Design and Analysis of Algorithms

Introduction to Course

8^{th} April, 2021

Dr. Ramesh Kumar
Associate Professor
Electronic Engineering Department, DUET Karachi

Introduction to Algorithm

- An algorithm is a finite set of instructions that commands machine to accomplish a particular task.
- In addition every algorithm must satisfy the following criteria:
  - Input: Zero/more quantities which are externally supplied
  - Output: At least one quantity is produced
  - Definiteness: Instruction must be clear and unambiguous
  - Finiteness: An algorithm must run instructions & terminate after a finite number of steps
  - Effectiveness: It must be a definite and feasible (carry out using paper and pencil)

Program Vs Algorithm Vs Flowchart

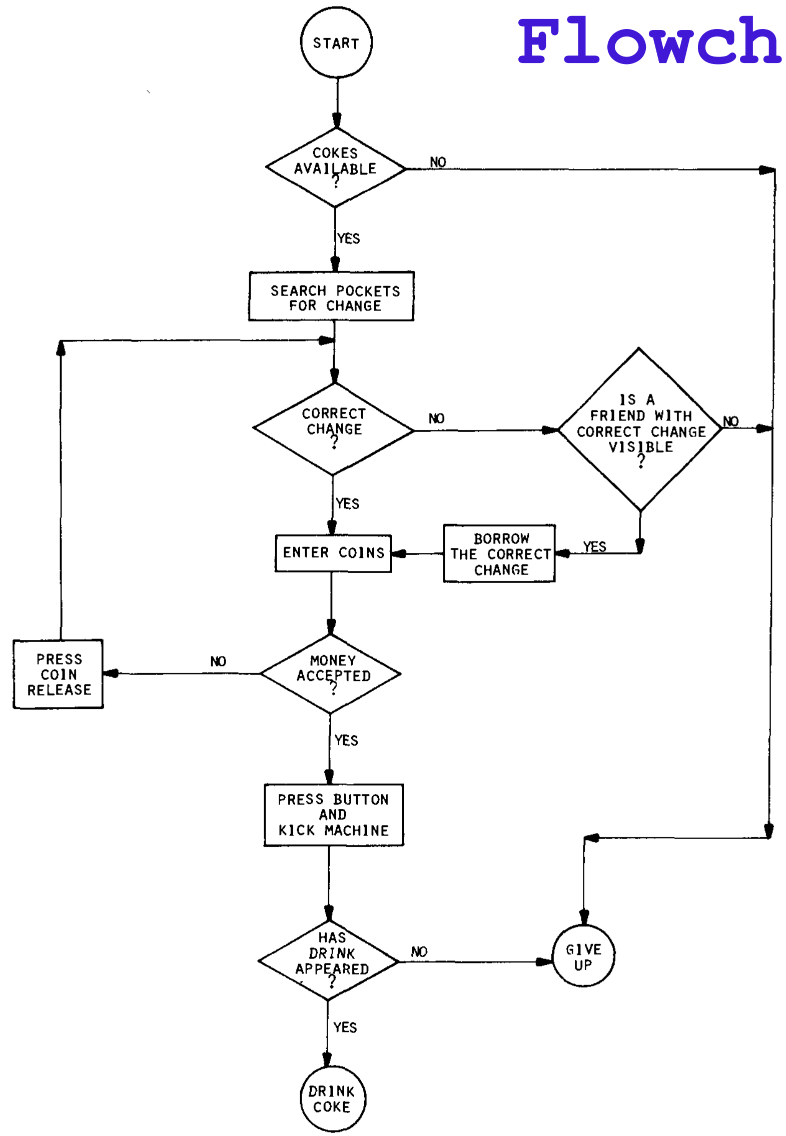
- Program:
  - Set of exact instructions written by using the syntax
  - Does not necessarily terminate in finite time. For example Operating system
- Algorithm:
  - Natural language (English) can be used for instruction
  - Instructions must be definite and terminate in finite time
  - Use some special characters also for specific purpose
- Flowchart:
  - Natural language coupled with graphical notations
  - This form places each processing step in a "box" and uses arrows to indicate the next step
  - Different shaped boxes stand for different kinds of operations.

Program Vs Algorithm Vs Flowchart

Algorithm

```
procedure FIBONACCI
   read (n)
   if n < 0 then [print ('error'); stop]
   if n = 0 then [print ('0'); stop]
   if n = 1 then [print ('1'); stop]
   fnm2 ← 0; fnm1 ← 1
   for i ← 2 to n do
      fn ← fnm1 + fnm2
      fnm2 ← fnm1
      fnm1 ← fn
   end
   print (fn)
end FIBONACCI
```

Flowchart



Program Vs Algorithm Vs Flowchart

```
#include "IntCell.h"

/**
 * Construct the IntCell with initialValue
 */
IntCell::IntCell( int initialValue ) : storedValue{ initialValue }
{
}

/**
 * Return the stored value.
 */
int IntCell::read( ) const
{
    return storedValue;
}

/**
 * Store x.
 */
void IntCell::write( int x )
{
    storedValue = x;
}
```

Algorithm Analysis

- Algorithm analysis means to estimate the resources used for an algorithm that solves a particular task
- An algorithm that solves a problem but:
  - Time constraint (e.g. year is too long to wait)
  - Space constraint (e.g. gigabytes of memory)
- Fast computations:
  - Computation time Big (O) notation
- Less memory storage

Data Structures

- A Stack is a last in first out (LIFO) data structure
  - Items are removed from a stack in the reverse order from the way they were inserted
- A Queue is a first in first out (FIFO) data structure
  - Items are removed from a queue in the same order as they were inserted
- A Deque is a double-ended queue—items can be inserted and removed at either end

Queue Data Structure

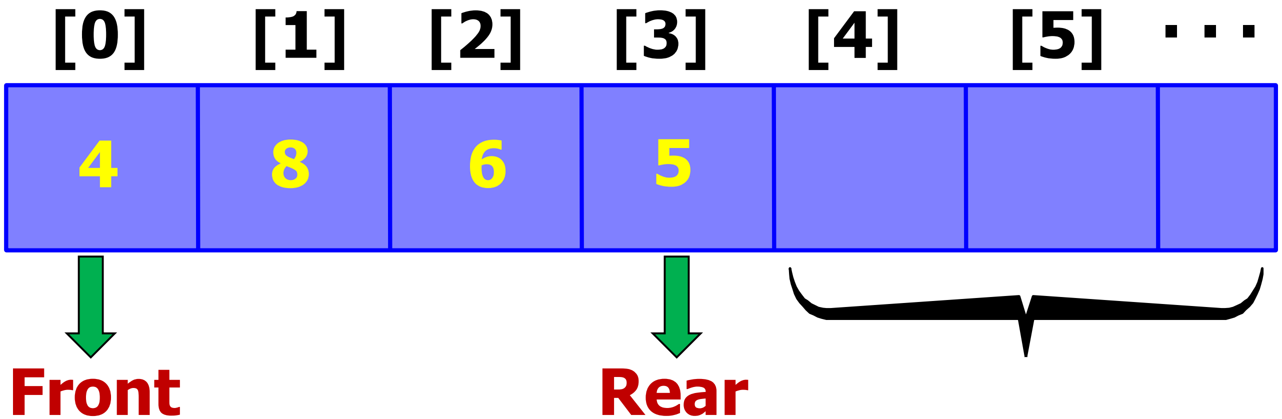
- It's a very important abstract data type (ADT)
- A FIFO data structure where:
  - Insertion of items can be done at one end (Rear)
  - Removal of items can be done at another end (Front)
- Major queue operations
  - Insertion of an item: Enqueue ()
  - Removal of an item: Dequeue ()
  - Search the items at start of a queue: Front ()
  - Check the size of queue: Size()
  - Check the completeness of queue IsFull ()
  - Check the presence of the items: IsEmpty ()
- Array and Linked-list

Queue (Applications)

- Operating systems:
  - Queue of print jobs to send to the printer
  - Queue of programs / processes to be run
  - Queue of network data packets to send
- Programming:
  - Modeling a line of customers or clients
  - Storing a queue of computations to be performed in order
- Real world examples:
  - People on an escalator/waiting in a line
  - Cars at a gas station (Assembly line)

Queue (Array Implementation)

- Use an integer array to implement a queue
- For example, the queue shown below contains the three elements 4 (Front), 8, 6, and 5 (Rear)

