TATA ELXSI
engineering creativity

TATA

C Programming

Presenter : Learning & Development Team

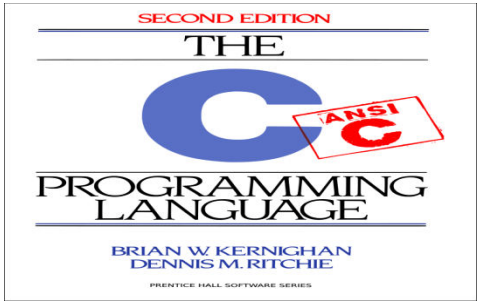
Day – 8 & 9

Disclaimer

- This material is developed for in-house training at TATA ELXSI.
- The training material therefore was solely developed for educational purposes. Currently, commercialization of the prototype does not exist, nor is the prototype available for general usage. TATA ELXSI, its staff, its students or any other participants can use it as a part of training. This material should not be found on any website.
- For the preparation of this material we have referred from the below mentioned links or books. Excerpts of these material has been taken where there is no copy right infringements. Some examples and concepts have been sourced from the below links and are open source material

  http://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html

ANSI C K&R



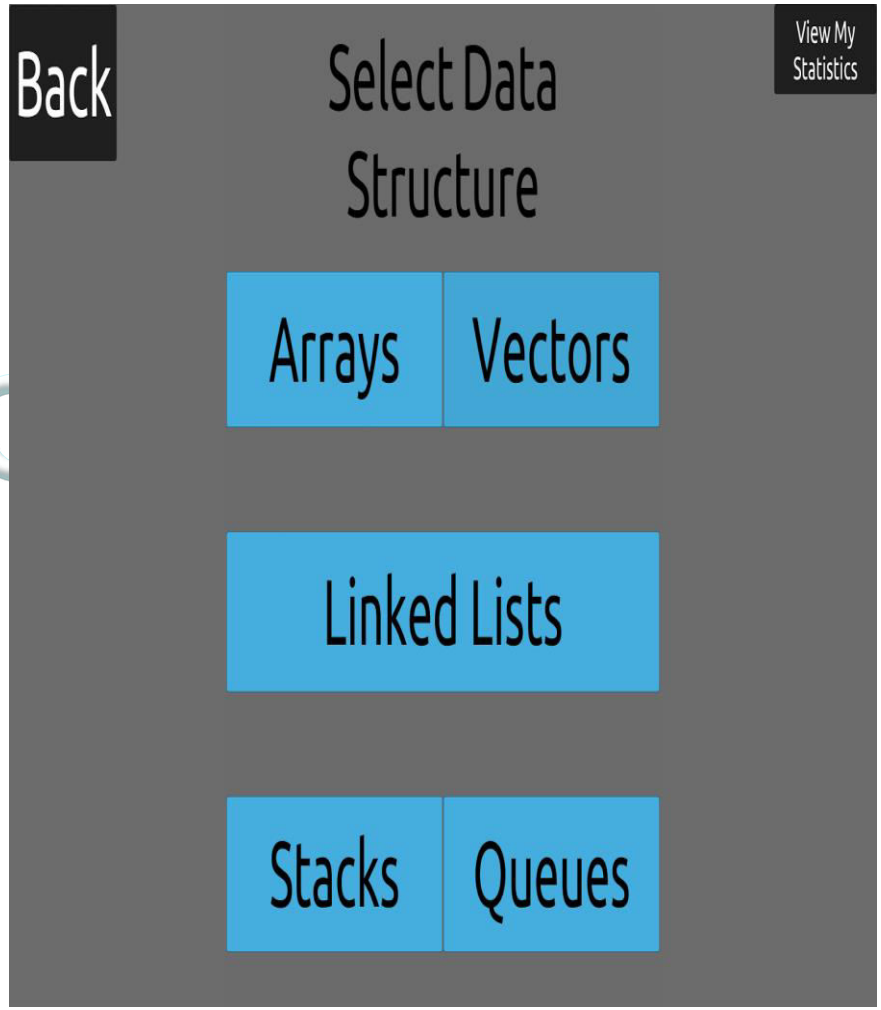
Day – 8 Agenda

- Definition of data structure
- Stack
- Queue
- Single linked list

Basic Data Structures

Intro to Data structure

- In computer science, a data structure is a particular way of organizing data in a computer so that it can be used efficiently.
- Some of the more commonly used data structures include lists, arrays, stacks, queues, heaps, trees, and graphs.
- The way in which the data is organized affects the performance of a program.
- Computer programmers have to decide which data structures to be use based on the nature of the data and the processes that need to be performed on that data.

