# BITE 1523 Computer Game Programming

HAMZAH ASYRANI BIN SULAIMAN

# LEARNING OUTCOMES

- Describe the different abstract data type & algorithms used in game programming and the effects towards performance
- Apply structured data and algorithm in game application that requires data structures
- Produce game application by applying suitable type of data structures and algorithms to solve game programming problems

# BOOKS / REFERENCES


Ron Penton, “Data Structure For Game Programmers”, Game Development Series, The Premier Press, 2003. (EBook)

Eric S. Robert, “Programming Abstraction in C++”, Prentice Hall 1st edition , 2013. (EBook).

Allen Sherrod, “Data Structures and Algorithms for Game Developers”, Game Development Series, Charles River Media, Thomson Learning Inc. , 2007. (EBook).

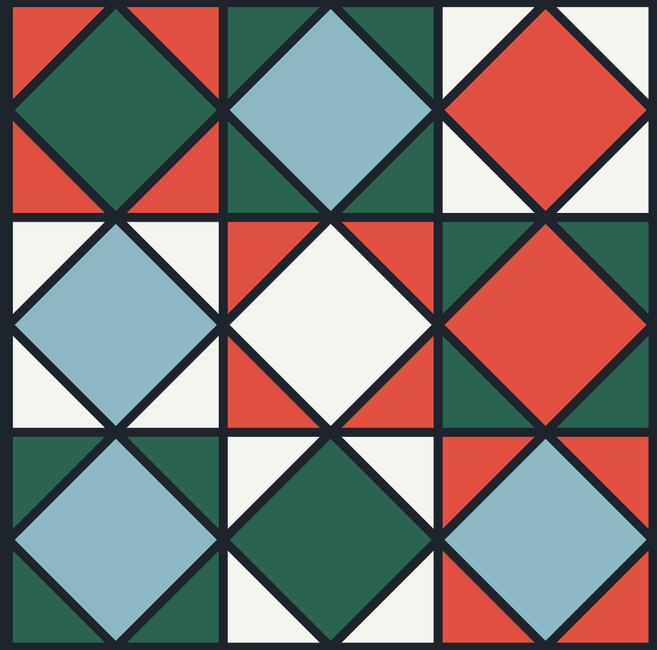
# COURSE SYLLABUS

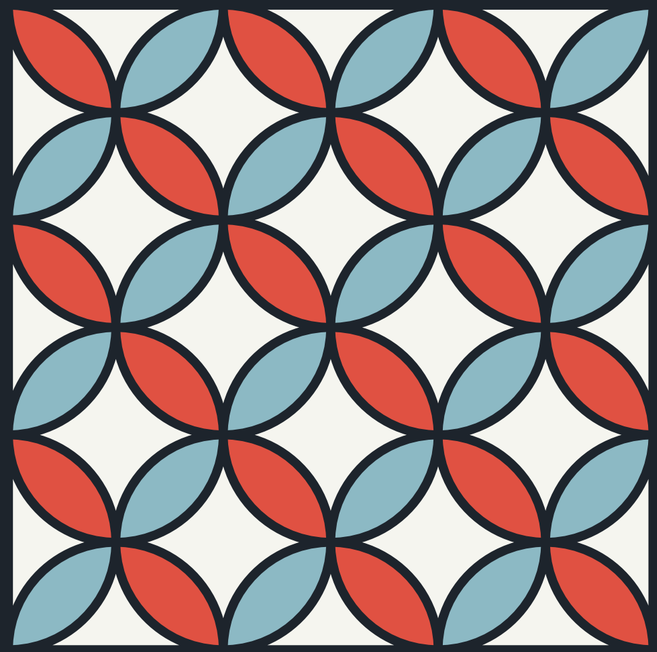
1. Concepts : Data Structure and Algorithm
2. Linked List (Part 1 & 2)
3. Stack
4. Queue
5. Random Numbers and Recursive
6. Sorting
7. Tree (Part 1 & 2)
8. Hash Tables
9. Graphs

BITE 1523 COMPUTER GAME PROGRAMMING

# CHAPTER 1:

# DATA STRUCTURE AND ALGORITHM





# OBJECTIVES:

By the end of the lesson the student will be able to:

**01**

Explain the concept of Abstract Data Type (ADT)

**02**

Explain the concept of Data Structure

**03**

Be able to describe data structure algorithm

# OUTLINE

- Concept
- Data
- Definition Data Structures
- Definition ADT
- Abstract View of DS
- Possible Implementation of DS
- Array vs Linked List

# WHAT IS DATA?

- Data is a set of values of qualitative or quantitative variables.
- Store, organize and group the data that is important.