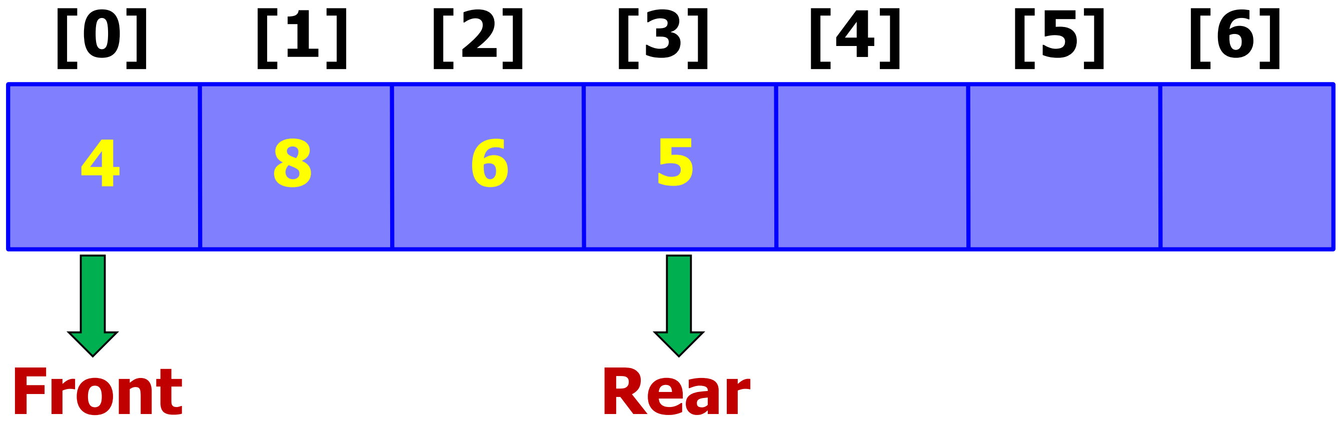


An array of integers to implement a queue of integers

- Time complexity is O(1)

Queue (Array Implementation)

```
int QueueArray[7];
int front=-1; int rear=-1;
boolean IsEmpty (){
      if (front=-1 && rear=-1)
            return true;
      else
            return false;
                        }
```



Queue (Array Implementation)

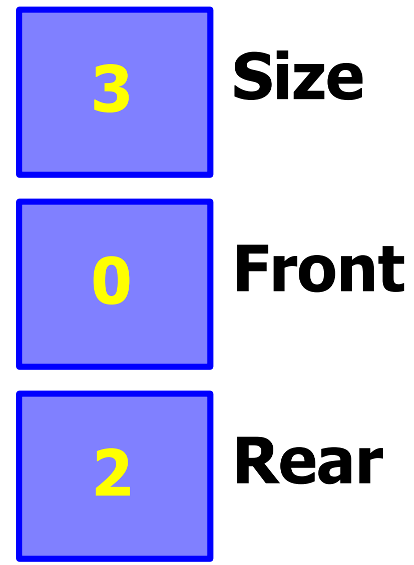
```
int QueueArray[7];
int front=-1; int rear=-1;
void Enqueue (x) {
      if (rear=size(QueueArray)-1)/IsFull()
      print "Queue is full";
      elseif IsEmpty(){
            front=0; rear=0;
            QueueArray[rear]=x;
                        }
      else {
            rear=rear+1;
            QueueArray[rear]=x;
            }
                        }
```

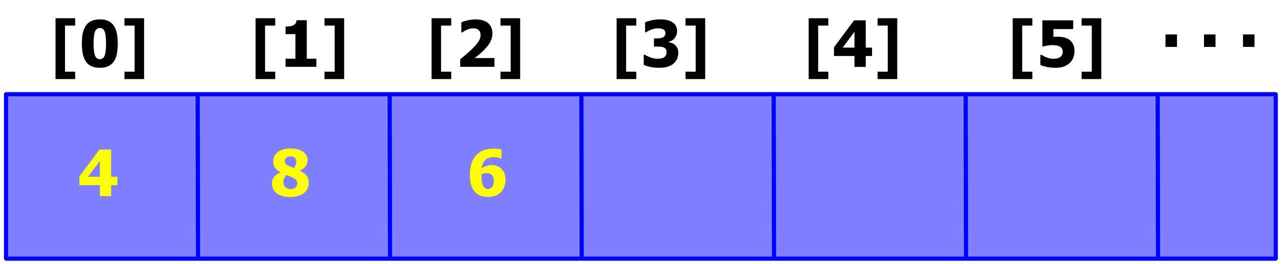
Queue (Array Implementation)

```
int QueueArray[7];
int front=-1; int rear=-1;
void Dequeue (x) {
      if IsEmpty()
            return;
      elseif (front==rear)
             front=-1; rear=-1;
      else
            front=front+1;
                        }
```

Queue (Array Implementation)

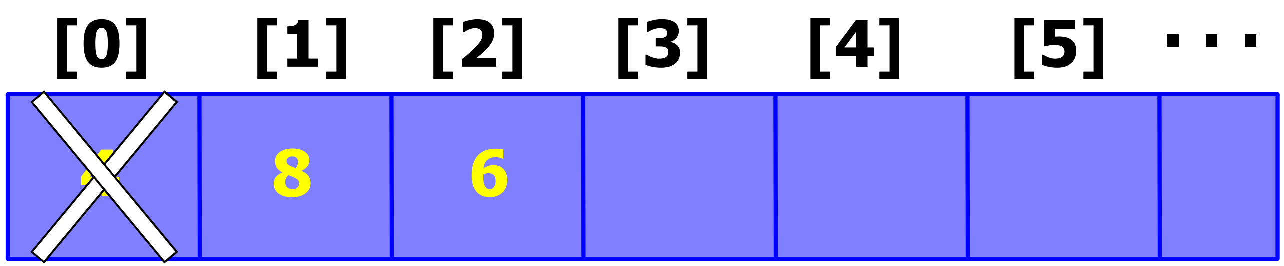
- The easiest implementation also keeps track of the number of items in the queue
- Index of the first element (at the front of the queue), the last element (at the rear).





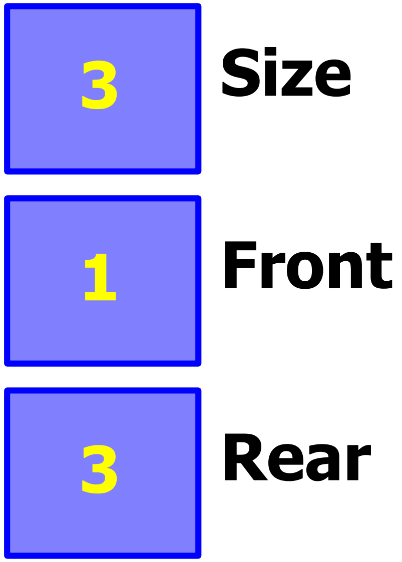
Queue (Array Implementation)

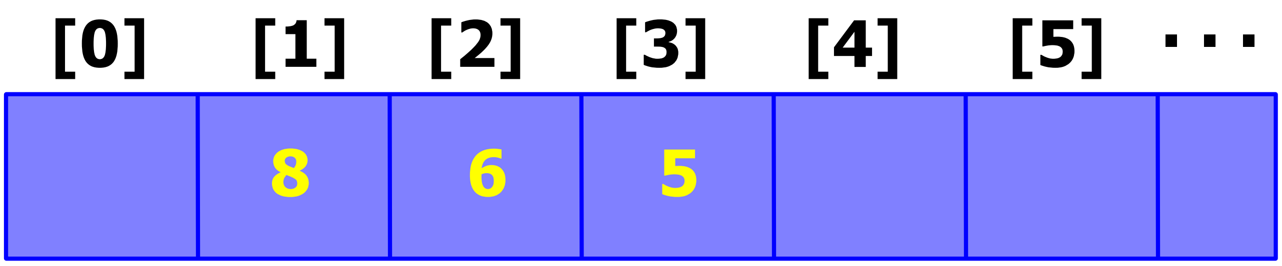
- Dequeue Operation: An element of the queue is removed through this operation
- Remove at Front of the queue
- The index of front element of a queue is updated by 1.



Queue (Array Implementation)

- Enqueue Operation: A new element is added to queue through this method
- Element is added at the rear of queue
- The index of rear is increased by 1





Queue (Array Implementation)

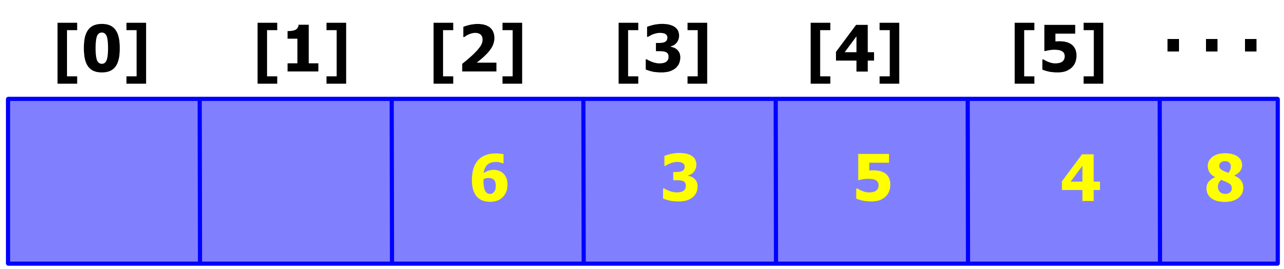
- IsEmpty Operation: Checks whether an element is present in the queue or to queue or not.
- If queue is empty, a dequeuer operation is not performed
  - Front_index= -1;
  - Rear_index= -1;