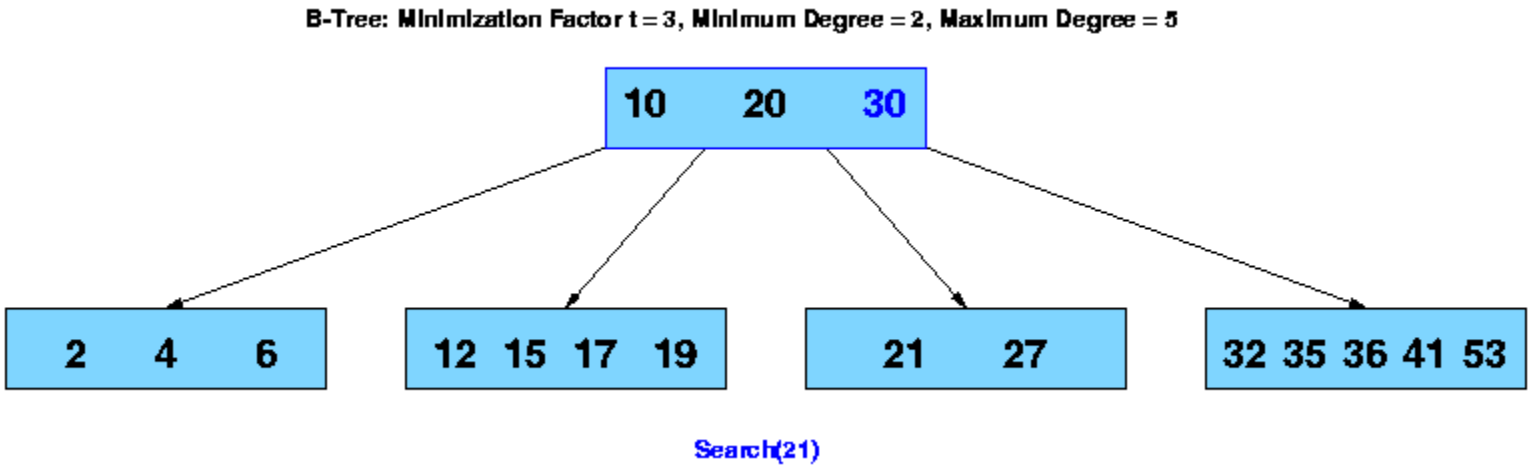


Example : Queue

- A *queue* is commonly used simple data structure.
- A queue has two ends, called the *front* and *back* of the queue.
- Data enters from one end and exists from the other.
- This is also called as FIFO type structure.

B-Tree

- Relational databases most commonly use B-tree indexes for data retrieval



Usage of data structure

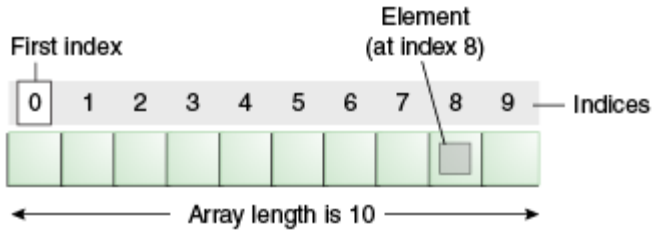
- Data structures provide a means to manage large amounts of data efficiently, such as large databases and internet indexing services.
- Usually, efficient data structures are a key to designing efficient algorithms.
- Storing and retrieving can be carried out on data stored in both main memory and in secondary memory.

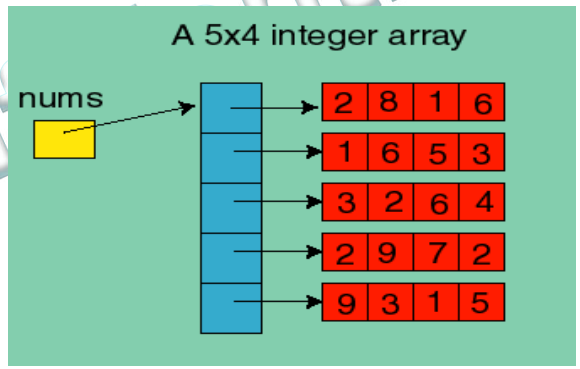
types of data structures:

- An array stores a number of elements in a specific order.
- Records (also called tuples or structs) , The elements of records are usually called fields or members.
- A hash table (also called a dictionary or map) , in which name-value pairs can be added and deleted freely.
- A union type Contrast with a record, which could be defined to contain a float and an integer; whereas, in a union, there is only one value at a time.

Array

- Array is a collection of similar data elements.
- Data insertion and deletion can happen from any end.
- Array can have any dimension.



