CS8391 – DATA STRUCTURES

II YEAR / III SEMESTER

R2017

UNIT – I & II

LINKED LISTS, STACK, QUEUE

Data Structure :-

Data Structure is a way of organizing the data, which also includes several operations to perform on that data.

Example:-

a) List – used in the implementation of File Allocation Table (FAT), Process Control Block (PCB) etc.

b) Stack – Normally every Program will use Stack data Structure implicitly, eventhough the Programmer has not declared it explicitly. All the data's Processed during the Program execution will use Stack.

c) Queue – many Task will be waiting for getting the attention of C.P.U. Queue is used to order the task as they arrive before they get the attention of C.P.U.

d) Trees – Implementation of Directory Structures is done by using tree data structure.

e) Graph – used in Networking and mapping of tasks to Processors in multiprocessor platforms.

Abstract Data Type (ADT):-

a) An Abstract Data Type (ADT) is a set of operations.

b) Objects such as Lists, Sets, Graph etc, along with their operations, can be Viewed as Abstract Data Types (ADT's). (Just as Integers, Reals, and Boolean are data types).

c) The basic idea is that the implementation of these operations is written once in the program, and any other part of the program that needs to perform an operation on the ADT can do so by calling the appropriate function.

Types OF Data Structures:-

Data Structures are classified into two categories,

a) Linear

b) Non-Linear.

1. A Data Structure is said to be linear if its elements form a sequence. (Example:- Arrays, Linked Lists)
2. Elements In Non-Linear Data structure do not form a Sequence. (Example:- Trees, Graphs)
3. There are two ways of representing linear data structures in memory.
   i) To have the linear relationship between the elements by means of Sequential memory locations. Such linear structures are called Arrays.
   ii) The other way is to have the linear relationship between the elements represented by means of pointers or links. [illegible] are called Linked Lists.

Arrays :-

a) An Array is a Linear Data structure which has a Finite collection of similar elements stored in adjacent memory locations.

b) An Array Containing 'n' number of elements is referenced using an index that Varies from 0 to n-1. The number of elements in the array is called its range.

c) Arrays are useful when the number of elements to be Stored is fixed. They are easy to traverse, Search and Sort.

d) An Array is further categorized as a one-dimensional array and multi-dimensional array. A multi-dimensional array can be a 2-D array, 3-D array, 4-D array, etc.

e) There are several operations that can be performed on an Array,

i) Traversal - Processing each element in the array.
ii) search - Finding the location of an element with a given Value.
iii) Insertion - Adding an New element to an array.
iv) Deletion - Removing an element from an array
v) Sorting - Organizing the elements in some order.
vi) merging - Combining two arrays into a Single array.
vii) Reversing - Reversing the elements of an array.

Elements in an array with their indices:-

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
34	2	7	-7	12	9	-8	72

Advantages of Arrays:-

a) Arrays are Simple to understand.

b) Elements of an array are easily accessible.

Dis-Advantages of Arrays:-

a) Arrays have fixed dimension. once the size of an array is decided it cannot be increased or decreased during execution.

b) Array elements are always stored in contigous memory locations. At times it might so happen that enough Contiguous locations might not be available for the array we are trying to create. Even though the total space requirement of the array can be met through a Combination of non-Contiguous blocks of memory, we would still not be allowed to create the array.

c) operations like insertion of a new element in an array or deletion of existing element from the array are tedious. Because during insertions or deletions each element after the specified position has to be shifted one position to the

Structure

a) A structure is a constructed data type which is a method for packing data of different types.

b) A structure is a convenient tool for handling a group of logically related data items.

c) The general format of a structure definition is as follows,

```
struct tag-name
{
    data_type  member 1;
    data_type  member 2;
    ...        ...
    ...        ...
};
```

Difference between Structure and Union:-

1. In structures, each member has its own storage location, whereas all the members of a union use the same location.

2. Structures can handle many member at a time, but union can handle only one member at a time.

Pointers :-

a) A Pointer is a variable that Contains an address which is a location of another variable in memory.

b) There are number of reasons for using Pointers.

i) A pointer enables us to access a variable that is defined outside the function.

ii) Pointers are more efficient in handling the data tables.

iii) Pointers reduce the length and Complexity of a Program.

iv) Pointers Increase the execution Speed.

v) The use of a pointer array to Character Strings results in saving of data Storage Space in memory.

