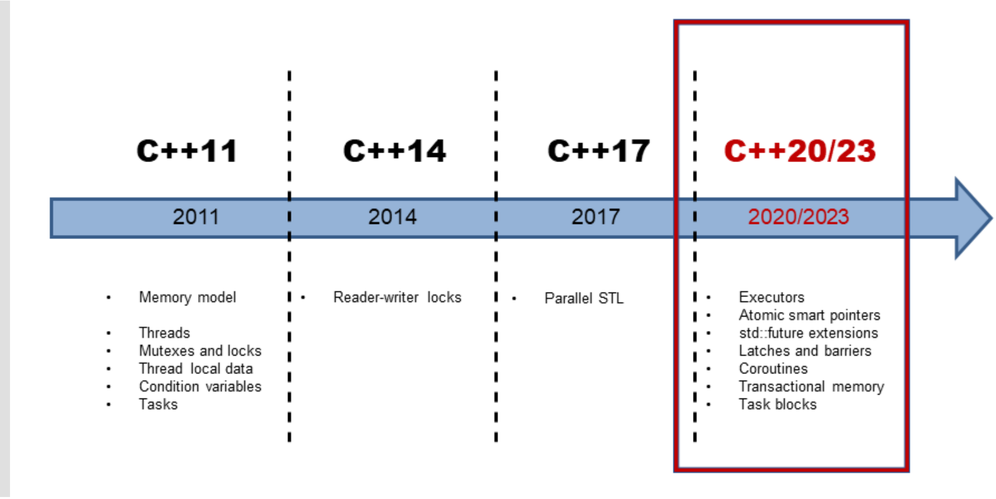
Data Structure and Algorithm in C^{++}



C++11

2011

- Memory model
- Threads
- Mutexes and locks
- Thread local data
- Condition variables
- Tasks

C++14

2014

- Reader-writer locks

C++17

2017

- Parallel STL

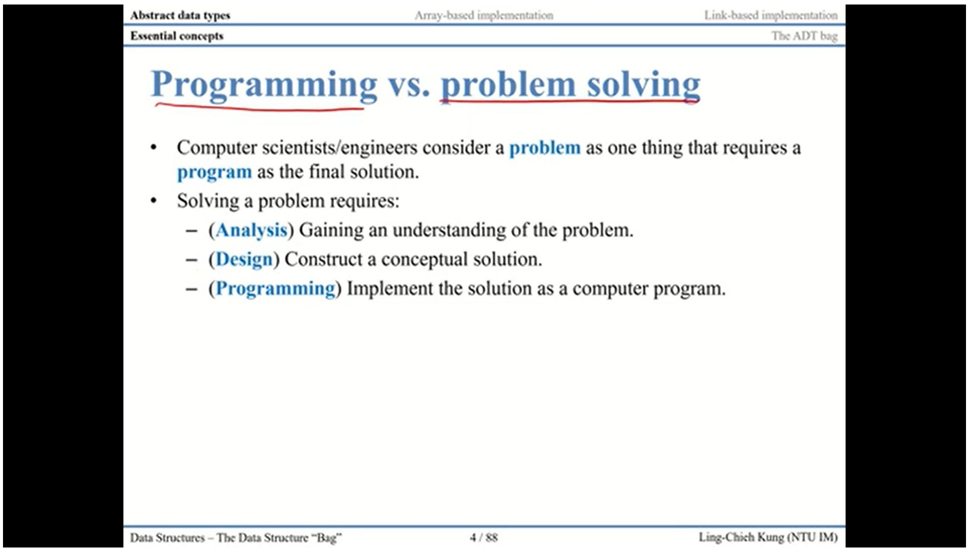
C++20/23

2020/2023

- Executors
- Atomic smart pointers
- std::future extensions
- Latches and barriers
- Coroutines
- Transactional memory
- Task blocks

C^{++} Algorithm

Problem Solving



Abstract data types Array-based implementation Link-based implementation

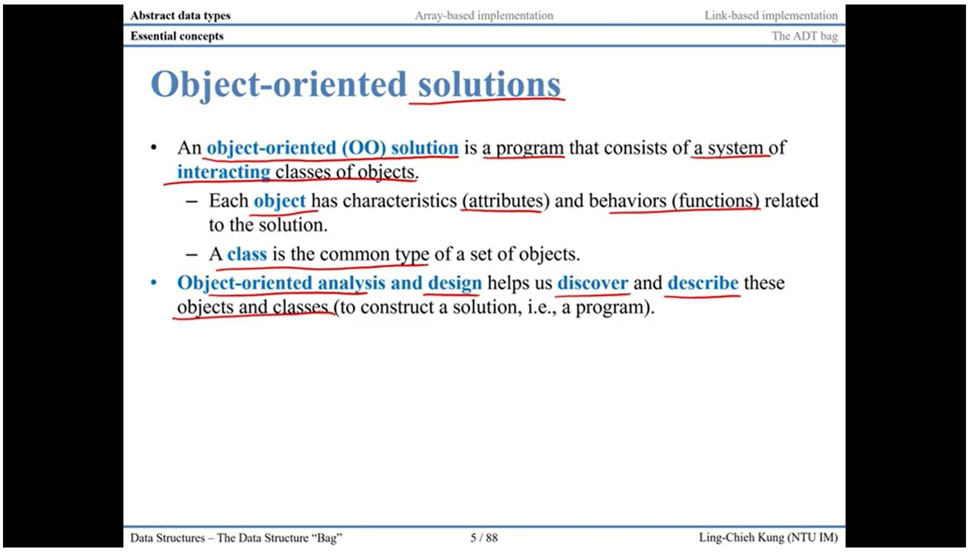
Essential concepts The ADT bag

Programming vs. problem solving

- Computer scientists/engineers consider a **problem** as one thing that requires a **program** as the final solution.
- Solving a problem requires:
  - (**Analysis**) Gaining an understanding of the problem.
  - (**Design**) Construct a conceptual solution.
  - (**Programming**) Implement the solution as a computer program.

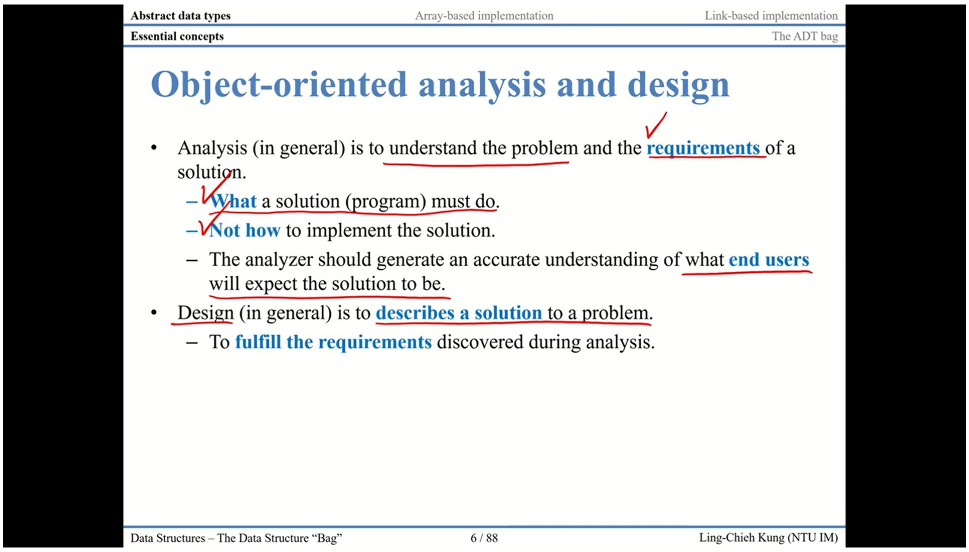
Data Structures – The Data Structure "Bag" 4 / 88 Ling-Chieh Kung (NTU IM)

Object-oriented Solution



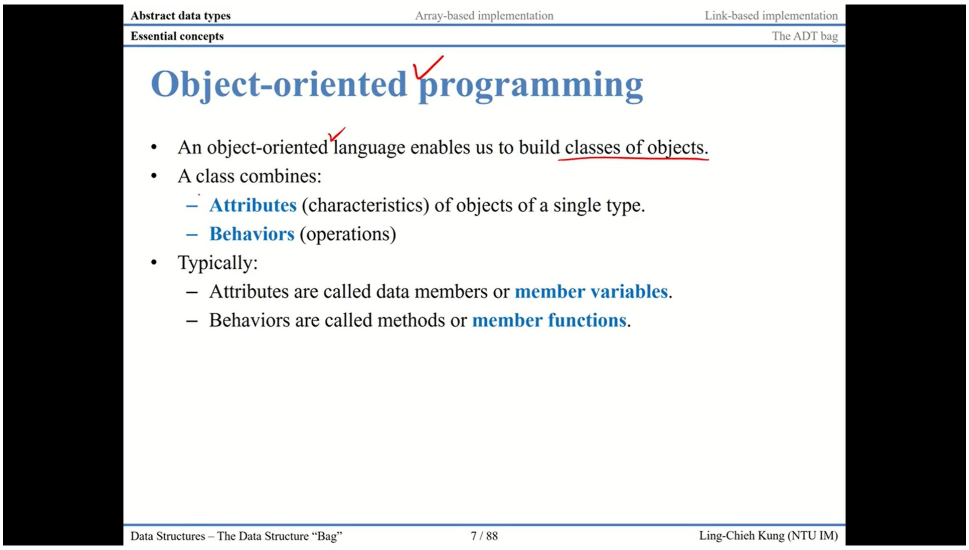
Object-oriented solutions

- An **object-oriented (OO) solution** is a program that consists of a system of **interacting** classes of objects.
  - Each **object** has characteristics (attributes) and behaviors (functions) related to the solution.
  - A **class** is the common type of a set of objects.
- **Object-oriented analysis and design** helps us **discover** and **describe** these objects and classes (to construct a solution, i.e., a program).



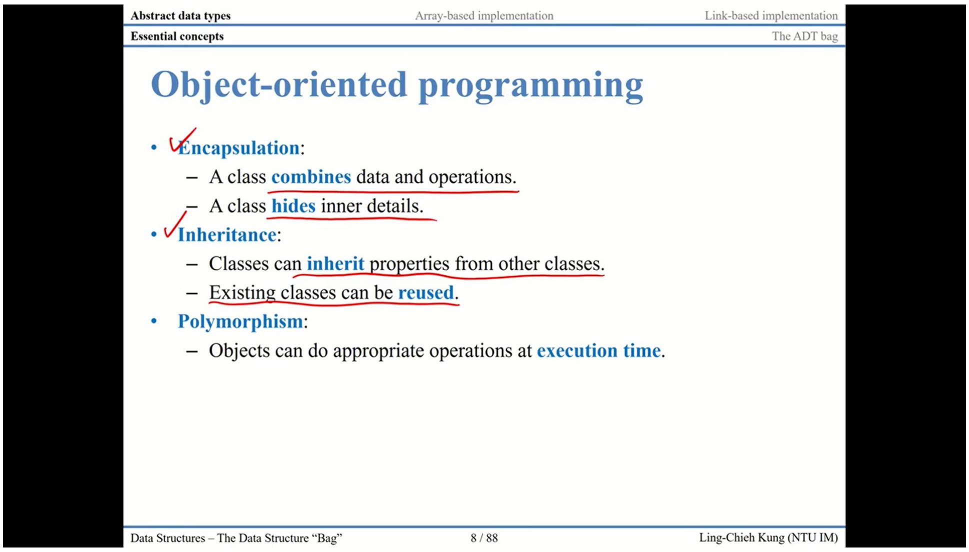
Object-oriented analysis and design

- Analysis (in general) is to understand the problem and the **requirements** of a solution.
  - **What** a solution (program) must do.
  - **Not how** to implement the solution.
  - The analyzer should generate an accurate understanding of what **end users** will expect the solution to be.
- Design (in general) is to **describes a solution** to a problem.
  - To **fulfill the requirements** discovered during analysis.



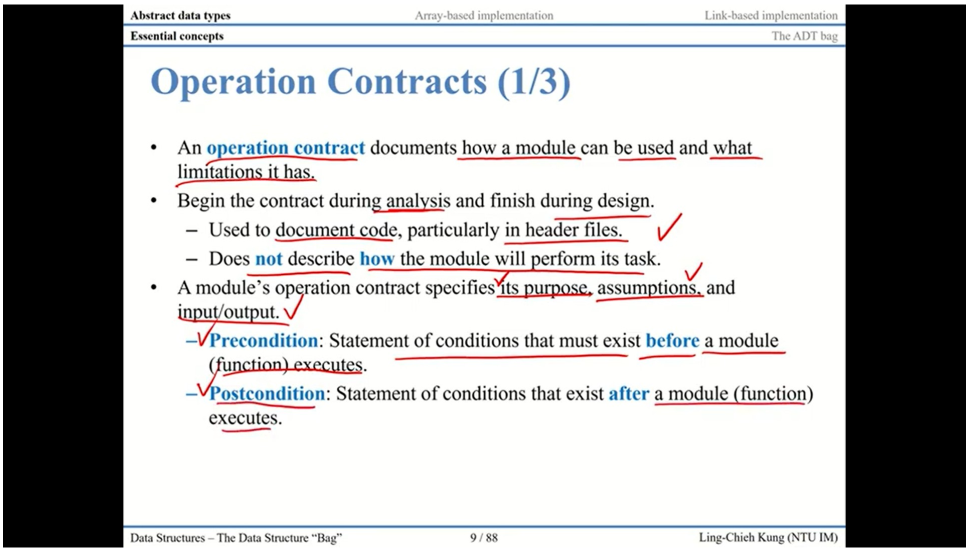
Object-oriented programming

- An object-oriented language enables us to build classes of objects.
- A class combines:
  - **Attributes** (characteristics) of objects of a single type.
  - **Behaviors** (operations)
- Typically:
  - Attributes are called data members or **member variables**.
  - Behaviors are called methods or **member functions**.



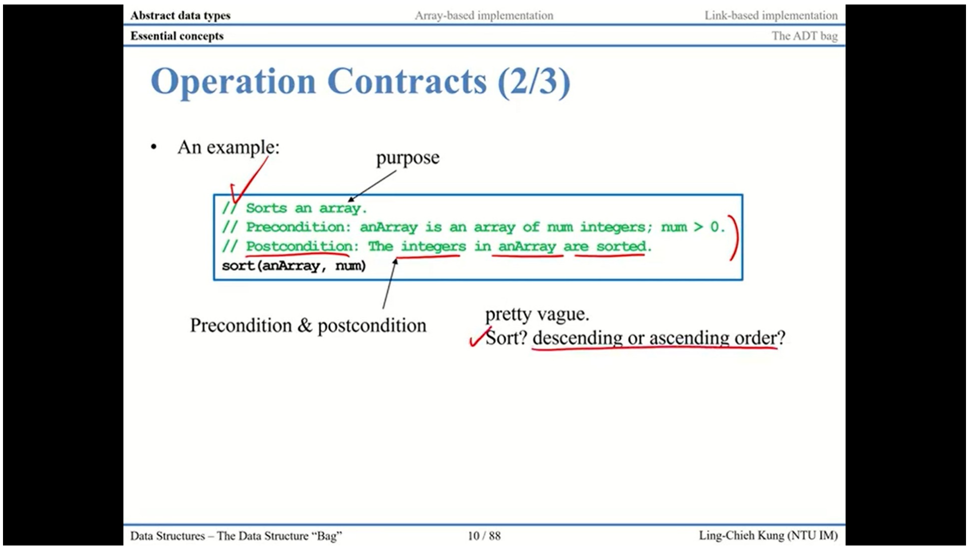
Object-oriented programming

- **Encapsulation**:
  - A class **combines** data and operations.
  - A class **hides** inner details.
- **Inheritance**:
  - Classes can **inherit** properties from other classes.
  - Existing classes can be **reused**.
- **Polymorphism**:
  - Objects can do appropriate operations at **execution time**.



Operation Contracts (1/3)

- An **operation contract** documents how a module can be used and what limitations it has.
- Begin the contract during analysis and finish during design.
  - Used to document code, particularly in header files.
  - Does **not** describe **how** the module will perform its task.
- A module's operation contract specifies its purpose, assumptions, and input/output.
  - **Precondition**: Statement of conditions that must exist **before** a module (function) executes.
  - **Postcondition**: Statement of conditions that exist **after** a module (function) executes.



