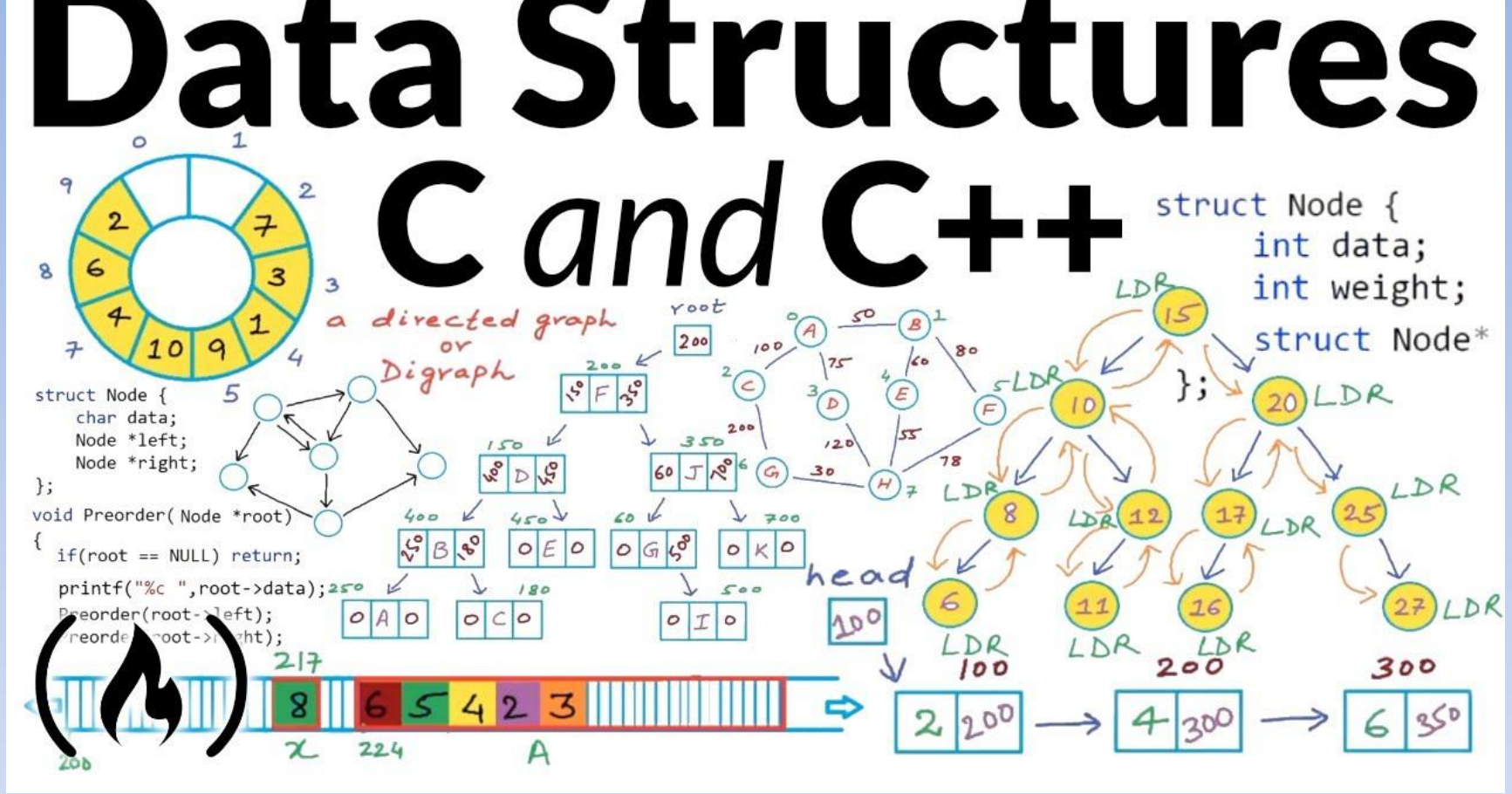
Data Structures C and C++



Prepared by
Dr. Ayesha Hakim
Assistant Professor
EXCP Dept., KJSCE

Data Structure

- A data structure is a specialized format for organizing, processing, retrieving and storing data.
- While there are several basic and advanced structure types, any data structure is designed to arrange data to suit a specific purpose so that it can be accessed and worked with in appropriate ways.

Data Structure

- In computer programming, a data structure may be selected or designed to store data for the purpose of working on it with various algorithms.
- Each data structure contains information about the data values, relationships between the data and functions that can be applied to the data.

Data Structure

- The data structure is basically a technique of organizing and storing of different types of data items in computer memory.
- It is considered as not only the storing of data elements but also the maintaining of the logical relationship existing between individual data elements.
- The Data structure can also be defined as a mathematical or logical model, which relates to a particular organization of different data elements.

Data Structure

- ***Data:***
  - Data is the basic entity of fact that is used in calculations or manipulation process.
  - The way of organizing of the data & performing the operations is called as data structure.

    Data structure=organized data+ operations
  - Operations
    - Insertion
    - Deletions
    - Searching
    - Traversing

Data Structure

- The organization must be convenient for users.
- Data structures are implemented in the real time in the following situations:
  - Car park
  - File storage
  - Machinery
  - Shortest path
  - Sorting
  - Networking
  - Evaluation of expressions

Data Structure

- Specification of data structure :
  - Data structures are considered as the main building blocks of a **computer program**.
    - **Organization of data**
    - **Accessing methods**
    - **Degree of associativity**
    - **Processing alternatives for information**

Algorithm + Data Structure = Program

Data Structure

- At the time of selection of data structure we should follow these two things so that our selection is efficient enough to solve our problem.
  - The data structure must be powerful enough to handle the different relationship existing between the data.
  - The structure of data also to be simple, so that we can efficiently process data when required.

Characteristics of data structures

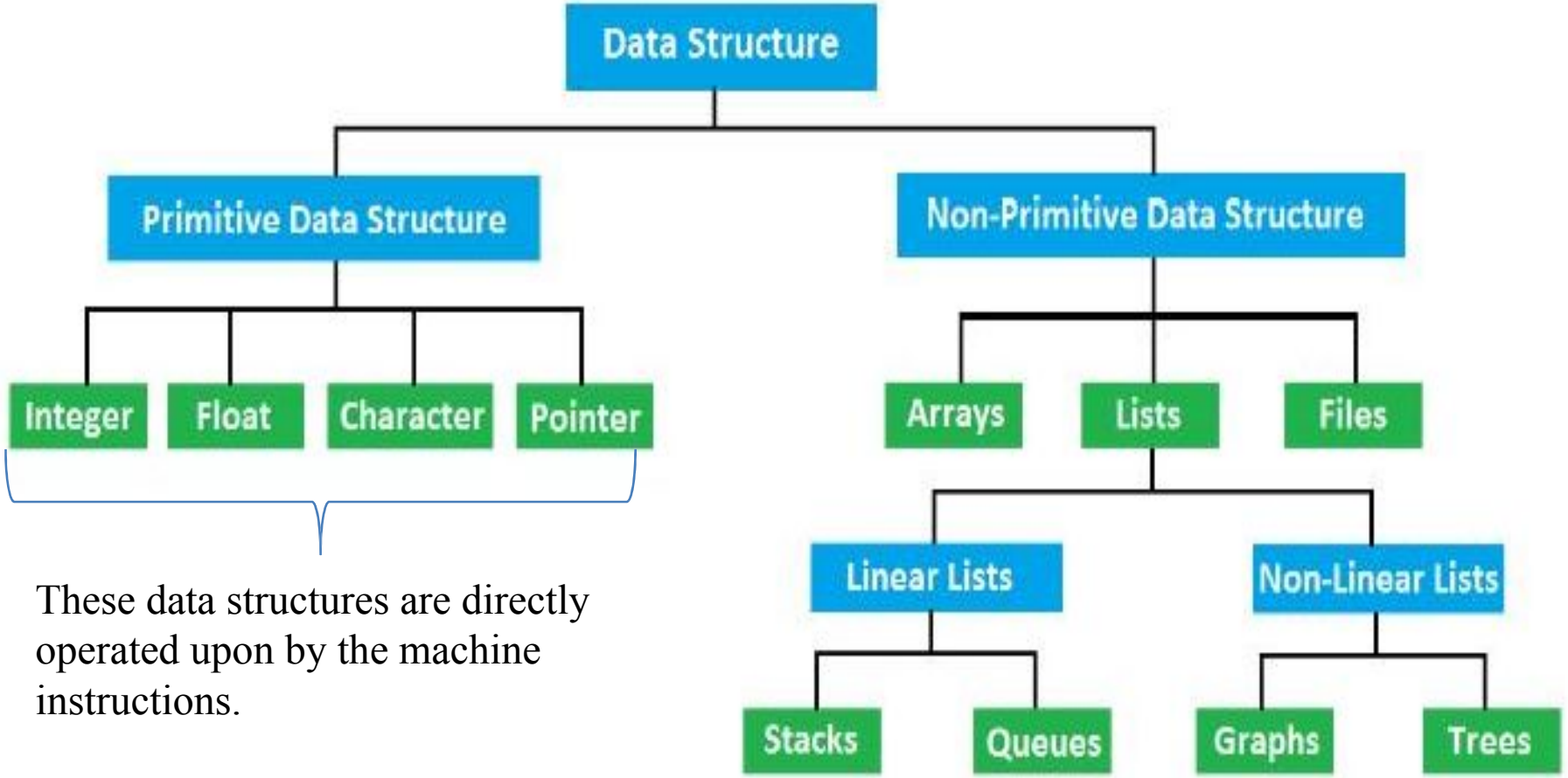
- **Linear or non-linear:** This characteristic describes whether the data items are arranged in chronological sequence, such as with an array, or in an unordered sequence, such as with a graph.