.

1. Word Dictionary

tional industrial labor union that was organized in C
in 1905 and disintegrated after 1920. *Abbr.:* I.W.W.
**in·dus·tri·ous** (in dus′trē əs), *adj.* **1.** hard-working
gent. **2.** *Obs.* skillful. [< L *industrius*, OL *indostru*
disputed origin] —**in·dus′tri·ous·ly**, *adv.* —**in·du**
**ous·ness**, *n.* —**Syn. 1.** assiduous, sedulous, energeti
busy. —**Ant. 1.** lazy, indolent.
**in·dus·try** (in′də strē), *n., pl.* **-tries** for 1, 2. **1.** the
gate of manufacturing or technically productive enter
in a particular field, often nam after its principal pr
**2.** any general business field. de or manufact
general. **4.** owners and managers ively. **5.** syste
work or labor. **6.** assiduous activi y work or
diligence. [ME *industrie* < L *indus* y of fem.
*dustrius* INDUSTRIOUS] —**Syn. 6.** effor
**In′dus val′ley civiliza′tion,** an
that flourished in the Indus River valley
1500 B.C. Also called **Indus civilization.**
**in·dwell** (in dwel′), *v.*, **-dwelt, -dwell·ing.**
inhabit. **2.** to possess (a person), as a principle,
force, etc. —*v.i.* **3.** to dwell. **4.** to abide within, as a

2. City Map

3. Cash Account

GENERAL LEDGER

Account Name: CASH

Account Number: 001

| Date | Explanation | Ref | Debit | Credit | Balance |
|---|---|---|---|---|---|
| 1/1/2011 | Beg. Balance | | | | - |
| 12/01/11 | see GJ #1 | GJ1 | 250,000 | | 250,000 |
| 12/05/11 | see GJ #3 | GJ3 | | 50,000 | 200,000 |
| 12/10/11 | see GJ #4 | GJ4 | | 20,000 | 180,000 |
| 12/12/11 | see GJ #5 | GJ5 | 100,000 | | 280,000 |
| 12/24/11 | see GJ #6 | GJ6 | 5,000 | | 285,000 |
| 12/27/11 | see GJ #8 | GJ8 | | 100,000 | 185,000 |
| 12/27/11 | see GJ#9 | GJ9 | | 5,000 | 180,000 |
| 12/29/11 | see GJ #10 | GJ10 | | 20,000 | 160,000 |
| 12/29/11 | see GJ #11 | GJ11 | 4,000 | | 164,000 |
| 12/29/11 | see GJ #13 | GJ13 | | 25,000 | 139,000 |
| 12/30/11 | see GJ #14 | GJ14 | 50,000 | | 189,000 |

* Different kind of DS (Data Structure) needed for different kind of data. (Texts, Images, Videos, Sounds etc).

