c) Make the pointer `head` point at the `new` node, making it the first node.

```
head = new;
```

Let us now add the colour GREEN at the end of the list.

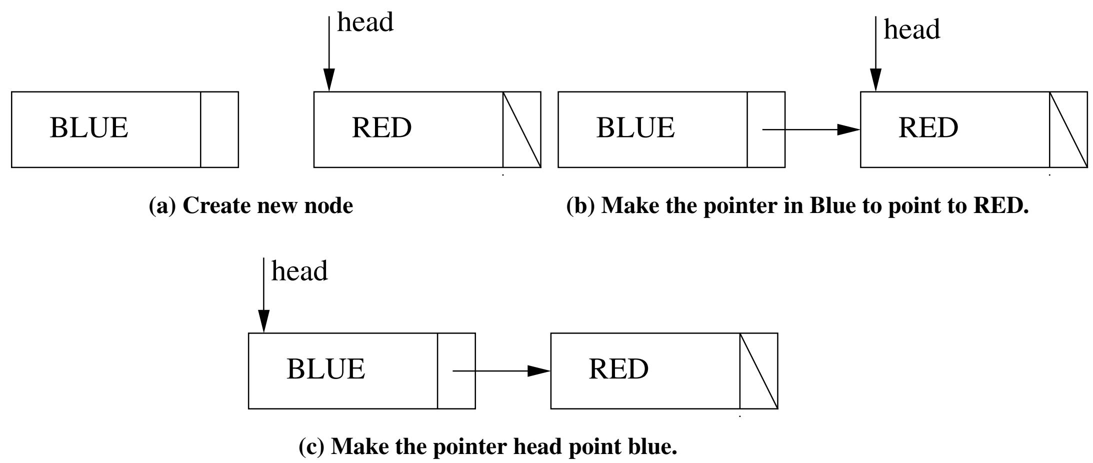
a) We need a pointer to the last node. We make pointer `current` point to `Node` and initially make it point to the same node as `head`. See Fig. 5a on the following page.

```
current = head;
```

b) Then, we advance `current` till it points to the last node in the list through a **while** loop:

```
while (current->next != NULL)
    current = current->next;
```

Initially, `current` points to the node containing BLUE. So, `current->next` is a pointer to where the pointer in the node containing BLUE points; this is the node containing RED. Since this isn't NULL, the statement in the `which` loop is executed. The effect of the statement '`current=current->next;`' is to make `current` point to the where the pointer in the node corresponding to BLUE points; this is the node containing RED. But, the pointer in the node corresponding to RED points to the NULL pointer and so the condition in the **while** loop is not satisfied. Now, we have the pointer `current` pointing to the last node, namely the one corresponding to RED.

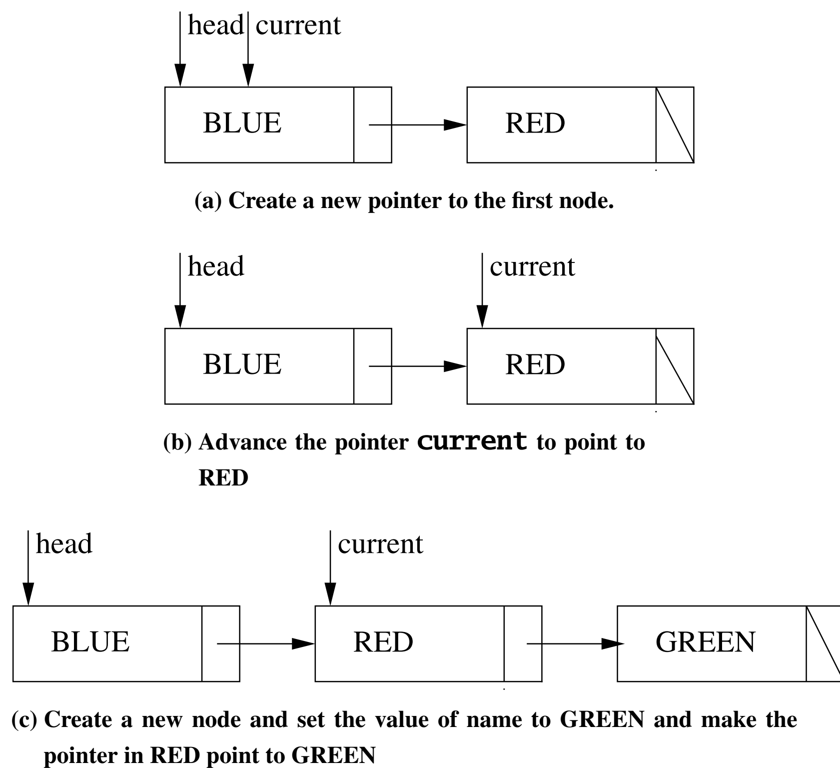


(a) Create new node

(b) Make the pointer in Blue to point to RED.