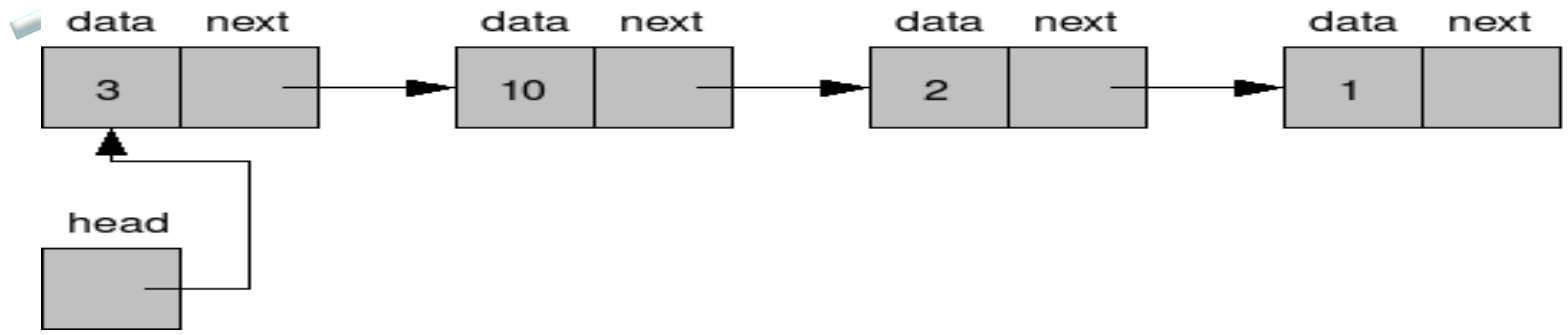


1) The size of the arrays is fixed.

2) Inserting a new element in an array of elements is expensive, because room has to be created for the new elements and to create room existing elements have to shifted.

Linked list

- Like arrays, Linked List is a linear data structure.
- Unlike arrays, linked list elements are not stored at contiguous location; the elements are linked using pointers.
- They are a dynamic in nature which allocates the memory when required.
- Insertion and deletion operations can be in any position and can be easily implemented.



Stack

- Stack is an ordered list of similar data type.

- Stack is a **LIFO** structure. (Last in First out).

OR [FILO]

- **push()** function is used to insert new elements into the Stack and **pop()** is used to delete an element from

- Used in many application in computer science.
  - 1) expression evaluation
  - 2) function call implementation
  - 3) Parsing in compiler design

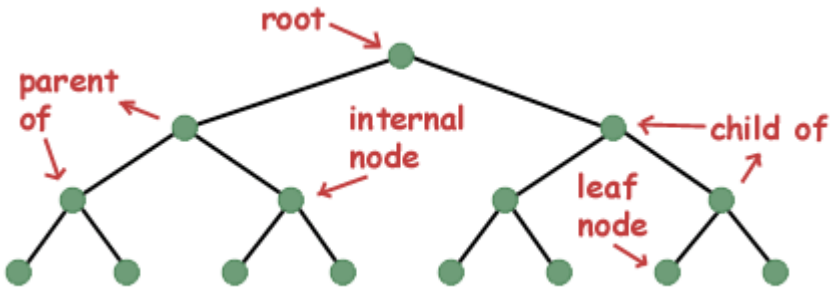
Queue

- Like Stack, Queue is also an ordered list of elements of similar data types.
- Queue is a FIFO(First in First Out) structure.
- Once a new element is inserted into the Queue, all the elements inserted before the new element in the queue must be removed, to remove the new element.

- Applications:
  - Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
  - Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive, First come first served.

Tree

- Tree is a Data structure keeps the data in organized way.



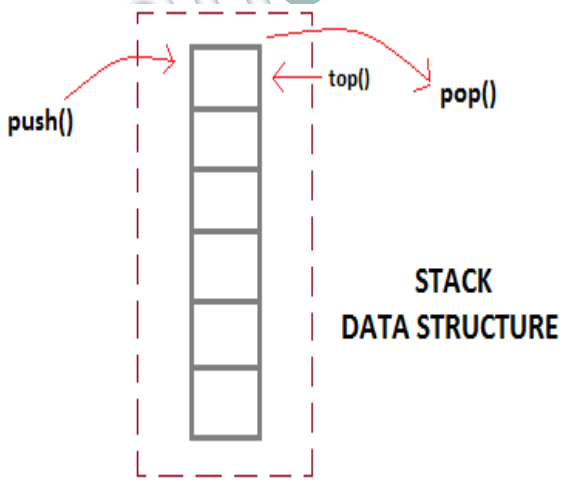
tree is hierarchical (or non-linear) data structure.
1) One reason to use trees might be because you want to store information that naturally forms a hierarchy.

Stack

What is stack?

- Stack is an abstract data type with a bounded(predefined) capacity.

- It is a simple data structure that allows adding and removing elements in FILO [LIFO] order.



- Operations on Stack:
- 1. **push :** is a function used to insert element
- 2. **pop :** is a function used to delete element from stack.
- 3. Stack is said to be in **Overflow** state when it is completely full and is said to be in **Underflow** state if it is completely empty.