Dynamic memory Allocation :-

a) The Process of allocating memory at run time is known as Dynamic memory allocation.

b) There are Four Library routines known as "memory management Functions" that can be used for allocating and freeing memory during Program execution.

i) malloc - Allocates requested size of bytes and returns a pointer to the First byte of the allocated Space.

ii) calloc - Allocates Space for array of elements, initializes them to zero and returns a pointer to the memory.

iii) free - Frees Previously allocated Space.

iv) realloc - modifies the Size of Previously allocated Space.

LIST ADT

a) A List is a series of related data items. It can be an ordered Collection of integer items like 30, 40, 75, 93, 87. (or) it can be a Collection of Structures like,

```
Struct Student
{
    int age;
    char name[40];
    int mark;
};
```

with each structure containing a set of related data items.

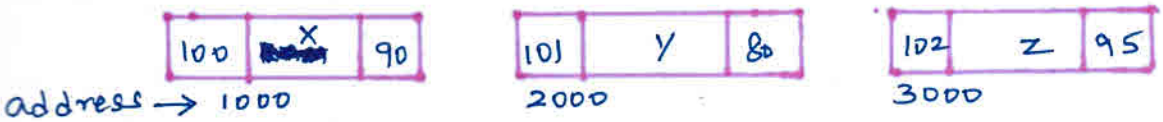
LINKED LISTS

a) A Linked List is a series of data items with each item containing a pointer giving the location of the next item in the list.

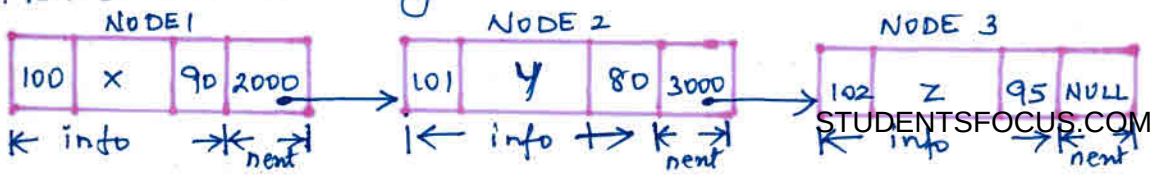
b) The data items need not necessarily be stored in Contiguous memory locations.

Example:

Three Items in a List

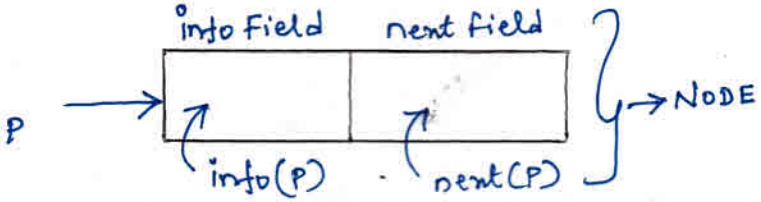


A Linked List Containing three items



c) In the Linked List representation, we have added an address with each item. (i.e) the first item contains the address 2000 which denotes the memory location of the second item and the second item contains the address 3000 which denotes the memory location of the third item. The third item contains a special address value 'NULL' meaning that it does not have a link to other items.

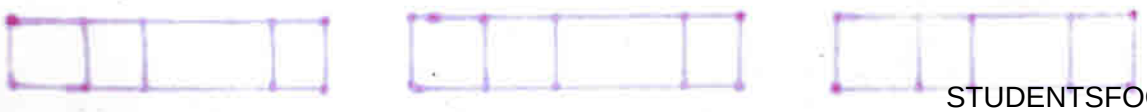
KEY TERMS ASSOCIATED WITH LINKED LISTS:



a) The above diagram is an illustration of a node with info and next fields.

b) The pointer variable 'P' gives the location of this node in memory.