# LOOK AT THE EXAMPLES

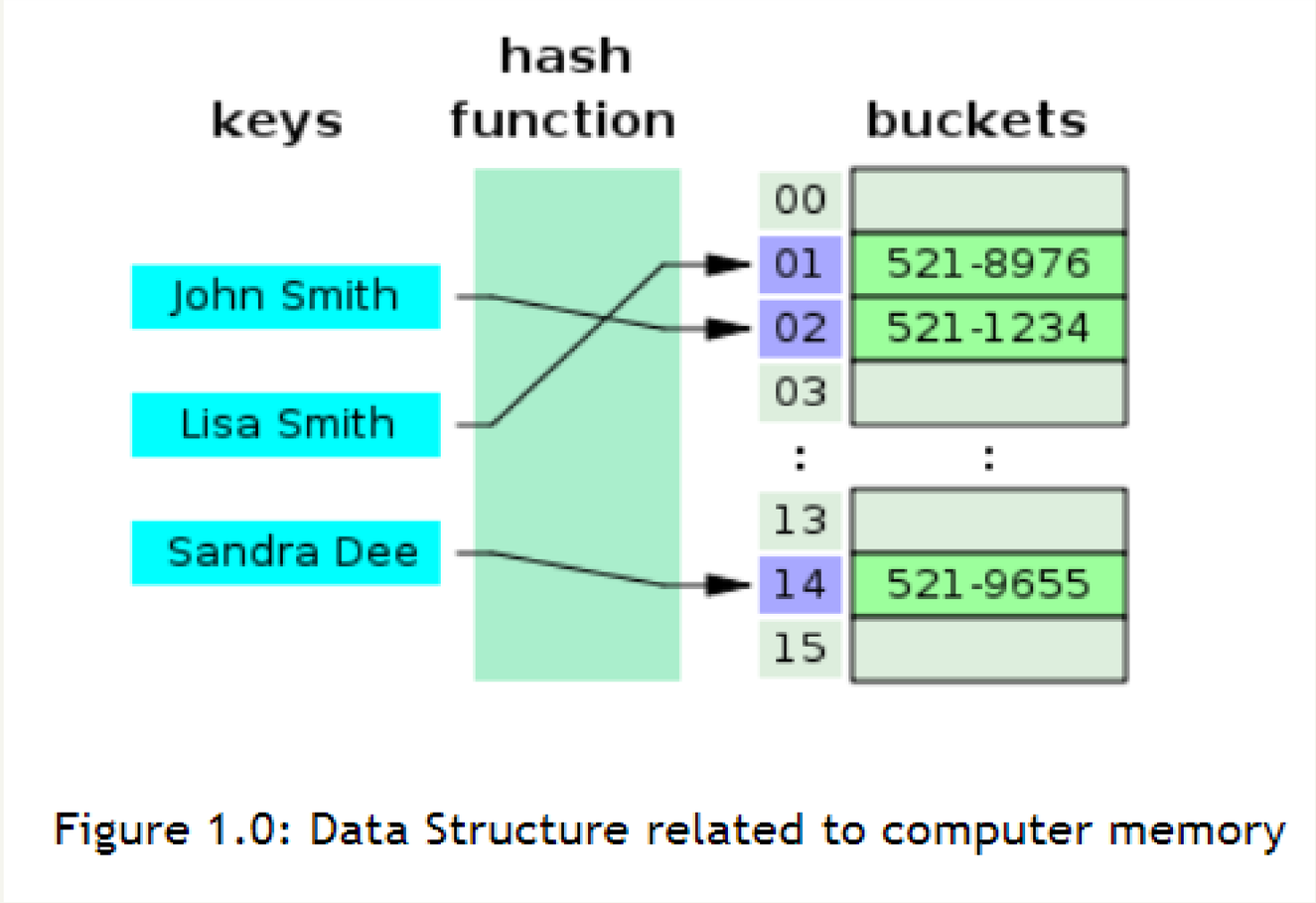



Figure 1.0: Data Structure related to computer memory

# DATA STRUCTURES AND ALGORITHMS

- Data structures: The building blocks of software engineering.
- Every application must manage and manipulate data in some meaningful way to perform a task.
- In modern video games this data is used to create a complex interactive experience.
- Data structures are meaningful when they are combined with algorithms.

# DEFINITION OF DATA STRUCTURES:

- A data structure defines how data is arranged in memory and can be operated on by using various algorithms. Example array.
- An array is a data structure because it defines how data is arranged in memory, operated on by various algorithms (e.g., insertion into the array, deletion, searching, sorting, and so forth).
- The data structures include the following:

  (Link lists, Queues , Stacks ,Heaps, Graphs, Scene graphs, Octrees, etc.)