Operation Contracts (2/3)

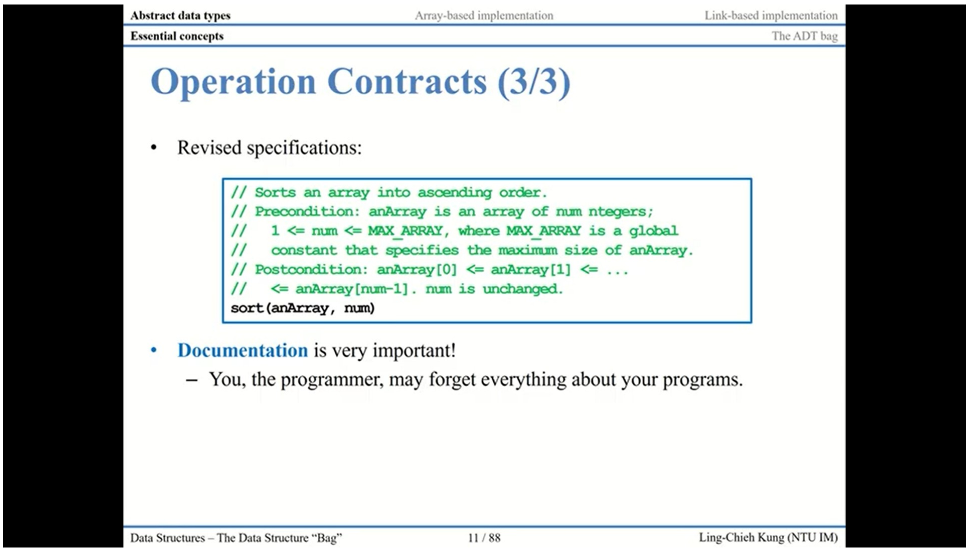
- An example:

purpose

```
// Sorts an array.
// Precondition: anArray is an array of num integers; num > 0.
// Postcondition: The integers in anArray are sorted.
sort(anArray, num)
```

Precondition & postcondition

pretty vague.
Sort? descending or ascending order?



Abstract data types | Array-based implementation | Link-based implementation

Essential concepts | The ADT bag

Operation Contracts (3/3)

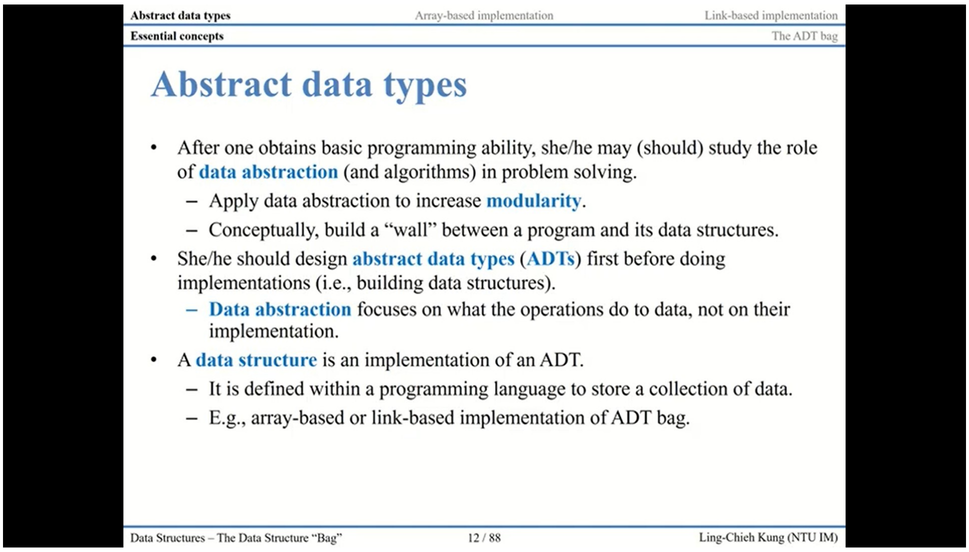
- Revised specifications:

```
// Sorts an array into ascending order.
// Precondition: anArray is an array of num ntegers;
//   1 <= num <= MAX_ARRAY, where MAX_ARRAY is a global
//   constant that specifies the maximum size of anArray.
// Postcondition: anArray[0] <= anArray[1] <= ...
//   <= anArray[num-1]. num is unchanged.
sort(anArray, num)
```

- **Documentation** is very important!
  - You, the programmer, may forget everything about your programs.

Data Structures – The Data Structure "Bag" 11 / 88 Ling-Chieh Kung (NTU IM)

Abstract Data Type



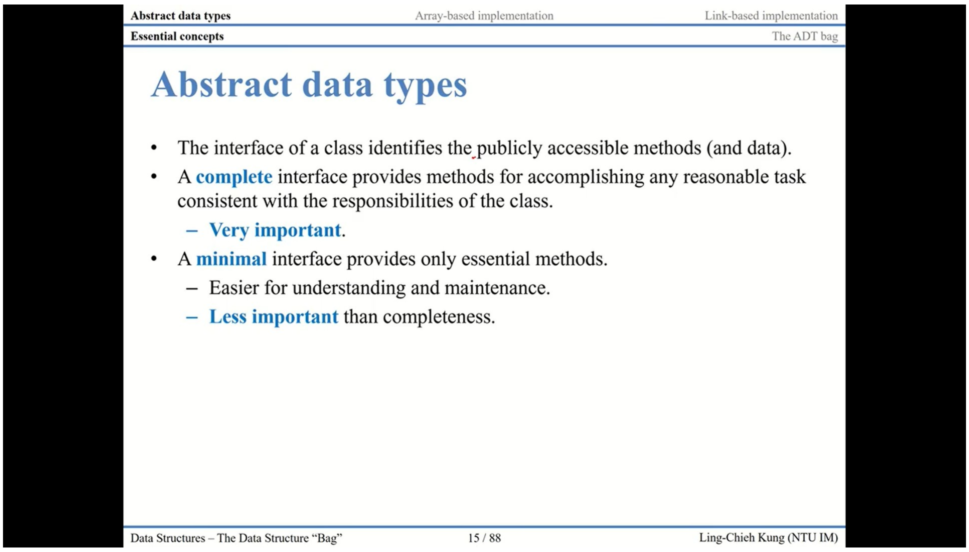
Abstract data types | Array-based implementation | Link-based implementation

Essential concepts | The ADT bag

Abstract data types

- After one obtains basic programming ability, she/he may (should) study the role of **data abstraction** (and algorithms) in problem solving.
  - Apply data abstraction to increase **modularity**.
  - Conceptually, build a "wall" between a program and its data structures.
- She/he should design **abstract data types** (**ADTs**) first before doing implementations (i.e., building data structures).
  - **Data abstraction** focuses on what the operations do to data, not on their implementation.
- A **data structure** is an implementation of an ADT.
  - It is defined within a programming language to store a collection of data.
  - E.g., array-based or link-based implementation of ADT bag.

Data Structures – The Data Structure "Bag" 12 / 88 Ling-Chieh Kung (NTU IM)



Abstract data types | Array-based implementation | Link-based implementation

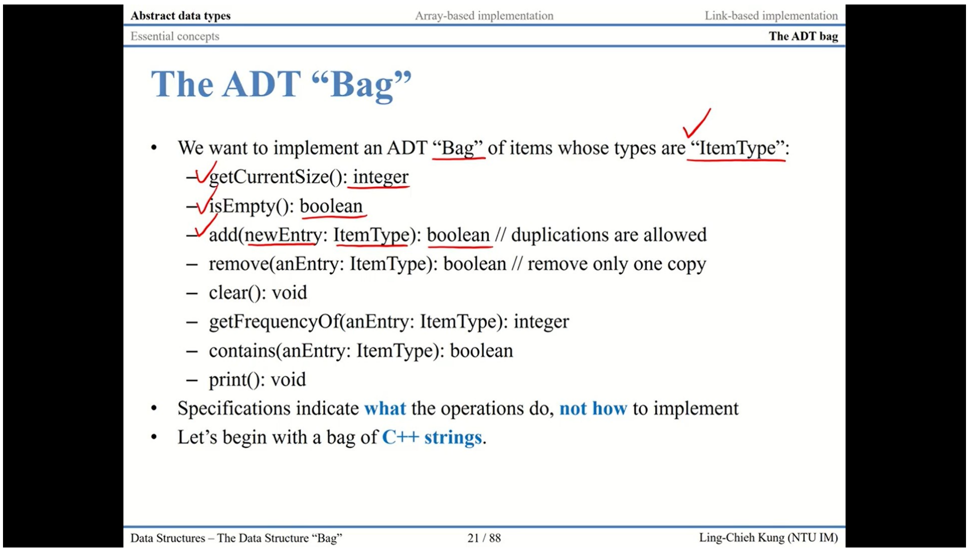
Essential concepts | The ADT bag

Abstract data types

- The interface of a class identifies the publicly accessible methods (and data).
- A **complete** interface provides methods for accomplishing any reasonable task consistent with the responsibilities of the class.
  - **Very important**.
- A **minimal** interface provides only essential methods.
  - Easier for understanding and maintenance.
  - **Less important** than completeness.

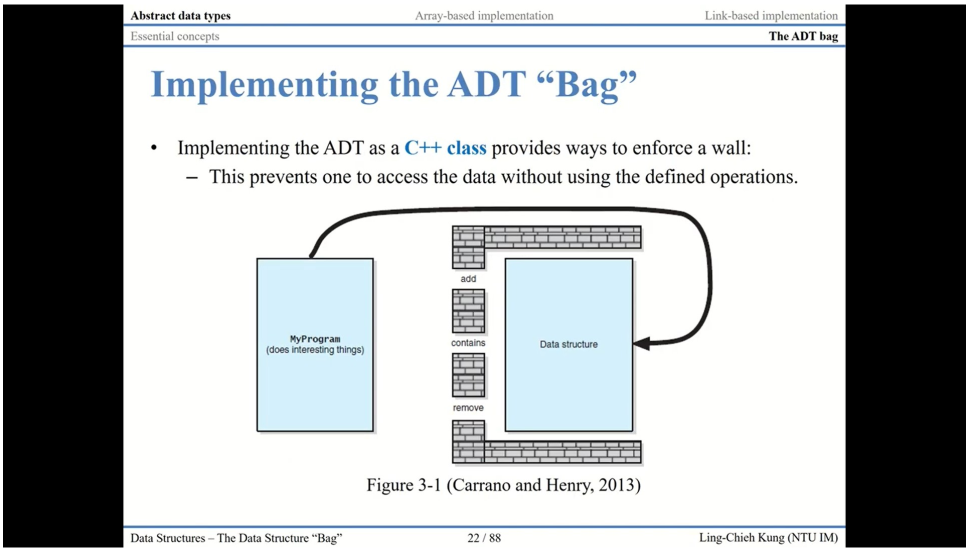
Data Structures – The Data Structure "Bag" 15 / 88 Ling-Chieh Kung (NTU IM)

ADT Bag



The ADT “Bag”

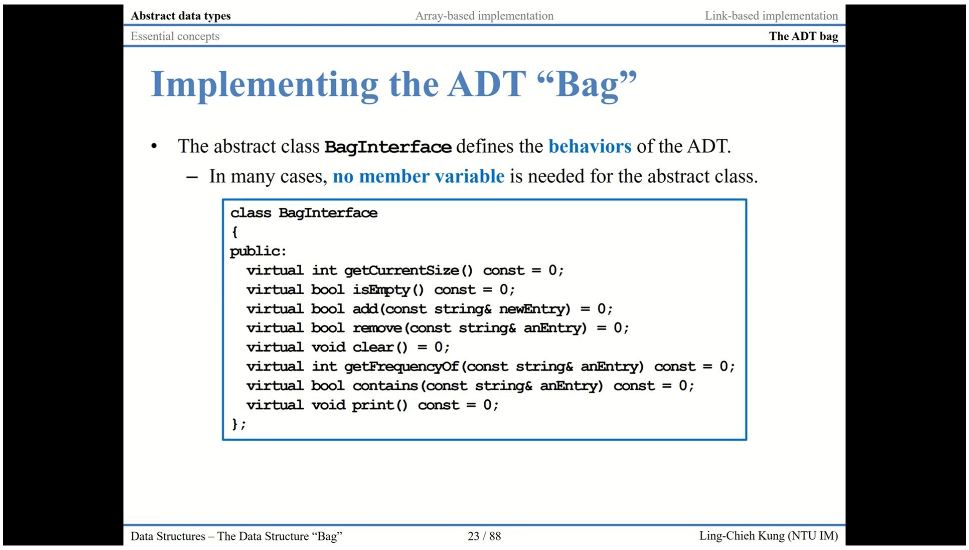
- We want to implement an ADT “Bag” of items whose types are “ItemType”:
  - getCurrentSize(): integer
  - isEmpty(): boolean
  - add(newEntry: ItemType): boolean // duplications are allowed
  - remove(anEntry: ItemType): boolean // remove only one copy
  - clear(): void
  - getFrequencyOf(anEntry: ItemType): integer
  - contains(anEntry: ItemType): boolean
  - print(): void
- Specifications indicate **what** the operations do, **not how** to implement
- Let’s begin with a bag of **C++ strings**.



Implementing the ADT “Bag”

- Implementing the ADT as a **C++ class** provides ways to enforce a wall:
  - This prevents one to access the data without using the defined operations.

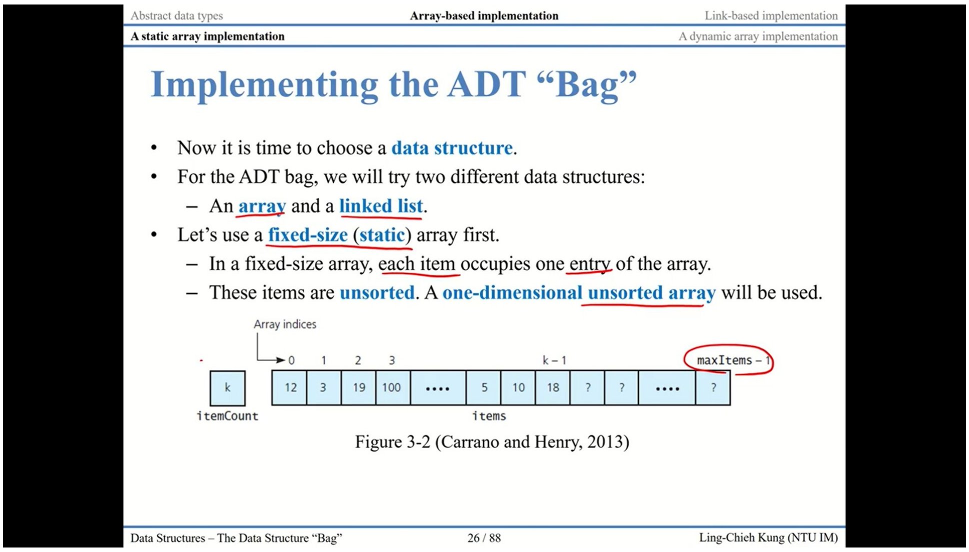
Figure 3-1 (Carrano and Henry, 2013)



Implementing the ADT “Bag”

- The abstract class `BagInterface` defines the **behaviors** of the ADT.
  - In many cases, **no member variable** is needed for the abstract class.

```
class BagInterface
{
public:
  virtual int getCurrentSize() const = 0;
  virtual bool isEmpty() const = 0;
  virtual bool add(const string& newEntry) = 0;
  virtual bool remove(const string& anEntry) = 0;
  virtual void clear() = 0;
  virtual int getFrequencyOf(const string& anEntry) const = 0;
  virtual bool contains(const string& anEntry) const = 0;
  virtual void print() const = 0;
};
```



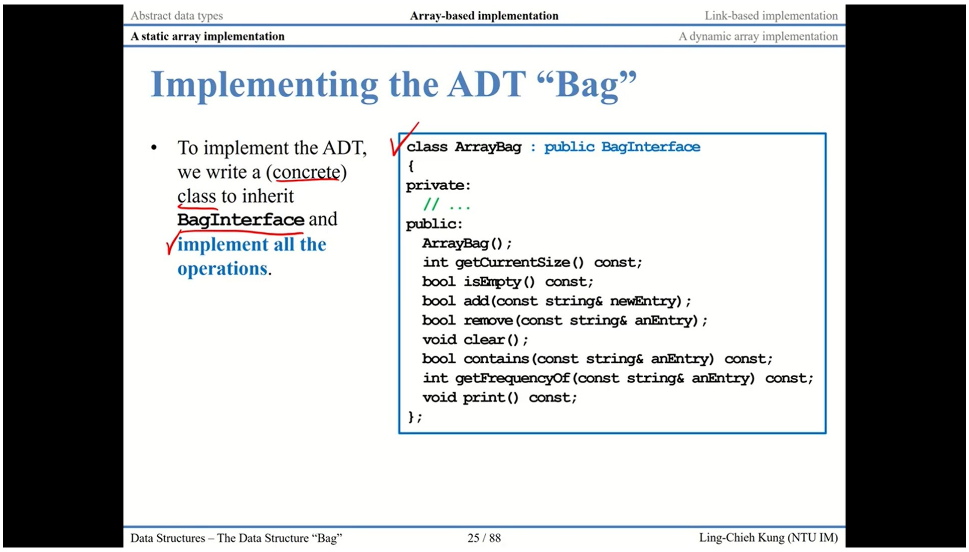
Abstract data types | **Array-based implementation** | Link-based implementation

A static array implementation | A dynamic array implementation

Implementing the ADT "Bag"

- Now it is time to choose a **data structure**.
- For the ADT bag, we will try two different data structures:
  - An **array** and a **linked list**.
- Let's use a **fixed-size** (**static**) array first.
  - In a fixed-size array, each item occupies one entry of the array.
  - These items are **unsorted**. A **one-dimensional unsorted array** will be used.