- **Homogeneous or non-homogeneous:** This characteristic describes whether all data items in a given repository are of the same type or of various types.

Characteristics of data structures

- **Static or dynamic:** This characteristic describes how the data structures are compiled. Static data structures have fixed sizes, structures and memory locations at compile time.
- Dynamic data structures have sizes, structures and memory locations that can shrink or expand depending on the use.

Types of data structures



These data structures are directly operated upon by the machine instructions.

Types of data structures

- ***Primitive data structure :***
  - The primitive data structures are known as basic data structures.
  - These data structures are directly operated upon by the machine instructions.
  - The primitive data structures have different representation on different computers.

Types of data structures

- ***Non-Primitive data structure :***
  - The non-primitive data structures are highly developed **complex** data structures.
  - Basically these are developed from the primitive data structure.
  - The non-primitive data structure is responsible for organizing the group of homogeneous and heterogeneous data elements.

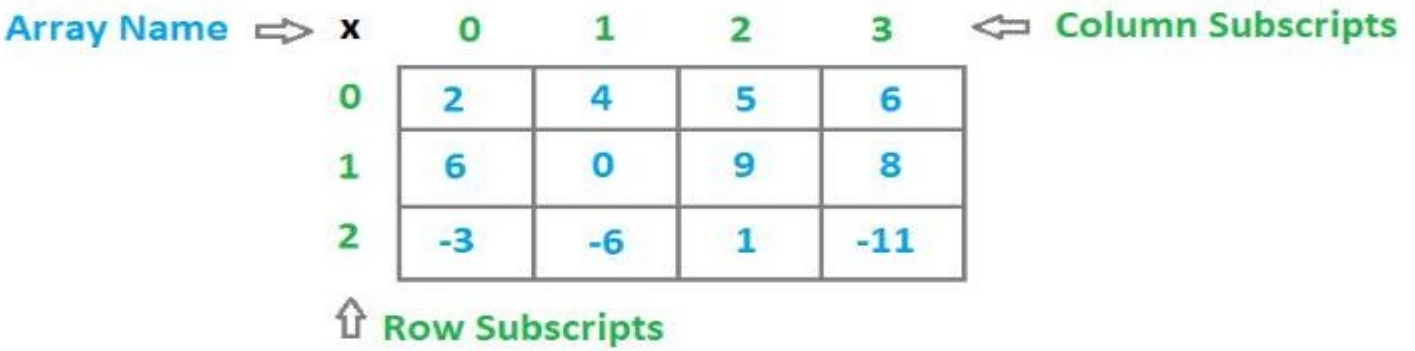
Types of data structures

- Data structure types are determined by what types of operations are required or what kinds of algorithms are going to be applied.
- ***Arrays-***
  - An array stores a collection of items at adjoining memory locations.
  - Items that are the same type get stored together so that the position of each element can be calculated or retrieved easily.
  - Arrays can be fixed or flexible in length.

Types of data structures

- **Arrays-**

One Dimenstional Array



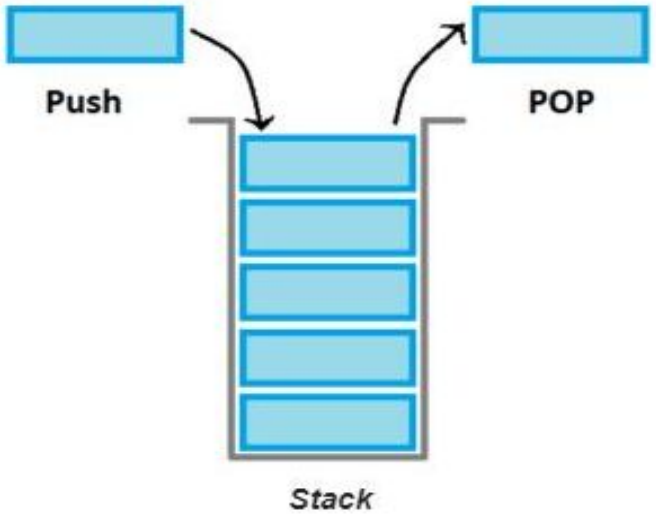
Two Dimensional Array

Types of data structures

- **Stacks-**

We can implement a stack using 2 ways:

- Static implementation (using arrays)
- Dynamic implementation (using pointers)

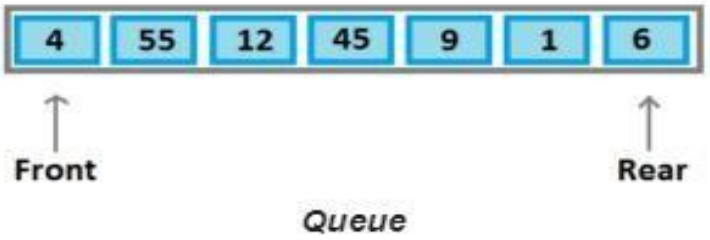


Types of data structures

- **Queues-**
  - A queue stores a collection of items similar to a stack; however the operation order can only be first in first out.

We can also implement queues using 2 ways :

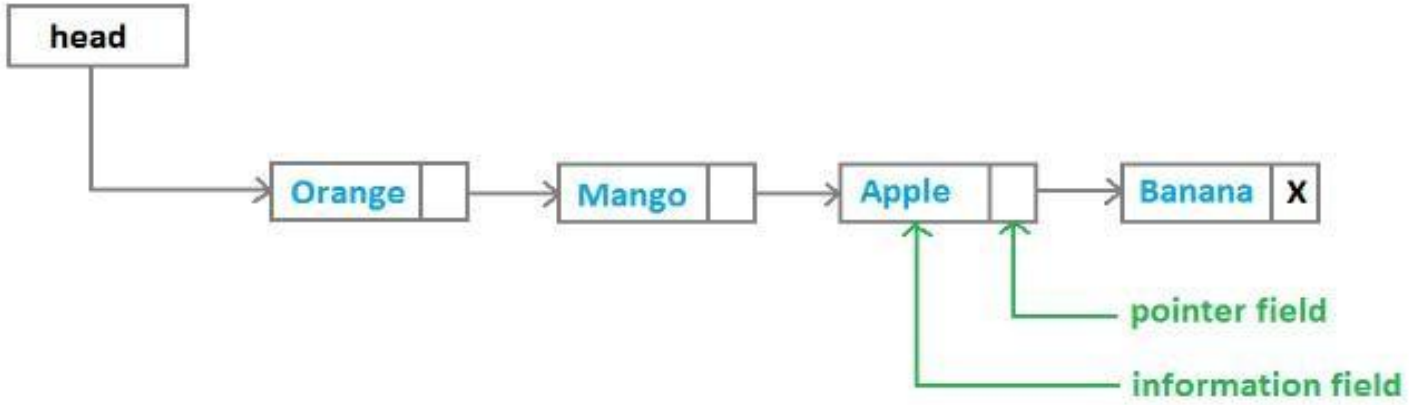
- **Using arrays**
- **Using pointers**



Queue

Types of data structures

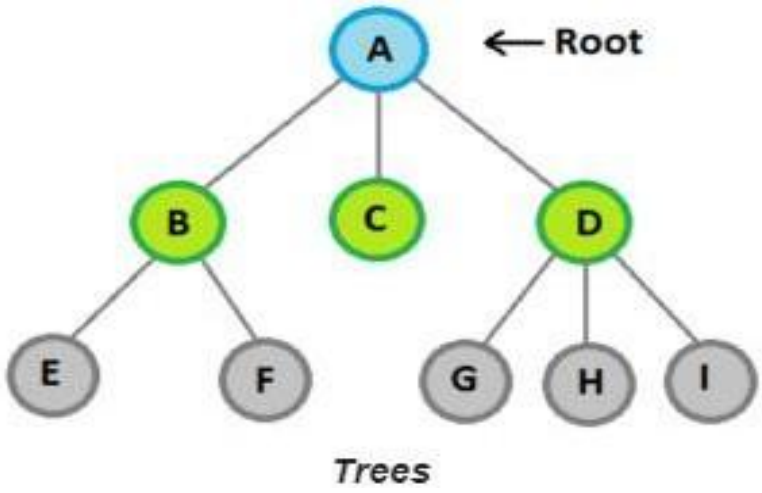
- **Linked lists-**
  - A linked list stores a collection of items in a linear order. Each element or node in a linked list contains a **data item as well as a reference or link to the next item** in the list.