5	**Size**
2	**Front**
6	**Rear**

[0]	[1]	[2]	[3]	[4]	[5]	[6]
		6	3	5	4	8

Queue (Array Implementation)

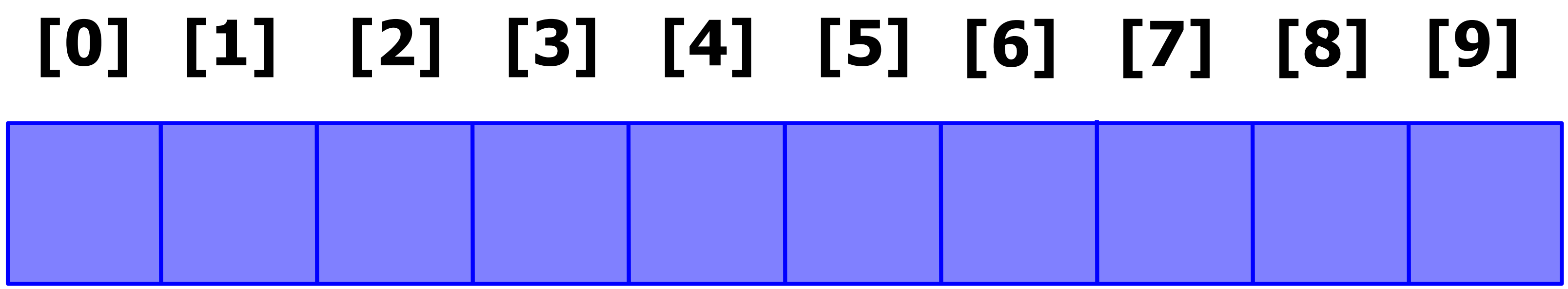
- IsFull Operation: Checks whether a new element can be added to queue or not.
- Usually, the last element of an array is added in the queue



Stack (Array Implementation)

- Using an Array `int StackArray [10];`

```
// Empty Stack
      top=-1;
Push (x)
      {
      top=top+1;
      StackArray(top)= x;
      }
```

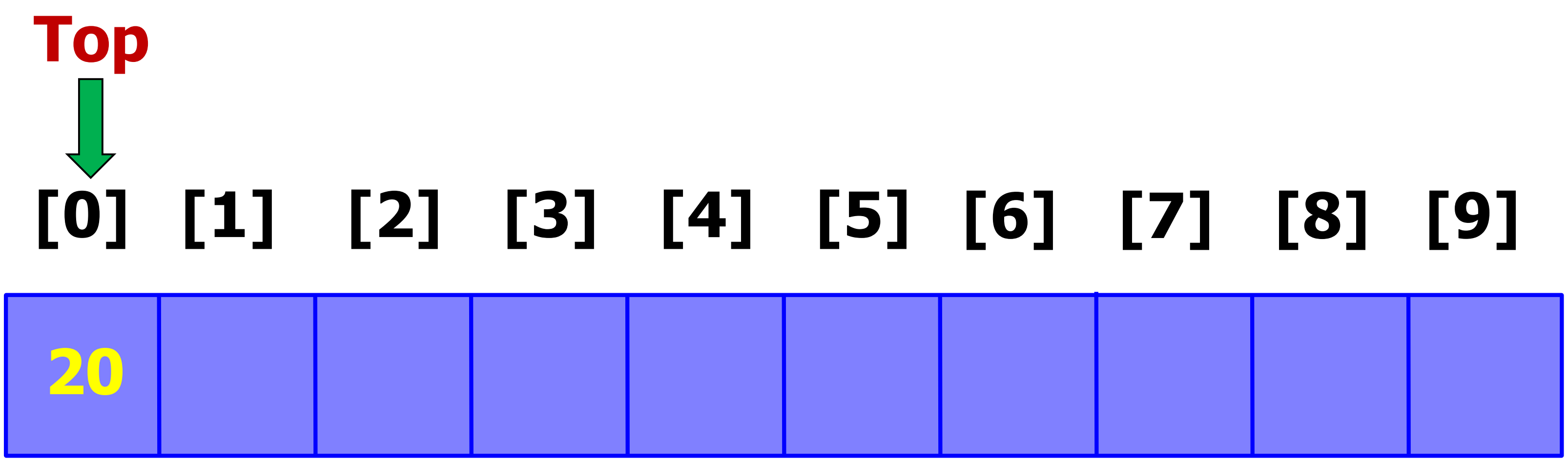


An integer array to implement a stack of integers

Stack (Array Implementation)

- Using an Array `int StackArray [10];`

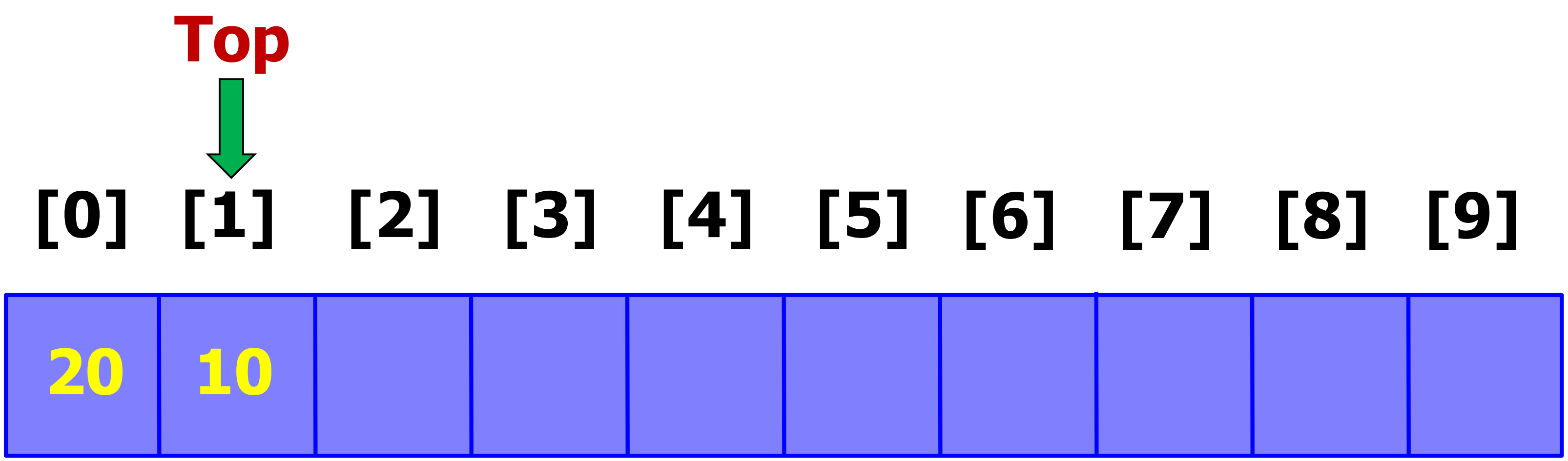
```
Push (20);        //1st time
```



Stack (Array Implementation)

- Using an Array `int StackArray [10];`

```
Push (20);        //1st time
Push (10);        //2nd time
```



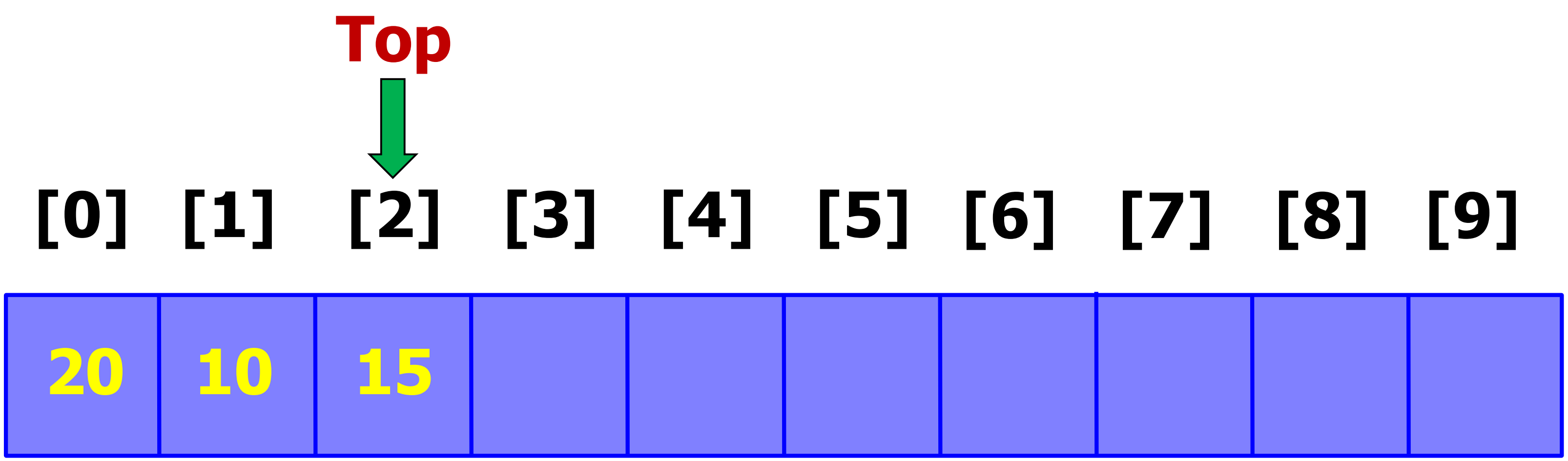
Stack (Array Implementation)