Figure 3-2 (Carrano and Henry, 2013)



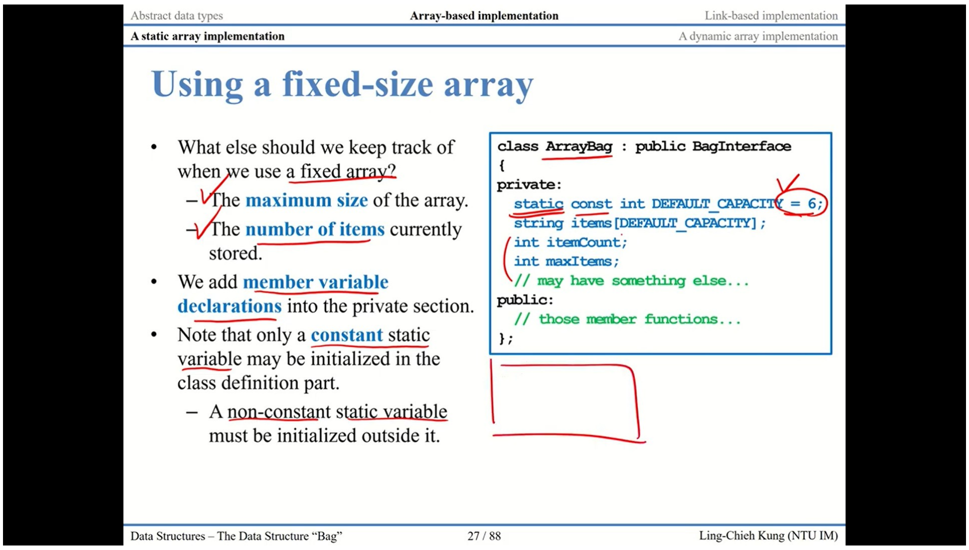
Abstract data types | **Array-based implementation** | Link-based implementation

A static array implementation | A dynamic array implementation

Implementing the ADT "Bag"

- To implement the ADT, we write a (concrete) class to inherit `BagInterface` and **implement all the operations**.

```
class ArrayBag : public BagInterface
{
private:
  // ...
public:
  ArrayBag();
  int getCurrentSize() const;
  bool isEmpty() const;
  bool add(const string& newEntry);
  bool remove(const string& anEntry);
  void clear();
  bool contains(const string& anEntry) const;
  int getFrequencyOf(const string& anEntry) const;
  void print() const;
};
```



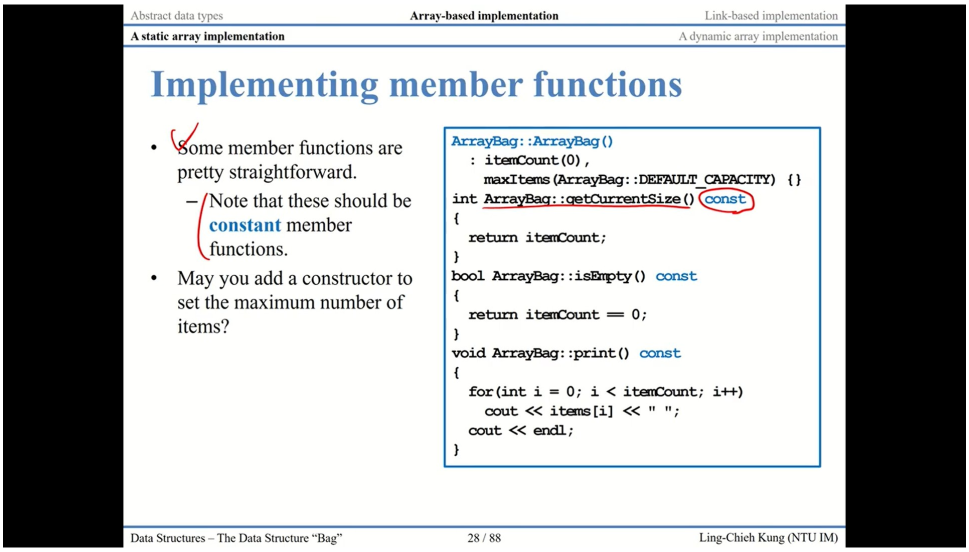
Abstract data types | **Array-based implementation** | Link-based implementation

A static array implementation | A dynamic array implementation

Using a fixed-size array

- What else should we keep track of when we use a fixed array?
  - The **maximum size** of the array.
  - The **number of items** currently stored.
- We add **member variable declarations** into the private section.
- Note that only a **constant** static variable may be initialized in the class definition part.
  - A non-constant static variable must be initialized outside it.

```
class ArrayBag : public BagInterface
{
private:
  static const int DEFAULT_CAPACITY = 6;
  string items[DEFAULT_CAPACITY];
  int itemCount;
  int maxItems;
  // may have something else...
public:
  // those member functions...
};
```



Abstract data types | **Array-based implementation** | Link-based implementation

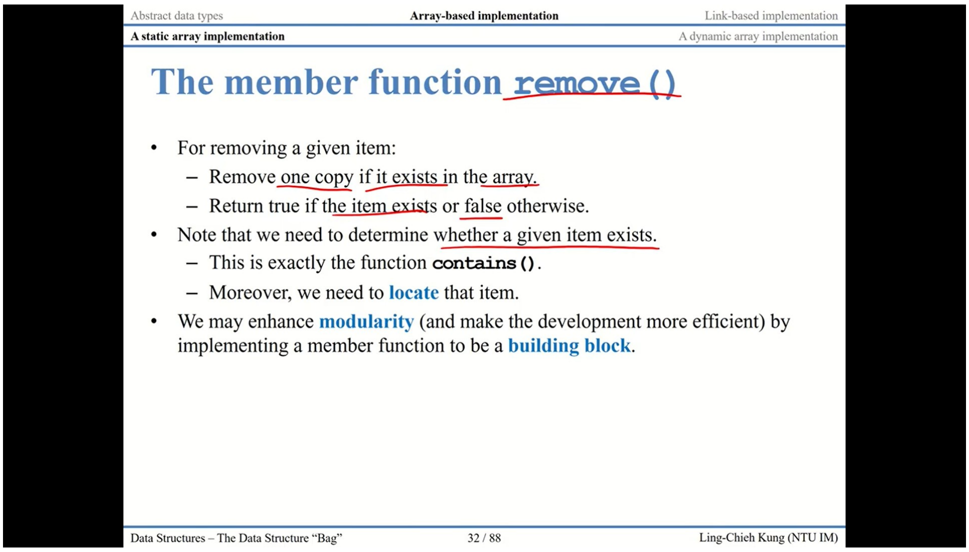
A static array implementation | A dynamic array implementation

Implementing member functions

- Some member functions are pretty straightforward.
  - Note that these should be **constant** member functions.
- May you add a constructor to set the maximum number of items?

```
ArrayBag::ArrayBag()
  : itemCount(0),
    maxItems(ArrayBag::DEFAULT_CAPACITY) {}
int ArrayBag::getCurrentSize() const
{
  return itemCount;
}
bool ArrayBag::isEmpty() const
{
  return itemCount == 0;
}
void ArrayBag::print() const
{
  for(int i = 0; i < itemCount; i++)
    cout << items[i] << " ";
  cout << endl;
}
```

Data Structures – The Data Structure "Bag" 28 / 88 Ling-Chieh Kung (NTU IM)



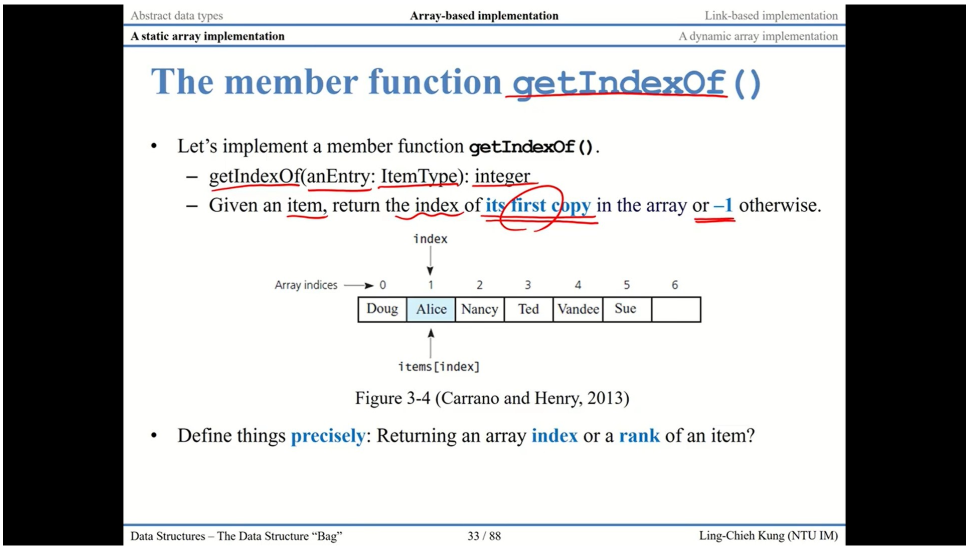
Abstract data types | **Array-based implementation** | Link-based implementation

A static array implementation | A dynamic array implementation

The member function remove()

- For removing a given item:
  - Remove one copy if it exists in the array.
  - Return true if the item exists or false otherwise.
- Note that we need to determine whether a given item exists.
  - This is exactly the function **contains()**.
  - Moreover, we need to **locate** that item.
- We may enhance **modularity** (and make the development more efficient) by implementing a member function to be a **building block**.

Data Structures – The Data Structure "Bag" 32 / 88 Ling-Chieh Kung (NTU IM)



Abstract data types | **Array-based implementation** | Link-based implementation

A static array implementation | A dynamic array implementation

The member function getIndexOf()

- Let's implement a member function **getIndexOf()**.
  - getIndexOf(anEntry: ItemType): integer
  - Given an item, return the index of **its first copy** in the array or **–1** otherwise.

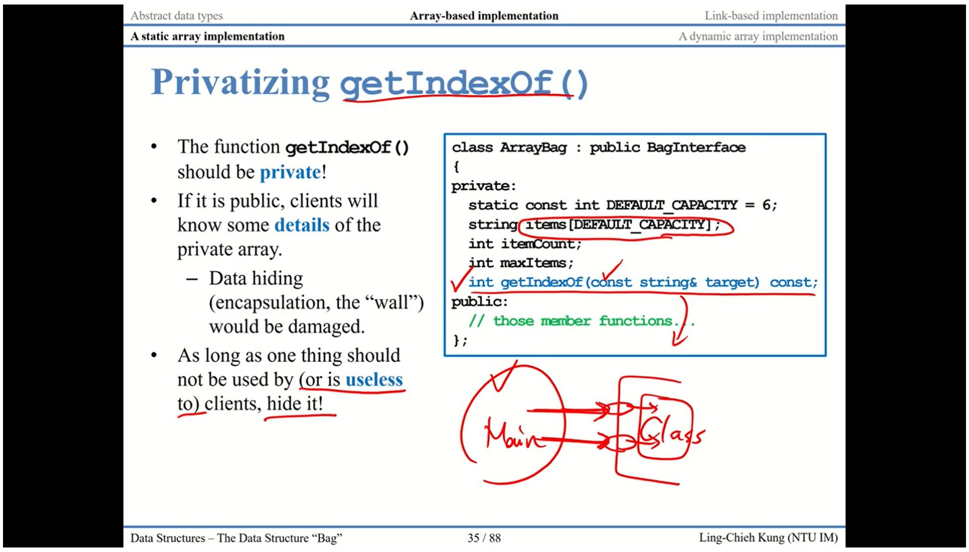
index
Array indices
items[index]

0	1	2	3	4	5	6
Doug	Alice	Nancy	Ted	Vandee	Sue	

Figure 3-4 (Carrano and Henry, 2013)

- Define things **precisely**: Returning an array **index** or a **rank** of an item?

Data Structures – The Data Structure "Bag" 33 / 88 Ling-Chieh Kung (NTU IM)



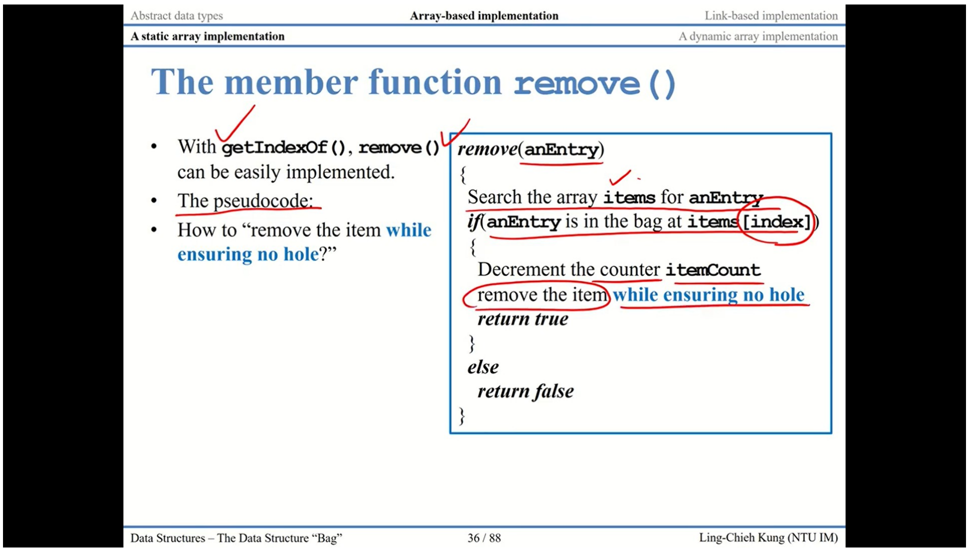
Privatizing `getIndexOf()`

- The function `getIndexOf()` should be **private**!
- If it is public, clients will know some **details** of the private array.
  - Data hiding (encapsulation, the “wall”) would be damaged.
- As long as one thing should not be used by (or is **useless** to) clients, hide it!

```
class ArrayBag : public BagInterface
{
private:
  static const int DEFAULT_CAPACITY = 6;
  string items[DEFAULT_CAPACITY];
  int itemCount;
  int maxItems;
  int getIndexOf(const string& target) const;
public:
  // those member functions...
};
```

Main

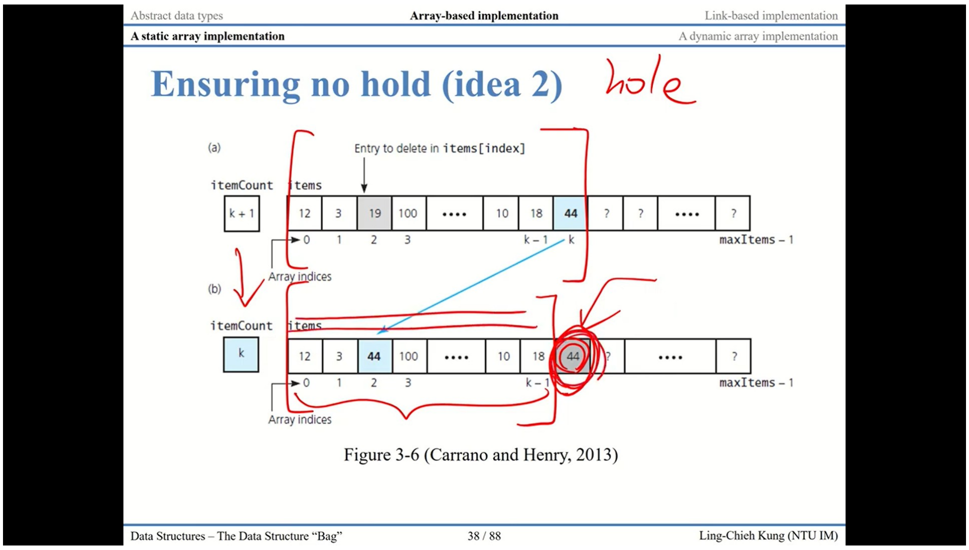
Class



The member function `remove()`

- With `getIndexOf()`, `remove()` can be easily implemented.
- The pseudocode:
- How to “remove the item **while ensuring no hole**?”