Linear linked lists

Types of data structures

- **Trees-**
  - A tree stores a collection of items in an abstract **hierarchical** way.
  - Each node is linked to other nodes and can have multiple sub-values also known as children.



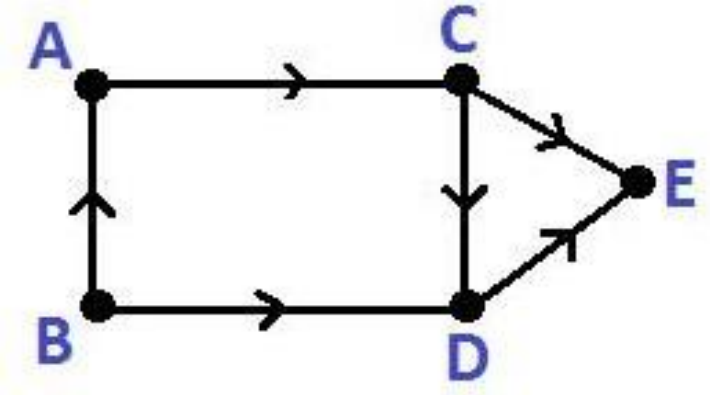
Trees

Types of data structures

- ***A Tree has the following characteristics :***
  - The top item in a hierarchy of a tree is referred as the root of the tree.
  - The remaining data elements are partitioned into a number of mutually exclusive subsets and they itself a tree and are known as the subtree.
  - Unlike natural trees trees in the data structure always grow in length towards the bottom.

Types of data structures

- **Graphs-**
  - A graph stores a collection of items in a non-linear fashion.
  - Graphs are made up of a finite set of nodes also known as vertices and lines that connect them also known as edges.
  - These are useful for representing real-life systems such as computer networks



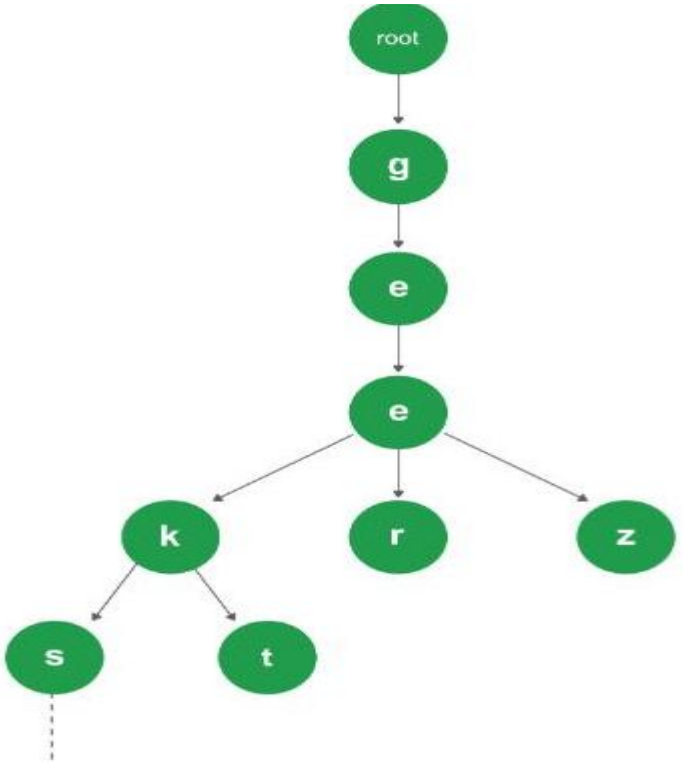
Graph

Types of data structures

- ***The different types of Graphs are :***
  - Directed Graph
  - Non-directed Graph
  - Connected Graph
  - Non-connected Graph
  - Simple Graph
  - Multi-Graph

Types of data structures

- **Tries-**
  - A trie or keyword tree, is a data structure that stores strings as data items that can be organized in a visual graph.



Types of data structures

- **Hash tables-**
  - A hash table or a hash map stores a collection of items in an **associative array** that **plots keys to values.**
  - A hash table uses a **hash function** to convert an index into an array of buckets that contain the desired data item.
  - Overcoming the drawbacks of linear data structures hashing is introduced.

Types of data structures

- ***Files :***
  - Files contain data or information, stored permanently in the secondary storage device such as Hard Disk and Floppy Disk.
  - It is useful when we have to store and processa large amount of data.
  - A file stored in a storage device is identified using a file name always

    like **HELLO.DAT** or **TEXTNAME.TXT** and so on.

Types of data structures

- ***Files :***
  - A file name normally contains a primary and a secondary name which is separated by a **dot(.)**.

Files

Fundamentals of data structures:

- **Fundamental Data Structures**
  - The following four data structures are used ubiquitously in the description of algorithms and serve as basic building blocks for realizing more complex data structures.
    - Sequences (also called as lists)
    - Dictionaries
    - Priority Queues
    - Graphs
  - Dictionaries and priority queues can be classified under a broader category called *dynamic sets*.
  - Binary and general trees are very popular building blocks for implementing dictionaries and priority queues.

Fundamentals of data structures:
Dictionaries

- A *dictionary* is a general-purpose data structure for storing a group of objects.
- A dictionary has a set of ***keys*** and each key has a single associated ***value***.
- When presented with a key the dictionary will return the associated value.
- A dictionary is also called a *hash*, a *map*, a *hashmap* in different programming languages.

Fundamentals of data structures:
Dictionaries

- For example, the results of a classroom test could be represented as a dictionary with pupil's names as keys and their scores as the values
- results = { 'Detra' : 17,
  'Nova' : 84,
  'Charlie' : 22,
  'Henry' : 75,
  'Roxanne' : 92,
  'Elsa' : 29 }
- Instead of using the numerical index of the datawe can use the dictionary names to return values
- >>> results['Nova']
  84
- >>> results['Elsa']
  29

Fundamentals of data structures:
Dictionaries

- The keys in a dictionary must be simple types (such as integers or strings) while the values can be of any type.
- Different languages enforce different type restrictions on keys and values in a dictionary.
- Dictionaries are often implemented as hash tables.
- Keys in a dictionary must be **unique**
- An attempt to create a duplicate key will typically overwrite the existing value for that key.

Fundamentals of data structures:
Dictionaries

- Dictionary is an abstract data structure that supports the following operations:
  - search(K key) (returns the value associated with the given key)
  - insert(K key, V value)
  - delete(K key)
- Each element stored in a dictionary is identified by a key of type K.
- Dictionary represents a mapping from keys to values.

Fundamentals of data structures: Dictionaries

- Dictionaries have numerous applications.
  - contact book
    - key: name of person; value:
  - telephone number table of program variable identiers
    - key: identier; value: address in memory
  - property-value collection
    - key: property name; value: associated value
  - natural language dictionary
    - key: word in language X; value: word in language Y

Fundamentals of data structures: *operations on dictionaries*

- Dictionaries typically support several operations:
  - **retrieve a value** (depending on language, attempting to retrieve a missing key may give a default value or throw an exception)
  - **insert or update a value** (typically, if the key does not exist in the dictionary, the key-value pair is inserted; if the key already exists, its corresponding value is overwritten with the new one)
  - **remove a key-value** pair
  - test for existence of a key
- Note that items in a dictionary are unordered, so loops over dictionaries will return items in an arbitrary order.

Abstract Data Type

A collection of related data is known as an *abstract data type* (ADT)

Data Structure = ADT + Collection of functions that operate on the ADT

Abstract data types are the entities that are definitions of data and operations but do not have implementation details

- Consist of the data **struct**ure definition and a collection of functions that operate on the **struct**
  - **We will never access the struct directly!**
- Separate *what* you can do with data from *how* it is represented
- Other parts of the program interacts with data through provided operations according to their specifications
- Implementation chooses how to represent data and implement its operations

Abstract Data Type

To handle the complex problems, the computer programmer uses **abstraction** to focus on what it does and ignoring how it does its job. ADTs are purely theoretical entities, used to simplify the description of algorithms, used to classify and evaluate the data structures.

An abstract data type (ADT) is an object with a **generic description independent of implementation details**. This description includes a specification of the components from which the object is made and also the behavioral details of the object.

ADTs as interfaces

- **abstract data type (ADT)**: A specification of a collection of data and the operations that can be performed on it.
  - Describes *what* a collection does, not *how* it does it.

- Java's collection framework uses interfaces to describe ADTs:
  - `Collection`, `Deque`, `List`, `Map`, `Queue`, `Set`

- An ADT can be implemented in multiple ways by classes:
  - `ArrayList` and `LinkedList` implement `List`
  - `HashSet` and `TreeSet` implement `Set`
  - `LinkedList`, `ArrayDeque`, etc. implement `Queue`

Why use ADTs?