Operations on Stacks

- Some of the typical operations performed on stacks are:
- **create (s)** – to create s as an empty stack
- **push (s, i)** – to insert the element i on top of the stacks
- **pop (s)** – to remove the top element of the stack and to return the removed element.
- **top (s)** – to return the top element of stack(s)
- **empty(s)** – to check whether the stack is empty or not. It returns true if the stack is empty

Function prototypes:

```
void create(Stack *);

int IsStackEmpty(Stack *);

int IsStackFull(Stack *);

StackElement pop(Stack *);

void push(Stack *, StackElement);

int menu(void);
```

Position of Top	Status of Stack
-1	Stack is Empty
0	Only one element in Stack
N-1	Stack is Full
N	Overflow state of Stack

```
void create(Stack *s)
{
        s -> top = -1;
}
```

Application of stack

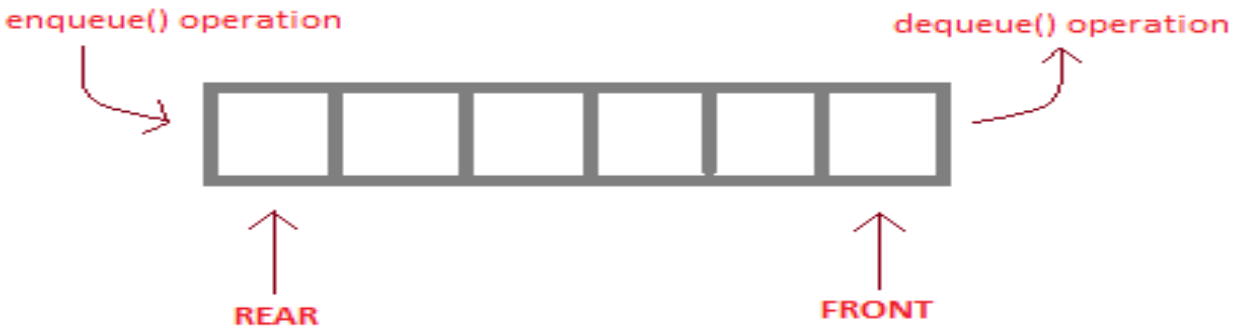
- **Infix , Prefix and postfix expression** conversion
- **Prefix and postfix expression** evaluation
- **Lexical analysis**
- **Recursion**

Queue

What is Queue?

- Queue is also an abstract data type or a linear data structure, in which the first element is inserted from one end called **REAR**(also called tail),

- Deletion of element takes place from the other end called as **FRONT**(also called head).



enqueue( ) is the operation for adding an element into Queue.
dequeue( ) is the operation for removing an element from Queue .

QUEUE DATA STRUCTURE

Basic features of Queue

- Like Stack, Queue is also an ordered list of elements of similar data types.
- Queue is a FIFO(First in First Out) structure.
- Once a new element is inserted into the Queue, all the elements inserted before the new element in the queue must be removed, to remove the new element.
- **peek( )** function is oftenly used to return the value of first element without dequeuing it.

Applications of Queue

- Serving requests on a mutually exclusive resource, like a printer, CPU task scheduling etc.

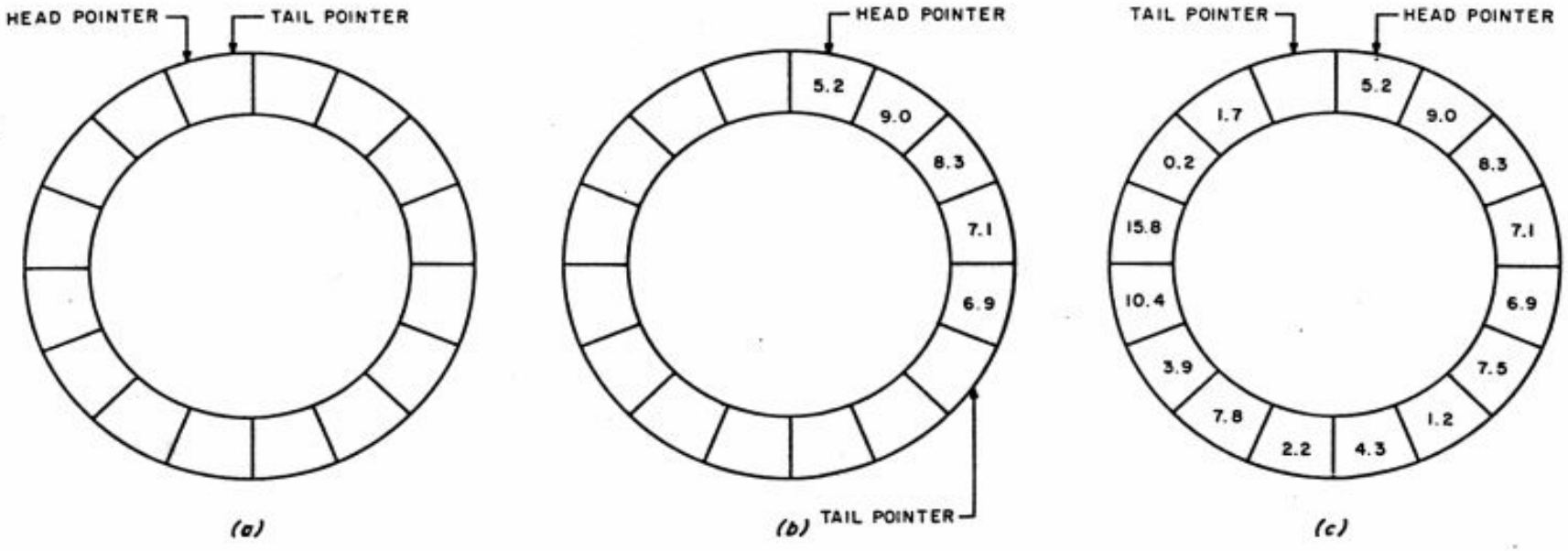
- Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive, First come first served.

