UNIT 2: LISTS

Structure

2.0 INTRODUCTION

In Unit 1 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 of data objects of the same type. Real-life event 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, in process management, in operating systems, in editors etc.

We shall discuss lists, Linked Lists such as singly, double and circularly Linked Lists, and their implementations: Using arrays and using pointer. Utility of concept of Lists also lines in understanding and implement other data structure like stacks, queues, trees etc., which we shall discuss in subsequent units.

These lists also motivate a discussion of storage allocation in computers. Storage or space management is of vital importance. The first step in this is the accquisition of adequate storage space. This is done by estimating the amount of space required for all the following categories of data:

- Operating system
- Special software
- User program
- Data files
- Temporary files

- Scratch area

The storage thus acquired is managed by the operating system. This Unit will study storage management methods under the following heads:

- Storage allocation
- Storage pools
- Garbage collection
- Blocks, Retrieval, Fragmentation
- Block splitting, Compaction, Relocation

2.1 OBJECTIVES

At the end of this Unit, you shall 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 Pascal and 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.

2.2 BASIC TERMINOLOGY

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.

The next element to the head of the List is called its successor. The previous element to the tail (if it is not the head of the List) of the List is called its predecessor.

Clearly a head does not have a predecessor, and a tail does not have a successor. Any other element of the List has both one successor and one predecessor.

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.

2.3 STATIC IMPLEMENTATION OF LISTS

It 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 as given in figure 1.

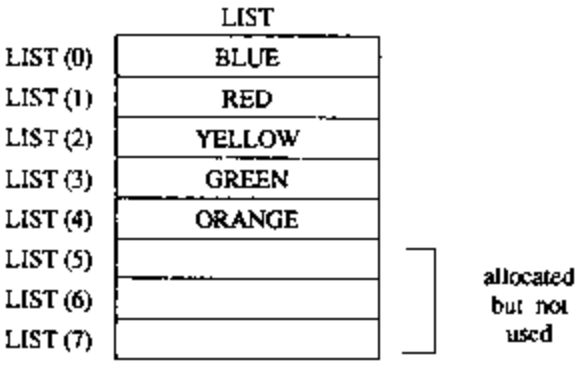


Figure 1: A List declared as an array

The elements are sequentially stored in LIST [0], LIST [1] . .... LIST [4]. LIST [5] through LIST [7] are allocated but not used.

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 given in Table 1.

	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	—	—	—	—
	—	—	—	—

TABLE 1

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 LIST [Last+l] does not exceed the value of LIST [LIST size- 1].

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

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

1. New_last = Last+ 1
2. If new last LIST size- 1, error: overflow
3. LIST [Last+1] = LIST [Last]
   LIST [Last] = LIST [Last-1]
   .....
   .....
   LIST [K+2] = LIST [K+1]
   LIST [K+1] = LIST [K]

   Shifting of elements from K+1 through Last+ 1
4. LIST [K] = element

   Inserting element at K_{th} position
5. tail=new_ last

You may try writing actual code for the insertion of an element in a List.

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

As an exercise you may write code for all the operations given in Section 2.2 using array implementation.

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 can not 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 requires a lot of movement of data, therefore, leading to inefficient and time consuming algorithms.

2.4 POINTER IMPLEMENTATION OF LISTS

For the list elements BLUE, RED , YELLOW , GREEN and ORANGE , we can form a linked List using pointers. The structure of such a list may be as below:

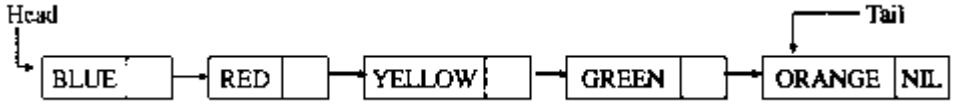


Figure 2: A Linked List

The Head and Tail are also pointers pointing to first and last element of the List respectively.

This is a singly linked List structure i.e. each of its elements have

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

For an empty List the Head and Tail pointer have the value NIL. When the List has one element, the Head and Tail point to the same.

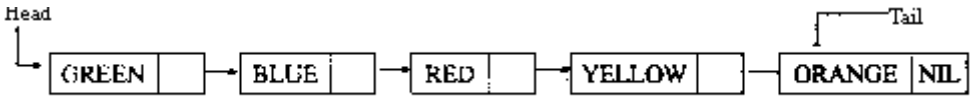
When the List has two elements the Head points to the first and the Tail points to the second element. The pointer of first element points to second element and that of second element holds a value NIL.

As an exercise write code for creation of a singly linked List by individually connecting its elements by pointers.