- Why would we want more than one kind of list, queue, etc.?

- **Answer**: Each implementation is more efficient at certain tasks.
  - `ArrayList` is faster for adding/removing at the end; `LinkedList` is faster for adding/removing at the front/middle. Etc.

  - You choose the optimal implementation for your task, and if the rest of your code is written to use the ADT interfaces, it will work.

  - Abstract data type in data structure type allows **reusability** of a **code** i.e. it makes it convenient for a programmer to write a shorter code.

Abstract Data Type (ADT)

Type defined in terms of its data items and associated operations, not its implementation.

Abstract Data Types (ADT)

- We are well acquainted with data types by now, like integers, arrays, and so on.
- To access the data, we've used operations defined in the programming language for the data type, for instance by accessing array elements by using the square bracket notation.
- An abstract data type is a data type whose representation is hidden from, and of no concern to the application code.

An Abstract Data Type (ADT) is a **blueprint for creating a data structure** that **defines the behavior and interface of the structure, without specifying how it is implemented.**

An ADT is a **set of operations that can be performed on a set of values**. This set of operations actually **defines the behavior of the data structure,** and they are used to manipulate the data in a way that suits the needs of the program. **ADT allows the programmers to use the functions while hiding the implementation details.**

Examples of abstract data type in data structures are List, Stack, Queue, etc.

Abstract Data Types (ADT)

- For example, when writing application code, we don't care how strings are represented: we just declare variables of type string, and manipulate them by using string operations.

- Once an abstract data type has been designed, the programmer responsible for implementing that type is concerned only with choosing a suitable data structure and coding up the methods.

- On the other hand, application programmers are concerned only with using that type and calling its methods without worrying much about how the type is implemented.

The Array As An ADT

- An array is probably the most versatile or fundamental Abstract Data Type

- An **array** is a finite sequence of storage cells, for which the following operations are defined:
  - create(A,N)creates an array A with storage for N items;
  - A[i]=item stores item in the i^{th} position in the array A; and
  - A[i] returns the value of the item stored in the i^{th} position in the array A.

Showing that an array is an ADT

Let A be an array of type T and has n elements then it satisfied the following operations

1. CREATE(A): Create an array A
2. INSERT(A,X): Insert an element X into an array A in any location
3. DELETE(A,X): Delete an element X from an array A
4. MODIFY(A,X,Y): modify element X by Y of an array A
5. TRAVELS(A): Access all elements of an array A
6. MERGE(A,B): Merging elements of A and B into a third array C

Thus by using 1D array we can perform above operations thus an array acts as an ADT.

Definition

- A stack is an ***ordered collection of items*** into which new items may be ***inserted*** and from which items may be ***deleted*** at one end, called the ***top*** of the stack.

- Stack is a linear data structure where all the ***insertions*** and ***deletions*** are done at end rather than in the middle.

Stack as an abstract data type

- A stack of elements of type T is a finite sequence of elements together with the operations

1. **CreateEmptyStack(S)**: create or make stack **S** be an empty stack
2. **Push(S,x):** Insert **x** at one end of the stack, called its **top**
3. **Top(S):** If stack **S** is not empty; then retrieve the element at its **top**
4. **Pop(S):** If stack **S** is not empty; then delete the element at its **top**
5. **IsFull(S):** Determine if **S** is full or not. Return **true** if **S** is full stack; return **false** otherwise
6. **IsEmpty(S)**: Determine if **S** is empty or not. Return **true** if **S** is an empty stack; return **false** otherwise.

Operations

- Push – Place something on stack on the top
- Pop – Remove the very top item
- Top – examine without removing top item
- IsEmpty – Tells us whether the stack is empty
- size – how many elements are there?

The advantages of ADT in Data Structures are:

- Provides **abstraction**, which simplifies the complexity of the data structure and **allows users to focus on the functionality.**
- Enhances **program modularity** by allowing the data structure implementation to be separate from the rest of the program.
- Enables **code reusability** as the same data structure can be used in multiple programs with the same interface.
- Promotes the concept of **data hiding** by encapsulating data and operations into a single unit, which **enhances security and control over the data.**
- Supports **polymorphism**, which allows the same interface to be used with different underlying data structures, providing **flexibility and adaptability** to changing requirements.

```
typedef struct {
   double real;
   double imag;
} complex;
```

Let us also illustrate the implementation of some arithmetic routines on complex numbers:

```
complex cadd ( complex z1 , complex z2 )
{
   complex z;
   z.real = z1.real + z2.real;
   z.imag = z1.imag + z2.imag;
   return z;
}

complex cmul ( complex z1 , comple z2 )
{
   complex z;
   z.real = z1.real * z2.real - z1.imag * z2.imag;
   z.imag = z1.real * z2.imag + z1.imag * z2.real;
   return z;
}

complex conj ( complex z1 )
{
   complex z;
   z.real = z1.real;
   z.imag = -z1.imag;
   return z;
}

void cprn ( complex z )
{
   printf("(%lf) + i(%lf)", z.real, z.imag);
}
```

Complex Numbers ADT Implementation

Matrix ADT Implementation

```
#define MAXROW 10
  #define MAXCOL 15
  typedef struct {
    int rowdim;
    int coldim;
    complex entry[MAXROW][MAXCOL];
  } matrix;
```

in order to define an ADT we need to specify:

- The components of an object of the ADT.
- A set of procedures that provide the behavioral description of objects belonging to the ADT.

Let us now implement some basic arithmetic operations on these matrices.

```
matrix msetid ( int n )
  {
    matrix C;
    int i, j;

    if ((n > MAXROW) || (n > MAXCOL)) {
      fprintf(stderr, "msetid: Matrix too big\n");
      C.rowdim = C.coldim = 0;
      return C;
    }
    C.rowdim = C.coldim = n;
    for (i = 0; i < C.rowdim; ++i) {
      for (j = 0; j < C.coldim; ++j) {
        A.entry[i][j].real = (i == j) ? 1 : 0;
        A.entry[i][j].imag = 0;
      }
    }
    return C;
  }
```

```
matrix madd ( matrix A , matrix B )
  {
    matrix C;
    int i, j;

    if ((A.rowdim != B.rowdim) ||
(A.coldim != B.coldim)) {
      fprintf(stderr, "madd: Matrices of
incompatible dimensions\n");
      C.rowdim = C.coldim = 0;
      return C;
    }

    C.rowdim = A.rowdim;
    C.coldim = A.coldim;
    for (i = 0; i < C.rowdim; ++i)
      for (j = 0; j < C.coldim; ++j)
        C.entry[i][j] =
cadd(A.entry[i][j],B.entry[i][j]);
    return C;
  }
```

```
matrix mmul ( matrix A , matrix B )
  {
    matrix C;
    int i, j, k;
    complex z;

    if (A.coldim != B.rowdim) {
      fprintf(stderr, "mmul: Matrices of incompatible dimensions\n");
      C.rowdim = C.coldim = 0;
      return C;
    }
    C.rowdim = A.rowdim;
    C.coldim = B.coldim;
    for (i = 0; i < A.rowdim; ++i) {
      for (j = 0; j < B.coldim; ++j) {
        C.entry[i][j].real = 0;
        C.entry[i][j].imag = 0;
        for (k = 0; k < A.coldim; ++k) {
          z = cmul(A.entry[i][k], B.entry[k][j]);
          C.entry[i][j] = cadd(C.entry[i][j],z);
        }
      }
    }
    return C;
  }
```