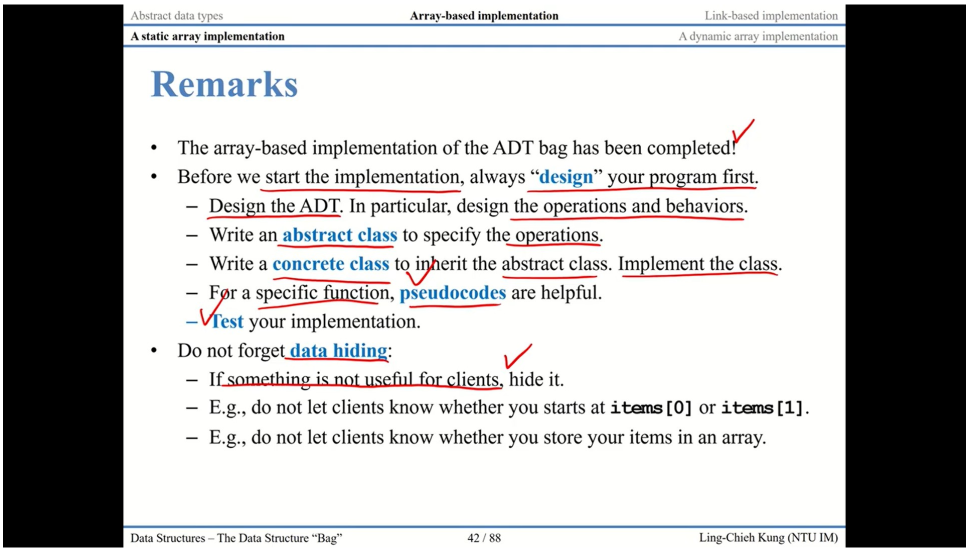
```
remove(anEntry)
{
  Search the array items for anEntry
  if(anEntry is in the bag at items[index])
  {
    Decrement the counter itemCount
    remove the item while ensuring no hole
    return true
  }
  else
    return false
}
```



Ensuring no hold (idea 2)

hole

Figure 3-6 (Carrano and Henry, 2013)



Abstract data types | Array-based implementation | Link-based implementation

A static array implementation | A dynamic array implementation

Remarks

- The array-based implementation of the ADT bag has been completed!
- Before we start the implementation, always "**design**" your program first.
  - Design the ADT. In particular, design the operations and behaviors.
  - Write an **abstract class** to specify the operations.
  - Write a **concrete class** to inherit the abstract class. Implement the class.
  - For a specific function, **pseudocodes** are helpful.
  - **Test** your implementation.
- Do not forget **data hiding**:
  - If something is not useful for clients, hide it.
  - E.g., do not let clients know whether you starts at `items[0]` or `items[1]`.
  - E.g., do not let clients know whether you store your items in an array.

Data Structures – The Data Structure "Bag" 42 / 88 Ling-Chieh Kung (NTU IM)

Implement by static Array

In [ ]:

```
#include <iostream>
#include <string>
using namespace std;

class BagInterface {
public:
    virtual int getCurrentSize() const = 0;
    virtual bool isEmpty() const = 0;
    virtual bool add(const string& newEntry) = 0;
    virtual bool remove(const string& anEntry) = 0;
    virtual void clear() = 0;
    virtual int getFrequencyOf(const string& anEntry) const = 0;
    virtual bool contains(const string& anEntry) const = 0;
    virtual void print() const =0;
};


class ArrayBag : public BagInterface{
private:
    static const int DEFAULT_CAPACITY = 6;
    string items[DEFAULT_CAPACITY];
    int itemCount;
    int maxItems;
    int getIndexOf(const string& target) const;
public:
    ArrayBag();
    int getCurrentSize() const;
    bool isEmpty() const;
    bool add(const string& newEntry);
    bool remove(const string& anEntry);
    void clear();
    int getFrequencyOf(const string& anEntry) const;
    bool contains(const string& anEntry) const;
    void print() const;
};

ArrayBag::ArrayBag()
    :itemCount(0),
     maxItems(ArrayBag::DEFAULT_CAPACITY){}
```

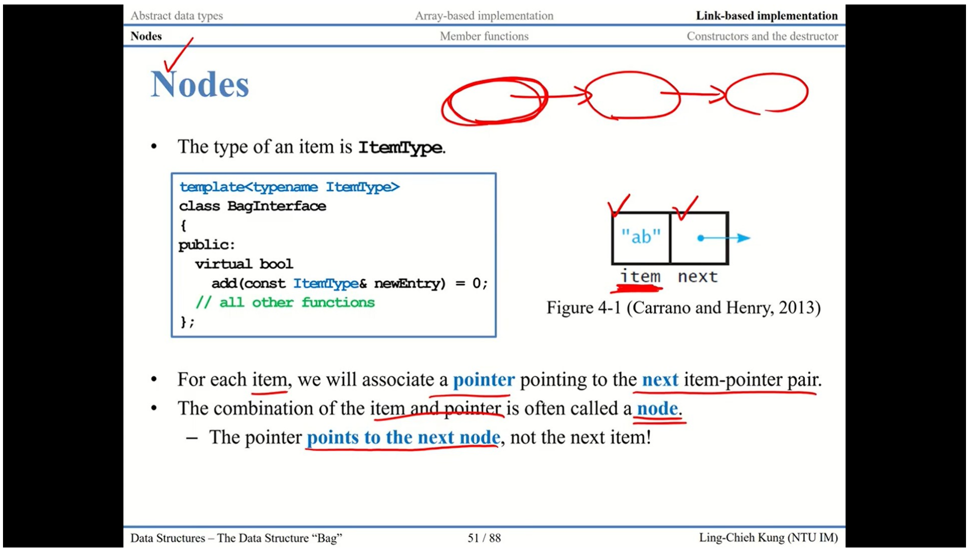
```
int ArrayBag::getCurrentSize() const {
    return itemCount;
}

bool ArrayBag::isEmpty() const {
    return itemCount == 0;
}

void ArrayBag::print() const {
    for (int i = 0; i < itemCount; i++){
        cout << items[i] << " ";
    }
    cout << endl;
}

bool ArrayBag::add(const string& newEntry){
    bool hasRoomToAdd = (itemCount < maxItems);
    if (hasRoomToAdd){
        items[itemCount] = newEntry;
        itemCount++;
    }
    return hasRoomToAdd;
}

int ArrayBag::getIndexOf(const string& target) const {
    bool found = false;
    int result = -1;
    int searchIndex = 0;
    while(!found && (searchIndex < itemCount)){
        if (items[searchIndex] == target){
            found = true;
            result = searchIndex;
        }
        else {
            searchIndex++;
        }
    }
    return result;
}

bool ArrayBag::remove(const string& anEntry){
    int locateIndex = getIndexOf(anEntry);
    bool canRemoveItem = (locateIndex > -1);
    if (canRemoveItem){
        itemCount--;
        items[locateIndex] = items[itemCount];
    }
    return canRemoveItem;
}

bool ArrayBag::contains(const string& anEntry) const {
    return getIndexOf(anEntry) > -1;
}

//void ArrayBag::clear(){
//    while(!isEmpty()){
//        remove(items[0]);
//    }
//}

void ArrayBag::clear(){
    itemCount = 0;
}

int ArrayBag::getFrequencyOf(const string& anEntry) const{
    int frequency = 0;
    int curIndex = 0;
    while(curIndex < itemCount){
        if(items[curIndex] == anEntry) {
            frequency++;
        }
        curIndex++;
    }
    return frequency;
}

int main(){
    ArrayBag bag;
    bag.add("aaa");
    bag.print();
    cout << bag.isEmpty() << endl;
    cout << bag.getCurrentSize() << endl;
    bag.add("bbb");
    bag.print();
    cout << bag.isEmpty() << endl;
    cout << bag.getCurrentSize() << endl;

    return 0;
}
```

Implement by Dynamic Memory Allocate

In [ ]:

```
#include <iostream>
#include <string>
using namespace std;

class BagInterface {
public:
    virtual int getCurrentSize() const = 0;
    virtual bool isEmpty() const = 0;
    virtual bool add(const string& newEntry) = 0;
    virtual bool remove(const string& anEntry) = 0;
    virtual void clear() = 0;
    virtual int getFrequencyOf(const string& anEntry) const = 0;
    virtual bool contains(const string& anEntry) const = 0;
    virtual void print() const =0;
};


class ArrayBag : public BagInterface{
private:
    static const int DEFAULT_CAPACITY = 6;
    string* items;
    int itemCount;
    int maxItems;
    int getIndexOf(const string& target) const;
public:
    ArrayBag();
    ~ArrayBag();
    int getCurrentSize() const;
    bool isEmpty() const;
    bool add(const string& newEntry);
    bool remove(const string& anEntry);
    void clear();
    int getFrequencyOf(const string& anEntry) const;
    bool contains(const string& anEntry) const;
    void print() const;
};

ArrayBag::ArrayBag():
          itemCount(0),
          maxItems(ArrayBag::DEFAULT_CAPACITY){
    items = new string[DEFAULT_CAPACITY];
}

ArrayBag::~ArrayBag(){
    delete [] items;
}

int ArrayBag::getCurrentSize() const {
    return itemCount;
}

bool ArrayBag::isEmpty() const {
    return itemCount == 0;
}

void ArrayBag::print() const {
    for (int i = 0; i < itemCount; i++){
        cout << items[i] << " ";
    }
    cout << endl;
}

bool ArrayBag::add(const string& newEntry){
    bool hasRoomToAdd = (itemCount < maxItems);
    if (hasRoomToAdd){
        string* oldArray = items;
        items = new string[2 * maxItems];
        for (int index = 0; index < maxItems; index++){
            items[index] = oldArray[index];
        }
        delete [] oldArray;
        maxItems = 2 * maxItems;

    }
    items[itemCount] = newEntry;
    itemCount++;
    return true;
}

int ArrayBag::getIndexOf(const string& target) const {
    bool found = false;
    int result = -1;
    int searchIndex = 0;
    while(!found && (searchIndex < itemCount)){
        if (items[searchIndex] == target){
            found = true;
            result = searchIndex;
        }
        else {
            searchIndex++;
        }
    }
    return result;
}

bool ArrayBag::remove(const string& anEntry){
    int locateIndex = getIndexOf(anEntry);
    bool canRemoveItem = (locateIndex > -1);
    if (canRemoveItem){
        itemCount--;
        items[locateIndex] = items[itemCount];
    }
    return canRemoveItem;
}

bool ArrayBag::contains(const string& anEntry) const {
    return getIndexOf(anEntry) > -1;
}

//void ArrayBag::clear(){
//    while(!isEmpty()){
//        remove(items[0]);
//    }
//}

void ArrayBag::clear(){
    itemCount = 0;
}

int ArrayBag::getFrequencyOf(const string& anEntry) const{
    int frequency = 0;
    int curIndex = 0;
    while(curIndex < itemCount){
        if(items[curIndex] == anEntry) {
            frequency++;
        }
        curIndex++;
    }
    return frequency;
}

int main(){
    ArrayBag bag;
    bag.add("aaa");
    bag.print();
    cout << bag.isEmpty() << endl;
    cout << bag.getCurrentSize() << endl;
    bag.add("bbb");
    bag.print();
    cout << bag.isEmpty() << endl;
    cout << bag.getCurrentSize() << endl;

    return 0;
}
```

> ## Implement by Linked List



Abstract data types | Array-based implementation | **Link-based implementation**

Nodes | Member functions | Constructors and the destructor

Nodes

- The type of an item is **ItemType**.

```
template<typename ItemType>
class BagInterface
{
public:
  virtual bool
    add(const ItemType& newEntry) = 0;
  // all other functions
};
```

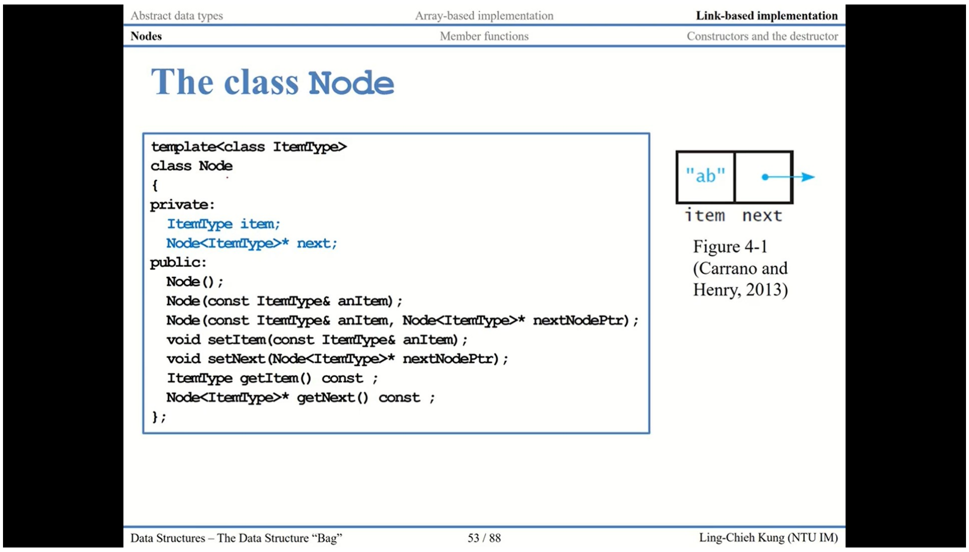
"ab"

item next

Figure 4-1 (Carrano and Henry, 2013)

- For each item, we will associate a **pointer** pointing to the **next** item-pointer pair.
- The combination of the item and pointer is often called a **node**.
  - The pointer **points to the next node**, not the next item!

Data Structures – The Data Structure "Bag" 51 / 88 Ling-Chieh Kung (NTU IM)



Abstract data types | Array-based implementation | **Link-based implementation**

Nodes | Member functions | Constructors and the destructor

The class Node

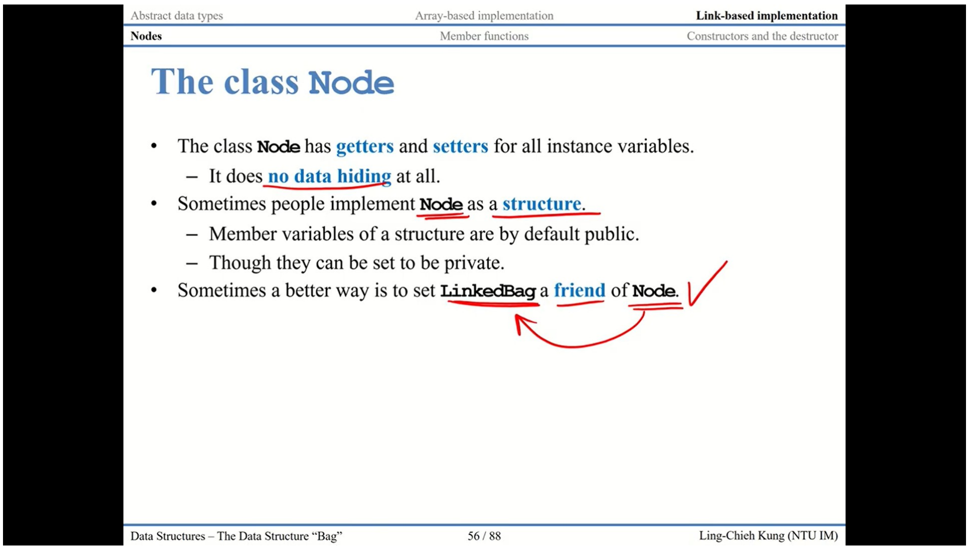
```
template<class ItemType>
class Node
{
private:
  ItemType item;
  Node<ItemType>* next;
public:
  Node();
  Node(const ItemType& anItem);
  Node(const ItemType& anItem, Node<ItemType>* nextNodePtr);
  void setItem(const ItemType& anItem);
  void setNext(Node<ItemType>* nextNodePtr);
  ItemType getItem() const ;
  Node<ItemType>* getNext() const ;
};
```

"ab"

item next

Figure 4-1 (Carrano and Henry, 2013)

Data Structures – The Data Structure "Bag" 53 / 88 Ling-Chieh Kung (NTU IM)



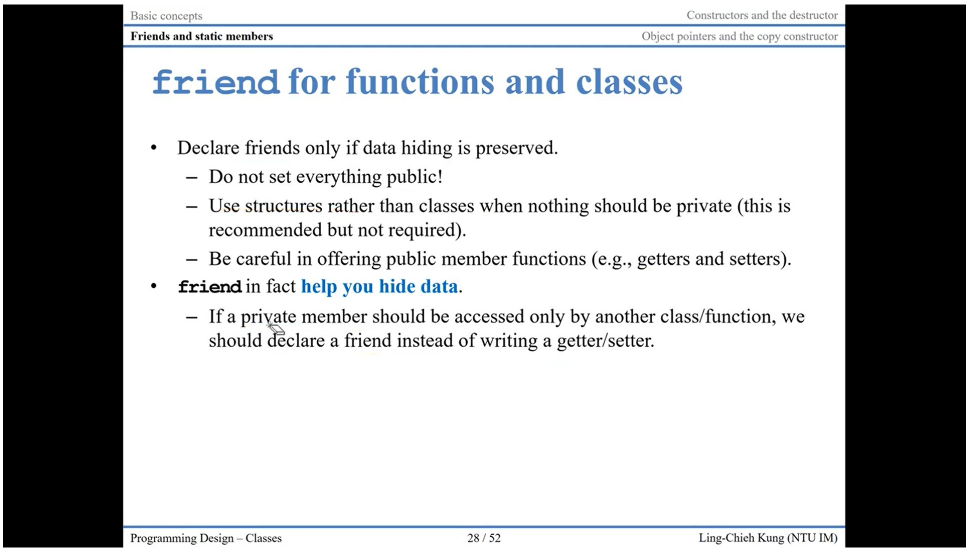
Abstract data types | Array-based implementation | **Link-based implementation**

Nodes | Member functions | Constructors and the destructor

The class Node

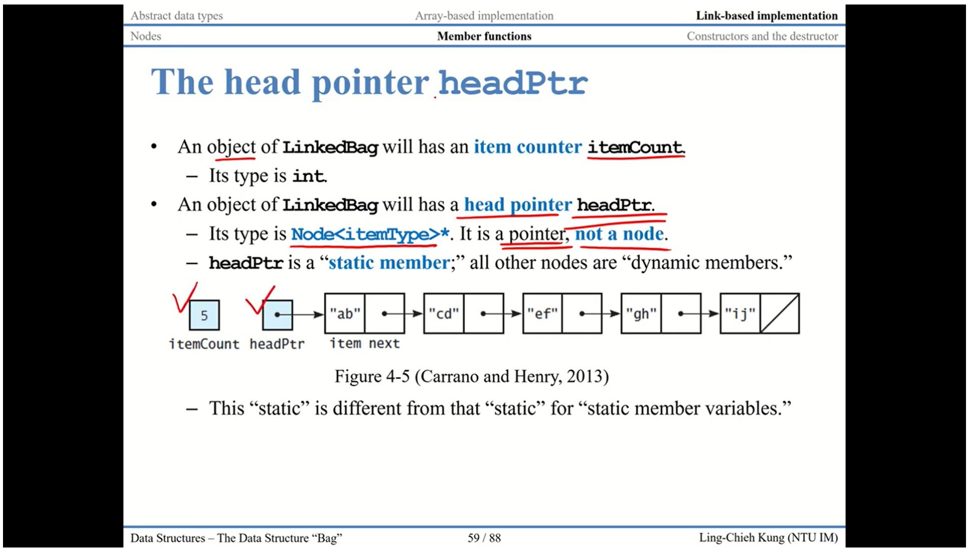
- The class **Node** has **getters** and **setters** for all instance variables.
  - It does **no data hiding** at all.
- Sometimes people implement **Node** as a **structure**.
  - Member variables of a structure are by default public.
  - Though they can be set to be private.
- Sometimes a better way is to set **LinkedBag** a **friend** of **Node**.