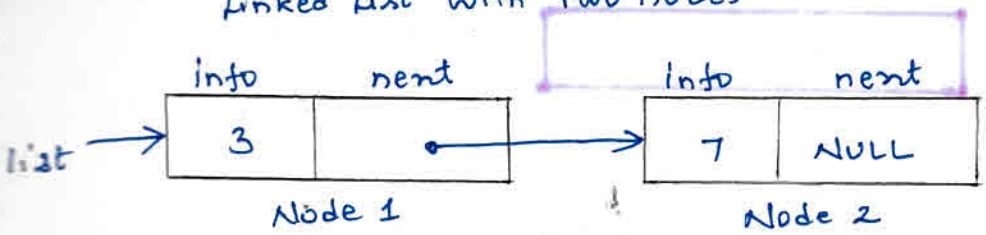
c) info(P) refers to the contents of the information field and next(P) refers to the contents of the next address field pointed to by the pointer variable 'P'.



Inserting a Node at the Front of a List

Example:

Linked List with two nodes



Consider a list containing two nodes with an integer item in their info field and a pointer (an address) in the next field giving the location of the next node in memory. We assume that the pointer variable 'list' points to the first node of the list.

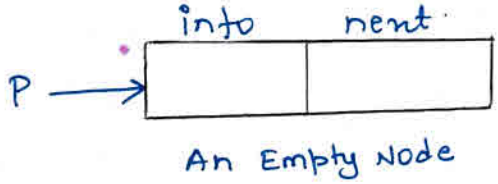
Solution:

a) How to get a new node so that it can be inserted at the front of the list?

The statement (or) operation,

```
P = getnode ( );
```

can obtain an empty node from memory with a pointer 'P' pointing to the address of that node.



b) How to set values for info and next fields for this new node?

To put an integer '5' into the info field we can write

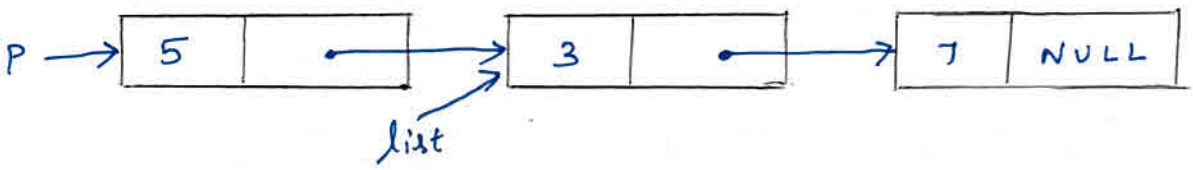
```
info(P) = 5;
```

(10)

c) To store the value of the 'list' pointer (which now points to the front of the list) into the next field, we can write

```
next(P) = list;
```

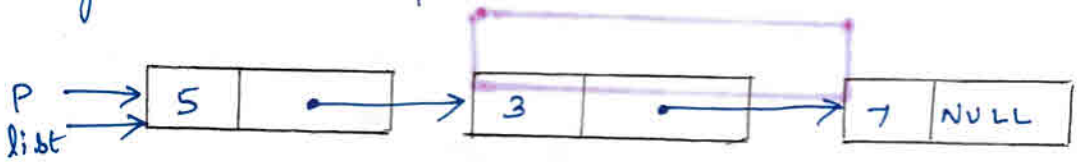
List after the application of above operations;



d) To complete our task, we must make the pointer 'list' to point to the front node of the list. we can do this by writing,

```
list = P;
```

List after the above operation,



e) Now, the auxiliary pointer 'P' is no longer needed for our operation and can be discarded.

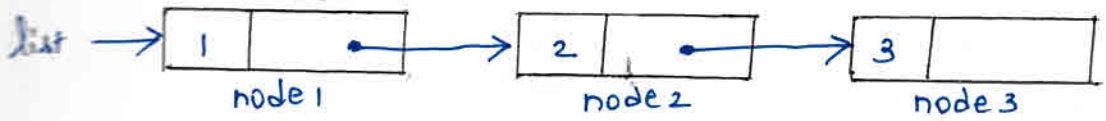
Thus, the algorithm for inserting a new node at the front of a List is,

```
P = getnode();
info(P) = value;
next(P) = list;
list = P
```

Deleting a node from the front of the list.