Queue Operations

- create(q) – which creates q as an empty queue
- enqueue(i) , push(i) – adds the element i to the rear end of the queue
- dequeue(q) , pop(q) and – removes the element at the front end of the queue (q) returns the removed element.
- empty (q) – it checks the queue (q) whether it is empty or not. It returns true if the queue is empty and returns false otherwise
- front(q)or peek() – returns the front element of the queue without changing the queue
- queuesize(q) – returns the number of entries in the queue

Circular Queue

- This is a particular kind of queue where new items are added to the rear of the queue as items are read off the front of the queue.
- So there is constant stream of data flowing into and out of the queue



Insert into Circular Queue

1. If (FRONT == 1 and REAR == N) or (FRONT == REAR + 1) Then
2. Print: Overflow
3. Else
4. If (REAR == 0) Then [Check if QUEUE is empty]
(a) Set FRONT = 1
(b) Set REAR = 1
5. Else If (REAR == N) Then [If REAR reaches end if QUEUE]
6. Set REAR = 1
7. Else
8. Set REAR = REAR + 1 [Increment REAR by 1]
[End of Step 4 If]
9. Set QUEUE[REAR] = ITEM
10. Print: ITEM inserted
[End of Step 1 If]
11. Exit

Delete from Circular Queue

1. If (FRONT == 0) Then [Check for Underflow]
2. Print: Underflow
3. Else
4. ITEM = QUEUE[FRONT]
5. If (FRONT == REAR) Then [If only element is left]
(a) Set FRONT = 0
(b) Set REAR = 0
6. Else If (FRONT == N) Then [If FRONT reaches end if QUEUE]
7. Set FRONT = 1
8. Else
9. Set FRONT = FRONT + 1 [Increment FRONT by 1]
[End of Step 5 If]
10. Print: ITEM deleted
[End of Step 1 If]
11. Exit