(c) Make the pointer head point blue.

Fig. 4: Inserting a new node at the beginning of a list.



(a) Create a new pointer to the first node.

(b) Advance the pointer `current` to point to RED

(c) Create a new node and set the value of name to GREEN and make the pointer in RED point to GREEN

Fig. 5: Adding a new node at the end of a list.

c) As before, we create a new node, set the first component of the node as GREEN.

```
new = CreateNode("GREEN");
current->next = new;
new->next = NULL;
```

Then, we use the statement `current->next=new` to make the pointer in the node corresponding RED point to the newly created node. Finally, we make the pointer in the newly added node point to NULL since this is the last node.

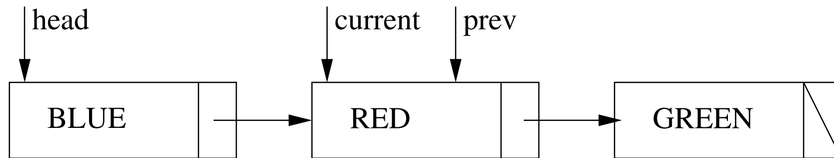
Let us see how we can add a node in the middle of a linked list. Let us add a node corresponding to YELLOW after RED. Here are the steps involved. See Fig. 6.

1) Make `current` point to the first node.

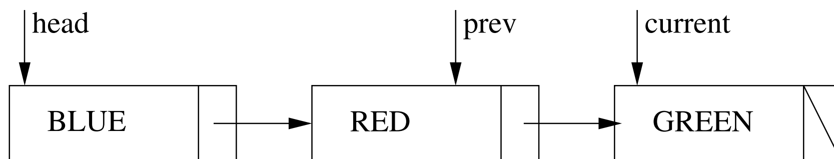
2) Advance `current` to point to the node containing RED. Use a new pointer called `prev`. Make `prev` and `current` point to the same node.

3) Advance `current` to point to the next node containing GREEN.

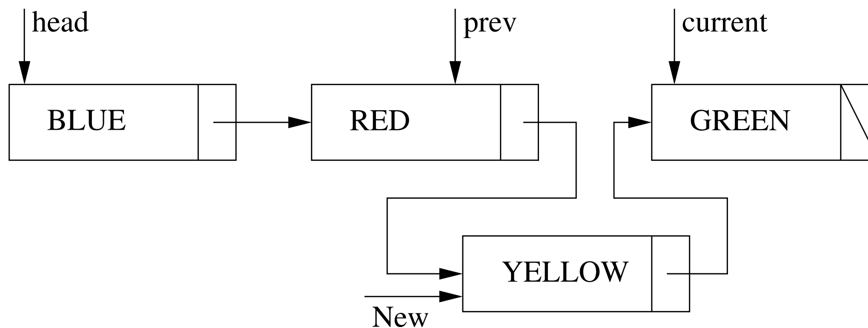
(a) Make `current` point to the first node.



(b) Make `current` and a new pointer `prev` point to the node after which we want to insert the new node.



(c) Advance `current` to point to the next node.



(d) Make the pointer in RED point to the new node and the pointer in the new node point to GREEN

Fig. 6: Adding a node in the middle.

4) Create a new node. We have to make this pointer in this new node point to the node containing RED. The pointer `current` points to GREEN. So, the statement `new->next=current;` makes `new->next` also point at GREEN. Then, we have to make the pointer in the node containing RED to point to the new node. The pointer `prev` points to RED; so we can achieve this using the statement `prev->next=new;`.