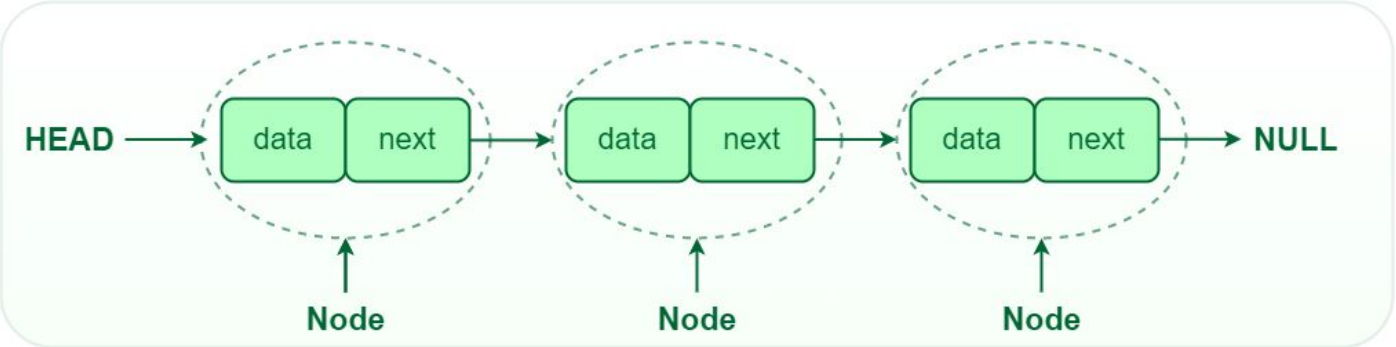
Linear Data Structures

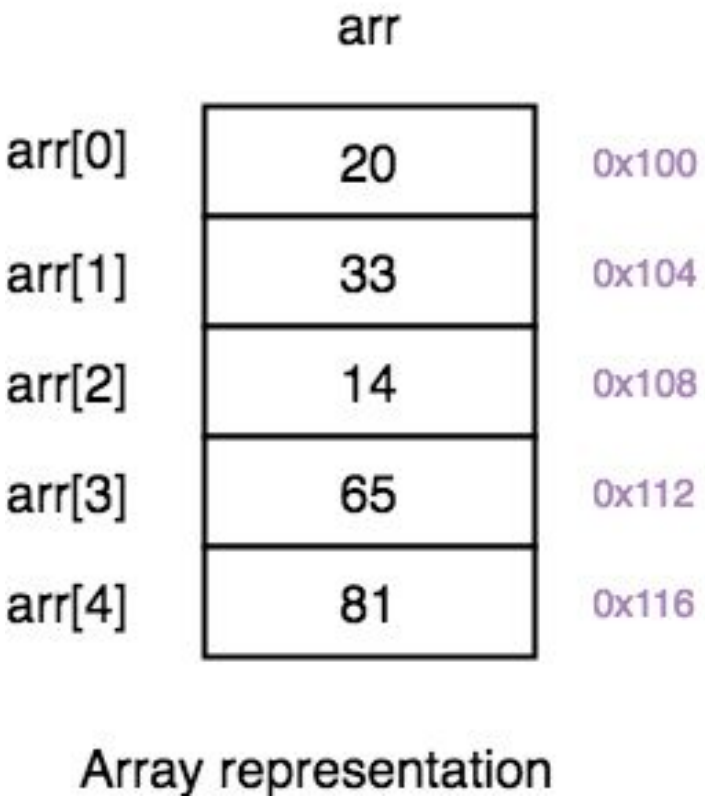
Linear list

- A sequence of elements
- There is first and last element
- Each element has previous and next
  - Nothing before first
  - Nothing after last

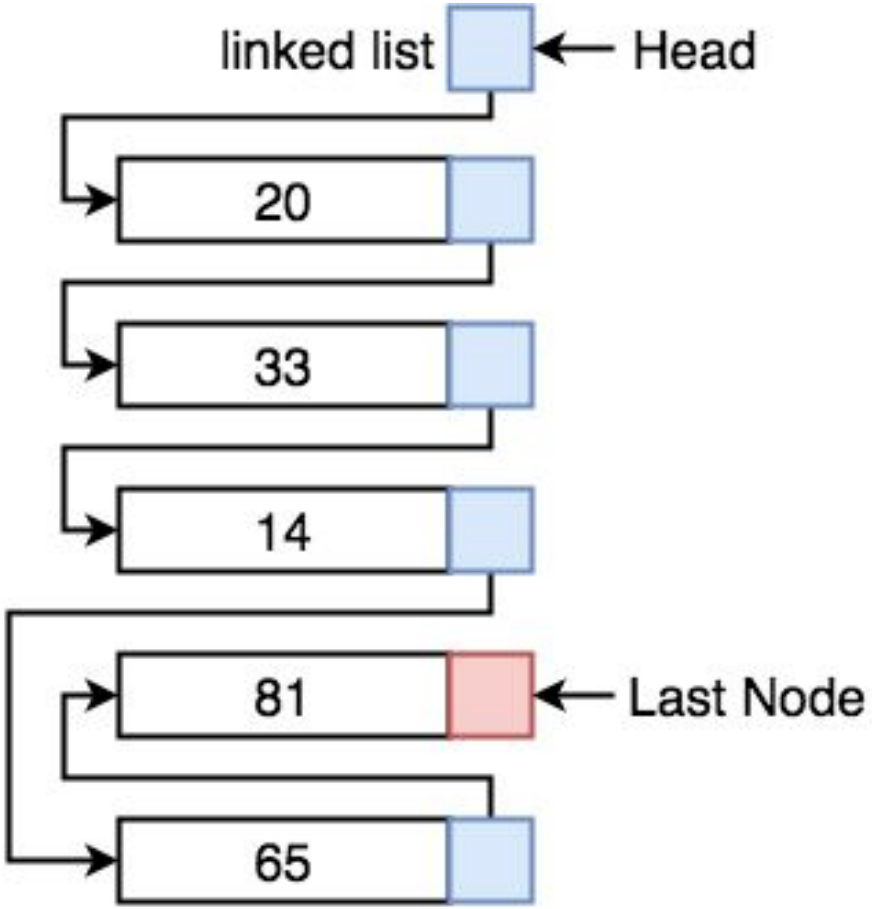
Why linked lists?

- A linked list is a dynamic data structure.
  - It can grow or shrink in size during the execution of a program.
  - It can be made just as long as required.
  - It does not waste memory space.
- Linked lists provide flexibility in allowing the items to be rearranged efficiently.
  - Insert an element.
  - Delete an element.





Array representation



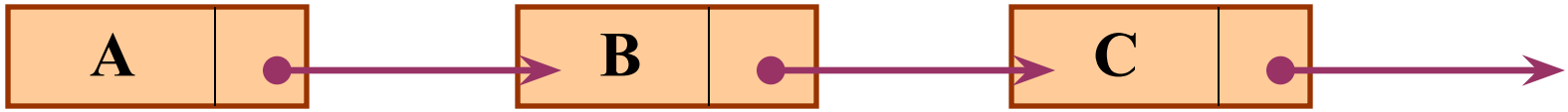
Array Vs. link list

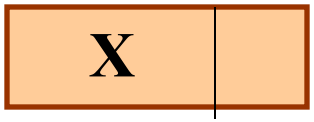
Arrays	Linked list
Fixed size: Resizing is expensive	Dynamic size
Insertions and Deletions are inefficient: Elements are usually shifted	Insertions and Deletions are efficient: No shifting
Random access i.e., efficient indexing	No random access → Not suitable for operations requiring accessing elements by index such as sorting
No memory waste if the array is full or almost full; otherwise may result in much memory waste.	Since memory is allocated dynamically(acc. to our need) there is no waste of memory.
Sequential access is faster [Reason: Elements in contiguous memory locations]	Sequential access is slow [Reason: Elements not in contiguous memory locations]

Operations on LL

- What we can do with a linear list?
  - Delete element
  - Insert element
  - Find element
  - Traverse list