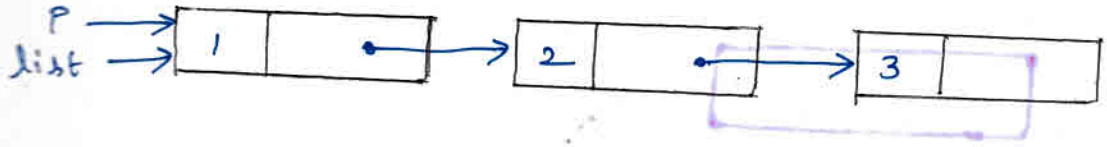
Example:-

A linked list with 3 nodes,



a) First, we make use an auxiliary pointer 'P' and make it point to the first node by writing,

```
P = list;
```



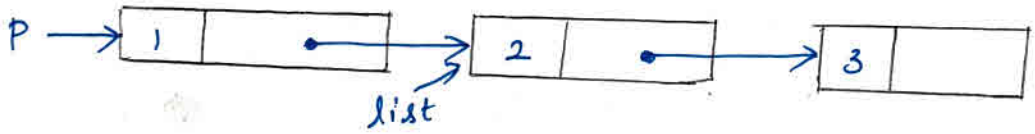
b) Before deleting the first node, we should save whatever that is contained in the info field of the first node, so we write as,

```
x = info (P)
```

c) Then, we should make the pointer 'list' to point to the second node in the list because this should be the node at the front of the list after removing the first node, we do this by writing,

```
list = next(P)
```

List after the above operation,

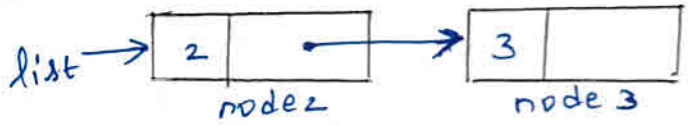


d) Now, we are ready to remove the first node. If we keep this node and no longer use it, then it means that we are wasting valuable Storage.

The operation that does the removal work is,

```
freenode(P);
```

List after the above operation becomes,



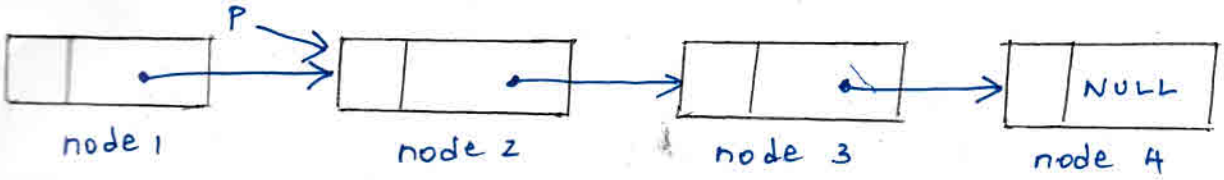
Thus, the algorithm for deleting a node from the front of a list is,

```
P = list;
x = info(P);
list = next(P);
freenode(P);
```

Inserting a Node in a List after the given node:

Example:

A List with 4 Nodes



Suppose we want to insert a node after the node pointed to by 'p' (i.e, between node 2 and node 3), we make use of an auxiliary pointer 'q'.

a) The operation `q = getnode();` can obtain a new node from memory.

q → Empty node obtained from memory with a pointer 'q'.