- Using an Array `int StackArray [10];`

`Push (20);` //1^{st} time

`Push (10);` //2^{nd} time

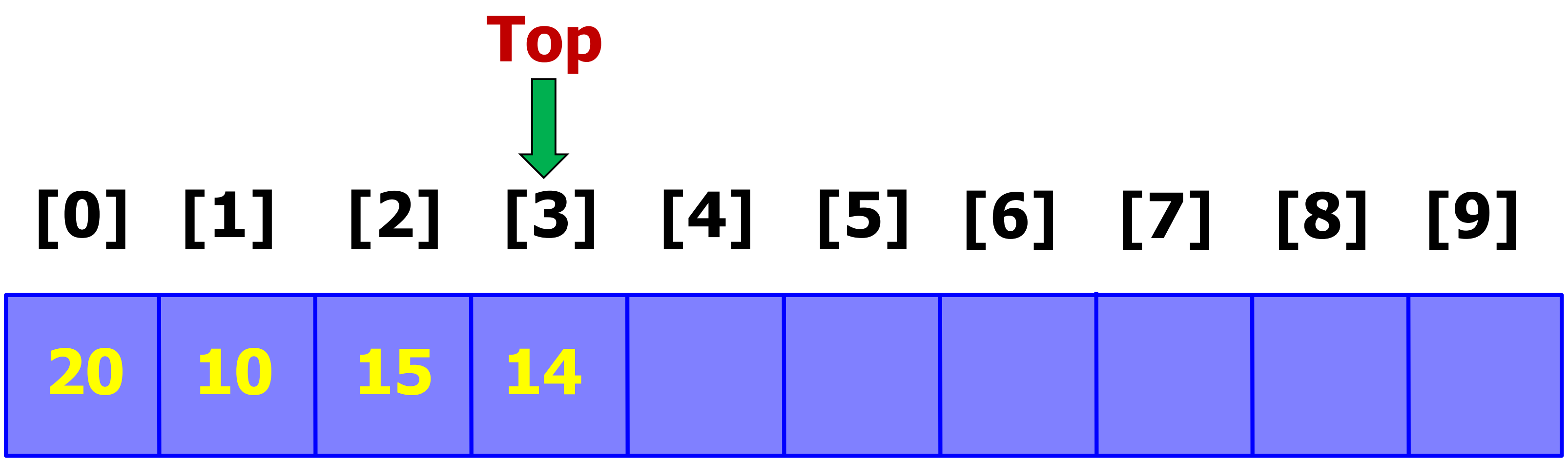
`Push (15);` //3^{rd} time



Stack (Array Implementation)

- Using an Array `int StackArray [10];`

```
Push (20);      //1st time
Push (10);      //2nd time
Push (15);      //3rd time
Push (14);      //4th time
```



Stack (Array Implementation)

- Using an Array `int StackArray [10];`

```
// Pop (x)
      {
      top=top-1;
      }
```

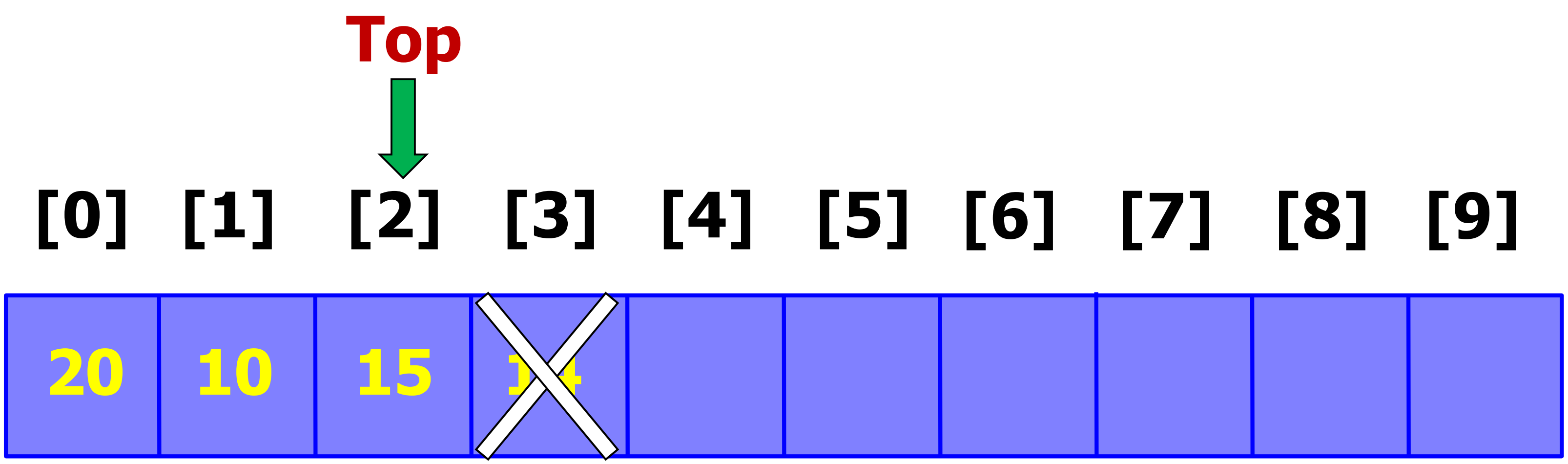
Top

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
20	10	15	14						

Stack (Array Implementation)

- Using an Array `int StackArray [10];`

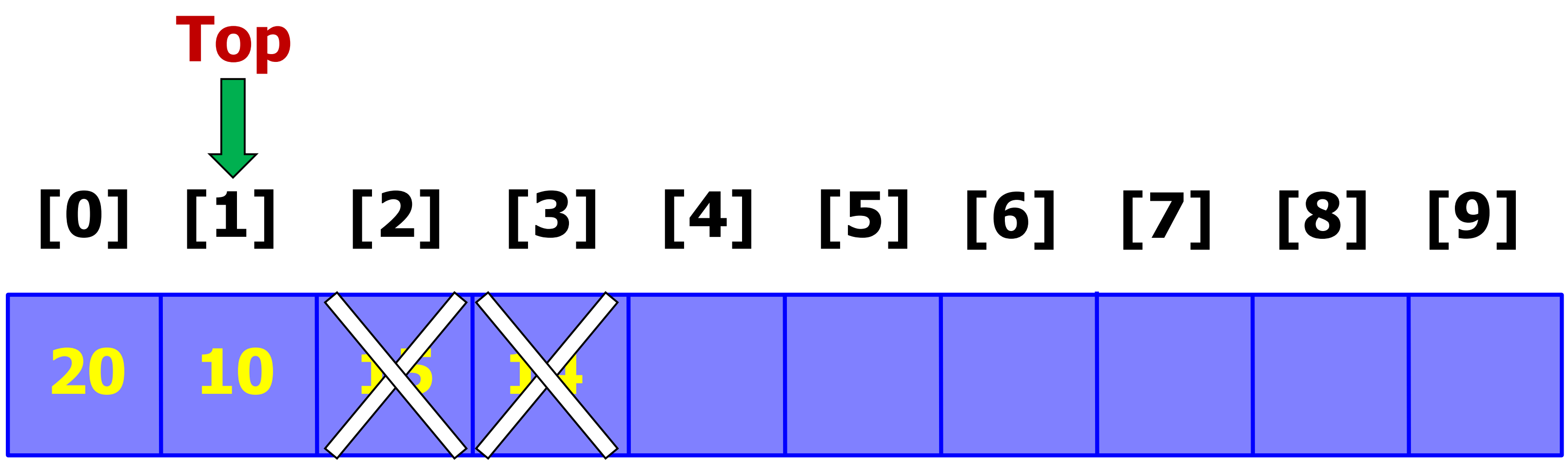
`Pop (); //1st time`



Stack (Array Implementation)