Here is the complete listing of the C program.

```
1  /*Program to demonstrate the creation and insertion of nodes
2  in a linked list.  File name:uni12-myfirstll-2.c*/
3  #include<stdio.h>
4  #include<string.h>
5  #include<stdlib.h>
6  #define MAX_WD 20
7  struct node{
8          char colour[MAX_WD];
9          struct node *next;};
10 typedef struct node Node;
11 Node *CreateNode(char *colour);
12 void Error(char *message);
13 void printlist(Node *);
14 int main()
15 {
16     Node *head, *new, *current, *prev;
```

```
17      head = NULL;
18      head = CreateNode("RED");
19      new = CreateNode("BLUE");
20      new->next = head;
21      head = new;
22      /*Insert node GREEN at the end. */
23      current = head;
24      while (current->next != NULL)
25          current = current->next;
26      new = CreateNode("GREEN");
27      current->next = new;
28      new->next = NULL;
29      printlist(head);
30      /*Insert YELLOW after RED */
31      current = head;
32      /*current points to BLUE */
33      current = current->next;
34      /*Current points to RED now */
35      new = CreateNode("YELLOW");
36      prev = current;
37      /*save the value of current in prev.
38         prev points to RED now. */
39      current = current->next;
40      /*current points to GREEN now. */
41      new->next = current;
42      /*Make the pointer in the new node also
43         to  point at GREEN */
44      prev->next = new;
45      /*The pointer in RED points to
46         the new node containing YELLOW.
47      Print the list for checking.*/
48      printlist(head);
49      return 0;
50  }
51  Node *CreateNode(char *Colour)
52  {
53      Node *ptr;
54      if((ptr = malloc(sizeof(Node)))){
55          ptr->next= NULL;
56          strcpy(ptr->colour,Colour);
57          return(ptr);
58      }
59      else{
60          Error("Unable to create node!");
61          return(ptr);
62      }
63  }
64  void Error(char *message)
65  {
66      fprintf(stderr,"Error! %s\n",message);
67  }
68  void printlist(Node *ptr)
69  {
70      while(ptr->next){
71          printf("%s\n",ptr->colour);
72          ptr=ptr->next;
73      };
74      printf("%s\n",ptr->colour);
75      printf("\n");
76  }
```


* * *

Here is an exercise to test your understanding of the earlier material.

E2) Write a program that creates a linked List with entries 1, 2, 3, 4 and 5.

The aim of the example above was to help you understand the process of creating nodes. Obviously, the procedure will be tedious if we want to create a list with 100 nodes. In practice, we will need a function that creates and insert a new node. We will see how to do this in the next example.

Example 4: In this example, let us write functions for inserting new nodes in a list. First let us write a function that adds a node at the beginning of an existing linked list.

```
void add_node(Node **headref, char cname[MAX_WD])
{
    Node *new = CreateNode(cname);
    new->next = *headref;
    *headref = new;
}
```

You may have noticed `**headref`, a pointer to pointer! Why do we need to do this? Remember that, in C, if we want a function to change an object, we have to pass a pointer to the object that we want to change. Here, when we add a new node, `head` will point to this new node instead of wherever it was pointing before. Since, we want to change the contents of the pointer variable `head` we have to pass a pointer to `head`, not just `head`. So, we will pass the value of `head` by the statement `add_node(&head)`. So, the function must accept a pointer to a pointer variable.

Let us now write a function that inserts a new node at the nth position, regardless of where it is, in the beginning, in the end or in the middle. The function will take the pointer to the `head` pointer, the colour of the new node and the position where we want to insert the new node as the arguments. The function should do the following:

If *position* = 0 call `add_node()` to insert a node at the beginning;
else
Create a new pointer called `current` and advance it point to *position*;
If `current` does not point to the last element of the list, pass the pointer `current` to `add_node_mid()`;
elseif
Pass the pointer `current` to `add_node_end()`.

Here is the function that inserts a node in the middle:

```
void add_node_mid(Node **headref, char name[MAX_WD])
{
    Node *current, *new, *prev;
    current = *headref;
    prev = current;
    current = current->next;
    new = CreateNode(name);
    prev->next = new;
    new->next = current;
}
```

Here is a function that inserts a node in the end.

```
void add_node_end(Node **headref, char name[MAX_WD])
{
    Node *current, *new;
    current = *headref;
    new = CreateNode(name);
```

```
    new->next = NULL;
    current->next = new;
}
```

Here is a function that checks the position where we want to insert the node and call appropriate functions to insert them in the correct position.

```
void ins_node(Node **headref, char cname[MAX_WD], int num)
    /*Function that inserts the colour cname
       after num nodes */
{
    Node *current;
    int count = 1;
    current = *headref;
    if (num == 0)
        add_node(headref, cname);
    else
        {
        while (count < num)
            {
            if (current->next == NULL)
                {
                printf("\nThere are only %d nodes.\n", count);
                printf("Cannot insert node after position %d.",
num);
                exit(1);
                };                      /*End of if */
            current = current->next;
            count += 1;
            };                  /*End of while */
        /*check if num is the last node. */
        if (current->next == NULL)
            add_node_end(&current, cname);
        else
            add_node_mid(&current, cname);
        }
}
```

***

Here are some exercises to test your understanding of the previous example.