# DATA STRUCTURE

## OPERATIONS TO THE DATA STRUCTURE

### TRAVERSING

Access and process every data in data structure at least once

### SEARCHING

Search for a location of data

### INSERTION

Insert item in the list of data

### DELETION

Delete item from a set of data

### SORTING

Sort data in certain order

### MERGING

Merge multiple group of data

# DATA TYPES

## BASIC DATA TYPES (C++)

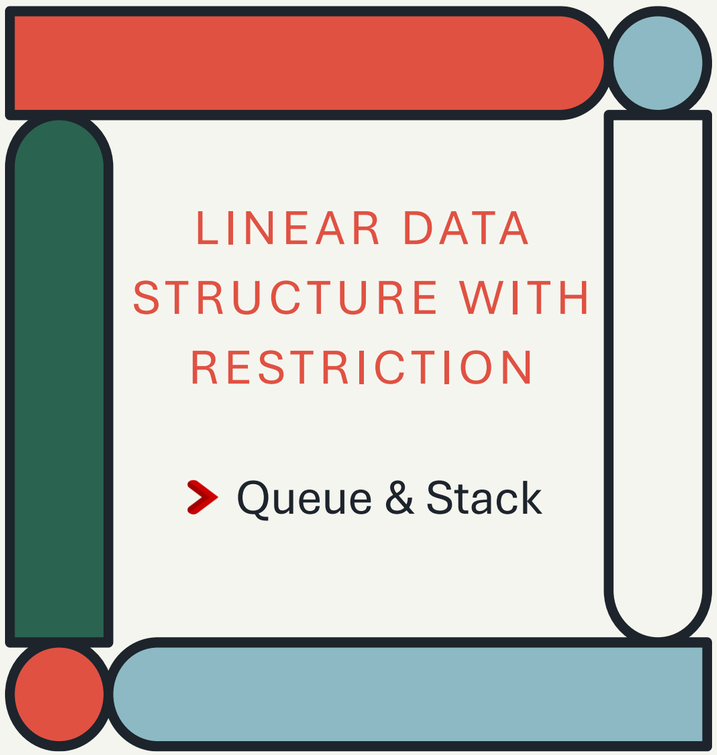
Store only a single data

- Boolean –bool
- Enumeration –enum
- Character –char
- Integer –short, int, long
- Floating point –float, double

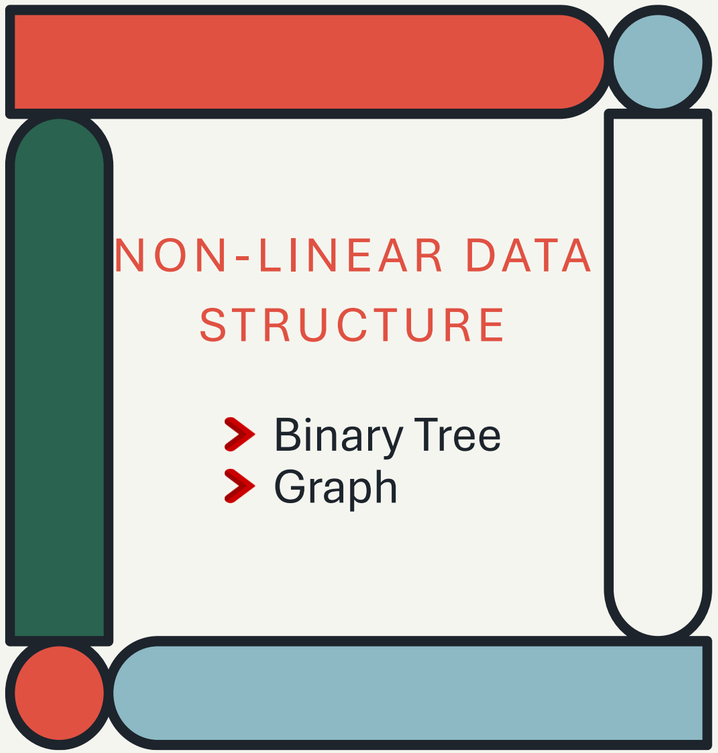
## STRUCTURED DATA TYPES

- Array –can contain multiple data with the same types
- Struct–can contain multiple data with different type

# DATA TYPES







# DEFINITION OF ALGORITHMS:

- An algorithm is code that manipulates data in data structures.
- Most algorithms apply to general data structures that include inserting items into a data structure, deleting items, sorting, and iterating.
- Example: Recursion, Insertions, Deletions, Merging, Various sorting algorithms and Various searching algorithms

# ALGORITHMS:

- It is a step-by-step procedure for performing a task within a finite period of time.
- Algorithms often operate on a collection of data, which is stored in a structured way in the computer memory (Data Structure)
- Algorithms: Problem solving using logic.