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



Charges : Head puts to Green, Green to Blue, Yellow to Orange

Figure 3

item to make room for the new item at the beginning; in a linked list, we just change three links, as shown in Figure 3.

The two versions shown in Figure 3 are equivalent; they are just drawn differently. We make the node containing GREEN point to BLUE, the node containing YELLOW point to ORANGE, and head point to GREEN. Even if the List was very long, we could make this structural change by changing just three links.

2.4.1 Insertion in a List

Let see how an Insertion algorithm works for inserting an element in a linked List after a specified element.

1. Check if List is empty, return if yes.
2. Search for the element after which the insertion is to be done.
3. Create a memory space for element to be inserted.
4. Assign the data value to its data field.
5. Assign its pointer to pointer of the element searched at step 2.
6. Assign pointer of the element searched at step 2 to this new element created.

Let us consider a List as given in Figure 2.

We want to insert PINK before YELLOW. We must create space for element with data PINK i.e. we will have

Now we need to assign the pointer of NEW_ELEMENT, which has a value NIL at present; to the value stored in structure containing data YELLOW i.e.

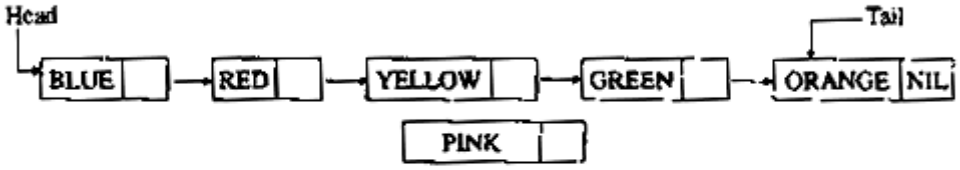


Figure 4 : Creating the node to be inserted

Now we need to delink YELLOW from RED and assign it to the new_element created. Therefore, we will have

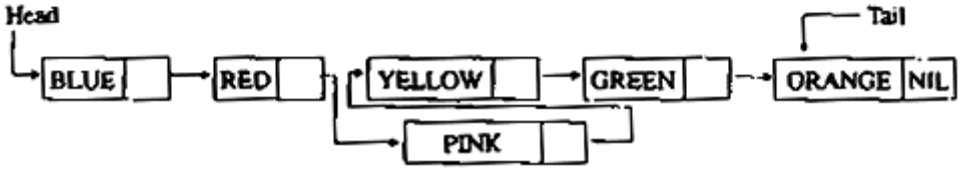


Figure 5: Insertion in a Linked List

We may notice that no data is physically moved, as we had seen in array implementation. Only the pointers are readjusted.

Query 1: Do you see any difficulty if the element is to be inserted before a specified element?

Query 2: What about insertions at the beginning or at the end of the linked List? Are these two insertions special cases of the insertions before and after an element respectively.

2.4.2 Deletion from a List

Deletion of an element from a single linked List requires

1. List to be checked for empty List.
2. Search for the element with key value same as the element to be deleted.
3. Its pointer to next element be saved and later assigned to next element of predecessor of the element searched at step
4. Dispose the memory space held by the deleted element.

The Code is left as an exercise.

There are other operations for which Linked Lists are not well-suited. The most obvious of these is "find the kth item" (find an item given its index): in an array this is done simply by accessing a[k], but in a list we have to travel through k links. Another operation that is unnatural on Linked Lists is 'find the item before a given item'. If all we have is the link to GREEN in our sample list in Figure 2, then the only way we can find the link to YELLOW is to start at head and travel through the list to find the node that points to GREEN. As a matter of fact, this operation is necessary if we want to be able to delete a given node from a linked list: how else do we find the node whose link must be changed? In many applications, we can get around this problem, by re-designing the deletion operaflon to be "delete the next node". A similar problem can be avoided for insertion by making the insertion operation "insert a given item after a given node" in the List.

Implementation

To illustrate how basic linked List primitives might be implemented, we begin by precisely specifying the format of the List nodes and building an empty List, as follows:

```
struct node
(int key; struct node *next; };
struct node *head, *z;
head = new node; z = new node;
head-next = z; z-next = z;
```

The struct declaration specifies that Lists are made up of nodes, each node containing an integer and a pointer to the next node on the List. The key is an integer here only for simplicity, and could be any type - the next pointer is the key to the LIST. The asterisks indicate that the variables head and z are declared to be pointers to nodes. Nodes are actually created only when the built-in function new is called. This hides a complex mechanism that is intended to relieve the programmer of the burden of "allocating" storage for the nodes as the List grows. We discuss this mechanism in some detail below. The "arrow" (minus sign follows by a greater-than sign) notation is used in C to follow pointers through structures. We write a reference to a link followed by this symbol to indicate a reference to the node pointed to by that link. Thus, the above code creates two new nodes referred to by head and z and sets both of them to point to z.