Illustration: Insertion





Item to be inserted

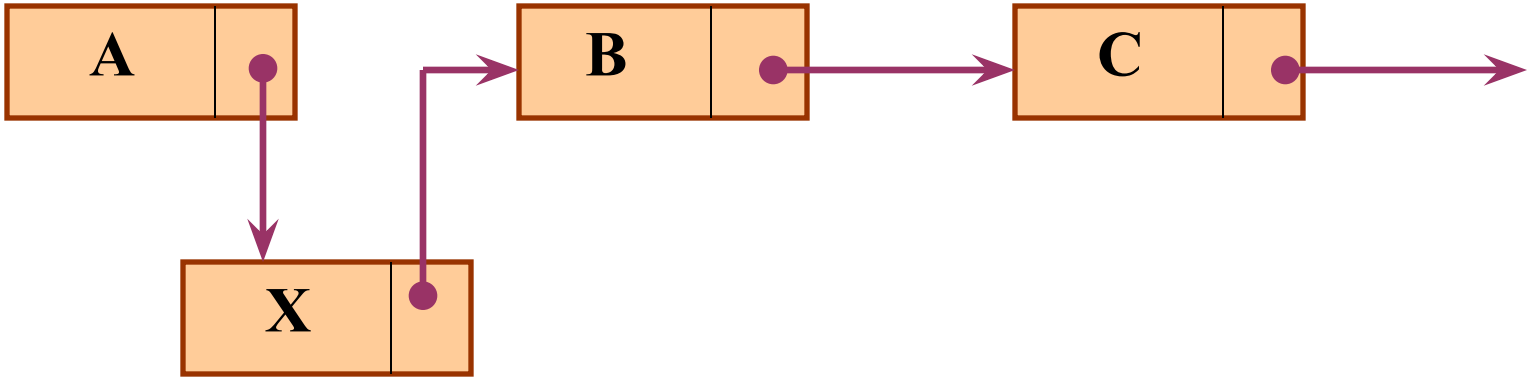
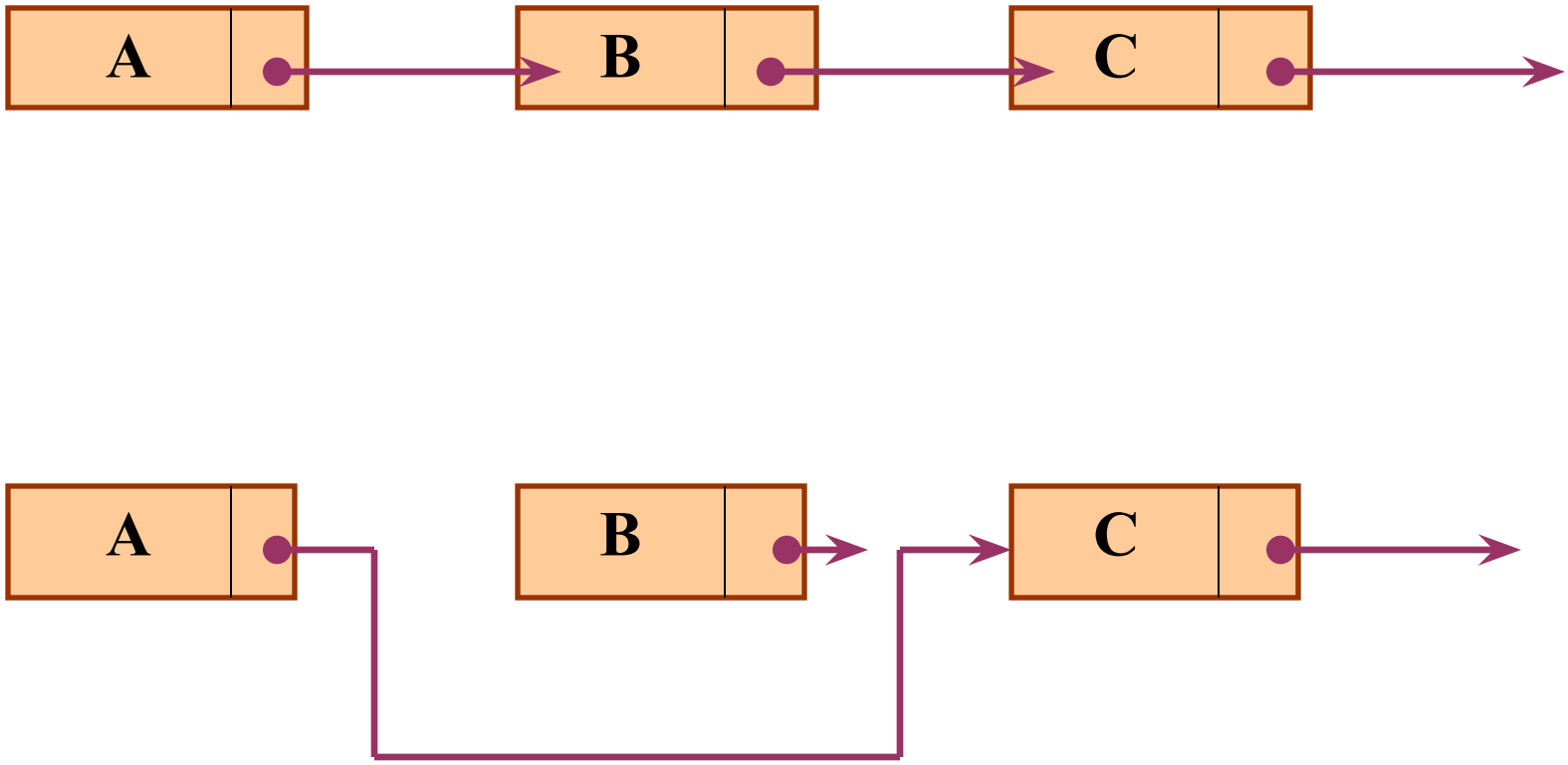
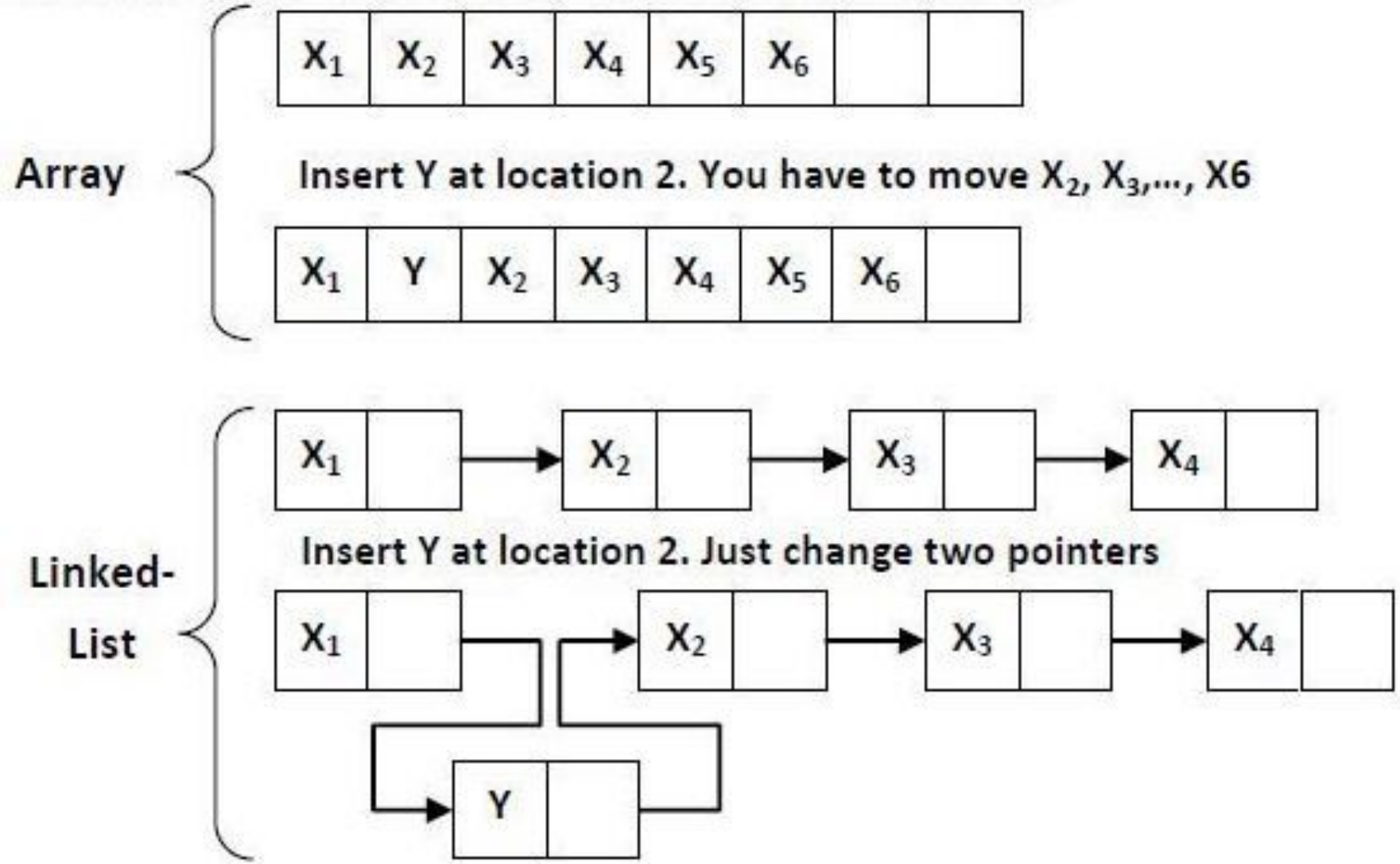


Illustration: Deletion



In essence ...

- For insertion:
  - A record is created holding the new item.
  - The next pointer of the new record is set to link it to the item which is to follow it in the list.
  - The next pointer of the item which is to precede it must be modified to point to the new item.
- For deletion:
  - The next pointer of the item immediately preceding the one to be deleted is altered, and made to point to the item following the deleted item.


Array
X1 X2 X3 X4 X5 X6
Insert Y at location 2. You have to move X2, X3,..., X6
X1 Y X2 X3 X4 X5 X6
Linked-List
X1 X2 X3 X4
Insert Y at location 2. Just change two pointers
X1 X2 X3 X4
Y

Traverse: list ⇨ elements in order

- get_first(list) -
  - returns first element if it exists
- get_next(list) -
  - returns next element if it exists
- Both functions return NULL otherwise
- Calling get_next in a loop we will get one by one all elements of the list

How we can implement a list?