- Using an Array `int StackArray [10];`

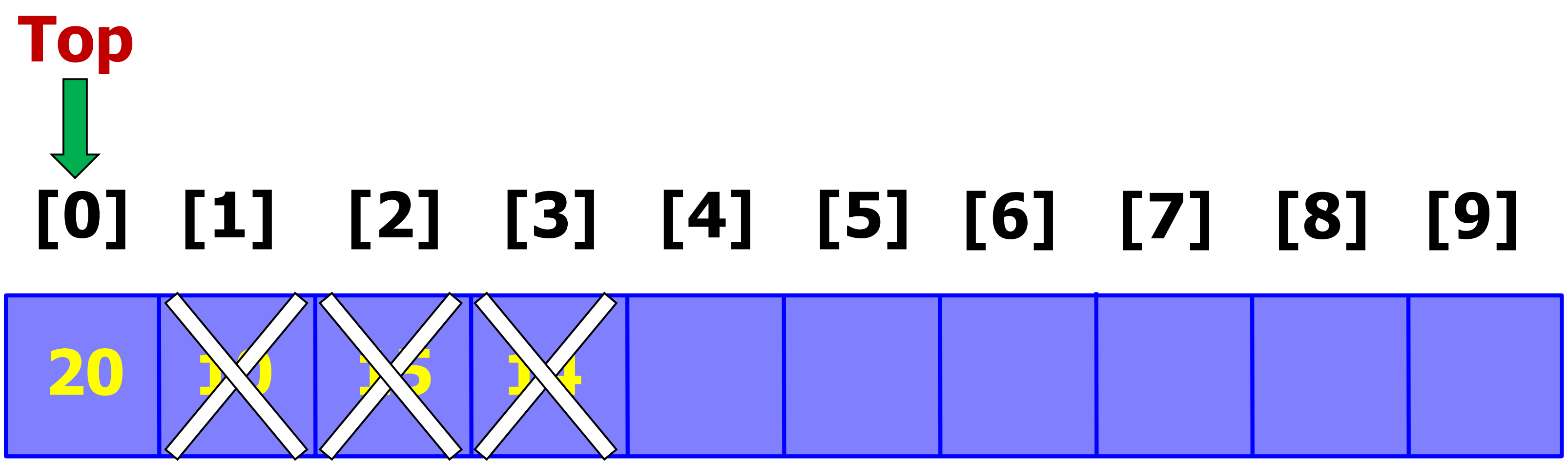
```
Pop (); //1st time
Pop (); //2nd time
```



Stack (Array Implementation)

- Using an Array `int StackArray [10];`

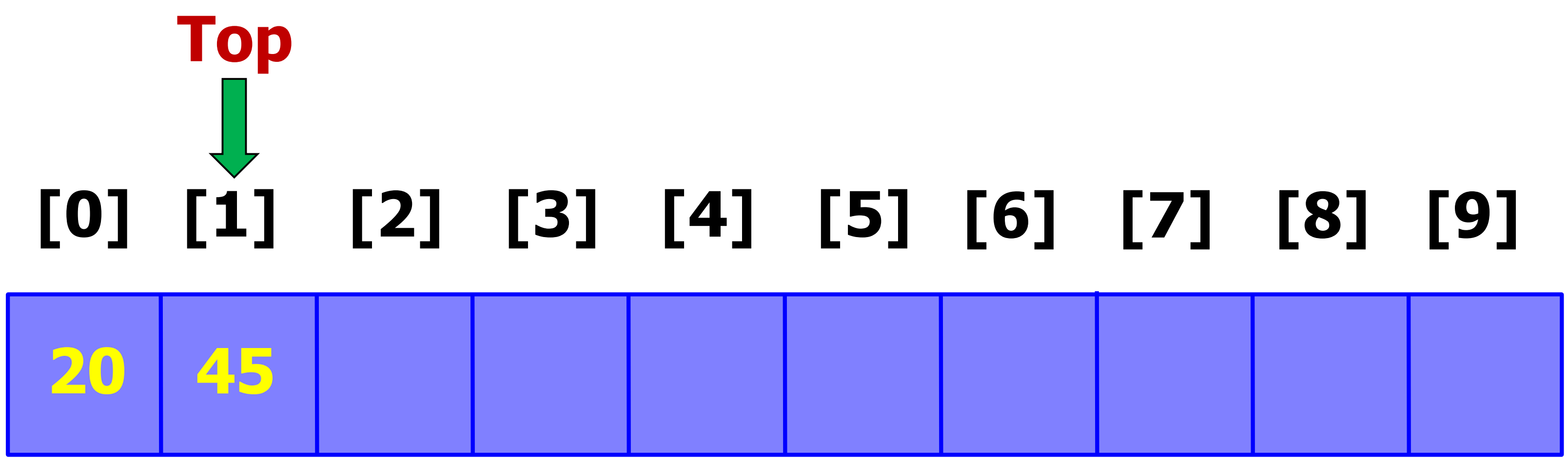
```
Pop (); //1st time
Pop (); //2nd time
Pop (); //3rd time
```



Stack (Array Implementation)

- Using an Array `int StackArray [10];`

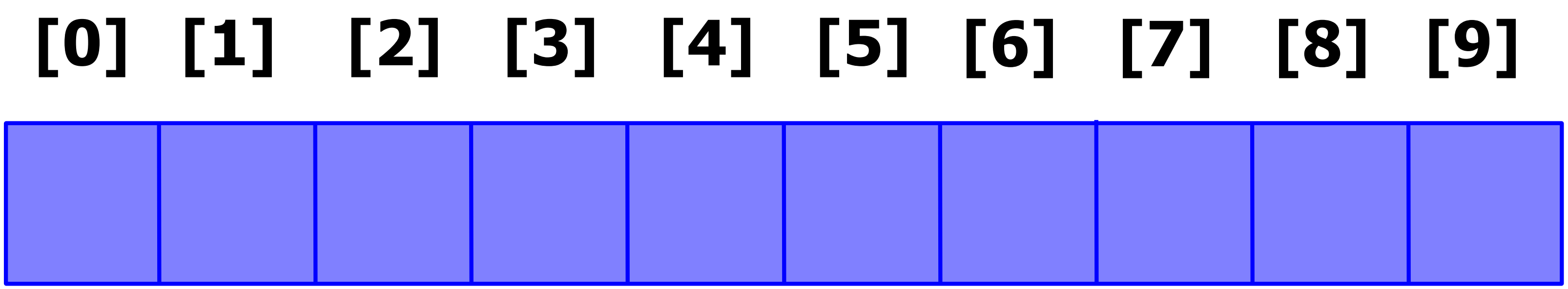
```
Pop ();   //1st time
Pop ();   //2nd time
Pop ();   //3rd time
Push (45);
```



Stack (Array Implementation)

- Using an Array `int StackArray [10];`

```
// Empty Stack
        top=-1;
Push (x)
        {
        top=top+1;
        StackArray(top)= x;
        }
```



An integer array to implement a stack of integers

Stack (Array Implementation)

- Using an Array `int StackArray [10];`

```
// IsEmpty Method
Boolean IsEmpty (){
        if (top==-1)
                return true;
        else
                return false;
                        }
// Top Method
int Top(){
        return StackArray[Top];
              }
```

Basic Implementation (Stack)

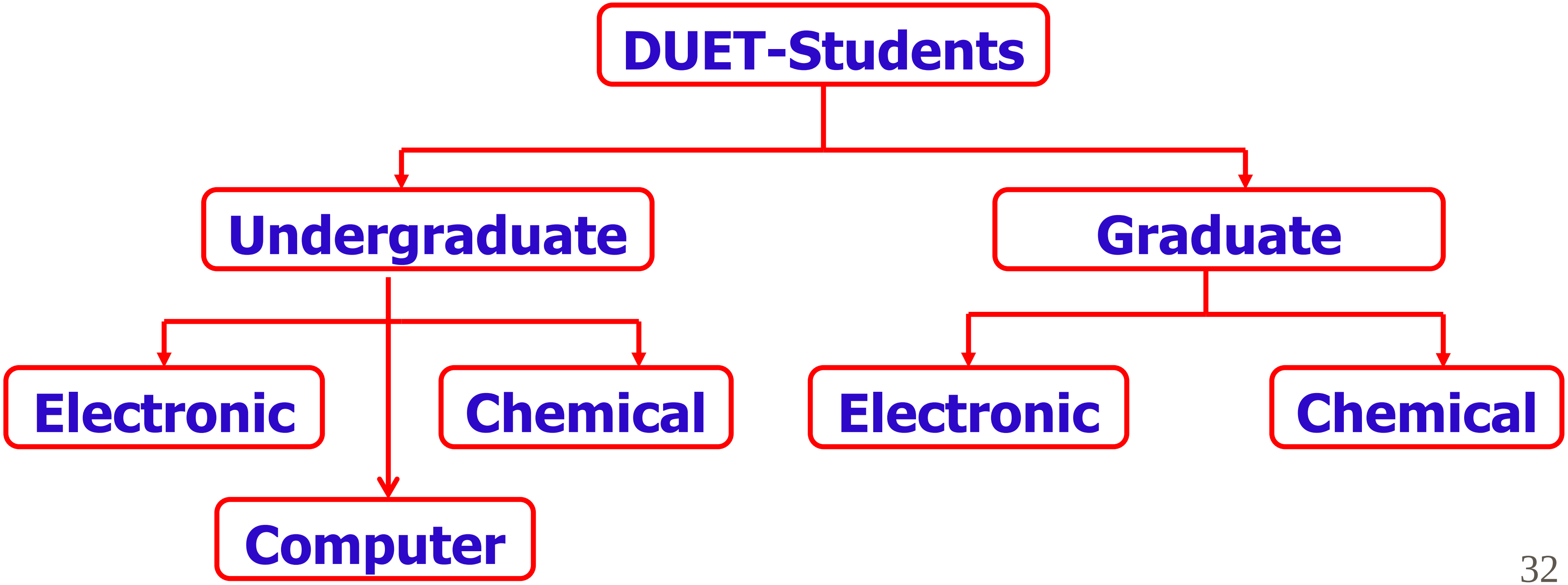
```
// Stack - Array based implementation.
#include<stdio.h>
#define MAX_SIZE 101
int A[MAX_SIZE];
int top = -1;
void Push(int x) {
    if(top == MAX_SIZE -1) {
        printf("Error: stack overflow\n");
        return;
    }
    A[++top] = x;
}
void Pop() {
    if(top == -1) {
        printf("Error: No element to pop\n");
        return;
    }
    top--;
}
```

```
int Top() {
    return A[top];
}

void Print() {
    int i;
    printf("Stack: ");
    for(i = 0;i<=top;i++)
        printf("%d ",A[i]);
    printf("\n");
}
int main() {
    Push(2);Print();
    Push(5);Print();
    Push(10);Print();
    Pop();Print();
    Push(12);Print();
}
```