E3) Write a function that returns the number of nodes in the list we created in the previous example.

E4) Write a function that will delete the node after the nth node.

Let us now discuss an application of linked list, namely addition of two polynomials. Linked lists are convenient when we have to add two polynomials of high degree with many coefficients zero. For example, consider the problem of adding two polynomials. $x^{25} - 12x^{13} + 5x^7 - 8x^3 + x + 1$ and $x^{27} - 12x^{14} + 7x^{10} + 7x^4 + 2x^3 + x + 9$. If we use arrays, we will need three arrays, one of size 27, another of size 25 and a third one of size 27 to hold the answer. Further, there are many terms which are 0 in both the polynomials, yet we have to take them into account. We will see how to use linked lists to add these polynomials.

Example 5: Let us first define the structures that will hold the terms. Note the use of **typedef** in this.

```
#include<stdio.h>
#include<stdlib.h>
typedef struct poly {
```

```
    int coeff;
    int deg;
    struct poly *next;
} Poly;
```

Let us now write a function in C that adds a term of a given degree and given coefficient to an existing polynomial. The function assumes that polynomial is constructed in such a way, starting from the head node, the terms are arranged in descending order with the head pointing to the highest degree term. Given a term, it checks if the polynomial has any terms. If it doesn't have any terms, it adds a new term. Otherwise, it inserts the term at an appropriate place. If there is already a term with given degree, it adds the coefficient of the new term to the existing term. Here is the full listing of the function with copious comments. Please go through it carefully.

```
void add_term(Poly ** poly1, int coeff, int degree)
{
    Poly *new, *current, *prev;
    current = *poly1;
    new = malloc(sizeof(Poly));
    new->coeff = coeff;
    new->deg = degree;
    new->next = NULL;
    /*The polynomial has no terms */
    if (*poly1 == NULL)
        *poly1 = new;
    else
/* If the term being added has higher degree than the
 highest degree term in poly1, add the term in the beginning */
    if (current->deg < degree) {
        new->next = current;
        *poly1 = new;
    } else {
/* Advance current till it points to the last term(node) or
 to a term of degree not greater than the degree of the terms
 we want to insert. */
        while (current->deg > degree && current->next != NULL)
            current = current->next;
/*If we have reached the last term of poly and the degree
of the last term is greater than that of the term we want
to insert, add the term at the end. */
        if (current->next == NULL && current->deg > degree)
            current->next = new;
        else {
/*If we have found a term with the same degree as the term we
want to insert, merely add the coefficient of the term we want
to insert to coefficient of the existing term of the same degree; */
            if (current->deg == degree) {
                current->coeff = current->coeff + coeff;
            } else {
/*otherwise add a new term. */
                prev = current;
                current = current->next;
                prev->next = new;
                new->next = current;
            };
        };
    };
}
```

* * *

Here are some exercises for you to check your understanding of the representing polynomials using linked lists.

E5) Using the function in the example above, write a program that represents the polynomial $x^{22} - 28x^{17} + 12x^{13} - 34x^{11} + x^8 + x^5 + x^2 + x + 1$ as linked list.

E6) Using the function in the example above, write functions for adding, subtracting and multiplying two polynomials.

12.4.1 Storage of Sparse Arrays using Linked List

It is often necessary to deal with large arrays in which many of the element has a zero value. Such arrays are called sparse arrays (see Unit 11 of this Block). We have already discussed one

$$
\begin{array}{cccccccc}
0 & 0 & 3.5 & 0 & 0 & 0 & 0 & 0 \\
0 & 1.2 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 5.5 \\
0 & 0 & 0 & 0 & 2.5 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 6.7 & 0
\end{array}
$$

Fig. 7: A sparse array.