To insert a new node with key value v into a linked List at a point following a given node t, we create the node (x = new node) and put in the key value (x-key = v), then copy t's link into it (x-next = t-next) and made t's link point to the new node (t-next = x).

To extract the node following a given node t from a linked List, we get a pointer to that node (x = t-next), copy its pointer into t to take it out of the List (t-next = x-next), and return it to the storage allocation system using the built-in procedure delete, unless the List was empty (if (x!=z) delete x).

2.4.3 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 1 of this Block). We have already discussed

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

one 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. Figure 7 illustrates the linked List for the sparse array given in Figure 6.

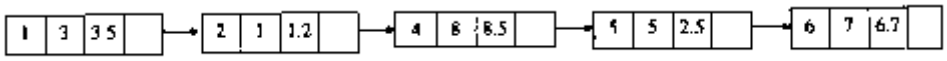


Figure 7: Linked List for Sparse Array

You may find it useful to develop algorithms/programs for the following:

1. Reversing a singly Linked List.

2. Concatenate two Linked Lists into one linked List.

3. Split a linked List into two Linked Lists such that the first List contains only elements where data is less than a given value and the second List contains the elements where data is greater than a given value.

2.5 DOUBLY LINKED LISTS

In the Linked Lists discussed in previous section, we 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 realized 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.

Thus the traversal of the List can be in any direction. We have a structure as given in Figure 8. Note that the left link of leftmost node and right link of rightmost node are NIL.

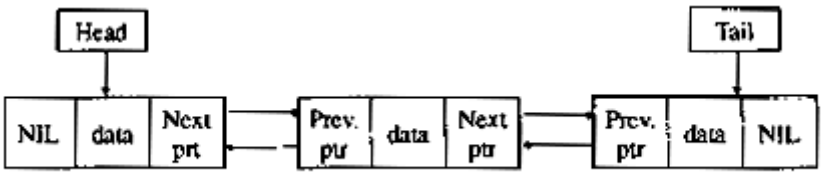


Figure 8: A Doubly Linked List

Inserting 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 HERE are nil. An insertion in this node is simply making left and right link point to the new node and left and right link of new node to be set to nil.

2. If there is a predecessor and a successor to the given node. In such a case we need to readjust links of the specified node and its successor node. The procedure may be depicted as given in Figure 9.

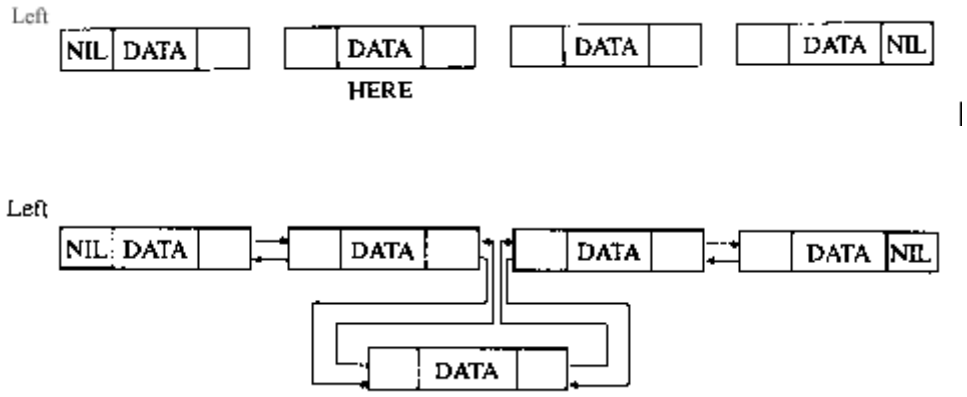


Figure 9 : Insertion in Doubly Linked List

3. If 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.

Deletion of a Node

Similar to insertion of a node, we have to consider several cases for deletion of a node.

2.6 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 (Figure 10).

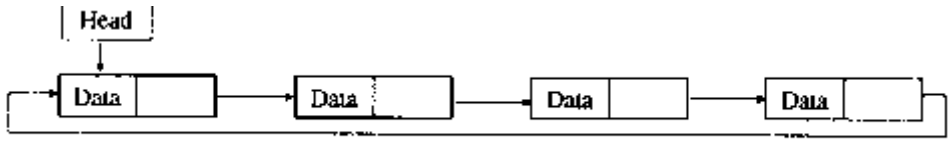


Figure 10 : A Circular List

A circularly linked List is a List in which the link field of the last element of the List contains a pointer to the first element of the List. The last element of the List no longer points to a NIL value.