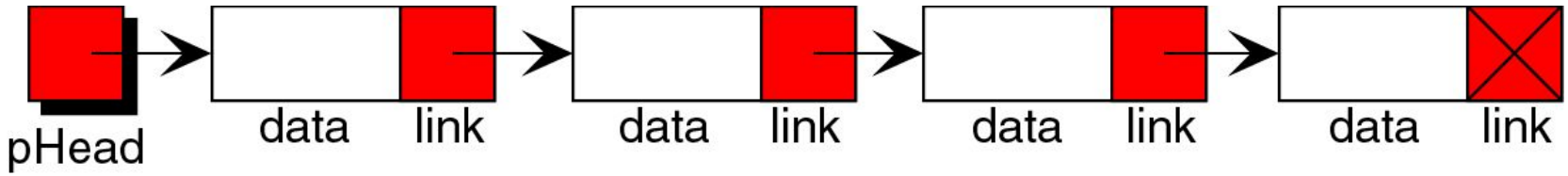
- Array?
- Search is easy (sequential or binary)
- Traversal is easy:

```
for(i = first; i <= last; ++i)
    process(a[i]);
```

- Insert and delete is *not* easy
  – a good part of the array has to be moved!
- Hard to guess the size of an array

A *linked list* implementation

- Linked list is a chain of elements
- Each element has data part and link part pointing to the next element



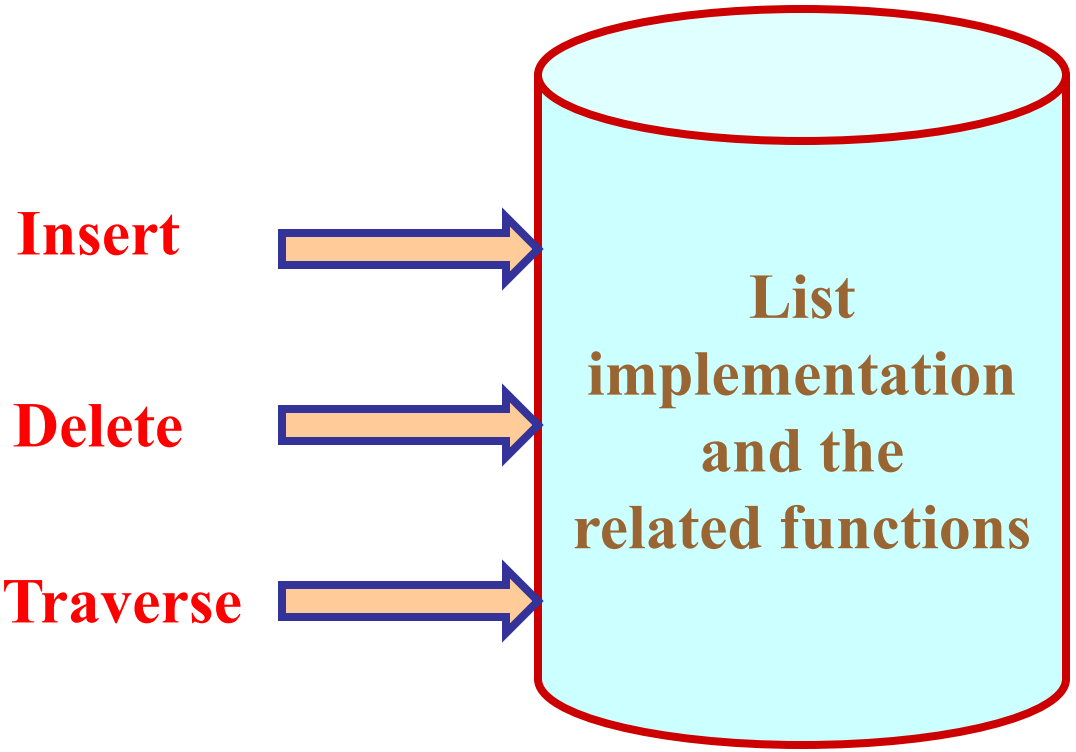
(a) A linked list with a head pointer: pHead

(b) An empty linked list

Main operations

- Create list
- Add node
  - beginning,middle or end
- Delete node
  - beginning,middle or end
- Find node
- Traverse list

Conceptual Idea



Example: Working with linked list

- Consider the structure of a node as follows:

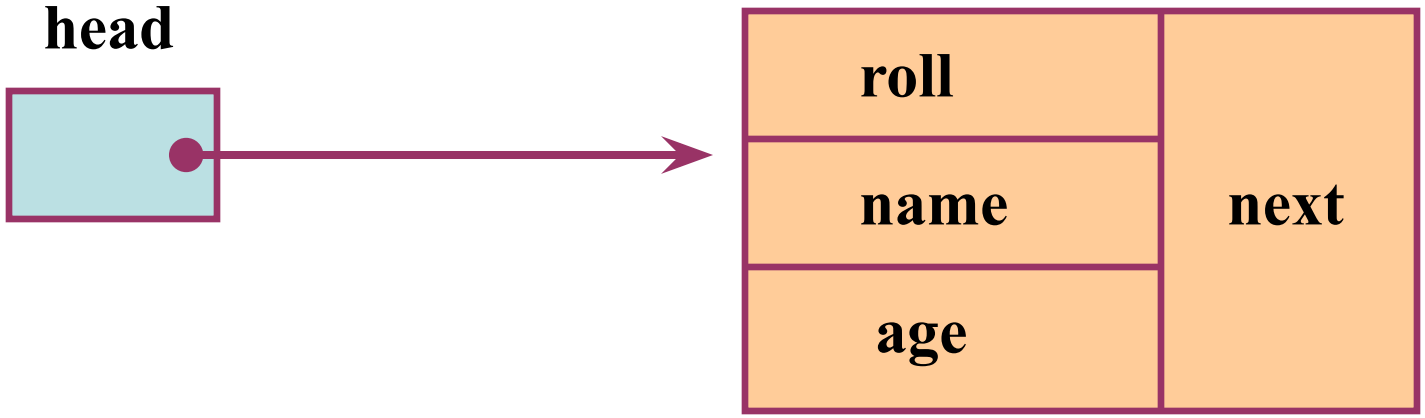
```
struct  stud  {
                int     roll;
                char  name[25];
                int     age;
                struct  stud  *next;
              };

        /* A user-defined data type called “node” */
typedef  struct  stud  node;
node  *head;
```

##typedef is used to define a data type in C.

Creating a List

- To start with, we have to create a node (the first node), and make head point to it.

```
head = (node *) malloc (sizeof (node));
```



Contd.

- If there are n number of nodes in the initial linked list:
  - Allocate n records, one by one.
  - Read in the fields of the records.
  - Modify the links of the records so that the chain is formed.

```
void  create_list  (node  *list)
{
   int  k, n;
   node  *p;
   printf  ("\n How many elements?");
   scanf ("%d", &n);

   list  =  (node *)  malloc (sizeof (node));
   p = list;
   for  (k=0; k<n; k++)
   {
      scanf ("%d %s %d", &p->roll,
                          p->name, &p->age);
      p->next  = (node *) malloc
                              (sizeof (node));
      p  =  p->next;
   }
   free (p->next);
   p->next  =  NULL;
}
```

To be called from the main() function as:

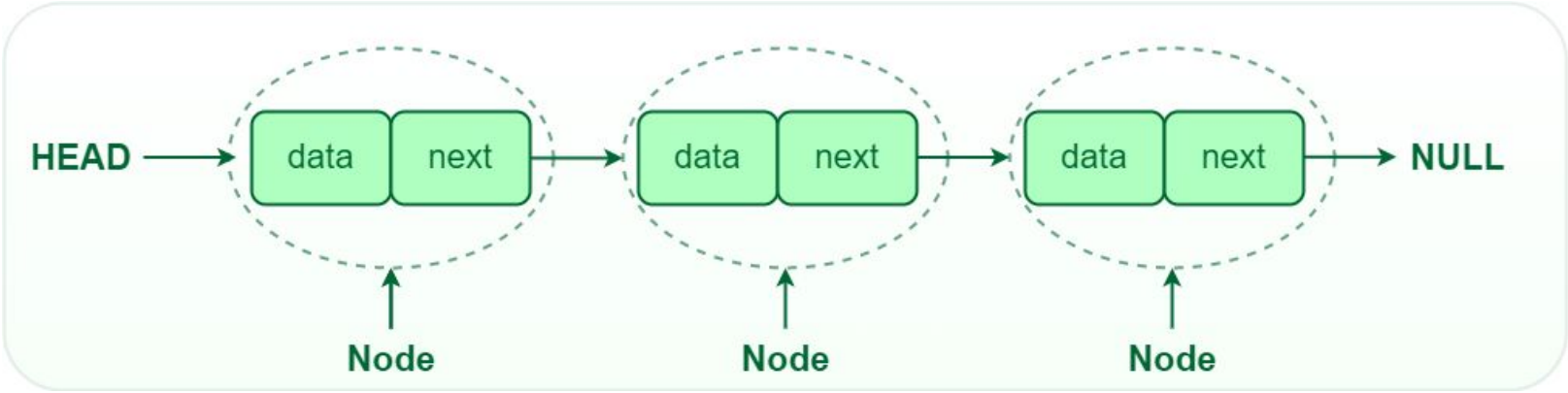
```
node  *head;
.......
create_list (head);
```

sizeof() is used to determine size in bytes of an element in C. Here it is used to determine size of each node and sent as a parameter to `malloc`.

malloc() is used to dynamically allocate a single block of memory in C, it is available in the header file `stdlib.h`.

Traversing the List

- Once the linked list has been constructed and head points to the first node of the list,
  - Follow the pointers.
  - Display the contents of the nodes as they are traversed.
  - Stop when the next pointer points to NULL.