# ALGORITHMS

## Well defined instructions in algorithms includes:

- When given an initial state, (INPUT)
- Proceed through a well-defined series of successive states, (PROCESS)
- Eventually terminating in an end-state (OUTPUT)

## Algorithm creation techniques:

- Flowchart, pseudo code, language, etc)

# ALGORITHMS

## 3 types of algorithm basic control structure:

- Sequential
- Selection
- Repetition (Looping)

## Basic algorithm characteristics:

- Finite solution
- Clear instructions
- Has input to start the execution
- Has output as the result of the executions
- Operate effectively

## Factors for measuring good algorithm:

- Running time
- Total memory usage

# ABSTRACT DATA TYPE (ADT)

- In computer science, an abstract data type (ADT) is a mathematical model for data types, where a data type is defined by its behavior (semantics) from the point of view of a user of the data, specifically in terms of possible values, possible operations on data of this type, and the behavior of these operations.
- This contrasts with data structures, which are concrete representations of data, and are the point of view of an implementer, not a user.
- Data structures can implement one or more particular abstract data types (ADT).

Imagine an **Abstract Data Type (ADT)** like a **recipe** for making a cake. The recipe doesn’t care how you store the ingredients or what kind of oven you use — it just tells you:

- **What the cake should have** (flour, sugar, eggs — like "data").
- **What you can do with the cake** (bake, cut, eat — like "operations" on data).
- **What the result should be** (a finished cake, clean slices, tasty — like "operation behavior").

On the other hand, a **data structure** is like the **kitchen and tools** you use — for example, whether you use a metal or plastic bowl, an electric or wood-fired oven. That’s more about **how you make it happen**.

Example:

- **ADT**: "List" — you can add, remove, or search for items.
- **Data structure**: You can implement that list using an **array** or a **linked list** — that's the "how."
- So, ADT focuses on **what you want to achieve**, while data structures handle **how you make it work**.

# LOOK AT THE EXAMPLES (ADT)

1. Mathematical / logical models - Abstract View or high level features and operations that defines the DS

Example :

- Turned on/off
- Receive signals

Abstract View

# OBJECT ORIENTED PROGRAMMING (OOP) APPROACH

- OOP implements abstract data type:
- Abstract data type (ADT)

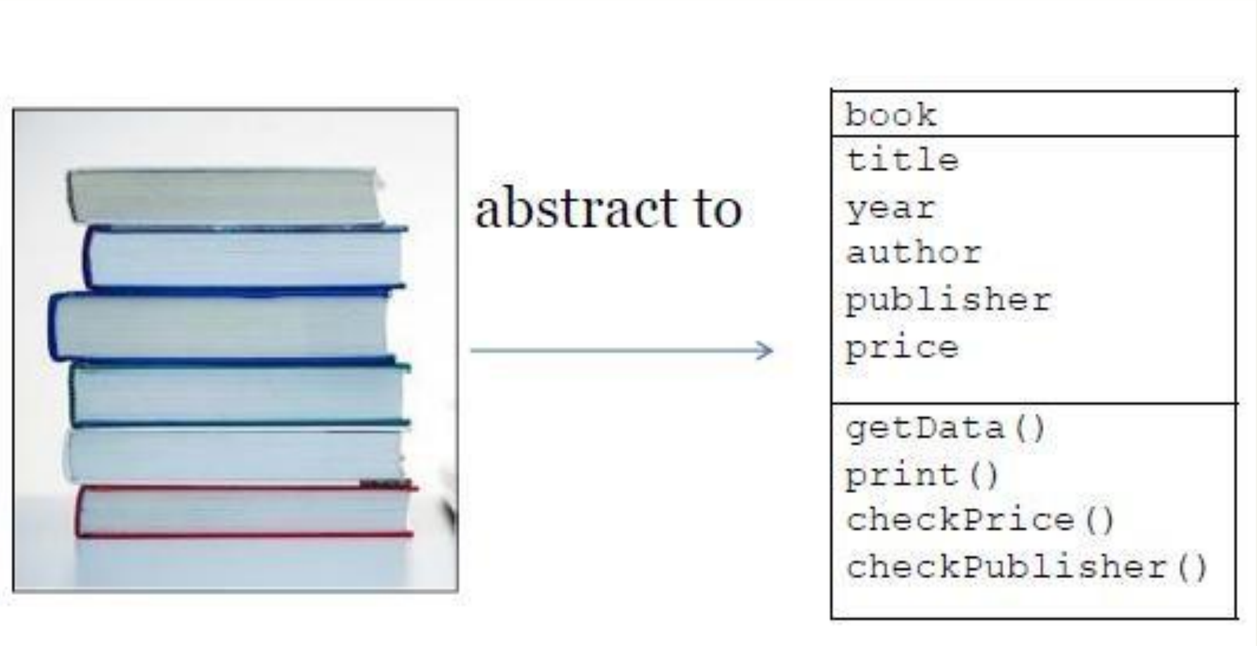
  A collection of data and a set of operations on the data