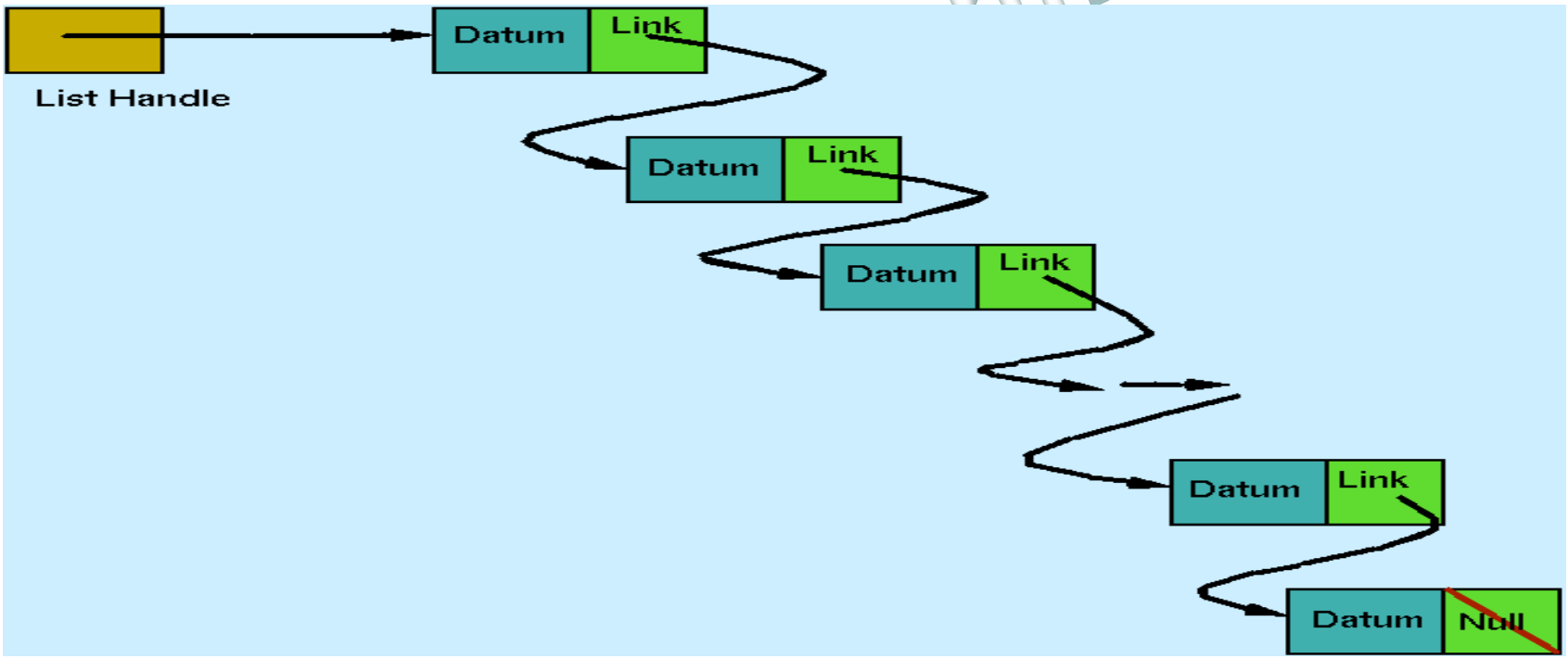
Linked List

Day - 9

Linked List

- Like arrays, Linked List is a linear data structure.

- Unlike arrays, linked list elements are not stored at contiguous location; the elements are linked using pointers



Why Linked List

- Arrays can be used to store linear data of similar types, but arrays have following limitations.

1) The size of the arrays is fixed, but List is dynamic in size.

2) Inserting a new element in an array of elements is expensive, because room has to be created for the new elements and to create room existing elements have to shifted.

- In list Insertion and deletion is fast and less expensive.

- It is a widely used data structure for applications which do not need random access.

Limitations of Linked List

1) Random access is not allowed.

We have to access elements sequentially starting from the first node.

So we cannot do binary search with linked lists.

2) Extra memory space for a pointer is required with each element of the list.

3) Slow in accessing the elements in the worst case , if list is too large

Applications of Linked Lists

- Linked lists are used to implement stacks, queues, graphs, etc.
- Linked lists let you insert elements at the beginning and end of the list.
- In Linked Lists we don't need to know the size in advance.

Operation on List

- The main primitive operations of a list are known as:
- Add : adds a new node
- Update :updates the contents of a node
- Remove : removes a node
- Sort : sorting the list members
- Insert : inserting the data at the given position.

Types of List implementation

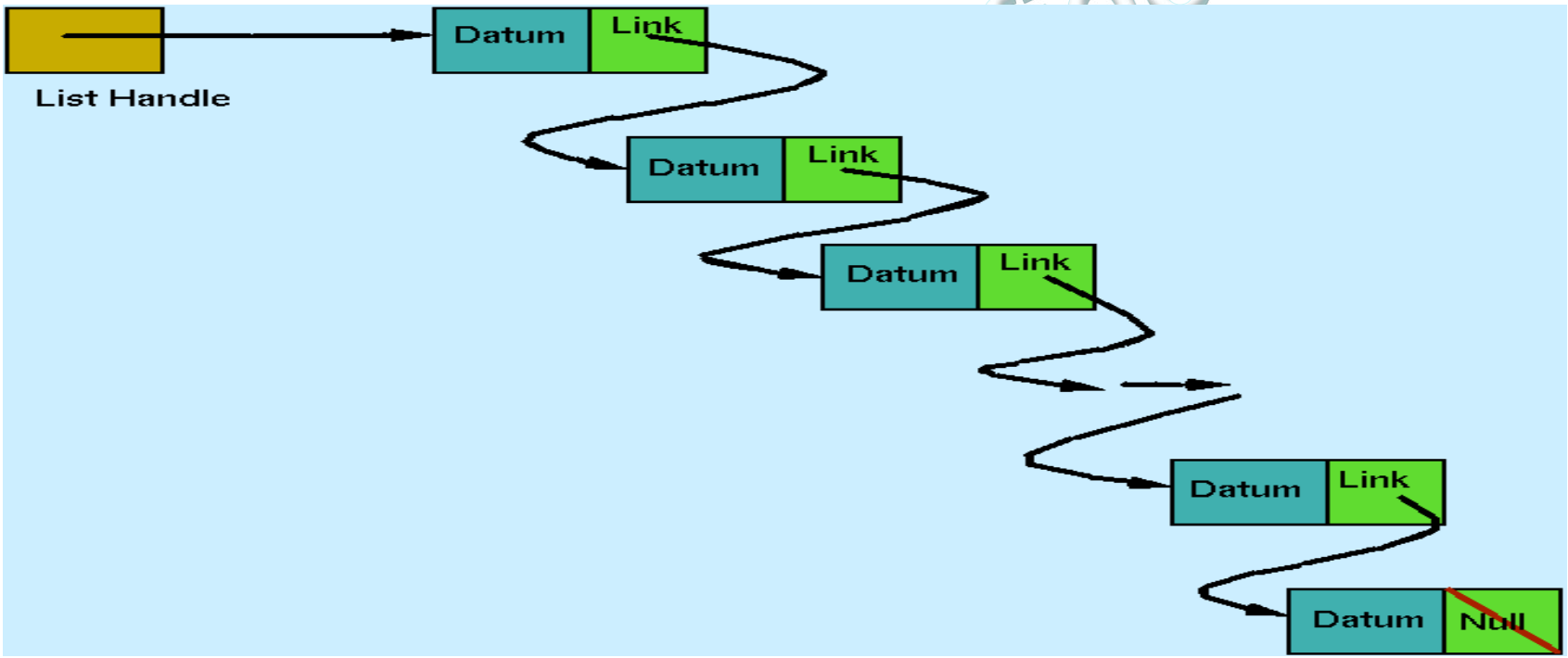
- Singular Linked list
- Single circular Linked list
- Double Linked list
- Double circular Linked list