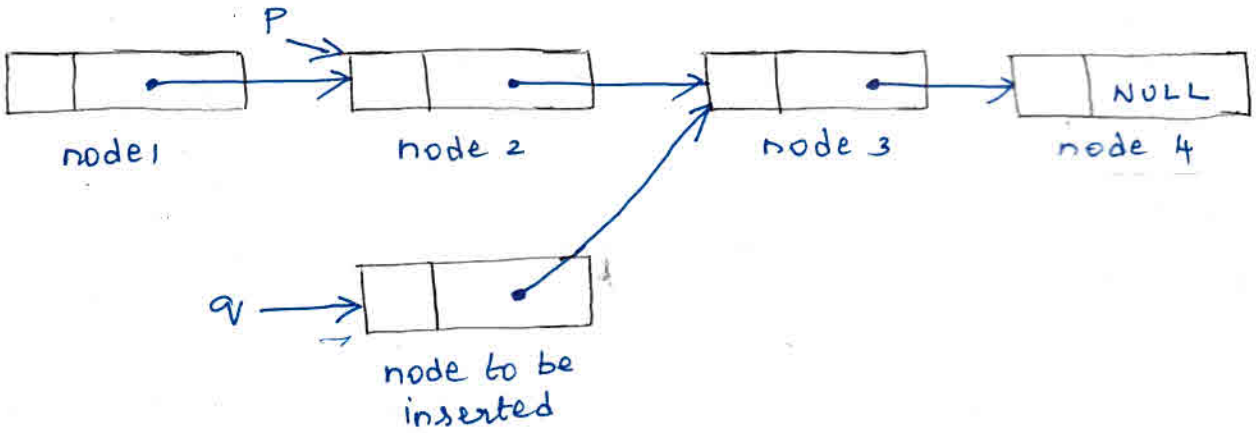
b) An item 'x' can be inserted into the info field of 'q' by the operation,

```
info(q) = x;
```

c) If this node is to be inserted between node 2 and node 3, we must make next(q) to point to node 3. For this, we can write,

```
next(q) = next(p);
```

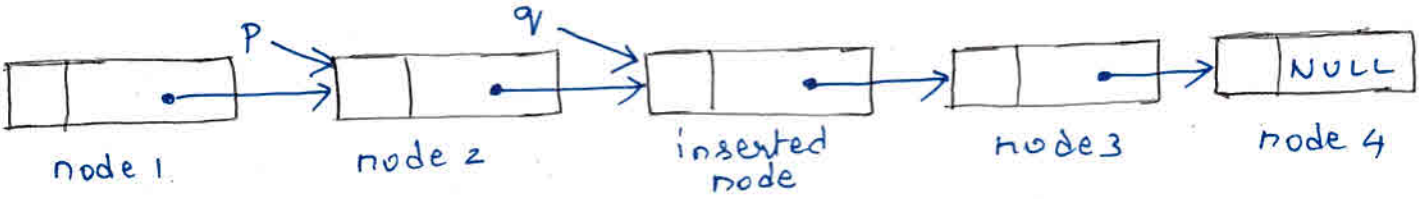
List after the above operation,



d) Now, we must make next(P) to point to the new node. This we do by writing,

```
next(P) = q;
```

List after the above operation,

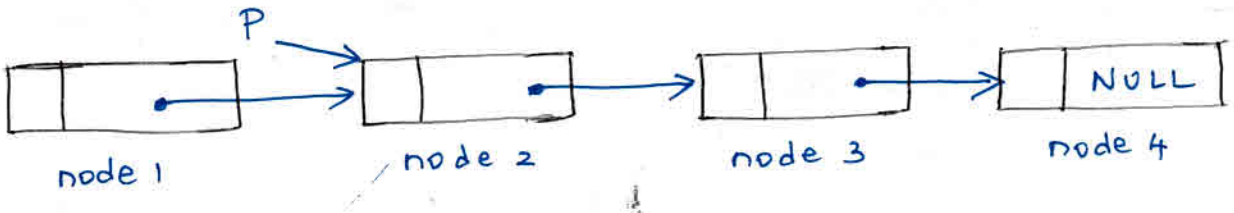


Summarizing all the above steps, we can write the algorithm for inserting an item 'x' into a new node in a list after the node pointed to by 'p' as,

```
q=getnode();
info(q)=x;
next(q)=next(P);
next(P)=q;
```

Deleting a node from a list after the node pointed to by 'p'