of the methods of storing these sparse arrays, i.e. by using a 3-tuple for each element. Often non zero elements need to be added or deleted. That requires a lot of data movement in a static storage system. An improvement over this would be to store these non zero element as a linked List of 3 tuples instead of using an array. Fig. 8 illustrates the linked List for the sparse array given in Fig. 7. As you can see, there is node for each non-zero element. In each node, the first element is the row number, the second element is the column number, the third element is the non-zero element and the last component is a pointer to the next node. For example, the first row has a non-zero element 3.5 in the third column. The first node represents this element. We

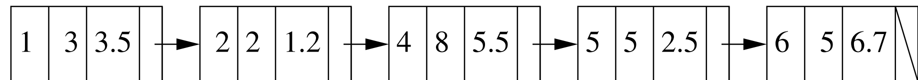


Fig. 8: Linked list for sparse array.

close the section here. In the next section, we discuss doubly linked lists.

12.5 DOUBLY LINKED LISTS

In the Linked Lists discussed in previous section, the traverse the List in one direction. In many applications it is required to traverse a List in both directions. This 2-way traversal can be realised by maintaining two link fields in each node instead of one. We call such a structure a **Doubly Linked List**. Each element of a doubly linked List structure has three fields

- data value
- a link to its successor
- a link to its predecessor

The predecessor link is called the **left link** and the successor link is known as the **right link**.

Here is the definition of the node of a doubly linked list that holds integer data.

```
typedef struct dlnode {
    int data;
    struct dlnode *left;
    struct dlnode *right;
} Dlnode;
```

Since there are nodes in both the directions, the traversal of the List can be in any direction. We have a structure as given in Fig. 9. Note that the left link of leftmost node and right link of rightmost node are NULL.

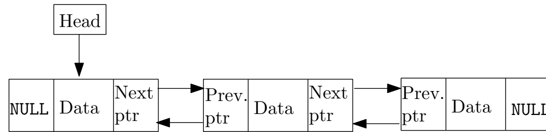
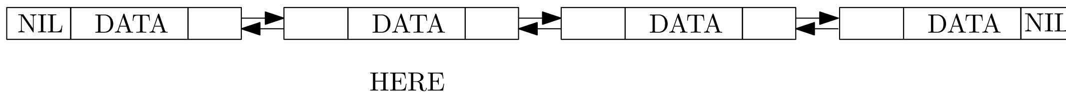


Fig. 9: A doubly linked list.

Insertion of a Node

To insert a node into a doubly linked List to the right of a specified node, we have to consider several cases. These are as follows:

1. If the List is empty, i.e. the left and right link of specified node pointed to by say variable HEAD are NULL. An insertion in this node is simply making left and right pointers point to the new node and left and right pointers of new node to be set to NULL.
2. If there is a predecessor and a successor to the given node. In such a case we need to readjust pointers of the specified node and its successor node. The procedure is shown in Fig. 10.



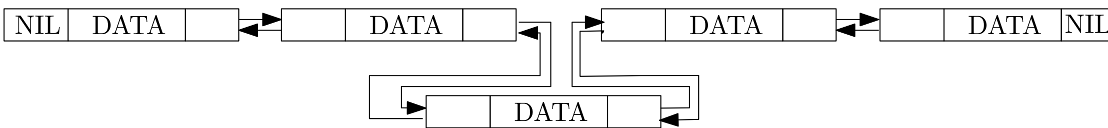


Fig. 10: Insertion in doubly linked list.

3. The insertion is to be done after the right most node in the List. In such a case only the right link of the specified node i.e. the rightmost node is to be changed.

Here is a function that creates a new node and returns a pointer to it.

```
Dlnode *getnode()
{
    Dlnode *p;
    p = malloc(sizeof(Dlnode));
    return (p);
}
```

Here is a function that inserts a node **before** the node at pos.

```
void ins_dlnode_lt(Dlnode ** dlptr, int x, int pos)
{
    Dlnode *current, *new, *prev;
    int count = 1;
```

```
    if (*dlptr == NULL) {
        printf("Unable to insert to the left.");
        exit(1);
    };
    if (pos == 1) {
/*When the node has to be inserted at the start of
        list.*/
        new = getnode();
        current = *dlptr;
        *dlptr = new;
        new->data = x;
        new->right = current;
        new->left = NULL;
    } else {
/*Advance the pointer.*/
        for (; count < pos; count++)
            current = current->right;
/*Create a new node.*/
        new = getnode();
        new->data = x;
/*prev stores the node to the left of the pos*/
        prev = current->left;
        current->left = new;
        new->left = prev;
        prev->right = new;
        new->right = current;
    }
}
```