  Given the operations' specifications, the ADT's operations can be used without knowing their implementations or how data is stored
- Abstraction

  The purpose of a module is separated from its implementation

  Specifications for each module are written before implementation

# LOOK AT THE EXAMPLES (OOP APPROACH)




Abstraction of a book

# ABTRACT DATA TYPES (ADT)

## INTRODUCTION

Formal Definition:
An ADT is a model of a data structure that specifies:

- The characteristics of the collection of data
- The operations that can be performed on the collection
- It's abstract because it doesn't specify how the ADT will be implemented
  Ex : integer, real number, matrix
  Ex : common entity in real world
  List - collections of objects of the same type:
  words, names, numbers
- A given ADT can have multiple implementations

- Data Structures : Implementation - concrete type, not abstract. Can implement same adt in multiple ways in the same language e.g. c++ .

Some of them :
- Arrays
- Linked-list
- Stack
- Queue
- Tree
- Graph

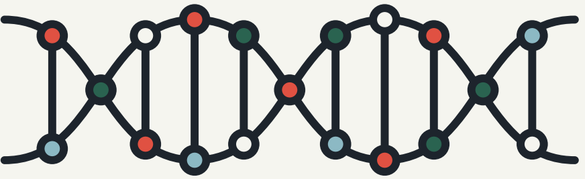
- Logical view
- Operations
- Implementations
- Cost of Operations (time)

# Recap

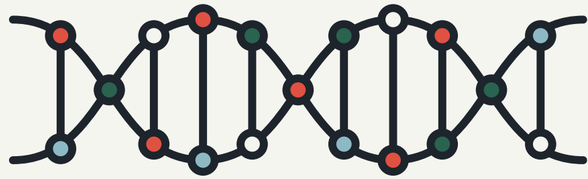
- Abstract Data Type (ADT) is the logical picture of the  data and the operations to manipulate the component  elements of the data.
  - ADT is in the logical level.
- Data Structure (DS) is the actual representation of the  data during the implementation and the algorithms to  manipulate the data elements.
  - Data Structure is in the implementation level.

# List as Abstract Data Type

- List - common entity in real world
  - Collections of objects of the same type :words, names, numbers
- Define the data in the list
- Collections of objects of the same type : words,  names, numbers, images, ID, command, resource  etc.



# LIST AS ABSTRACT DATA TYPE



- Operations of List as ADT

  - Empty list – size 0
  - Insert
  - Remove
  - Count
  - Read / Modify element at a position

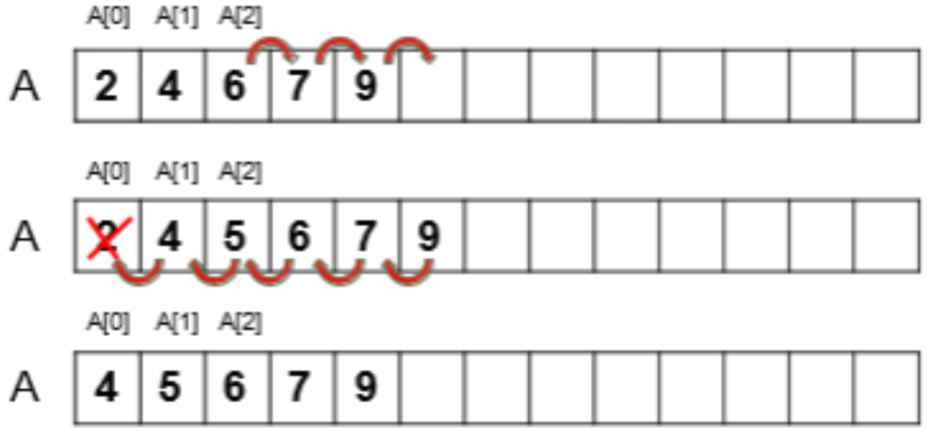
DS -Arrays

Ex)

```
int M[10];
A[i] = 4;
Print A[i];
```

We need a list that is dynamic, will have many more features, handle more scenarios. How to implement data structure for dynamic list ?
Array is static and fixed size. Is it possible to use array?