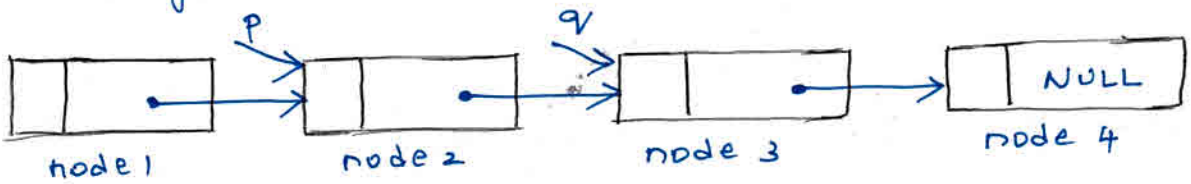
Example:



a) Suppose, we want to delete the node after the node pointed to by 'p' (i.e. in our example node 3). As usual, we use an auxilary pointer 'q' and set its value to next(p) by,

```
q = next(p);
```

List after the above operation,



b) Before deleting node 3, we must save the 'info' field Contents by the operation,

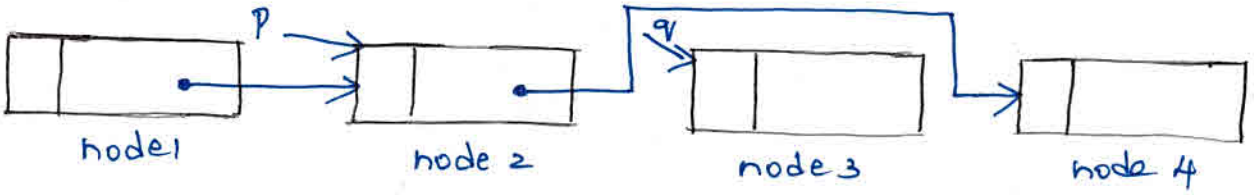
```
x = info(q);
```

c) Since we are to delete node 3, we must make next(p) to Point to node 4. Therefore we write,

```
next(p) = next(q);
```

(16)

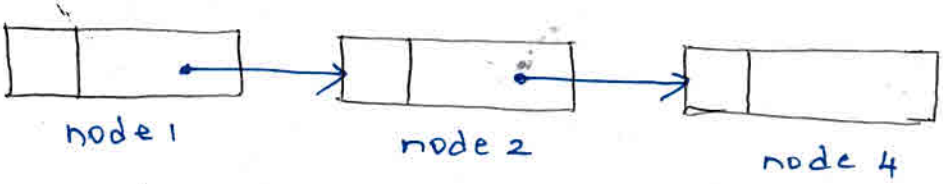
List after the above operation,



d) We no longer need the node pointed to by q. we can delete it and return it to available list for later use. The freenode(q) operation can do this work.

```
Freenode (q);
```

List after the above operation,



Summarizing all the above steps, we can write the algorithm for deleting a node after the node pointed to by 'p' as,

```
q = next(P);
x = info(q);
next(P) = next(q);
freenode(q);
```

Difference between Arrays and Lists

Array	List.
1. We can access n^{th} item in an array by a single operation.	Accessing n^{th} item in a list requires to pass through the First $(n-1)$ items in the list to reach the n^{th} item.
2. Inserting an item into an array or deleting an item from an array in a given position involves a lot of operations on moving the array elements.	Insertion or deletion in a list requires only the pointers to be changed appropriately and saving the contents of a node.

Advantages of Linked Lists.

1. A Linked List can grow and shrink in size during its lifetime. In other words, there is no maximum size of a Linked list.
2. As nodes are stored at different memory locations it hardly happens that we fall short of memory when required.
3. Unlike arrays, while inserting or deleting the nodes of the linked list, shifting of nodes is not required.

Operations that can be made with Linked Lists:

1. makelist (or, create list)
2. Insertions
3. Deletion
4. Counting the number of nodes
5. Printlist (Display the nodes in the list)
6. Finding a Node

Applications of Linked List

1. Searching
2. Sorting.
3. Performing polynomial operations.
4. Implementation of Process Control Block (PCB) and File Allocation Table (FAT) etc.

Implementation of Linked Lists

1. Array Implementation of Lists.

a) A Linked list can be thought of as an array of nodes with each node containing a structure with two fields "info" and "next".