Data Structures – The Data Structure "Bag" 56 / 88 Ling-Chieh Kung (NTU IM)



friend for functions and classes

- Declare friends only if data hiding is preserved.
  - Do not set everything public!
  - Use structures rather than classes when nothing should be private (this is recommended but not required).
  - Be careful in offering public member functions (e.g., getters and setters).
- friend in fact **help you hide data**.
  - If a private member should be accessed only by another class/function, we should declare a friend instead of writing a getter/setter.

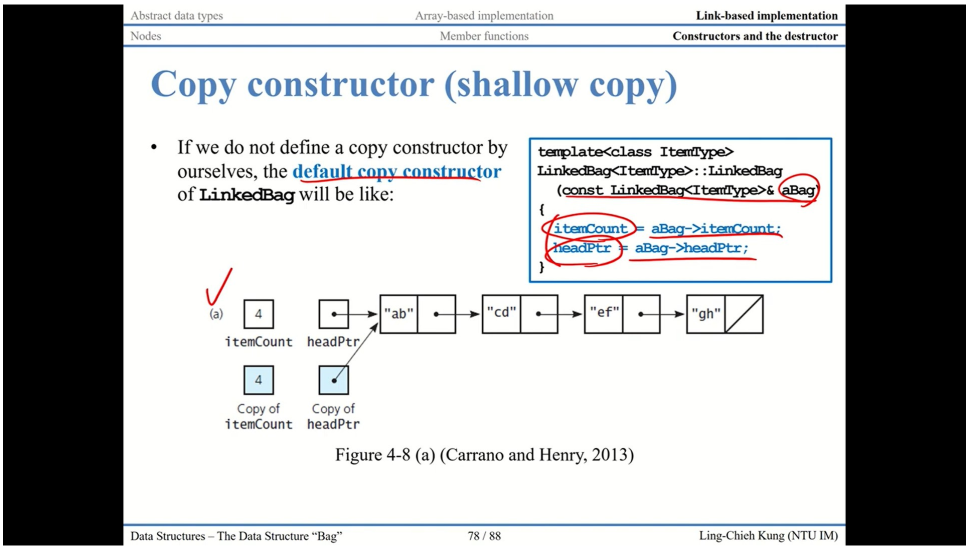


The head pointer headPtr

- An object of LinkedBag will has an **item counter** itemCount.
  - Its type is int.
- An object of LinkedBag will has a **head pointer** headPtr.
  - Its type is Node<itemType>*. It is a pointer, **not a node**.
  - headPtr is a "**static member**;" all other nodes are "dynamic members."

Figure 4-5 (Carrano and Henry, 2013)

- This "static" is different from that "static" for "static member variables."

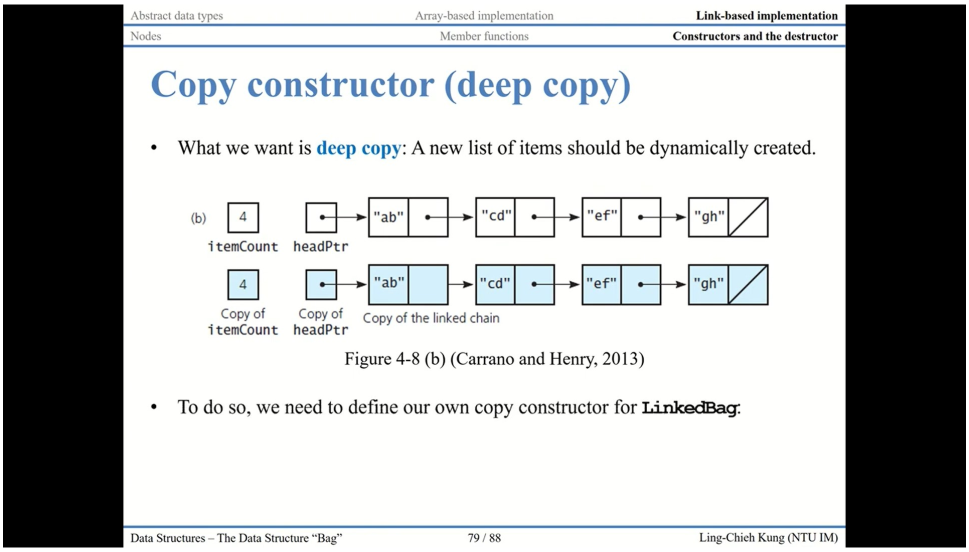


Copy constructor (shallow copy)

- If we do not define a copy constructor by ourselves, the **default copy constructor** of LinkedBag will be like:

```
template<class ItemType>
LinkedBag<ItemType>::LinkedBag
  (const LinkedBag<ItemType>& aBag)
{
  itemCount = aBag->itemCount;
  headPtr = aBag->headPtr;
}
```

Figure 4-8 (a) (Carrano and Henry, 2013)



Abstract data types Array-based implementation **Link-based implementation**

Nodes Member functions **Constructors and the destructor**

Copy constructor (deep copy)

- What we want is **deep copy**: A new list of items should be dynamically created.

Figure 4-8 (b) (Carrano and Henry, 2013)

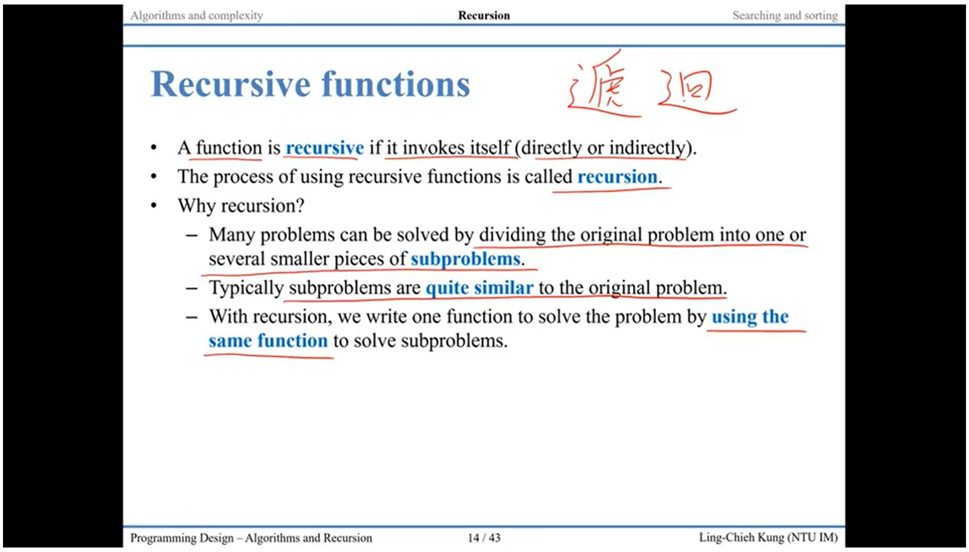
- To do so, we need to define our own copy constructor for `LinkedBag`.

Data Structures – The Data Structure "Bag" 79 / 88 Ling-Chieh Kung (NTU IM)

In [ ]:

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;


//Class Node
template<typename ItemType>
class Node{
private:
    ItemType item;
    Node<ItemType>* next;
public:
    Node();
    Node(const ItemType& anItem);
    Node(const ItemType& anItem, Node<ItemType>* nextNodePtr);
    void setItem(const ItemType& anItem);
    void setNext(Node<ItemType>* nextNodePtr);
    ItemType getItem() const;
    Node<ItemType>* getNext() const;
};

//Constructors of Node
template<typename ItemType>
Node<ItemType>::Node()
    : next(nullptr) {}

template<typename ItemType>
Node<ItemType>::Node(const ItemType& anItem)
    : item(anItem), next(nullptr) {}

template<typename ItemType>
Node<ItemType>::Node(const ItemType& anItem, Node<ItemType>* nextNodePtr)
    : item(anItem), next(nextNodePtr) {}

// Getters and Setters
template<typename ItemType>
void Node<ItemType>::setItem(const ItemType& anItem){
    item = anItem;
}

template<typename ItemType>
void Node<ItemType>::setNext(Node<ItemType>* nextNodePtr){
    next = nextNodePtr;
}

template<typename ItemType>
ItemType Node<ItemType>::getItem() const {
    return item;
}

template<typename ItemType>
Node<ItemType>* Node<ItemType>::getNext() const {
    return next;
}


//Class BagInterface
template<typename ItemType>
class BagInterface {
public:
    virtual int getCurrentSize() const = 0;
    virtual bool isEmpty() const = 0;
    virtual bool add(const ItemType& newEntry) = 0;
    virtual bool remove(const ItemType& anEntry) = 0;
    virtual void clear() = 0;
    virtual int getFrequencyOf(const ItemType& anEntry) const = 0;
    virtual bool contains(const ItemType& anEntry) const = 0;
    virtual void print() const =0;
};


//class LinkedBag
template<typename ItemType>
class LinkedBag : public BagInterface<ItemType> {
private:
    Node<ItemType>* headPtr;
    int itemCount;
public:
    LinkedBag(); // constructor
    LinkedBag(const LinkedBag<ItemType>& aBag); // copy constructor
    virtual ~LinkedBag(); // destructor
    int getCurrentSize() const;
    bool isEmpty() const;
    bool add(const ItemType& newEntry);
    vector<ItemType> toVector() const;
    bool remove(const ItemType& anEntry);
    void clear();
    Node<ItemType>* getPointerTo(const ItemType& anEntry) const;
    int getFrequencyOf(const ItemType& anEntry) const;
    bool contains(const ItemType& anEntry) const;
    void print() const;
};


template<typename ItemType>
LinkedBag<ItemType>:: LinkedBag(){
    headPtr = nullptr;
    itemCount = 0;
}

template<typename ItemType>
LinkedBag<ItemType>::LinkedBag(const LinkedBag<ItemType>& aBag){
    itemCount = aBag->itemCount;
    Node<ItemType>* origChainPtr = aBag->headPtr;
    if(origChainPtr == nullptr){
        headPtr = nullptr;
    }
    else {
        headPtr = new Node<ItemType>();
        headPtr->setItem(origChainPtr->getItem);
        Node<ItemType>* newChainPtr = headPtr;
        while(origChainPtr != nullptr){
            origChainPtr = origChainPtr->getNext();
            ItemType nextItem = origChainPtr->getItem();
            Node<ItemType>* newNodePtr = new Node<ItemType>(nextItem);
            newChainPtr->setNext(newNodePtr);
            newChainPtr = newChainPtr->getNext();
        }
        newChainPtr->setNext(nullptr);
    }
}

template<typename ItemType>
LinkedBag<ItemType>::~LinkedBag(){
    clear();
}

template<typename ItemType>
bool LinkedBag<ItemType>::isEmpty() const{
    return itemCount == 0;
}

template<typename ItemType>
int LinkedBag<ItemType>::getCurrentSize() const {
    return itemCount;
}

//add to the beginning
template<typename ItemType>
bool LinkedBag<ItemType>::add(const ItemType& newEntry){
    Node<ItemType>* newNodePtr = new Node<ItemType>();
    newNodePtr->setItem(newEntry);
    newNodePtr->setNext(headPtr);
    headPtr = newNodePtr;
    itemCount++;
    return true;
}

template<typename ItemType>
vector<ItemType> LinkedBag<ItemType>::toVector() const{
    vector<ItemType> bagContents;
    Node<ItemType>* curPtr = headPtr;
    while(curPtr != nullptr ){
        bagContents.push_back(curPtr->getItem());
        curPtr = curPtr->getNext();
    }
    return bagContents;
}

template<typename ItemType>
bool LinkedBag<ItemType>::remove(const ItemType& anEntry){
    Node<ItemType>* entryNodePtr = getPointerTo(anEntry);

    bool canRemoveItem = !isEmpty() && (entryNodePtr != nullptr);
    if (canRemoveItem) {
        entryNodePtr->setItem(headPtr->getItem());//Replace
        Node<ItemType>* nodeToDeletePtr = headPtr;
        headPtr = headPtr->getNext();
        delete nodeToDeletePtr;
        nodeToDeletePtr = nullptr;
        itemCount--;
    }
    return canRemoveItem;
}

template<typename ItemType>
Node<ItemType>* LinkedBag<ItemType>::getPointerTo(const ItemType& anEntry) const {
    bool found = false;
    Node<ItemType>* curPtr = headPtr;
    while(!found && (curPtr != nullptr)){
        if(anEntry == curPtr->getItem()){
            found = true;
        }
        else
            curPtr = curPtr->getNext();
    }
    return curPtr;
}

template<typename ItemType>
bool LinkedBag<ItemType>::contains(const ItemType& anEntry) const {
    return (getPointerTo(anEntry) != nullptr);
}

template<typename ItemType>
int LinkedBag<ItemType>::getFrequencyOf(const ItemType& anEntry) const {
    int frequency = 0;
    Node<ItemType>* curPtr = headPtr;
    while(curPtr != nullptr){
        if(anEntry == curPtr->getItem()){
            frequency++;
        }
        curPtr = curPtr->getNext();
    }
    return frequency;
}

template<typename ItemType>
void LinkedBag<ItemType>::clear(){
    Node<ItemType>* nodeToDeletePtr = headPtr;
    while(headPtr != nullptr){
        headPtr = headPtr->getNext();
        delete nodeToDeletePtr;
        nodeToDeletePtr = headPtr;
    }
    itemCount = 0;
}

void testBag(BagInterface<string>* bagPtr){
    string item[] = {"aa", "bb", "cc"};
    for(int i = 0; i < 3; i++){
        bagPtr->add(item[i]);
    }
    cout << bagPtr->isEmpty();
    cout << bagPtr->getCurrentSize();

    bagPtr->remove("two");
    cout << bagPtr->isEmpty();
    cout << bagPtr->getCurrentSize();
}