# LOOK AT THE EXAMPLES




```
int A [MAXSIZE];
int end = -1;
insert(2);
insert(4);
insert(6);
insert(7);
insert(9);
insert(5,2);
remove(0);
```

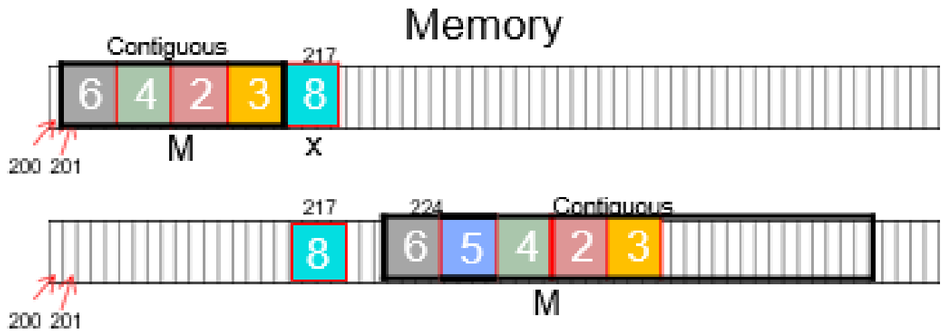
When array is full, create a new larger array, copy previous array into the new array and free the memory of the previous array

Creating, copying, freeing is costly (time) – should be avoided if want to develop a good design of software system

# LOOK AT THE EXAMPLES

- Using array as dynamic list has limitation. What is the limitation ?

```
int x;
x = 8;
int M[4];
```

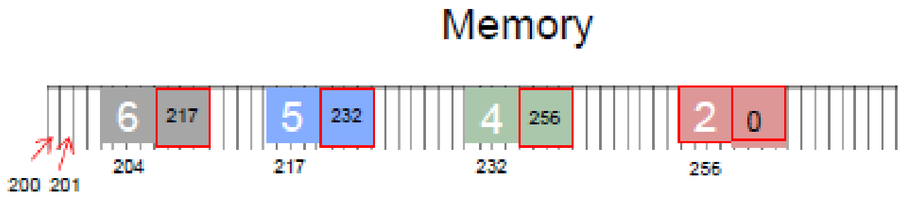



- Create new array
- Copy
- Free the memory of old array
- Again and again
- Some memory wasted

Solution ? Linked list

# LINKED LIST

6, 5, 4, 2

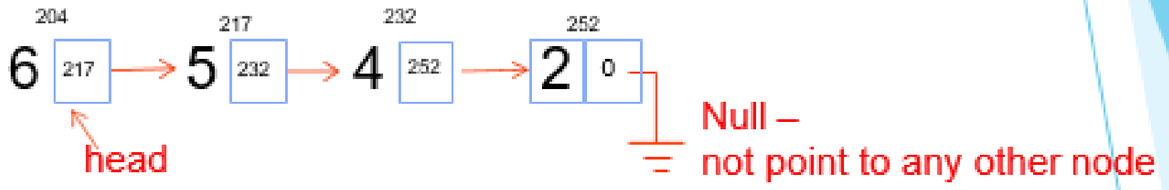



- Not contiguous block. Need to store information which is first block stores information about first element, second block stores information about second element and so on.
- Need to link these element/blocks together. Store extra information in each block – data and the address of next block

```
struct Node
{
    int data;//4 bytes
    Node* next; //4 bytes
}
```