We can declare a linked list containing 200 nodes as

```
#define LIST_SIZE 200
Struct nodefields
{
    int info;
    int next;
};
Struct nodefields node[LIST_SIZE];
```

b) We use the following notations,

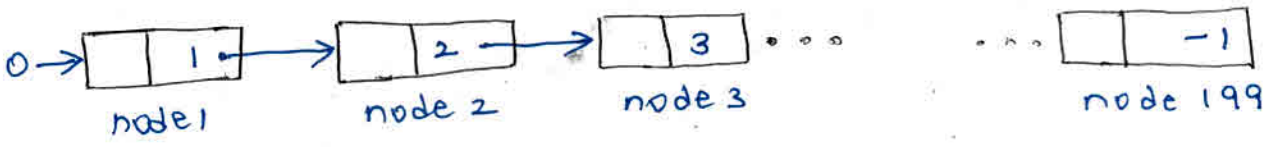
1. node[P] to refer to the node pointed to by P (i.e) node(P).
2. node[P].info to refer to the 'info' field of the node Pointed to by P.
3. node[P].next to refer to the next field of the node Pointed to by P.
4. The NULL Pointer is represented by the integer -1.

Initial placement of nodes in an available list:

a) Initially, nodes may be placed in an available list and linked in natural order.

b) The algorithm for linking nodes in the available can be written as.

```
avail = 0;
For (i=0; i < LIST_SIZE -1; i++)
node[i].next = i+1;
node[LIST_SIZE -1].next = -1;
```

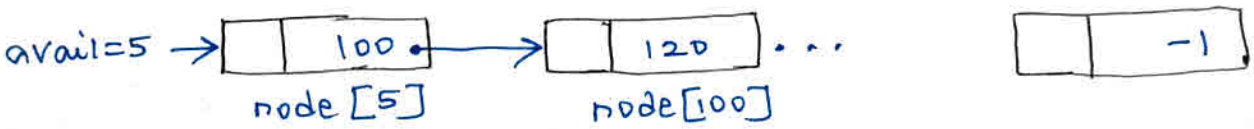


Algorithm for obtaining a node from the available list.

a) When a node is needed for use in Particular list, it can be obtained from the available list.

b) The getnode function removes a node from the available list and returns a Pointer to it.

Example:-

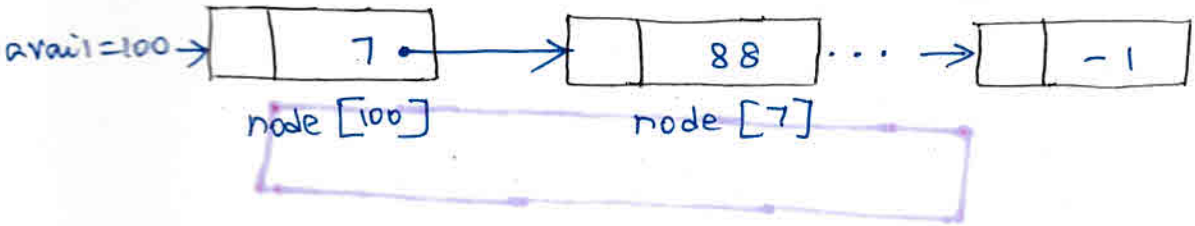


Algorithm for returning a node to the available list

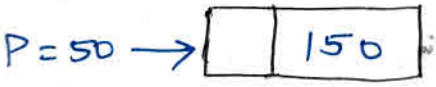
a) The free node function accepts a pointer to a node and returns that node.