int main(){
    pass

    return 0;
}
```

Recursive Function

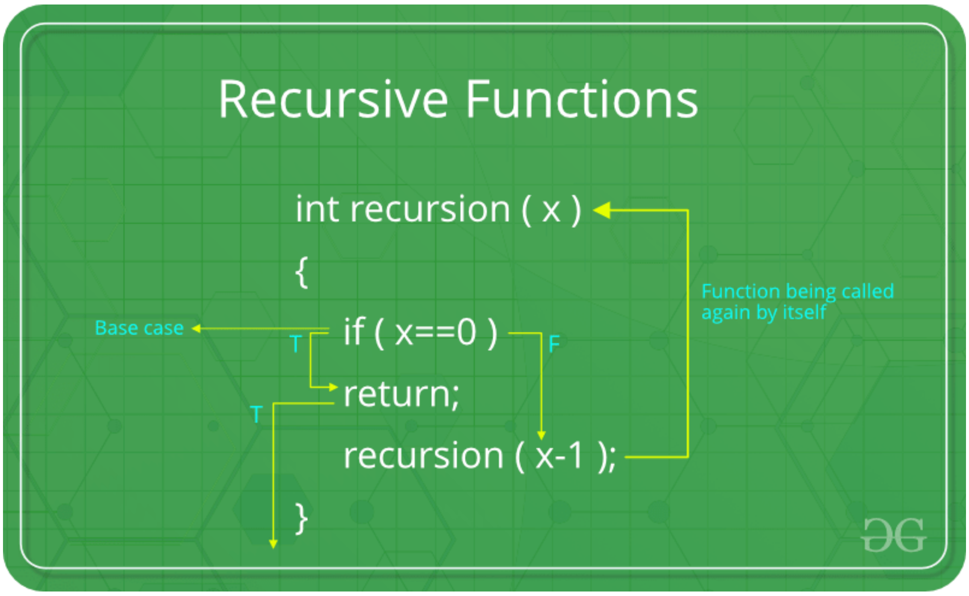


Algorithms and complexity | Recursion | Searching and sorting

Recursive functions 遞迴

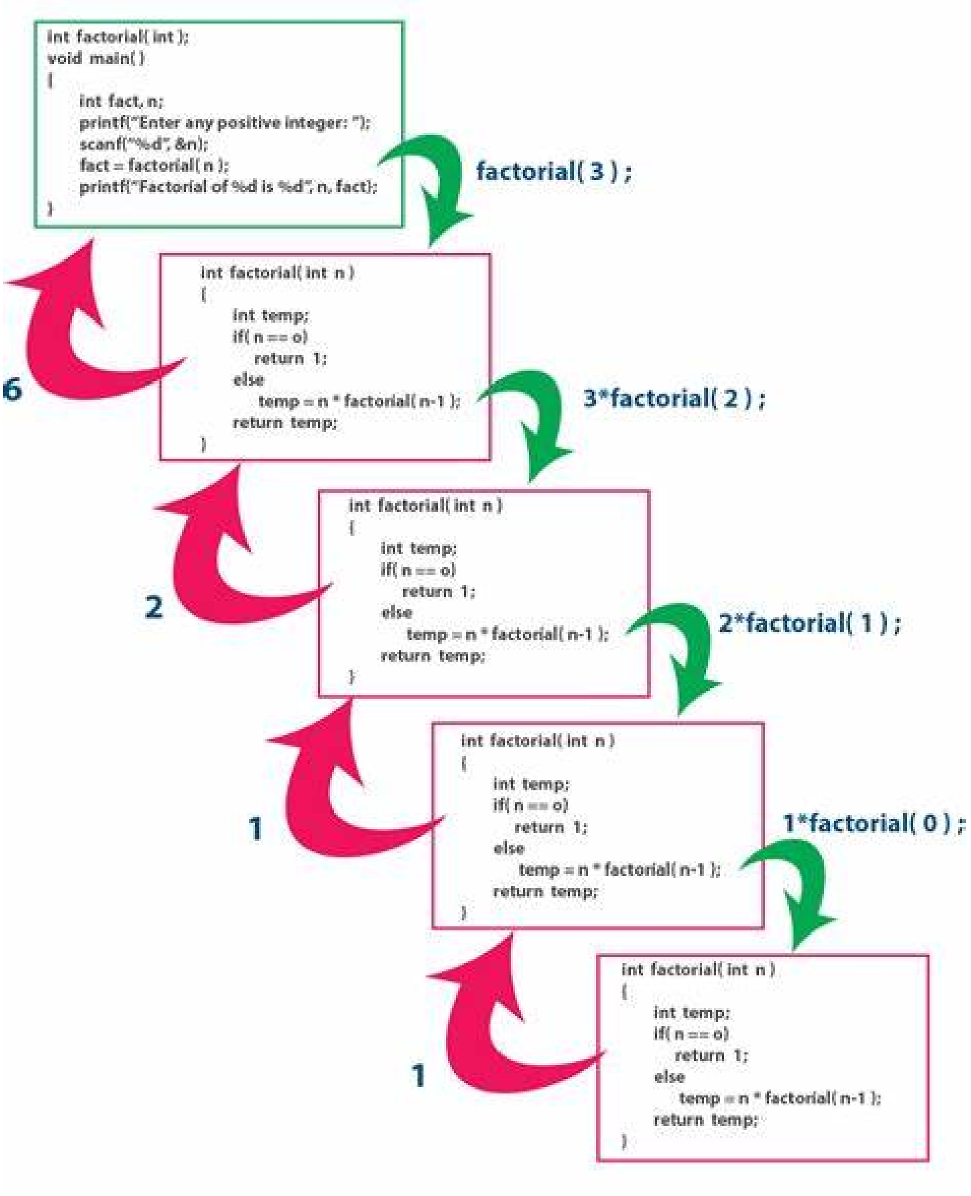
- A function is **recursive** if it invokes itself (directly or indirectly).
- The process of using recursive functions is called **recursion**.
- Why recursion?
  - Many problems can be solved by dividing the original problem into one or several smaller pieces of **subproblems**.
  - Typically subproblems are **quite similar** to the original problem.
  - With recursion, we write one function to solve the problem by **using the same function** to solve subproblems.

Programming Design – Algorithms and Recursion 14 / 43 Ling-Chieh Kung (NTU IM)



Recursive Functions

```
int recursion ( x )
{
    if ( x==0 )
    return;
    recursion ( x-1 );
}
```



```
int factorial( int );
void main( )
{
    int fact, n;
    printf("Enter any positive integer: ");
    scanf("%d", &n);
    fact = factorial( n );
    printf("Factorial of %d is %d", n, fact);
}
```

```
int factorial( int n )
{
    int temp;
    if( n == o)
        return 1;
    else
        temp = n * factorial( n-1 );
    return temp;
}
```

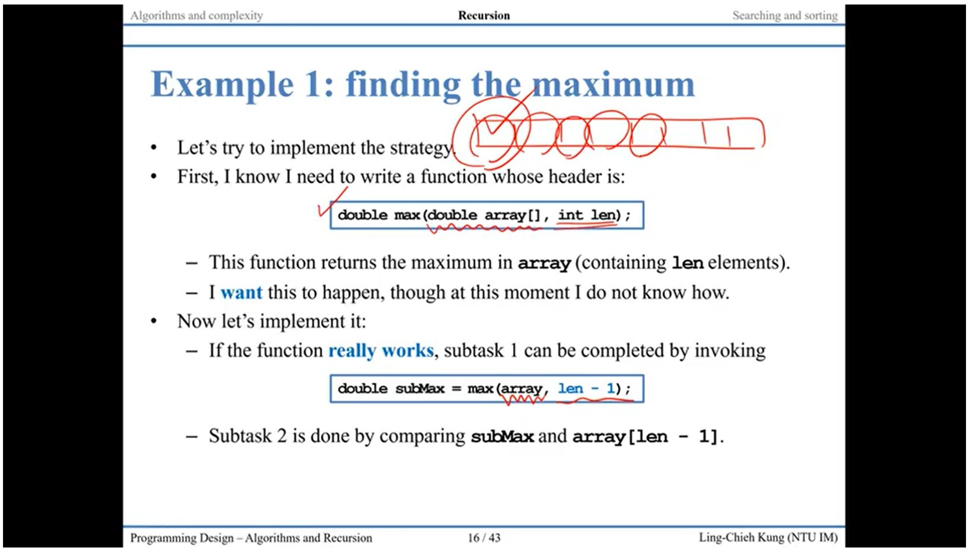
```
int factorial( int n )
{
    int temp;
    if( n == o)
        return 1;
    else
        temp = n * factorial( n-1 );
    return temp;
}
```

```
int factorial( int n )
{
    int temp;
    if( n == o)
        return 1;
    else
        temp = n * factorial( n-1 );
    return temp;
}
```

```
int factorial( int n )
{
    int temp;
    if( n == o)
        return 1;
    else
        temp = n * factorial( n-1 );
    return temp;
}
```

In [ ]:

```
#include <iostream>
using namespace std;


int fact(int n);


int main(){

    cout << "fact(5): " << fact(5) << endl;

    return 0;
}

int fact(int n){
    if(n==0)
        return 1;
    else
        return n * fact(n-1);
}
```



Algorithms and complexity
Recursion
Searching and sorting

Example 1: finding the maximum

- Let's try to implement the strategy.
- First, I know I need to write a function whose header is:

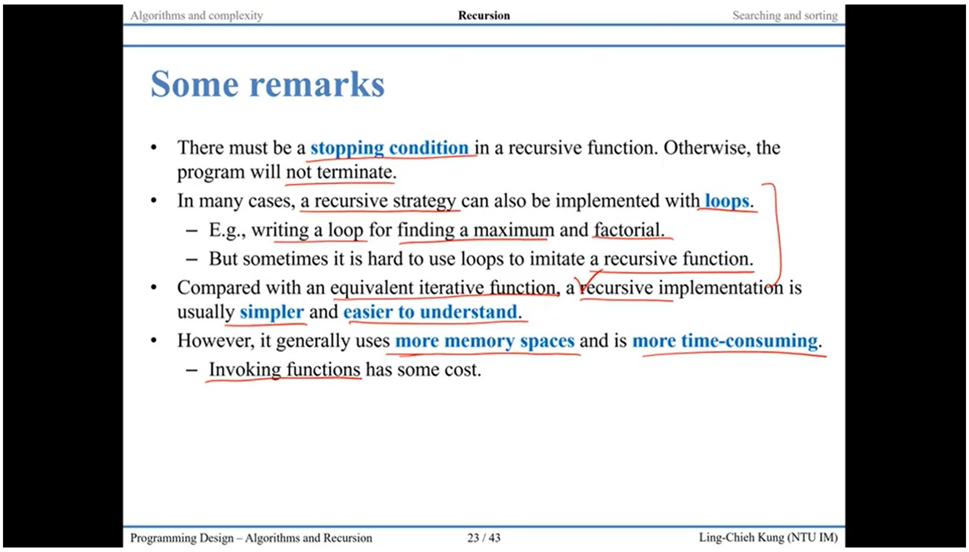
```
double max(double array[], int len);
```

  - This function returns the maximum in **array** (containing **len** elements).
  - I **want** this to happen, though at this moment I do not know how.
- Now let's implement it:
  - If the function **really works**, subtask 1 can be completed by invoking

```
double subMax = max(array, len - 1);
```

  - Subtask 2 is done by comparing **subMax** and **array[len - 1]**.

Programming Design – Algorithms and Recursion
16 / 43
Ling-Chieh Kung (NTU IM)



Algorithms and complexity
Recursion
Searching and sorting

Some remarks

- There must be a **stopping condition** in a recursive function. Otherwise, the program will not terminate.
- In many cases, a recursive strategy can also be implemented with **loops**.
  - E.g., writing a loop for finding a maximum and factorial.
  - But sometimes it is hard to use loops to imitate a recursive function.
- Compared with an equivalent iterative function, a recursive implementation is usually **simpler** and **easier to understand**.
- However, it generally uses **more memory spaces** and is **more time-consuming**.
  - Invoking functions has some cost.

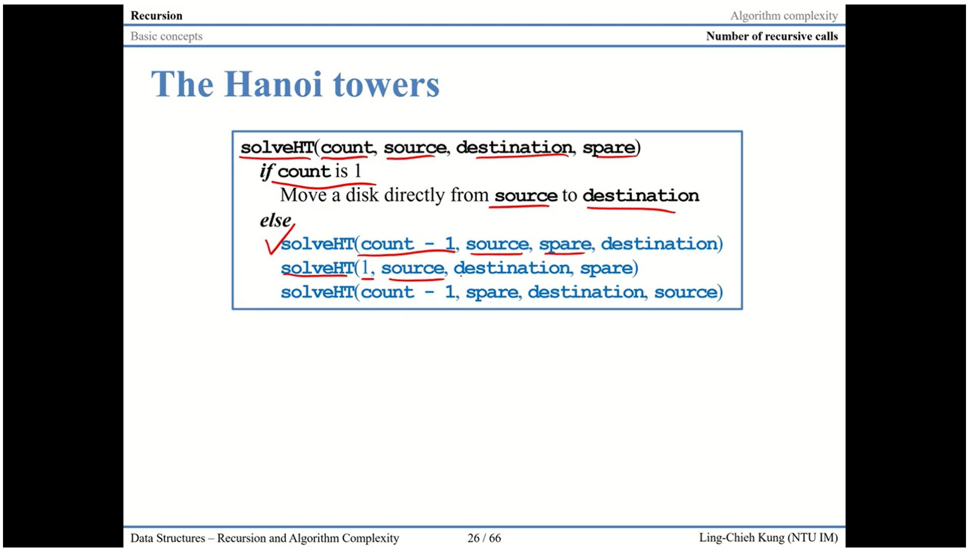
In [ ]:

```
#include <iostream>
using namespace std;

double findMax(double array[], int len);

int main(){
    double a[5] = {5, 7, 2, 4, 3};
    cout << "The max item is " << findMax(a, 5) << endl;

    return 0;
}

double findMax(double array[], int len){
 if (len == 1){
        return array[0];
 }
 else {
    double subMax = findMax(array, len-1);
    if (array[len-1] > subMax){
        return array[len -1];
    }
    else {
        return subMax;
    }
```

```
  }
}
```

In [ ]:

```
#include <iostream>
using namespace std;


int choose(int n, int k);


int main(){

    cout << "Choose(3 from 5 items): " << choose(5, 3) << endl;

    return 0;
}

int choose(int n, int k){
    if(k == 0 || k == n)
        return 1;
    else if(k > n)
        return 0;
    else
        return choose(n-1, k-1) + choose(n-1, k);
}
```



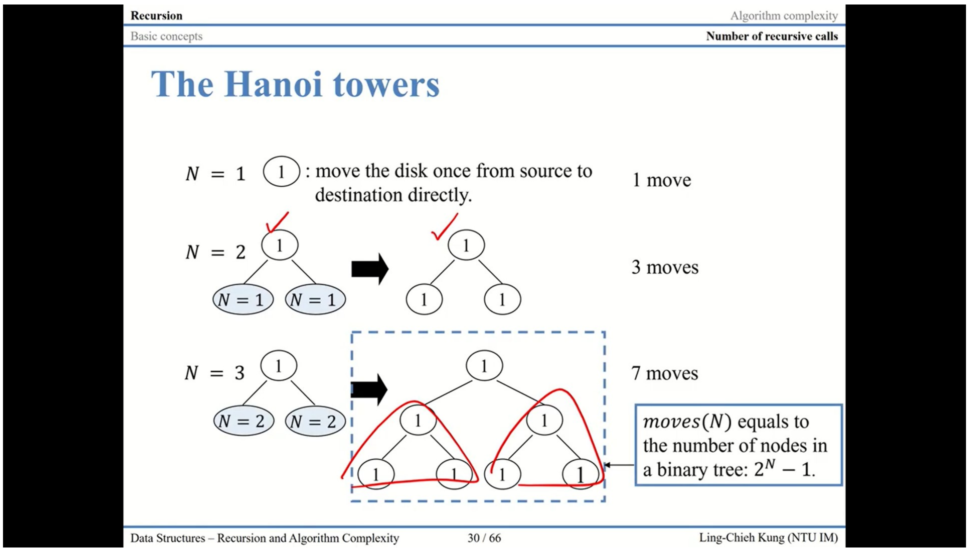
Recursion | Algorithm complexity

Basic concepts | Number of recursive calls

The Hanoi towers