Tree Data Structure

- Non-linear data structure
- Store hierarchical data
  - E.g. hierarchy of DUET Students (tree)



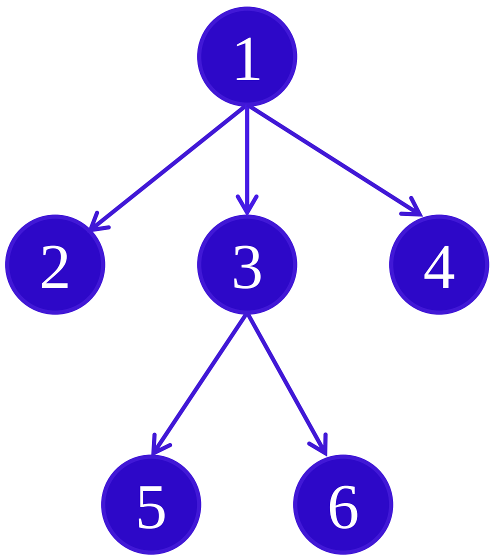
Tree (Applications)

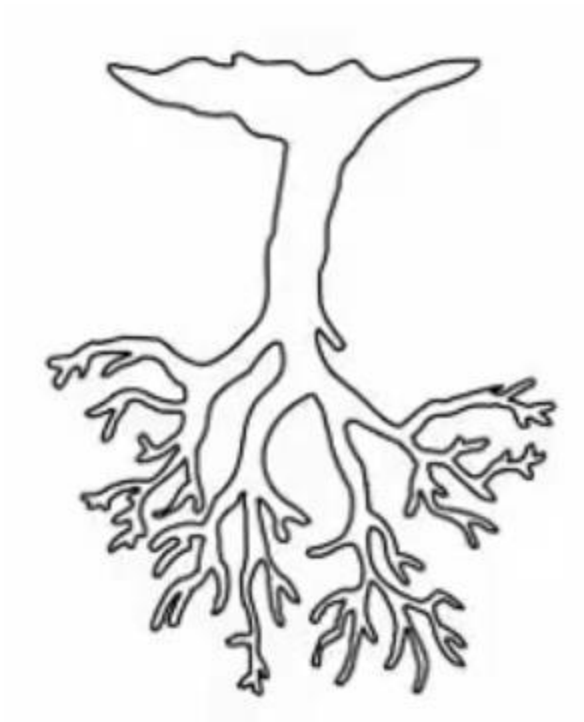
- Storing naturally hierarchical data
  - File System in Computer
- Organize data
  - For quick search, insertion, and deletion
    - Binary search tree
- Network routing algorithm
- Trie
  - Dictionary

Tree Data Structure

- Root Node connected via links with branching nodes
- Efficient for storing data that is hierarchical nature
- Each node may contain data and link to other nodes

Root
Parent
Child
Sibling
Leaf

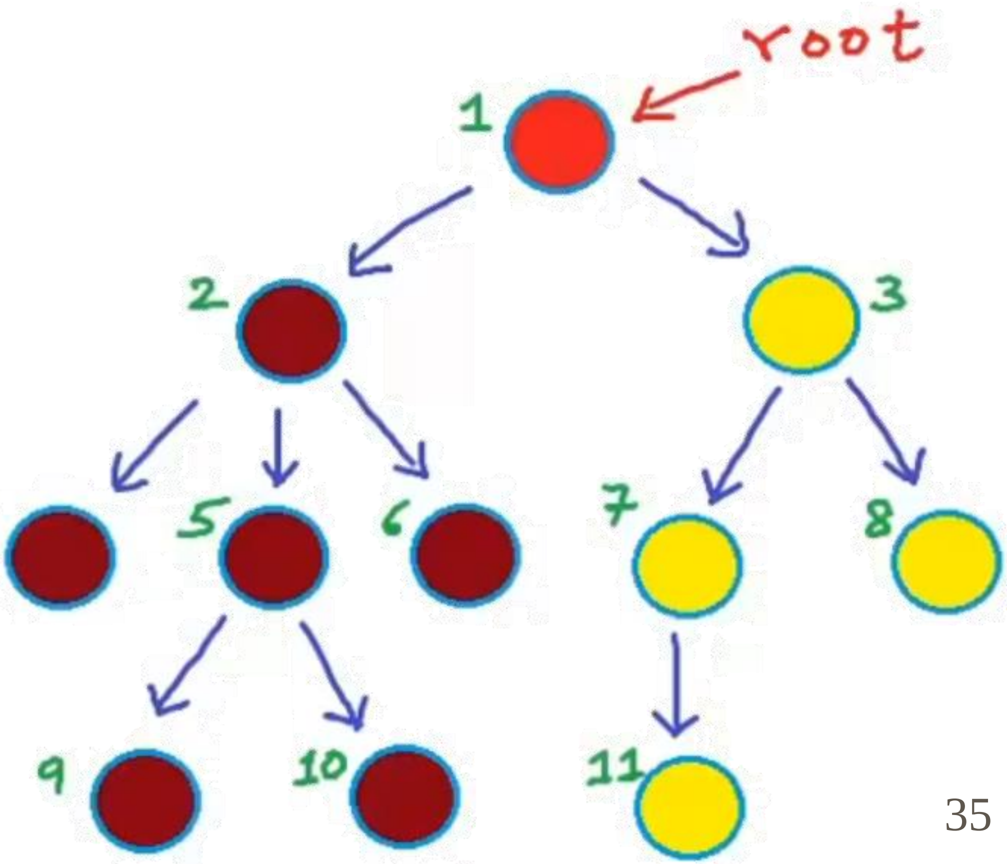




DAWOOD UNIVERSITY OF ENGINEERING & TECHNOLOGY

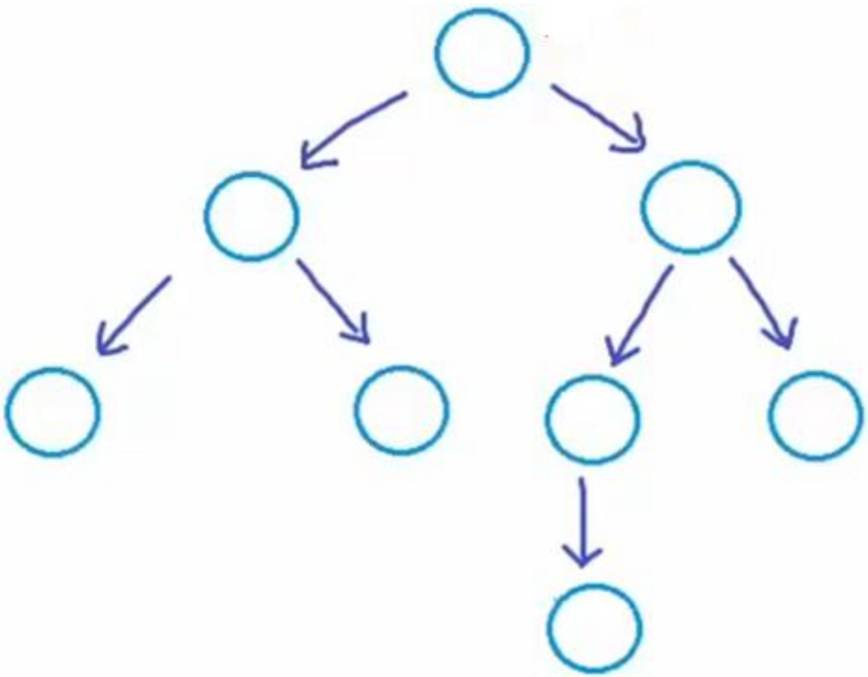
Tree (Properties)