Single-linked list

```
void  display_list  (node  *list)
{
   int  k = 0;
   node  *p;

   p = list;
   while  (p != NULL)
   {
     printf (“Node %d:  %d %s %d”, k, p->roll,
                                        p->name, p->age);
    k++;
     p  =  p->next;
   }
}
```

Inserting a Node in the List

- The problem is to insert a node before a specified node.
  - Specified means some value is given for the node (called key).
  - Here it may be roll.
- Convention followed:
  - If the value of roll is given as negative, the node will be inserted at the end of the list.

- When a node is added at the beginning,
  - Only one next pointer needs to be modified.
    - head is made to point to the new node.
    - New node points to the previously first element.
- When a node is added at the end,
  - Two next pointers need to be modified.
    - Last node now points to the new node.
    - New node points to NULL.
- When a node is added in the middle,
  - Two next pointers need to be modified.
    - Previous node now points to the new node.
    - New node points to the next node.

```
void  insert_node  (node  *list)
{
   int  k = 0, rno;
   node  *p, *q, *new;

   new  =  (node *)  malloc (sizeof (node));
   scanf ("%d %s %d", &new->roll, new->name,
                                        &new->age);

   printf ("\nInsert before roll (-ve for end):");
   scanf ("%d", &rno);

   p = list;
   if  (p->roll  ==  rno)      /* At the beginning */
   {
      new->next  =  p;
      list  =  new;
   }
```

```
   while  ((p != NULL) && (p->roll != rno))
   {
      q = p;
      p  =  p->next;
   }

   if  (p == NULL)          /* At the end */
   {
      q->next = new;
      new->next = NULL;
    }

    if  (p->roll  == rno)     /* In the middle */
   {
      q->next = new;
      new->next = p;
   }
}
```

The pointers q and p always point to consecutive nodes.

Deleting an Item

- Here also we are required to delete a specified node.
  - Say, the node whose roll field is given.
- Here also three conditions arise:
  - Deleting the first node.
  - Deleting the last node.
  - Deleting an intermediate node.

```
void  delete_node  (node  *list)
{
   int  rno;
   node  *p, *q;

   printf ("\nDelete for roll :");
   scanf ("%d", &rno);

   p = list;
   if  (p->roll == rno)          /* Delete the first element */
   {
      list = p->next;
      free (p);
   }
```

```
  while  ((p != NULL) && (p->roll != rno))
   {
      q = p;
      p  =  p->next;
   }


   if  (p == NULL)              /* Element not found */
       printf ("\nNo match :: deletion failed");

   if  (p->roll == rno)         /* Delete any other element */
   {
      q->next  =  p->next;
      free (p);
   }
}
```

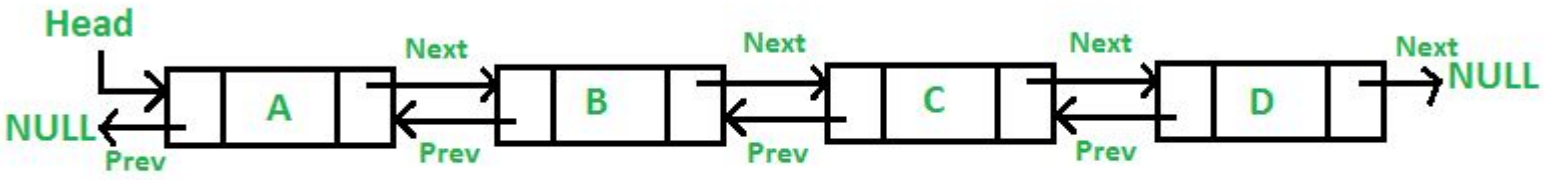
https://www.log2base2.com/data-structures/linked-list/linked-list-in-c.html

Types of Linked List

Singly Linked List: A Singly Linked List is the most common type of Linked List. Each node has data and a pointer field containing an address to the next node.

Doubly Linked List: A Doubly Linked List consists of an information field and two pointer fields. The information field contains the data. The first pointer field contains an address of the previous node, whereas another pointer field contains a reference to the next node. Thus, we can go in **both directions** (backward as well as forward).

Doubly linked list



A **doubly linked list** is a linked data structure that consists of a set of sequentially linked records called nodes. Each node contains three fields: two link fields (references to the previous and to the next node in the sequence of nodes) and one data field.

In a doubly linked list, each node contains references to both the next and previous nodes. This allows for traversal in both forward and backward directions, but it **requires additional memory for the backward reference.**

- **Circular Linked List:** The Circular Linked List is similar to the Singly Linked List. The only key difference is that the **last node contains the address of the first node,** forming a **circular loop** in the Circular Linked List. It can be either singly or doubly linked.

Linked List Applications

- Dynamic memory allocation
- Implemented in stack and queue
- In undo functionality of softwares
- Hash tables, Graphs

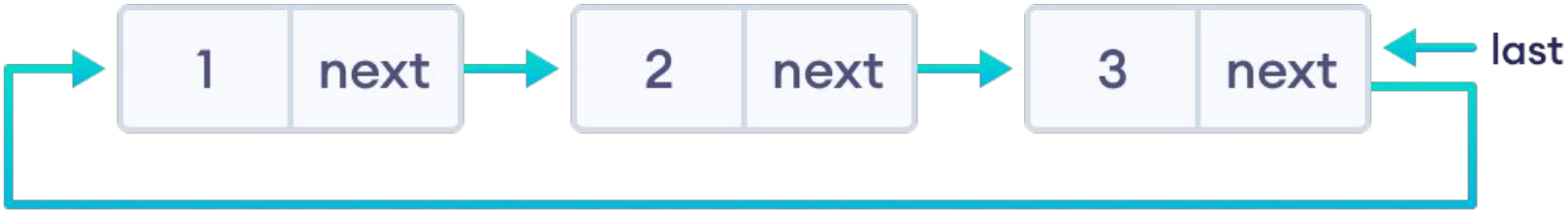
Advantages of Linked Lists

. **Dynamic Size:** Linked lists can grow or shrink dynamically, as memory allocation is done at runtime.
. **Insertion and Deletion:** Adding or removing elements from a linked list is efficient, especially for large lists.
. **Flexibility:** Linked lists can be easily reorganized and modified without requiring a contiguous block of memory.

Disadvantages of Linked Lists

. **Random Access:** Unlike arrays, linked lists do not allow direct access to elements by index. Traversal is required to reach a specific node.
. **Extra Memory:** Linked lists require additional memory for storing the pointers, compared to arrays.

Circular Linked List



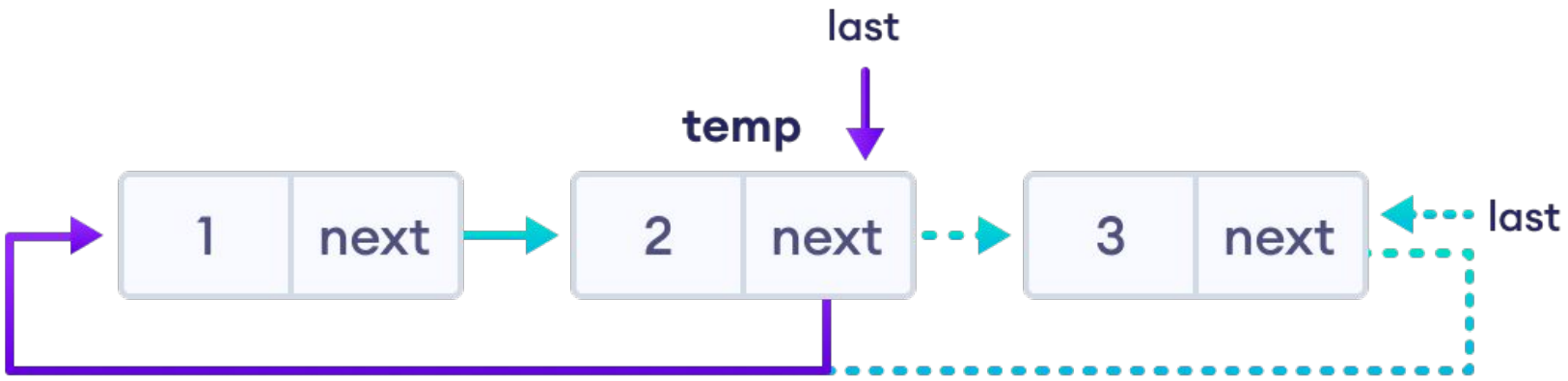
1. If the node to be deleted is the only node

- free the memory occupied by the node
- store NULL in `last`

2. If last node is to be deleted

- find the node before the last node (let it be `temp`)
- store the address of the node next to the last node in `temp`
- free the memory of last
- make `temp` as the last node

Deletion of Last Node



3. If any other nodes are to be deleted

- travel to the node to be deleted (here we are deleting node 2)
- let the node before node 2 be `temp`
- store the address of the node next to 2 in `temp`
- free the memory of 2

Any question?