```
solveHT(count, source, destination, spare)
    if count is 1
        Move a disk directly from source to destination
    else
        solveHT(count − 1, source, spare, destination)
        solveHT(1, source, destination, spare)
        solveHT(count − 1, spare, destination, source)
```

Data Structures – Recursion and Algorithm Complexity 26 / 66 Ling-Chieh Kung (NTU IM)



Recursion | Algorithm complexity

Basic concepts | Number of recursive calls

The Hanoi towers

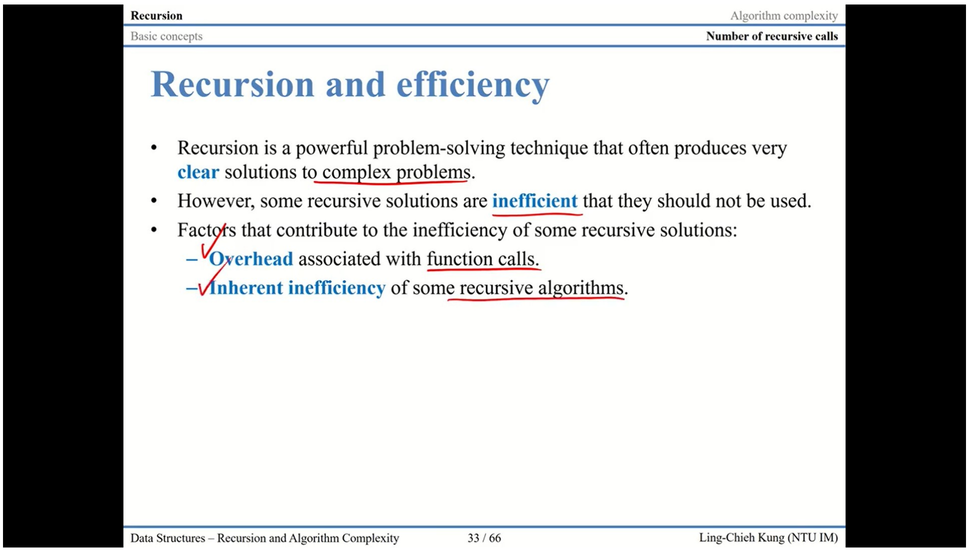
$N = 1$ (1) : move the disk once from source to destination directly. 1 move

$N = 2$ 3 moves

$N = 3$ 7 moves

$moves(N)$ equals to the number of nodes in a binary tree: $2^N - 1$.

Data Structures – Recursion and Algorithm Complexity 30 / 66 Ling-Chieh Kung (NTU IM)



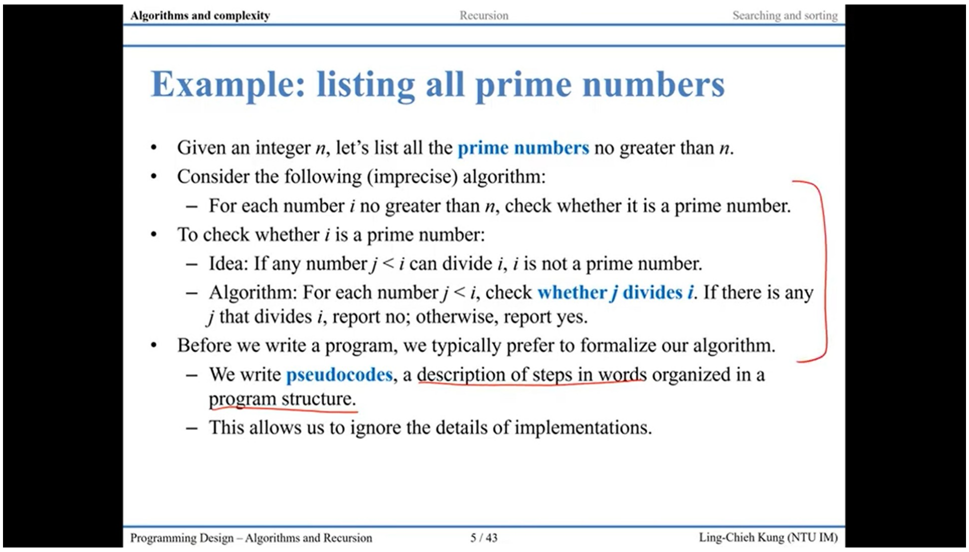
Recursion | Algorithm complexity
Basic concepts | Number of recursive calls

Recursion and efficiency

- Recursion is a powerful problem-solving technique that often produces very **clear** solutions to complex problems.
- However, some recursive solutions are **inefficient** that they should not be used.
- Factors that contribute to the inefficiency of some recursive solutions:
  - ✓**Overhead** associated with function calls.
  - ✓**Inherent inefficiency** of some recursive algorithms.

Data Structures – Recursion and Algorithm Complexity 33 / 66 Ling-Chieh Kung (NTU IM)

Find Prime Number

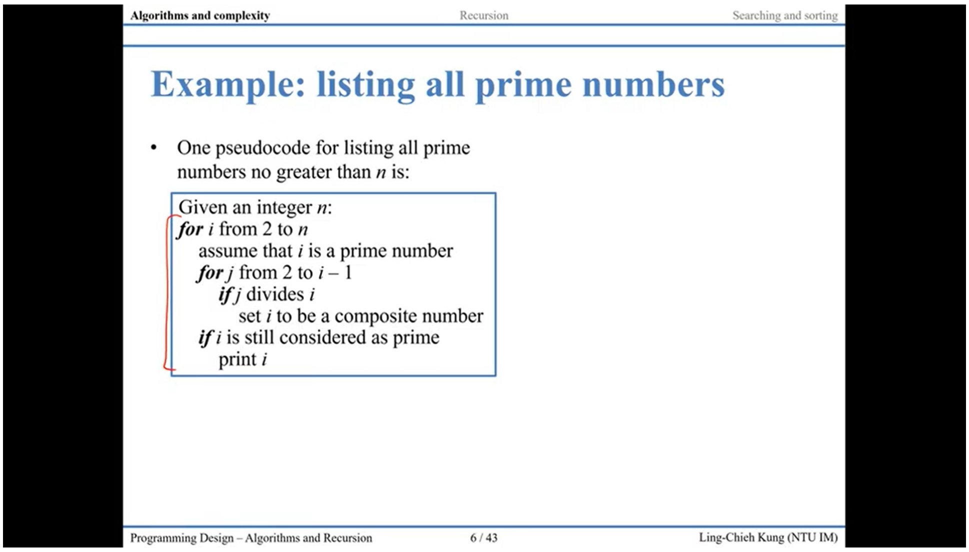


Algorithms and complexity | Recursion | Searching and sorting

Example: listing all prime numbers

- Given an integer n, let's list all the **prime numbers** no greater than n.
- Consider the following (imprecise) algorithm:
  - For each number i no greater than n, check whether it is a prime number.
- To check whether i is a prime number:
  - Idea: If any number $j < i$ can divide i, i is not a prime number.
  - Algorithm: For each number $j < i$, check **whether j divides i**. If there is any j that divides i, report no; otherwise, report yes.
- Before we write a program, we typically prefer to formalize our algorithm.
  - We write **pseudocodes**, a description of steps in words organized in a program structure.
  - This allows us to ignore the details of implementations.

Programming Design – Algorithms and Recursion 5 / 43 Ling-Chieh Kung (NTU IM)



Algorithms and complexity | Recursion | Searching and sorting

Example: listing all prime numbers

- One pseudocode for listing all prime numbers no greater than n is:

```
Given an integer n:
for i from 2 to n
  assume that i is a prime number
  for j from 2 to i – 1
    if j divides i
      set i to be a composite number
  if i is still considered as prime
    print i
```

Programming Design – Algorithms and Recursion 6 / 43 Ling-Chieh Kung (NTU IM)

In [ ]:

```
#include <iostream>
using namespace std;

int main(){
    int num = 0;
    cout << "Learning Algorithm and Data Structure" << endl;
    cout << "Enter your Integer Number: ";
    cin >> num;

    for (int i = 2; i <= num; i++){
        bool isPrime = true;
        for (int j = 2; j < i; j++){
            if ((i % j) == 0){
                isPrime = false;
                break;
            }
        }
        if (isPrime == true) {
            cout << i << " is prime" << endl;
        }
    }
    return 0;
}
```

In [ ]:

```
#include <iostream>
using namespace std;

bool isPrime(int x);

int main(){
    int num = 0;
    cout << "Learning Algorithm and Data Structure" << endl;
    cout << "Enter your Integer Number: ";
    cin >> num;

    for (int i = 2; i <= num; i++){
        if (isPrime(i) == true) {
            cout << i << " is prime" << endl;
        }
    }
    return 0;
}

bool isPrime(int x) {
    for (int i = 2; i < x; i++){
        if ((x % i) == 0){
            return false;
        }
    }
    return true;
}
```

In [ ]:

```
#include <iostream>
using namespace std;

bool isPrime(int x);

int main(){
    int num = 0;
    cout << "Learning Algorithm and Data Structure" << endl;
    cout << "Enter your Integer Number: ";
    cin >> num;

    for (int i = 2; i <= num; i++){
        if (isPrime(i) == true) {
            cout << i << " is prime" << endl;
        }
    }
    return 0;
}

bool isPrime(int x) {
    for (int i = 2; i * i < x; i++){
        if ((x % i) == 0){
            return false;
        }
    }
    return true;
}
```

In [ ]:

```
#include <iostream>
using namespace std;

const int MAX_LEN = 10000;

void ruleOutPrime(int x, bool isPrime[], int n);

int main(){
    int num = 0;
    cout << "Learning Algorithm and Data Structure" << endl;
    cout << "Enter your Integer Number(2~10000): ";
    cin >> num;

    bool isPrime[MAX_LEN] = {0};

    for (int i = 2; i < num; i++){
        isPrime[i] = true;
    }

    for (int i = 2; i <= num; i++){
        if(isPrime[i] == true){
            cout << i << " is prime" << endl;
            ruleOutPrime(i, isPrime, num);
        }
    }

    return 0;
}

void ruleOutPrime(int x, bool isPrime[], int n){
    for (int i = 1; x * i < n; i++){
        isPrime[x * i] = false;
    }
}
```

Dynamic Programming

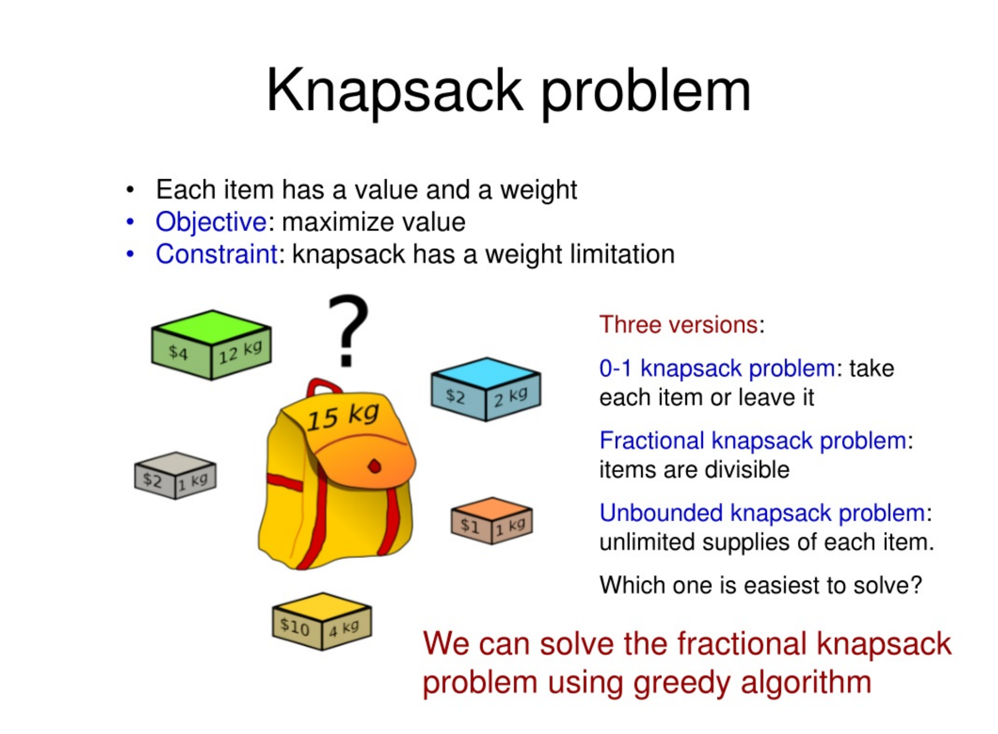
In [ ]:

```
#include <iostream>
#include <stdexcept>
using namespace std;


int fib(int n);


int main(){

    cout << "fib(5): " << fib(5) << endl;

    return 0;
}

int fib(int n){

    if(n < 0)
        throw logic_error("Error!");
    if(n == 0)
        return 0;
    if (n == 1)
        return 1;
    else
        return fib(n-1) + fib(n-2);
}
```

In [ ]:

```
#include <iostream>
#include <stdexcept>
using namespace std;


int fib(int n);


int main(){

    cout << "fib(6): " << fib(6) << endl;