Here is a function that inserts a node **after** the node at `pos` postion.

```
void ins_dlnode_rt(Dlnode ** dlptr, int x, int pos)
{
    Dlnode *prev, *current, *new;
    int count = 1;
    current = *dlptr;
    if (*dlptr == NULL) {
        if (pos > 1) {
            printf("Unable to insert new node.");
            exit(1);
        }
        if (pos == 1) {
            new = getnode();
            new->right = NULL;
            new->left = NULL;
            new->data = x;
            *dlptr = new;
        }
    } else {
        new = getnode();
        new->data = x;
        for (; count < pos; count++)
            current = current->right;
        prev = current;
        current = current->right;
        new->left = prev;
        prev->right = new;
        new->right = current;
    }
}
```

Here is a small program that illustrates the use of these functions.


```
#include <stdio.h>
```

```
#include <stdlib.h>
typedef struct dlnode {
    int data;
    struct dlnode *left;
    struct dlnode *right;
} Dlnode;
Dlnode *getnode();
void ins_dlnode_rt(Dlnode ** dlptr, int x, int pos);
void ins_dlnode_lt(Dlnode ** dlptr, int x, int pos);
int print_dllist(Dlnode * dlptr);
int main()
{
    Dlnode *head = NULL;
    ins_dlnode_rt(&head, 1, 1);
    print_dllist(head);
    ins_dlnode_lt(&head, 2, 1);
    print_dllist(head);
    ins_dlnode_rt(&head, 3, 2);
    print_dllist(head);
    ins_dlnode_lt(&head, -1, 1);
    print_dllist(head);
    return 0;
}
```

You may find it instructive to make diagrams like the ones we did in example 4 for these operations.

You may have noticed that the program uses a function to print the list. You may like to write one on your own. We have left it as an exercise to you. Try the following exercises.

E7) Can you guess what will be the contents of the list at the end of the program above?

E8) Write a function that prints the contents of the doubly linked list.

Another way of overcoming the disadvantages of singly linked List is a circular List. We discuss circular Lists in the next section.

12.6 CIRCULAR LINKED LIST

Another alternative to overcome the drawback of singly linked List structure is to make the last element point to the first element of the List. The resulting structure is called a **circularly linked List** (Fig. 11). A circularly linked List is a List in which the link field of the last element

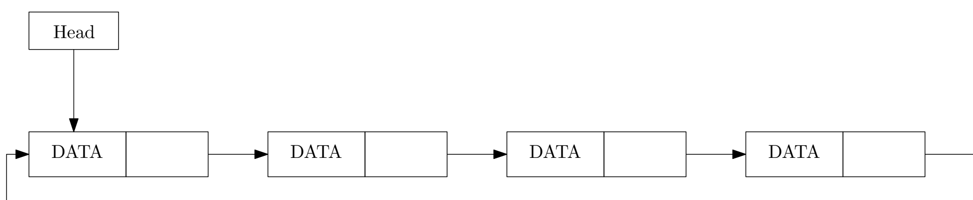


Fig. 11: A Circular list.

of the List contains a pointer to the first element of the List. The last element of the List no longer points to a `NULL` value.

The definition of circular list is similar to that of circular list. Insertion of nodes in the middle is similar. However, insertion at the end or in the beginning is different. We leave it to you to modify the functions that we wrote for singly linked list for a circular list.

E9) Modify the functions that we wrote for inserting nodes in a linked list for a circular list.

We conclude this section here. In the next section, we discuss storage allocation.

12.7 STORAGE ALLOCATION

Initially, the operating system determines the areas available for allocation to users. We will use the term 'node' to designate a Unit of the storage space.

Nodes are allocated to users in response to their requests. The number of nodes and the size of each node are decided keeping the following factors in mind.

1) Contiguity of space improves performance, especially for sequential access.
2) Having a large number of nodes leads to greater storage management effort.
3) Having fixed size nodes can lead to wastage of space.

The trade off can be summarised as:

1) Large nodes provide contiguous space and hence improve performance. They should be variable in size to prevent excessive wastage of space.
2) Small nodes improve flexibility. Much space is not wasted but their management is complex.