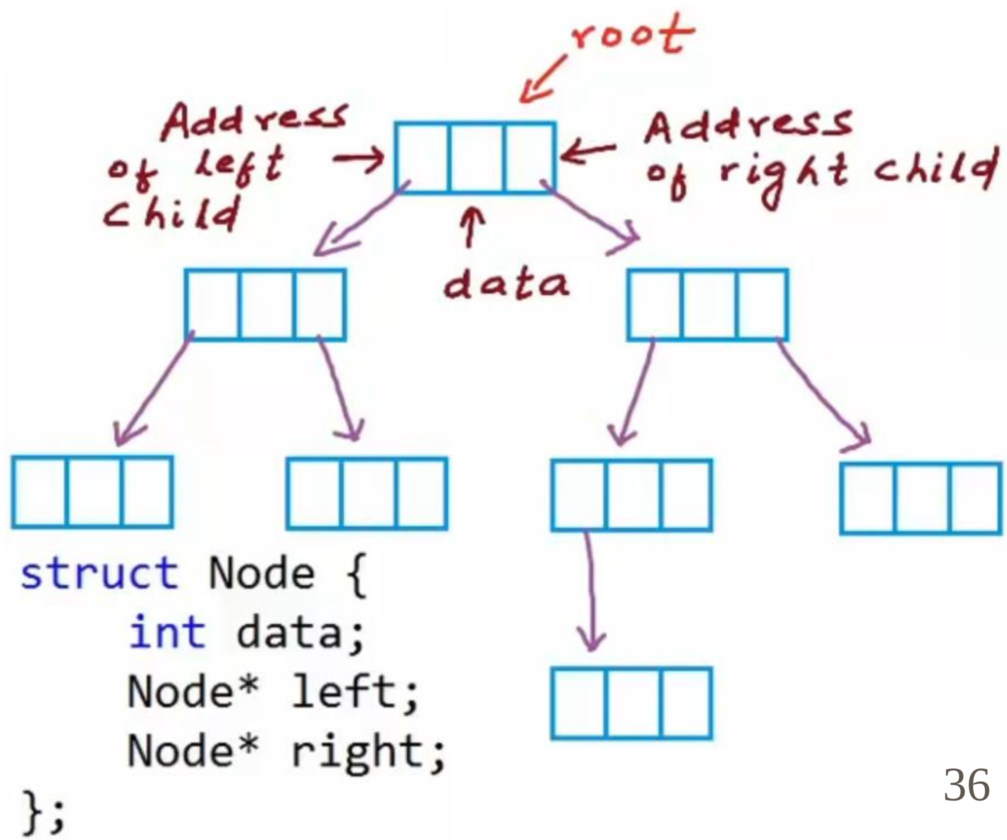
- Recursive data structure
  - Root (1)
    - Subtree (2) and (3)
- N nodes, $(N\text{-}1)$ Links
- Depth of node
  - Length of path from root to x node
- Height of node
  - No of edges in longest path from x node to a leaf
- Height of tree
  - Height of root node



Binary Tree

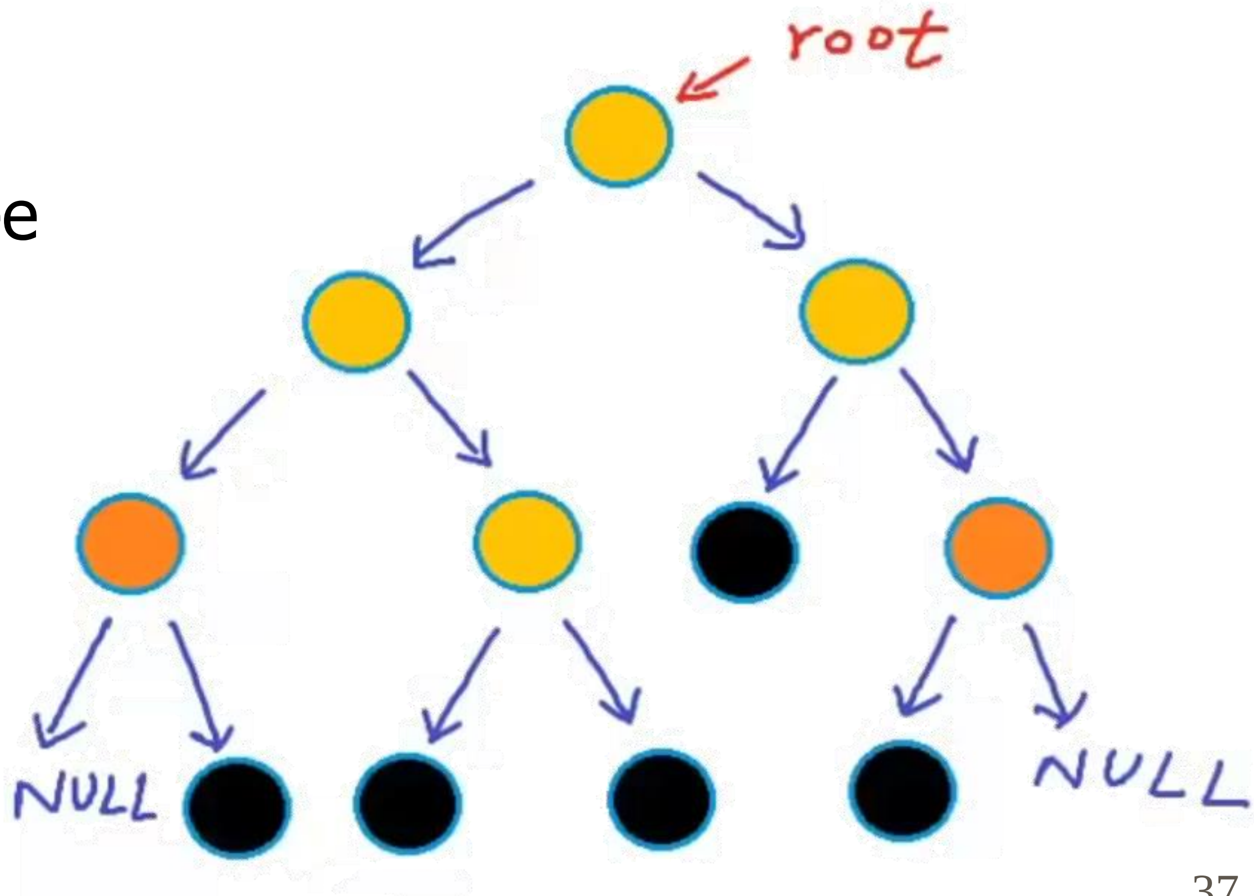
- A tree which has at most two child nodes is called binary tree (Most Famous type of tree)