Exercises:

1. Show how Linked Lists can be used to represent the following polynomial

(i) $5x^4 + 7x^2 + 9x + 3$

(ii) $9x^3y + 2x^2y - 1lx + 3y^3$

2. Develop algorithms to add two polynomials

(1) in one variable

(2) in two variables

Check Your Progress 1

Question 1: What are the advantages of circularly linked List as compared to singly linked List.

Question 2: Write an algorithm and a program to count the number of elements in a circularly singly linked List.

2.7 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 tradeoff can be summarized 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.

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.

2.8 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 '0' 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 updations. 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 al 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.

2.9 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.

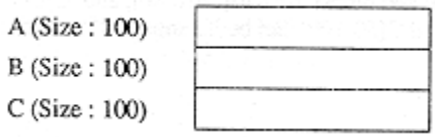
For example Deallocation as in (1) occurs in a C, program with the statement 'free(x)'where x is space earlier allocated by a 'malloc' call. (2) is usually implemented by the method of **Garbage Collection.** This requires the presence of a 'Marking' bit on each node. It runs in two phases. In the first 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.

2.10 FRAGMENTATION, RELOCATION AND COMPACTION

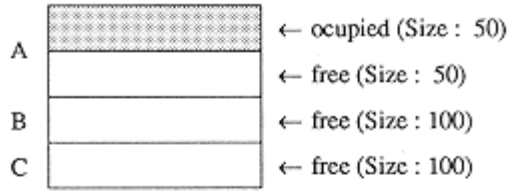
Fragmentation literally means splitting. We have seen from our discussions in 4.2 that the 'Best fit'& 'First fit' algorithms result in creation of small blocks of space. These constitute wastage of space as they can be used to satisfy only requests for small blocks. This can be illustrated by the following example:

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

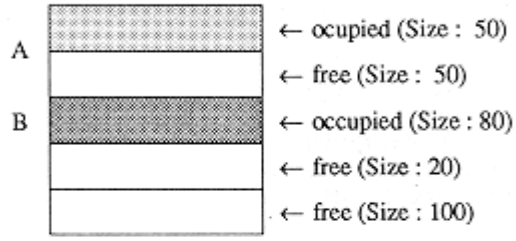


The following have to be allocated using First fit:

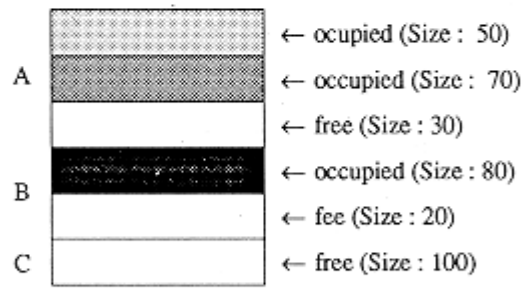
1. Size 50
This can be allocated from A. So we get



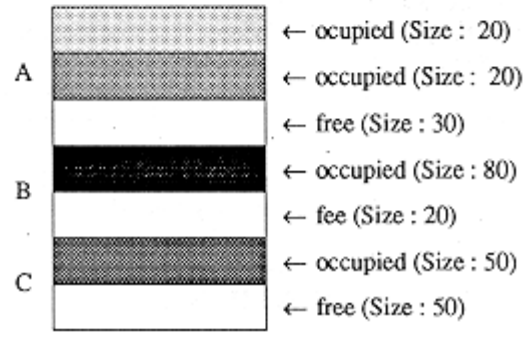
2. Size 80
This can be allocated from B.



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.

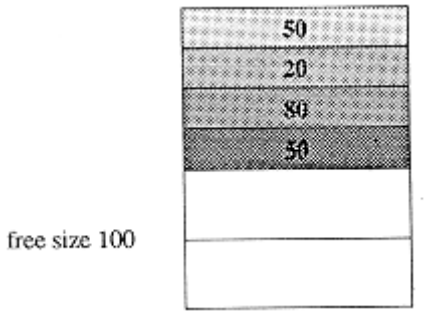


Fig. 6

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.

Relocation & 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[70-1001, B[80-100], and C[S0-100] are the garbage nodes which are 'compacted' to release space.

2.11 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 contiguous chunk of free space.

Check Your Progress 2

Question 1: Consider the following declaration

```
Struct   list_node
{ int data;
  struct list_node      *link;
 };

  struct list_node     *head, * scan,* item
```

(i) Write a segment of code to delete the first element of the Linked List printed by head.

(ii) Write a code to insert a value N after the last node in the Linked List.

MODEL ANSWERS

Check Your Progress 1 :

No Model answer is given.

Check Your Progress 2

Answer 1: No model answer is given.