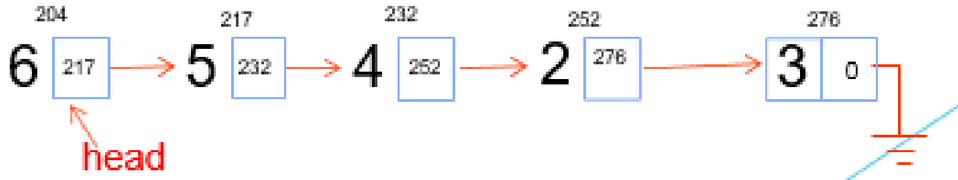
# LINKED LIST

## Logical View




First node is the head node

- Address in the head node give access to the complete list
- Address in the last node is Null or zero.
- To traverse – Starts with head get data and the address of the next node, then go to the second node, get data and address to the third node and so on.
- To insert an element (3) at the end of the list – create new node independently in the memory.



# 🎯 Array vs Linked List

| Feature | Array 🧱 | Linked List 🔗 |
|---|---|---|
| **Memory Allocation** | Fixed size (pre-allocated) | Grows or shrinks dynamically |
| **Access Time** | Fast (direct index access) 🔥 | Slower (node-by-node traversal) 🐢 |
| **Insertion/Deletion** | Slow (needs shifting elements) 🛠️ | Fast (update pointers) 🚀 |
| **Memory Usage** | More efficient (stores only data) 💾 | Extra memory for pointers 🔗 |
| **Cache Performance** | Better (continuous memory) ⚡ | Worse (scattered nodes) 🔍 |
| **Implementation** | Easier to set up 🛠️ | More complex (needs pointer logic) 🧠 |

# 🎯 Array vs Linked List

Array → Like a row of seats in a theater — easy to find your seat (index), but hard to add a new row.

Linked List → Like a chain of people holding hands — easy to add someone in the middle, but you need to start from the first person to find someone.

# THANK YOU

# Understanding Pointers in C++ - House Analogy

Making pointers easy to visualize

# What is a Pointer?

A pointer is a variable that stores the **address** of another variable.

Think of it like a person holding a paper with a house’s address.

# Step 1: Setting Up

- char ch1 = 'A', ch2 = 'Z';
- char *ptr1, *ptr2;

- House 1 (ch1) contains 'A'.
- House 2 (ch2) contains 'Z'.
- ptr1 and ptr2 are people without addresses yet.

# Step 2: Assign Address

- ptr1 = &ch1;
- ptr1 receives the address of House 1.

| Person | Holds Address Of | House Contains |
|---|---|---|
| ptr1 | House 1 (ch1) | 'A' |
| ptr2 | ??? | ??? |

# Step 3: Modify the Content

- *ptr1 = 'B';
- ptr1 goes to House 1 and changes the letter to 'B’.

| Person | Holds Address Of | House Contains |
|---|---|---|
| ptr1 | House 1 (ch1) | **'B'** |
| ptr2 | ??? | ??? |

# Step 4: Copy the Address

- ptr2 = ptr1;
- ptr2 now holds the same address as ptr1 — both point to House 1.

| Person | Holds Address Of | House Contains |
|---|---|---|
| ptr1 | House 1 (ch1) | 'B' |
| ptr2 | House 1 (ch1) | 'B' |

# Step 5: Change Address

- ptr1 = &ch2;
- ptr1 gets a new address — now it points to House 2.

| Person | Holds Address Of | House Contains |
|---|---|---|
| ptr1 | House 2 (ch2) | 'Z' |
| ptr2 | House 1 (ch1) | 'B' |

# Step 6: Copy Content

- *ptr1 = *ptr2;
- ptr1 copies the letter from House 1 ('B') into House 2.

| Person | Holds Address Of | House Contains |
|---|---|---|
| ptr1 | House 2 (ch2) | **'B'** |
| ptr2 | House 1 (ch1) | **'B'** |

# Final Output

- cout << ch1 << "\t" << ch2 << "\t" << *ptr1 << "\t" << *ptr2 << endl;

- Output:
- B B B B

# Key Takeaways

- A pointer stores an **address**, not a value.
- *ptr means **go to that address** and **access/change the value**.
- Pointers can switch addresses anytime.
- Changing content through a pointer changes the **original variable**.

- 🎯 Pointers are like people carrying addresses to different houses.

# Thank You!

- Questions or doubts? Let’s clear them up now!