Example:-

Available list of nodes at a given point of time



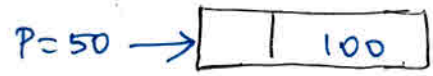
b) Suppose we desire that the node, 'node(P)' has to returned,



(we do not bother about 'info' field)

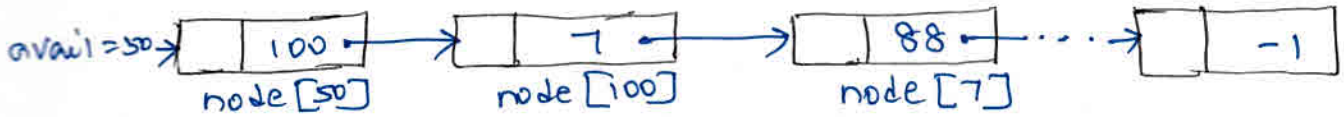
c) Now, we will keep the 'avail' pointer value in next(P). For this, we write `node[P].next = avail;`

The above operation yields,



(we lost 150 in p; we do not need it)

d) Then, we store the value of 'P' into avail by writing `avail = P;` by this we get avail = 50. If we return this node, then the available list would be like,



The algorithm for freenode() operation is,

```
freenode (P)
int P;
```

c) we make use of an auxilary pointer 'p' and store 'avail' value in it (i.e) in our example, we keep value '5' into 'p'. by writing `P = avail;`

d) After removing node[5] from the available list, the Pointer 'avail' must be made to point to the next node (i.e) node[100] in the available list. so we write as `avail = node[avail].next;`

e) Now, the first node 'node[5]' can be removed with its pointer value available in the auxilary Pointer p.

The algorithm for getnode() function can be written as,

```
getnode()
{
    int P;
    if (avail == -1)
    {
        Printf ("overflow");
        exit (1);
    }
    P = avail;
    avail = node[avail].next;
    return (P);
}
```

Algorithm for inserting a node next to the given node:

Suppose we want to insert a node next to the node pointed to by 'p' (i.e) node(p).

```
q = getnode();
node[q].info = x;
node[q].next = node[p].next;
node[p].next = q;
```

Algorithm for deleting a node next to the given node:

```
q = node[p].next;
x = node[q].info;
node[p].next = node[q].next;
freenode(q);
```

Disadvantages of Array implementation of Lists

1. Array implementation demands the amount of node requirements at before hand, which is not possible all the time.
2. Storage will remain allocated for the declared number of nodes even our program uses only lesser number of nodes.

Note:

The Solution to the above Problem is to allow nodes that are dynamic rather than static. In dynamic ... allocated as and when needed

Pointer Implementation of Lists.

a) Let us consider the declaration,

```
Struct node
{
   int info;
   Struct node * next;
};
```

We have declared the next field of a node as a Pointer to the same structure. Now, we can set a Pointer to the above structure by the declaration.

```
Struct node * P;
```

we use 'P' to point to the first node.
P→next is a pointer to another structure of the same type
we use P→next to point to the next node.

We can now make the assignment P = P→next. Then 'P' will point to the second node and P→next will Point to the third node.

b) Declaration of nodes in this way is identical to the nodes of the array implementation except that the next field is a pointer containing the address of the next node in the list. The advantage of such a declaration is that it eliminates declaration of a collection of nodes.

c) we use the type definition (typedef) feature in 'C' for declaring pointer variables of type struct node * simply by writing,

```
nodeptr P;
```

in place of writing the lengthier expression

```
struct node *P;
```

To define 'nodeptr' we should first write the code for type definition as,

```
typedef struct node *nodeptr;
```

The getnode() function:

a) The Programmer need not be concerned with managing the available storage. In fact, we no longer need the 'avail' pointer because the system governs allocation and removal of nodes.

b) There is no need to test overflow condition because it will be detected during the execution of the function 'malloc' and is system dependent.

c) The 'sizeof' operator returns the number of bytes required for the entire structure.

d) The 'malloc' function will allocate number of bytes (as told by sizeof) and returns a pointer of type void

(26)

Comparison of Array implementation and pointer implementation of Lists.

a) Array implementation is also known as static implementation because the storage allocation remains fixed for the declared number of nodes.

Pointer implementation is also known as dynamic implementation because the storage will be allocated as and when needed, and will be released when it is no longer needed.

b) Operations like 'insert' and 'delete' take a constant number of steps for a linked list.

Same operations requires time proportional to the number of elements when array implementation is used.

c) Array implementation may waste space, since it uses the maximum amount of space independent of the number of elements actually on the list at any time.

Pointer implementation uses only as much space as is needed for the elements currently on the list, but requires space for the pointer in each cell.