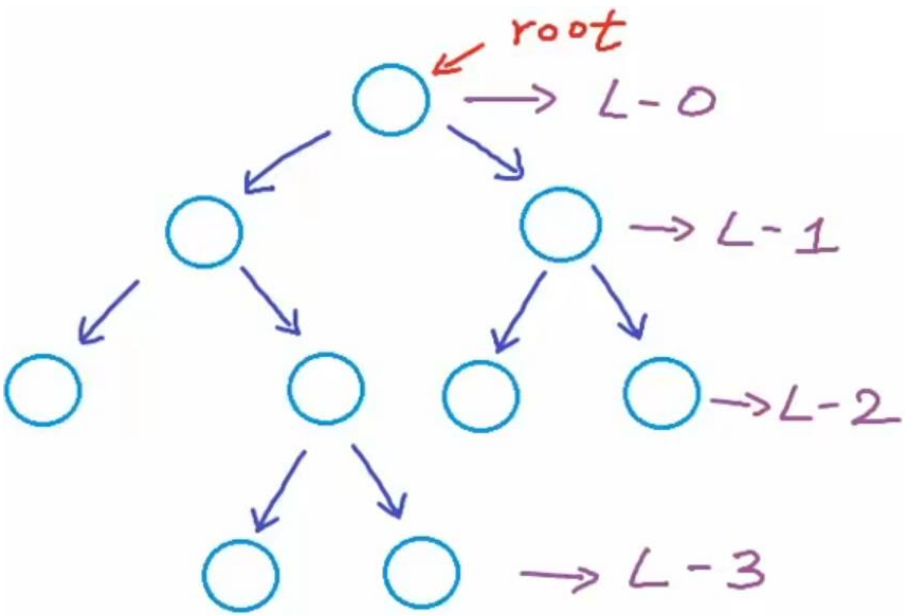
Binary Tree

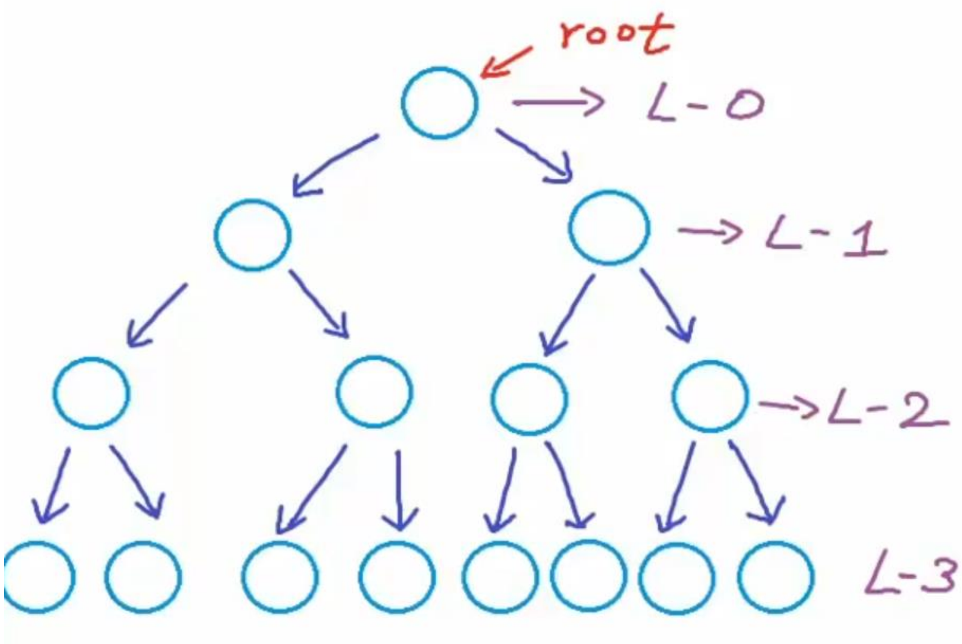
- Two children
  - Left child
  - Right child
  - No child (leaf)
- Strict binary tree
  - 0 or 2 children



Binary Tree

- Maximum no. of nodes at i^{th} level = 2^i

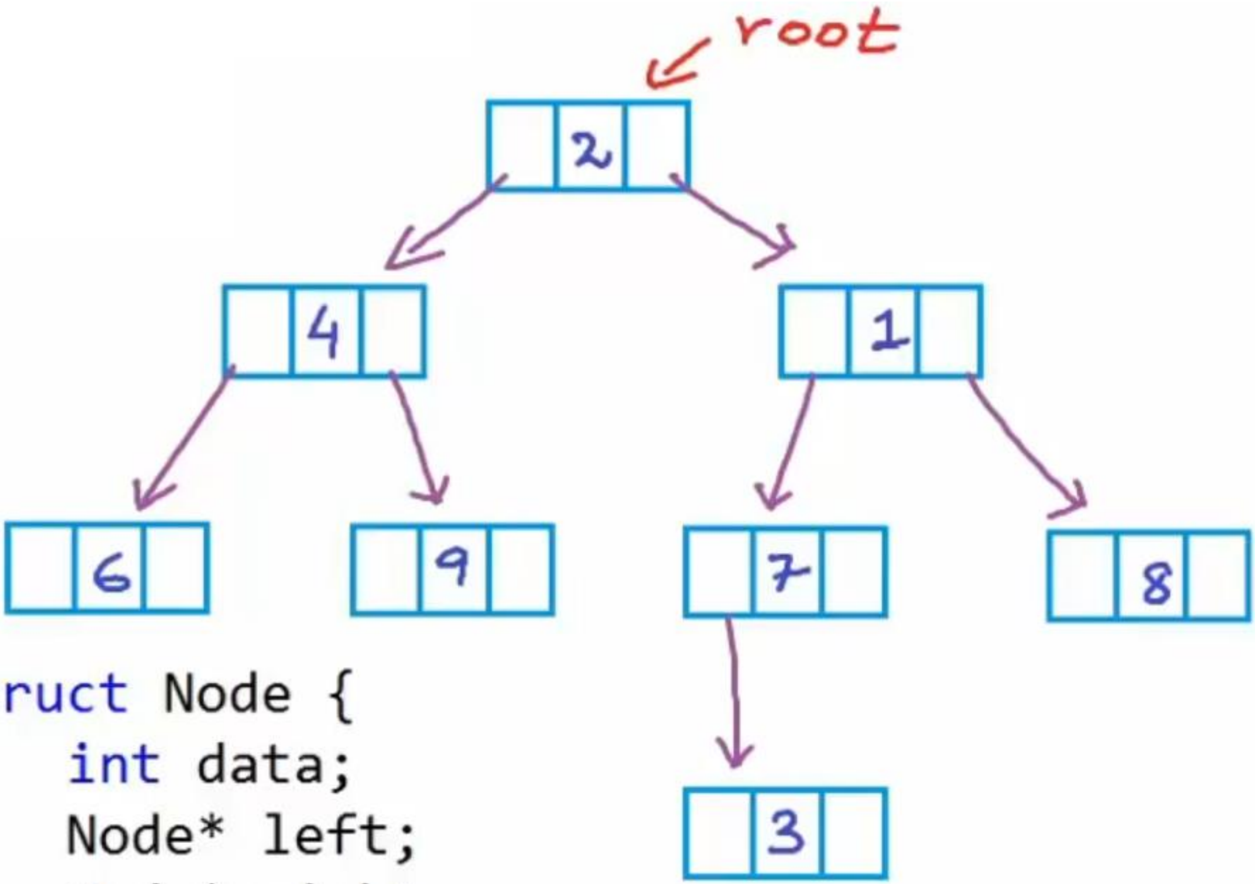




Perfect binary tree

Binary Tree (Implementation)

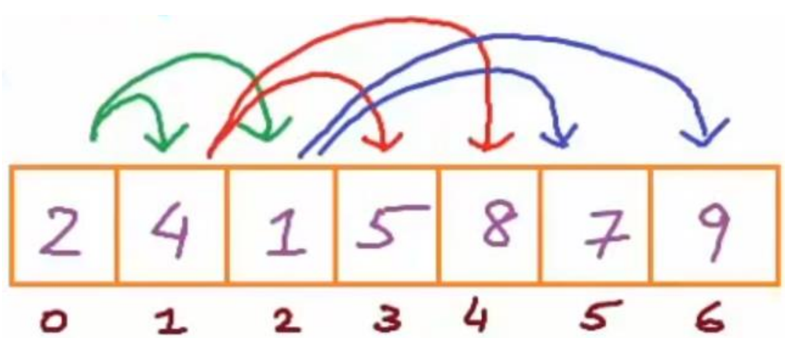
- Dynamically created nodes using pointers

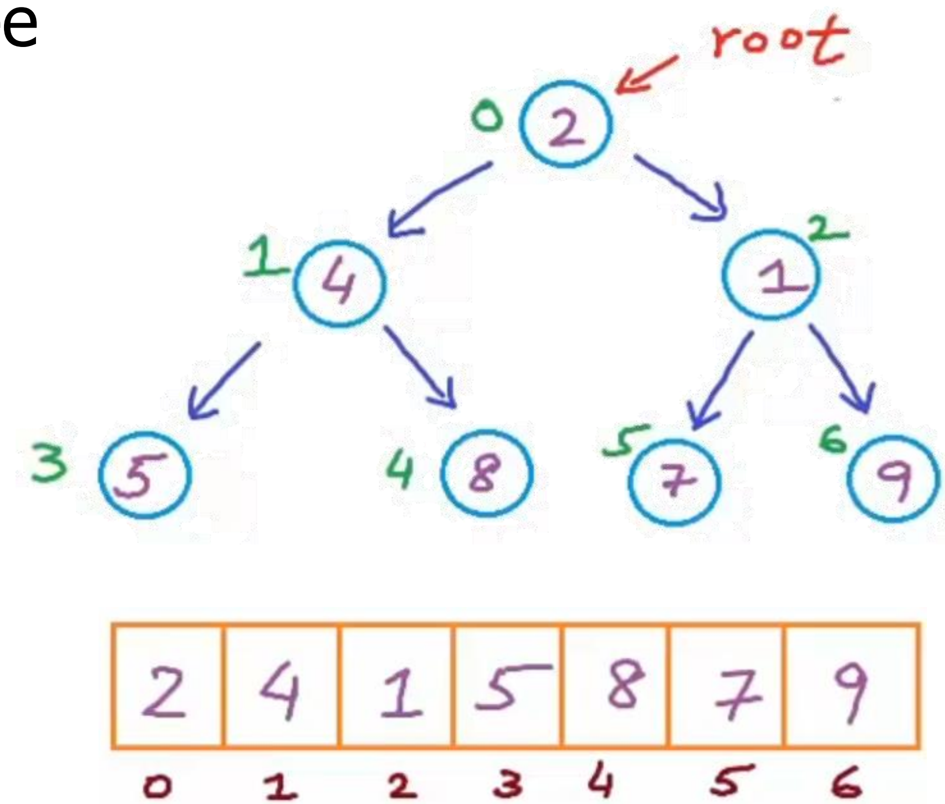



```
struct Node {
    int data;
    Node* left;
    Node* right;
};
```

Binary Tree (Implementation)

- Using Arrays
- Case: complete binary tree
  - At node x at i^{th} index:
    - Left_Child= $2i+1$
    - Right_Child= $2i+2$





Binary Search Tree

- Efficient data structure for a quick-search & quick-update