CONTENTS

CHAPTER – 24

LINKED LISTS

INDTRODUCTION

Linked lists are collections of data items **linked** or connected to one another. Insertions and deletions are made any where in a **linked list**. A **linked list** before adding any **list** elements can be empty to start with. In which case the pointer pointing to the **linked list** will point to **NULL**. **Linked list** can be considered as a higher level of data structure from the previously defined data structures.

An array is also a collection of data elements which are consecutively allocated memory space at compile time. Array cannot expand nor shrink as the necessity grows and shrinks. There has to be accurate allocation of memory at compile time. Memory will be wasted if excess memory is allocated at compile time or there will be shortage when the need grows unexpectedly. **Linked lists** will eliminate this disadvantage, memory is allocated or de-allocated as and when necessary. Memory utilization will be lot better with **linked lists**.

A linked list is appropriate when the number of data elements to be represented in the data structure at once is unpredictable. **Linked lists** are dynamic, so the length of a **list** can increase or decrease as necessary. The size of an array, however cannot be altered, because array memory is allocated at compile time. **Linked lists** become full when the system has insufficient memory to satisfy dynamic storage allocation requests.

Linked lists nodes are normally not stored contiguously in memory. Logically, however, the nodes of a **linked list** appear to be contiguous. **Lists** of data can be stored in arrays, but **linked lists** provided several advantages.

SELF REFERENTIAL STRUCTURES

A ***self referential structures*** contain a pointer member that points to a structure of the structure type. Self referential structure can be **linked** together to form useful data structures as lists, queues, stacks, and trees

Example:

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

A structure of type struct node has two members, one is the data member, which in this case an int field and the other is pointer data type nextptr. The nextptr field will point to a structure of the same type in which it is defined. When the nextptr is not pointing to any structure it may be assigned point to a **NULL** value that means it is not pointing to anything. A **NULL** pointer normally indicates the end of a data structure just as the **NULL** character indicates the end of string.

MEMORY MANAGEMENT

One of the most important functions of a programming language is to provide facilities for managing ***memory*** and the objects that are stored in memory. C provides three distinct ways to allocate memory for objects:

Static memory allocation:

space for the object is provided in the binary at compile-time; these objects have an extent (or lifetime) as long as the binary which contains them is loaded into memory

Automatic memory allocation:

Temporary objects can be stored on the stack, and this space is automatically freed and reusable after the block in which they are declared is exited

Dynamic memory allocation:

Blocks of memory of arbitrary size can be requested at run-time using library functions such as ***malloc*** from a region of memory called the ***heap***; these blocks persist until subsequently freed for reuse by calling the library function ***free***

These three approaches are appropriate in different situations and have various tradeoffs. For example, static memory allocation has no allocation overhead, automatic allocation may involve a small amount of overhead, and dynamic memory allocation can potentially have a great deal of overhead for both allocation and de-allocation. On the other hand, stack space is typically much more limited and transient than either static memory or heap space, and ***dynamic memory allocation*** allows allocation of objects whose size is known only at run-time. Most C programs make extensive use of all three.

Where possible, automatic or static allocation is usually preferred because the storage is managed by the compiler, freeing the programmer of the potentially error-prone chore of manually allocating and releasing storage. However, many data structures can grow in size at runtime, and since static allocations must have a fixed size at compile-time, there are many situations in which **dynamic allocation** must be used.