Having once decided the number and size of the nodes we now turn our attention to the actual allocation of these nodes. We make the assumption that we have multiple nodes of varying sizes and we want a storage space of size M. This can be done in one of the following ways:

1. Best Fit Method

All available nodes are checked. Assuming the size of a node is represented by N.

Let $D = N - M$

The node whose value of N gives the least value for D is chosen and allocated.

The advantages of this method are obviously the minimal wastage of space.

The disadvantages are

- it involves searching all available nodes.
- it tends to increase the number of very small free blocks, when D not equal to 0.2.

2. First Fit Method

The available nodes are checked till we find one whose size N is greater than or equal to M, the requested size.

The advantage of this is obviously a smaller search among the available List of nodes. The disadvantage, like in the Best fit method could be that very small free blocks could be created. The way out of this is to fix a reasonable size C such that on getting the node of size NM,

If $N(N - M) <= C$

allocate the entire node of size N

Else

Allocate space of size M

reserve node of size (n-M) for further use.

12.8 STORAGE POOLS

The collection of all nodes available is the Storage Pool. We will now deal with the management of this pool. This can be done in one of the following ways:

1) **Bit Tables**
   This method uses an array containing one bit per node. It is generally used when all nodes are of the same size, usually 1 block. A bit value of '0' indicates that the corresponding node is free, and a value of '1' indicates that it has been allocated. A separate file mechanism is needed to indicated the nodes allocated to a specific file. The advantages of this system are that the table can be kept in core memory so that allocation and deallocation (i.e. setting the bits to '1' or '0') costs are minimal.

2) **Table of Contents**
   This uses a file per unit (device, file system etc.) to describe the space allocation for the unit. This file will have (typically) the following data for each node - its size, whether allocated or not, if allocated - the name of the file, owner's identification, date of creation etc.
   Like the bit table, this table of contents has to be searched to find free space for allocation. This problem may be overcome by keeping the records of free nodes in the 'Free Space Table of Contents'. Allocation then means getting a suitable node from the 'Free Space Table of Contents' and moving it to the 'Table of Contents' after suitable updating. Freeing a node implies the reverse process.

3) **Linked allocation**
   Nodes can be linked together to overcome the limitations of the above two methods. In this method, each node will have a link to the next node in the List. Initially all nodes will be part of a free nodes List. On allocation to a file, a node will be detached from the free space List and added to the allocated List for that file. When a node is deallocated, it is detached from the allocated List of the file and attached to the free nodes List.
   Linked Lists have been discussed in an earlier section. Thus we know Linked Lists can be singly or doubly linked depending upon the needs. This method provides an implicit gain in storage - in cases where Tables or files overlap, sharing common parts. The set of common nodes can be part of the allocation Lists of all the sharing tables.
   The advantages of linked allocation are directly related to the case of operations on Linked Lists. Simple insertion and deletion from a linked List implies simplicity in inserting/deleting from a file. Ease of combination of Lists implies ease of joining files.

12.9 GARBAGE COLLECTION

Deallocation of nodes can take place in two levels:

1) The application which claimed the node releases it back to the operating system.

2) The operating system calls storage management routines to return free nodes to the free space.

For example Deallocation as in (1) occurs in a C, program with the statement `free(x)` where `x` is space earlier allocated by a `malloc` call. (2) is usually implemented by the method of **Garbage Collection**. This requires the presence of a 'Marking' bit on each node. It runs in two phases. In the first phase, all non-garbage nodes are marked. In the second phase all non-marked nodes are collected and returned to the free space. Where variable size nodes are

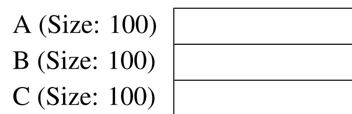
used it is desirable to keep the free space as one contiguous block. In this case, the second phase is called **Memory compaction**.

Garbage Collection is usually called when some program runs out of space. It is a slow process and its use should be obviated by efficient programming methods.

12.10 FRAGMENTATION, RELOCATION AND COMPACTION

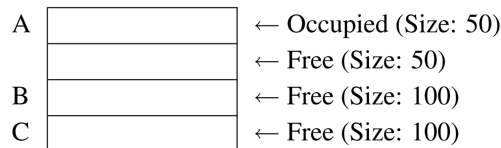
Fragmentation literally means splitting. We have seen from our discussions in Sec. 12.7 that the 'Best fit' and 'First fit' algorithms result in creation of small blocks of space. These constitute wastage of space as they can be used to satisfy only requests for small blocks. This can be illustrated by the following example:

Consider the following space divided into three nodes of size 100 each.