Declaring the List node

```
// A linked list node
struct node
{
    int data;
    struct node *next;
};
```

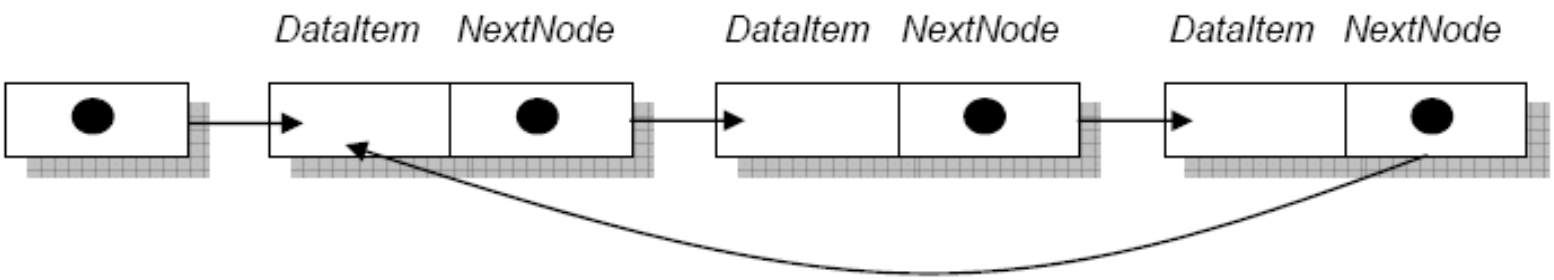
Single Linked list

- Singly linked lists contain nodes which have a data part as well as an address part

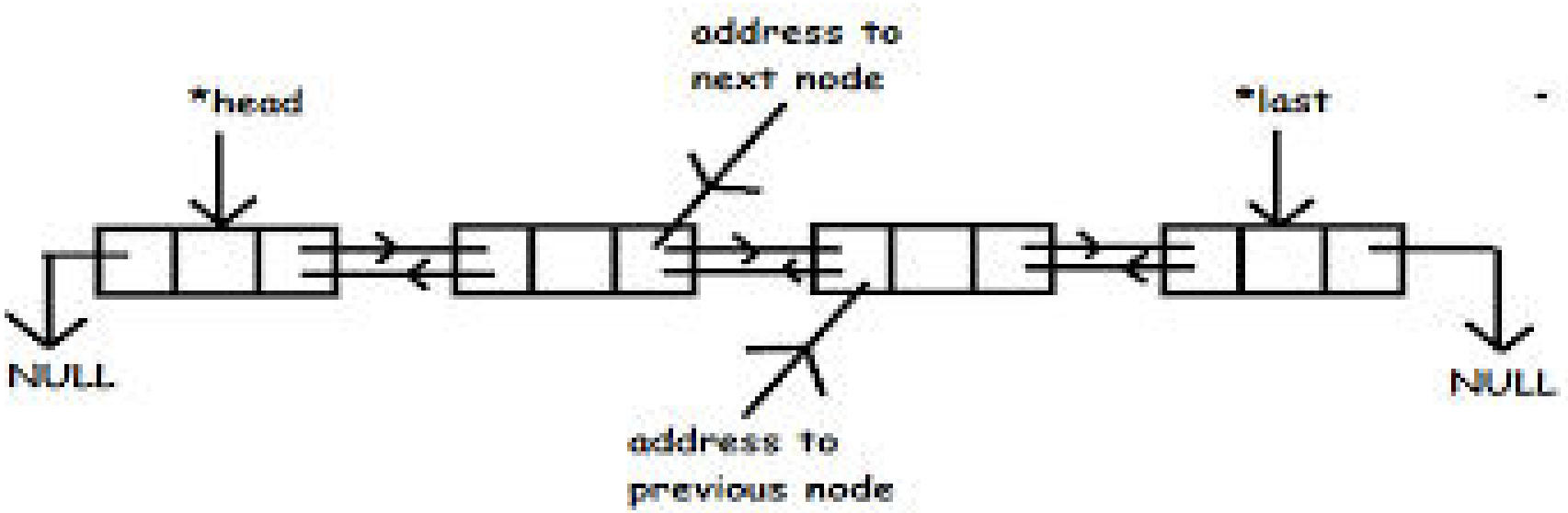


Single Circular linked lists

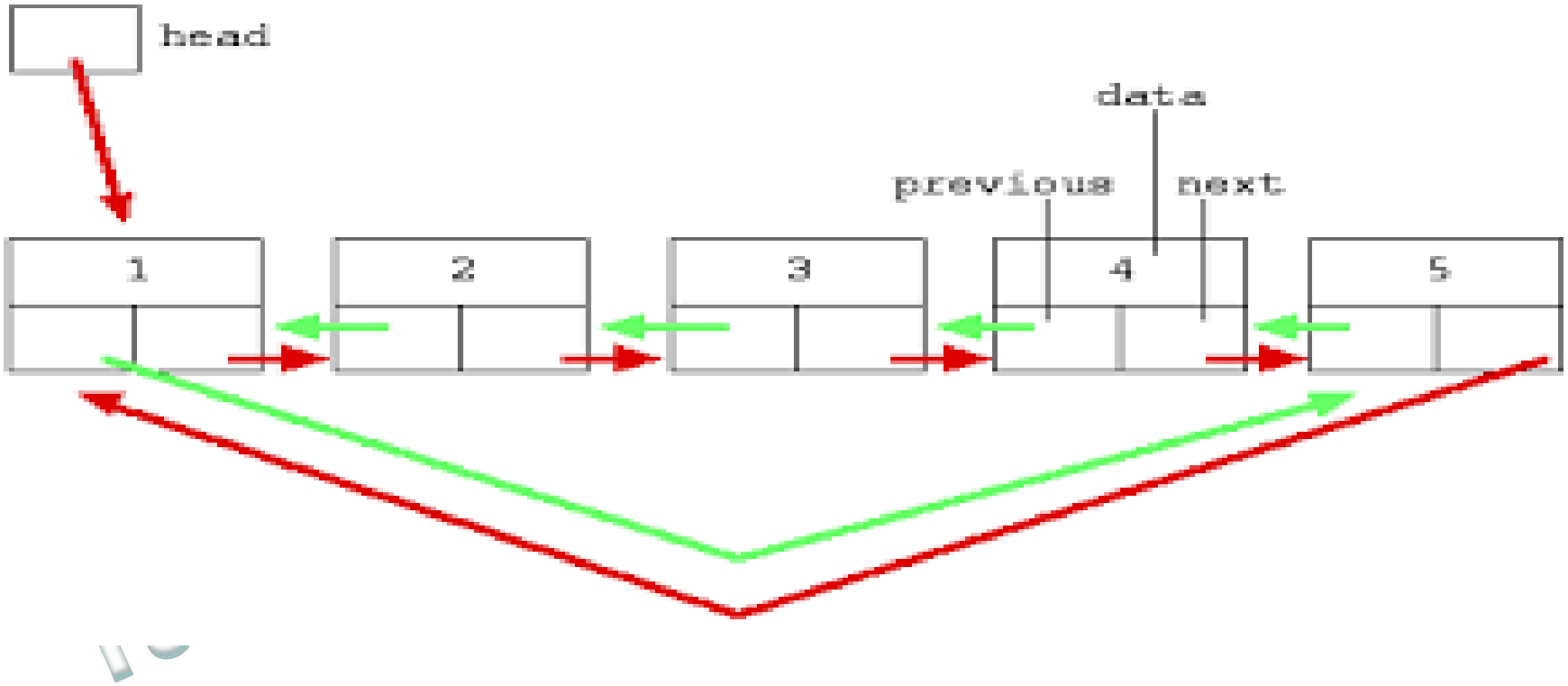
- The tail of the list always points to the head of the list



Double linked lists



Double circular linked list



Summary

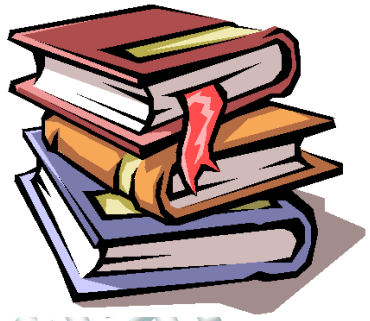
In this session, you learnt to:

Describe the limitations of using a data structure such as an array
Define self-referential structures and their advantages

Write code to:

- Create a sorted linked list,
- Insert nodes into a sorted linked list
- Traverse a linked list
- Delete nodes from a linked list

references



- The GNU C Reference Manual : https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html

- The GNU C Reference Manual : http://www.cprogramming.com/

- C Programming Language

by Brian W. Kernighan , Dennis Ritchie

- C: The Complete Reference

by Herbert Schildt

Thank you

TATA ELXSI

ITPB Road Whitefield
Bangalore 560 048 India
Tel +91 80 2297 9123
Fax +91 80 2841 1474
e-mail info@tataelxsi.com

www.tataelxsi.com