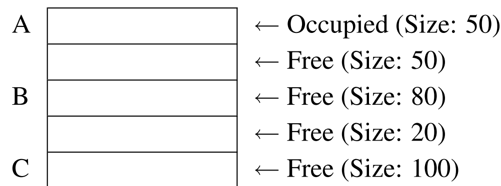


The following have to be allocated using first fit:

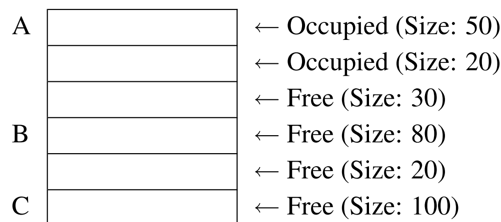
1) Size 50. This can be allocated from A. So, we get



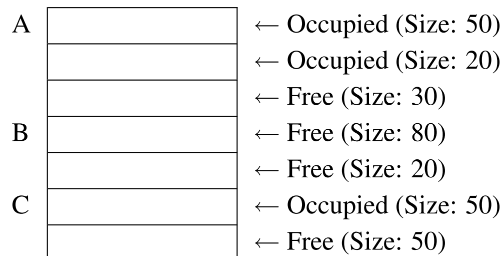
2) Size 80. This can be allocated from B.



3) Size 20. This can be allocated from A.



4) Size 50. This can be allocated from C.



5) Size 75

Now there is no contiguous block of size 75 though the actual space available is $30 + 20 + 50 = 100$

In this state of fragmentation, only requests of size ≤ 50 can be satisfied.

This calls for 'Relocation' of the allocated blocks A[0-70], B[0-80], and C[0-50], so that the free blocks A[70-100], B[80-100] and C[50-100] can be 'compacted' to form one free block of size 100. The relocation of B and C will result in a change in their address. This change must be reflected wherever B and C are used.

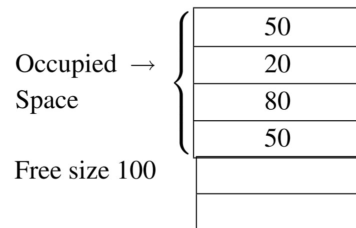


Fig. 12

Relocation and Compaction usually form the second phase of Garbage collection, A[0-70], B[0-80], and C[0-50] constitute the 'non-garbage' nodes or 'marked nodes' which are relocated and A[70-100], B[80-100], and C[50-100] are the garbage nodes which are 'compacted' to release space.

12.11 SUMMARY

In this Unit we dealt with List and Linked List structures storage management. List structures allow us to store individual data elements. In Linked Lists these elements are interconnected by pointers. Beyond singly linked structure, we find several variations of List structures, e.g. doubly Linked Lists and circular Linked Lists. The Linked Lists allow us great flexibility in organising our information. We also discussed a related concept, i.e. of storage management in this Unit.

Storage is available for allocation on peripheral devices and in the main memory. It is managed either by means of a bit-table, table of contents file or by Linked Lists. Space is allocated either using the best fit or first fit algorithms. Free space management is done by garbage collection which relocates fragmented free space and compacts it to get a continuous chunk of free space.

12.12 SOLUTIONS/ANSWERS

E1) Solution to i) is given in the listing below:

```
int print_array(char array[MAX_L][MAX_WD])
{
    int i;
    printf("\n");
    for (i = 0; i < last; i++)
        printf("%s\n", array[i]);
    return 0;
}
```

E3)
```
/*Function that counts the number of nodes.
File name: unit12-ans-ex3.c.*/
int cnt_node(Node *head)
{
    int count=0;
    for(count=0;head != NULL;count++)
        head=head->next;
    return(count);
}
```

E4)
```
/*Function that deletes node after the.
nth node File name: unit12-ans-ex-4.c.*/
void del_node(Node **headref, int num)
{
    int count;
    Node *current,*temp;
    current=*headref;
/*current now points to the head of the list.*/
    if(num==0){
/*We want to delete the first node.*/
        *headref=current->next;
/*Make head point at the second node.*/
        free(current);
/*Free the memory used by the first node.*/
    }
    else{
        for(count = 0;count < num-1 ;count++)
            current=current->next;
        temp=current->next;
        current->next=current->next->next;
        free(temp);
    }
}
```