    return 0;
}

int fib(int n){

    int A[20];
    A[0] = 0;
    A[1] = 1;
    for(int i = 2; i <= n; i++){
        A[i] = A[i-1] + A[i-2];
    }
    return A[n];
}
```

> # The Knapsack Problem



Knapsack problem

- Each item has a value and a weight
- Objective: maximize value
- Constraint: knapsack has a weight limitation

Three versions:

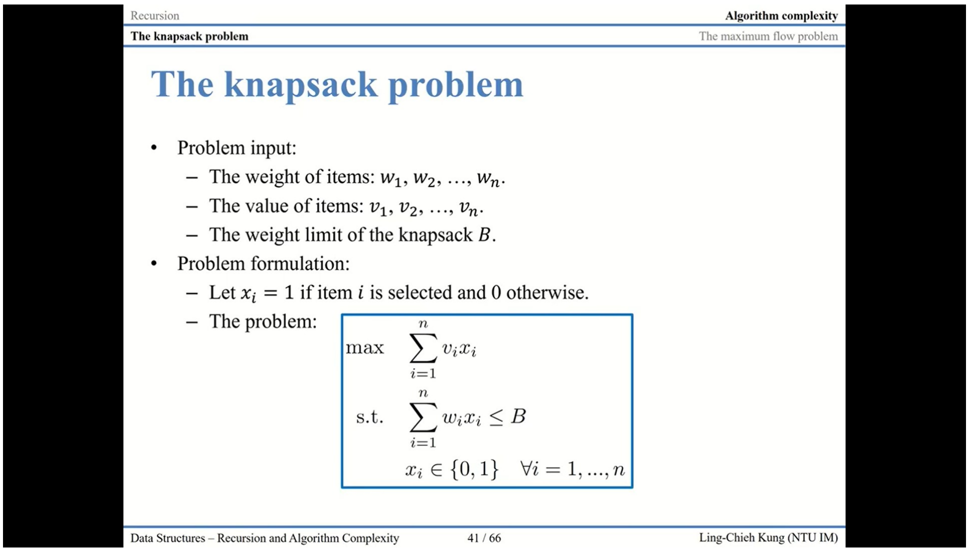
0-1 knapsack problem: take each item or leave it

Fractional knapsack problem: items are divisible

Unbounded knapsack problem: unlimited supplies of each item.

Which one is easiest to solve?

We can solve the fractional knapsack problem using greedy algorithm



Recursion | Algorithm complexity

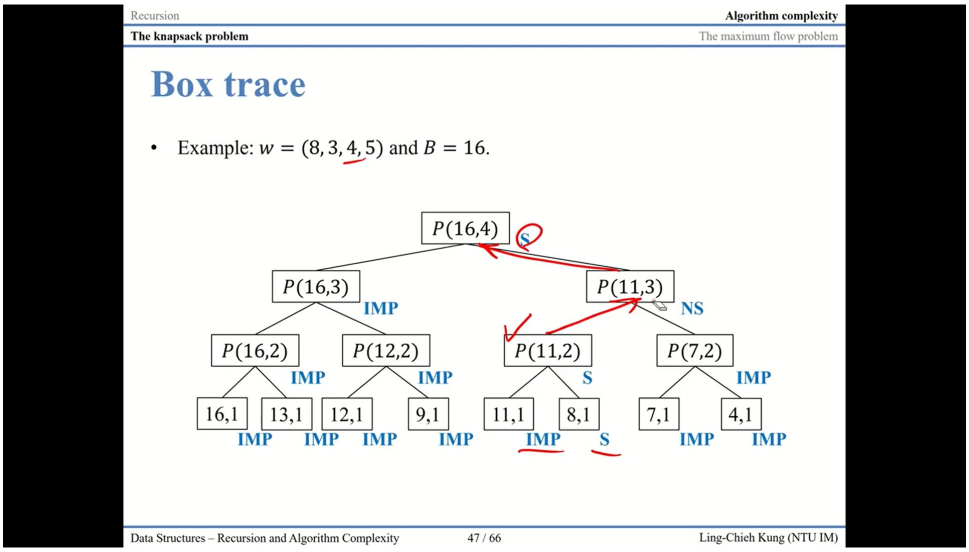
The knapsack problem | The maximum flow problem

The knapsack problem

- Problem input:
  - The weight of items: $w_1, w_2, \ldots, w_n$.
  - The value of items: $v_1, v_2, \ldots, v_n$.
  - The weight limit of the knapsack B.
- Problem formulation:
  - Let $x_i = 1$ if item i is selected and 0 otherwise.
  - The problem:

$$\boxed{\begin{array}{rl} \max & \sum_{i=1}^{n} v_i x_i \\ \text{s.t.} & \sum_{i=1}^{n} w_i x_i \leq B \\ & x_i \in \{0,1\} \quad \forall i = 1, \ldots, n \end{array}}$$

Data Structures – Recursion and Algorithm Complexity 41 / 66 Ling-Chieh Kung (NTU IM)



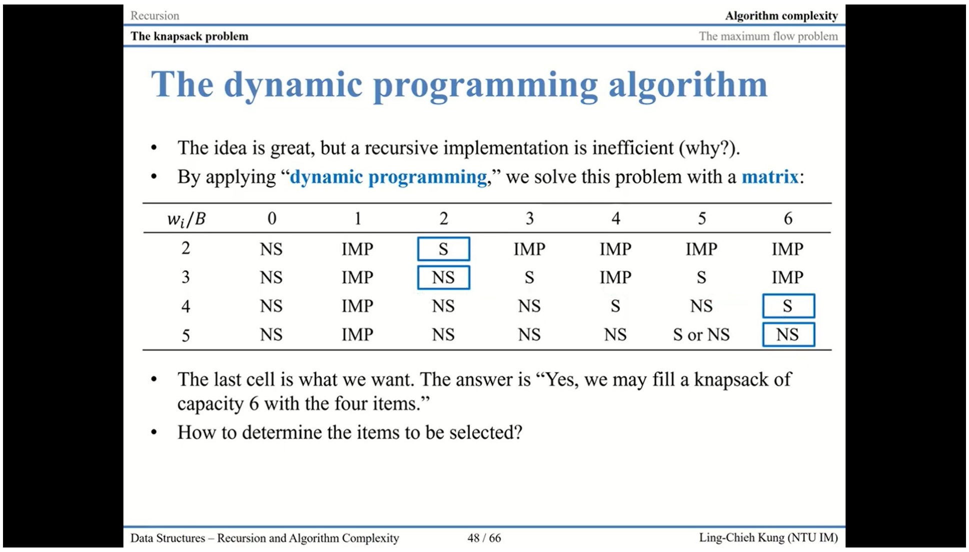
Recursion | Algorithm complexity

The knapsack problem | The maximum flow problem

Box trace

- Example: $w = (8, 3, 4, 5)$ and $B = 16$.

Data Structures – Recursion and Algorithm Complexity 47 / 66 Ling-Chieh Kung (NTU IM)



Recursion | Algorithm complexity
The knapsack problem | The maximum flow problem

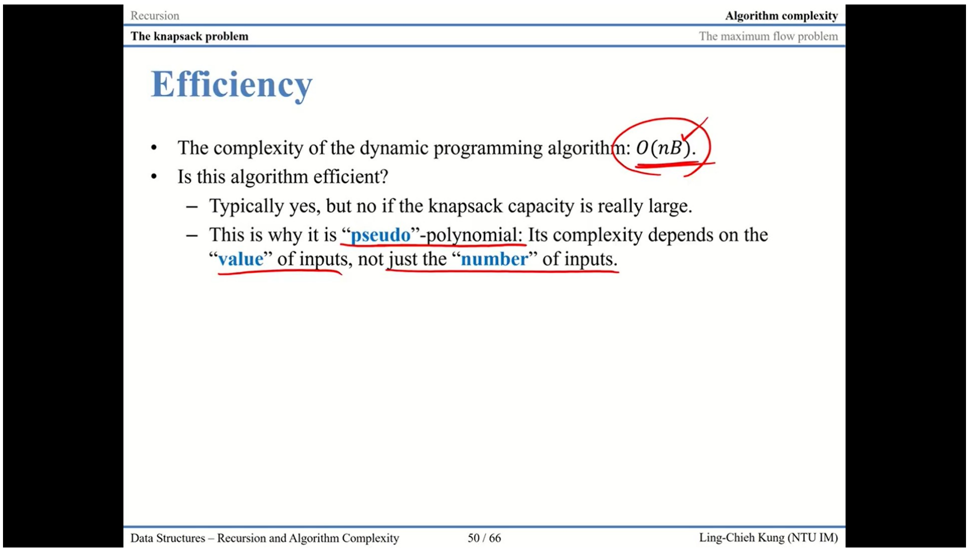
The dynamic programming algorithm

- The idea is great, but a recursive implementation is inefficient (why?).
- By applying "**dynamic programming**," we solve this problem with a **matrix**:

w_i/B	0	1	2	3	4	5	6
2	NS	IMP	S	IMP	IMP	IMP	IMP
3	NS	IMP	NS	S	IMP	S	IMP
4	NS	IMP	NS	NS	S	NS	S
5	NS	IMP	NS	NS	NS	S or NS	NS

- The last cell is what we want. The answer is "Yes, we may fill a knapsack of capacity 6 with the four items."
- How to determine the items to be selected?

Data Structures – Recursion and Algorithm Complexity 48 / 66 Ling-Chieh Kung (NTU IM)



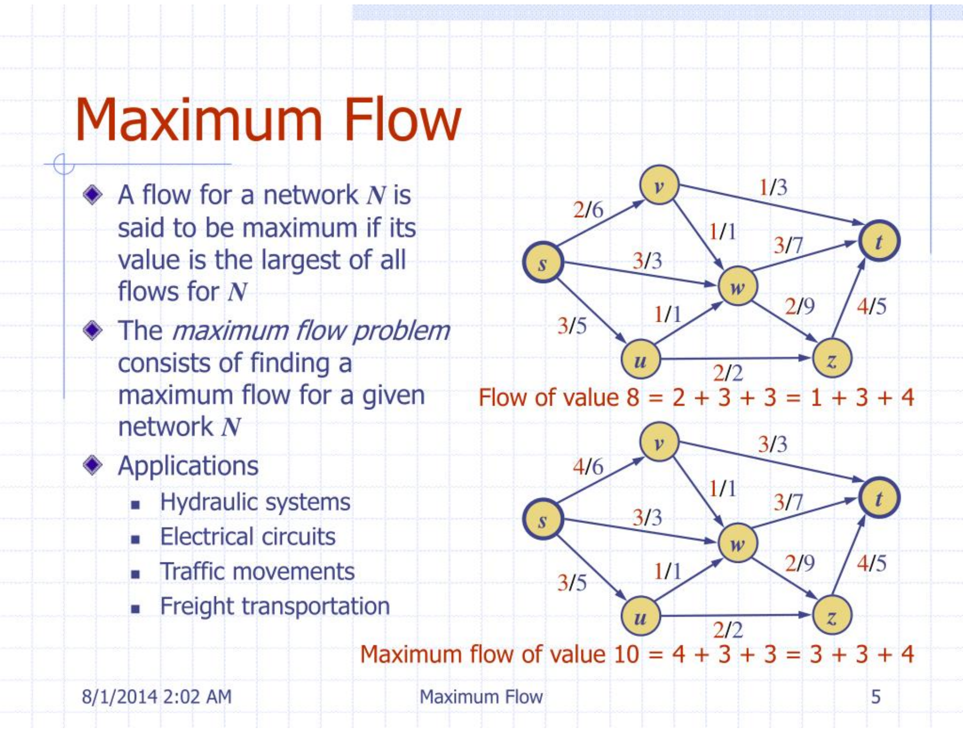
Recursion | Algorithm complexity
The knapsack problem | The maximum flow problem

Efficiency

- The complexity of the dynamic programming algorithm: $O(nB)$.
- Is this algorithm efficient?
  - Typically yes, but no if the knapsack capacity is really large.
  - This is why it is "**pseudo**"-polynomial: Its complexity depends on the "**value**" of inputs, not just the "**number**" of inputs.

Data Structures – Recursion and Algorithm Complexity 50 / 66 Ling-Chieh Kung (NTU IM)

The Max Flow Problem



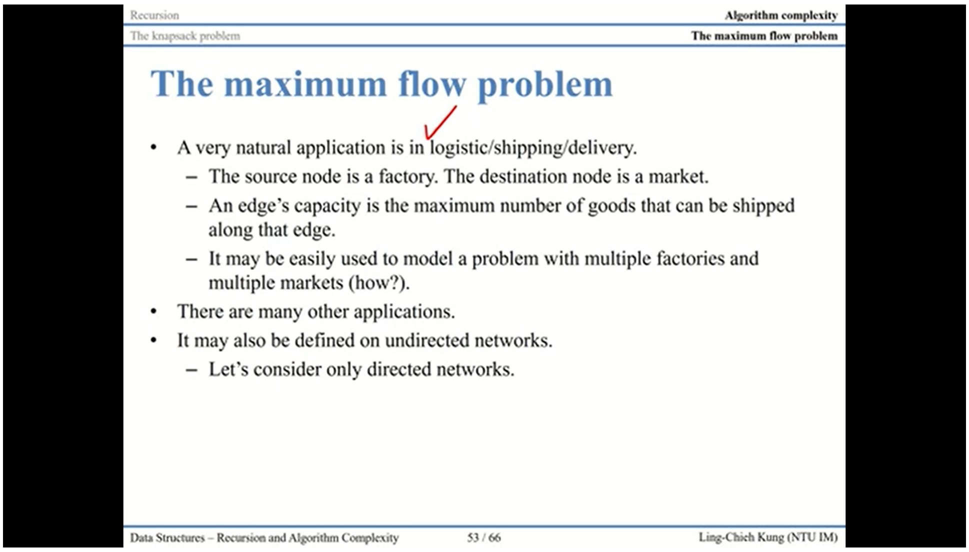
Maximum Flow

- A flow for a network N is said to be maximum if its value is the largest of all flows for N
- The *maximum flow problem* consists of finding a maximum flow for a given network N
- Applications
  - Hydraulic systems
  - Electrical circuits
  - Traffic movements
  - Freight transportation

Flow of value 8 = 2 + 3 + 3 = 1 + 3 + 4

Maximum flow of value 10 = 4 + 3 + 3 = 3 + 3 + 4

8/1/2014 2:02 AM Maximum Flow 5



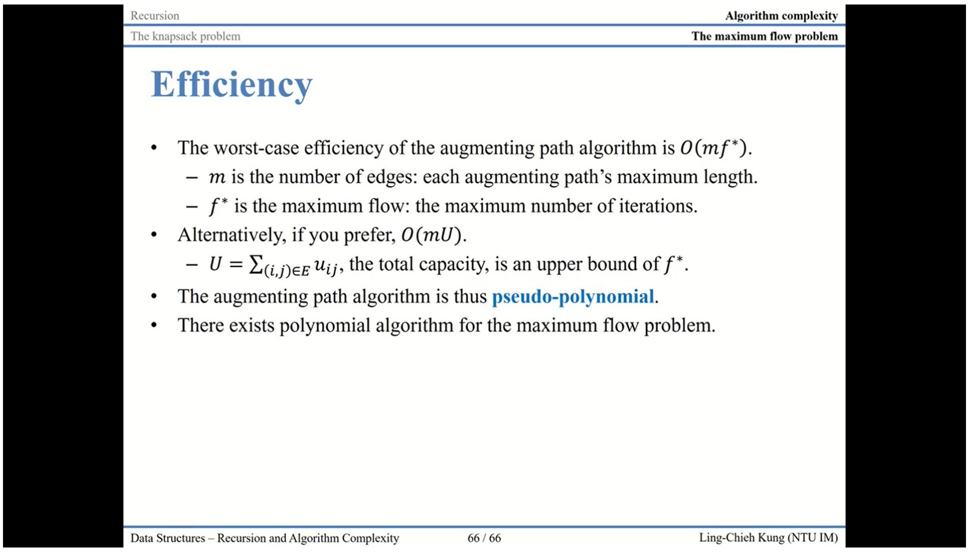
Recursion | Algorithm complexity

The knapsack problem | The maximum flow problem

The maximum flow problem

- A very natural application is in ✓ logistic/shipping/delivery.
  - The source node is a factory. The destination node is a market.
  - An edge's capacity is the maximum number of goods that can be shipped along that edge.
  - It may be easily used to model a problem with multiple factories and multiple markets (how?).
- There are many other applications.
- It may also be defined on undirected networks.
  - Let's consider only directed networks.

Data Structures – Recursion and Algorithm Complexity 53 / 66 Ling-Chieh Kung (NTU IM)



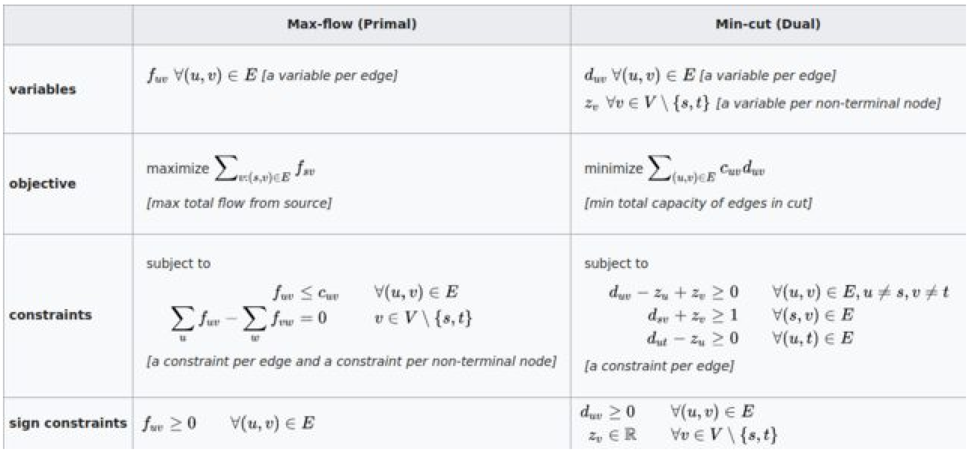
Recursion | Algorithm complexity

The knapsack problem | The maximum flow problem

Efficiency

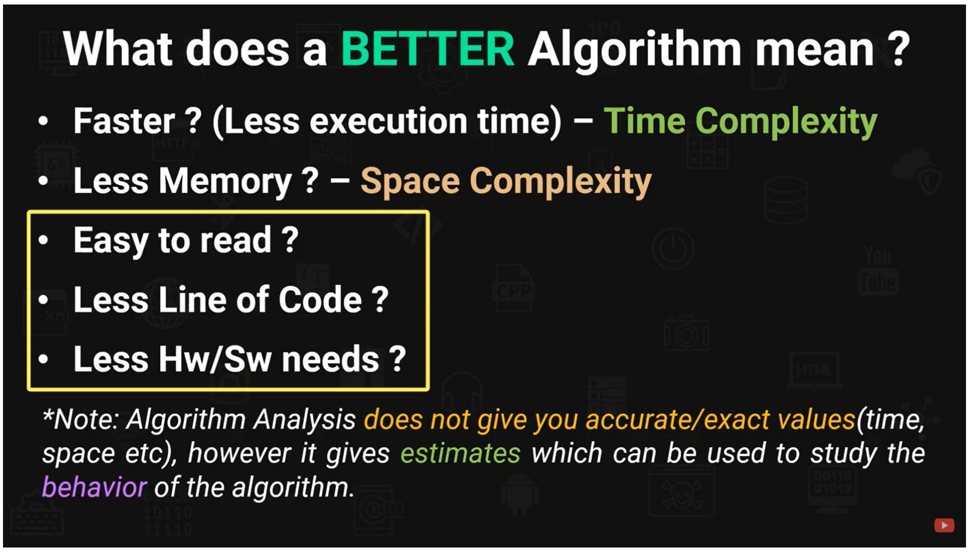
- The worst-case efficiency of the augmenting path algorithm is $O(mf^*)$.
  - m is the number of edges: each augmenting path's maximum length.
  - f^* is the maximum flow: the maximum number of iterations.
- Alternatively, if you prefer, $O(mU)$.
  - $U = \sum_{(i,j)\in E} u_{ij}$, the total capacity, is an upper bound of f^*.
- The augmenting path algorithm is thus **pseudo-polynomial**.
- There exists polynomial algorithm for the maximum flow problem.

Data Structures – Recursion and Algorithm Complexity 66 / 66 Ling-Chieh Kung (NTU IM)



	Max-flow (Primal)	Min-cut (Dual)
variables	$f_{uv}\ \forall(u,v)\in E$ *[a variable per edge]*	$d_{uv}\ \forall(u,v)\in E$ *[a variable per edge]* $z_v\ \forall v\in V\setminus\{s,t\}$ *[a variable per non-terminal node]*
objective	maximize $\sum_{v:(s,v)\in E} f_{sv}$ *[max total flow from source]*	minimize $\sum_{(u,v)\in E} c_{uv}d_{uv}$ *[min total capacity of edges in cut]*
constraints	subject to $f_{uv}\le c_{uv} \quad \forall(u,v)\in E$ $\sum_u f_{uv}-\sum_w f_{vw}=0 \quad v\in V\setminus\{s,t\}$ *[a constraint per edge and a constraint per non-terminal node]*	subject to $d_{uv}-z_u+z_v\ge 0 \quad \forall(u,v)\in E, u\ne s, v\ne t$ $d_{sv}+z_v\ge 1 \quad \forall(s,v)\in E$ $d_{ut}-z_u\ge 0 \quad \forall(u,t)\in E$ *[a constraint per edge]*
sign constraints	$f_{uv}\ge 0 \quad \forall(u,v)\in E$	$d_{uv}\ge 0 \quad \forall(u,v)\in E$ $z_v\in\mathbb{R} \quad \forall v\in V\setminus\{s,t\}$

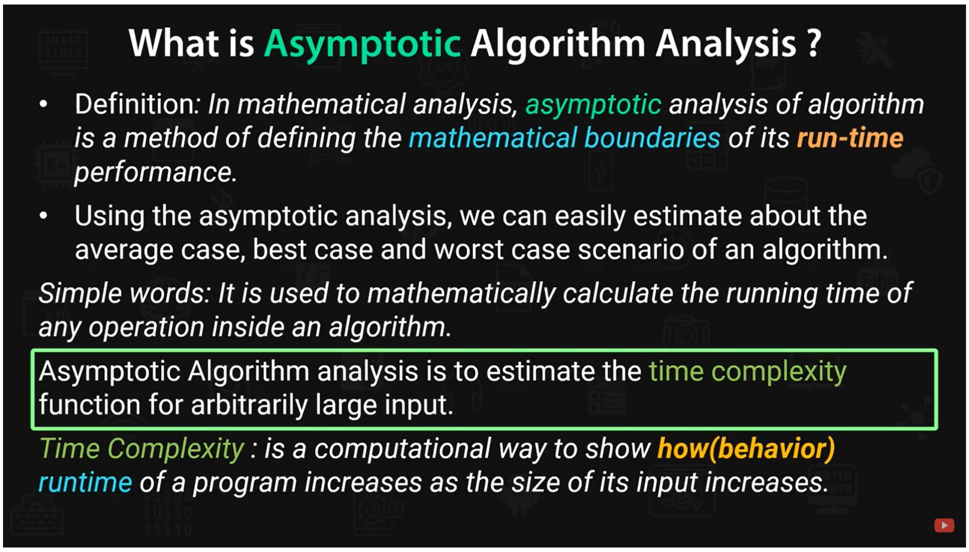
C⁺⁺ Performance Evaluation - Space Complexity and Time Complexity



What does a BETTER Algorithm mean ?

- Faster ? (Less execution time) – Time Complexity
- Less Memory ? – Space Complexity
- Easy to read ?
- Less Line of Code ?
- Less Hw/Sw needs ?

**Note: Algorithm Analysis does not give you accurate/exact values(time, space etc), however it gives estimates which can be used to study the behavior of the algorithm.*



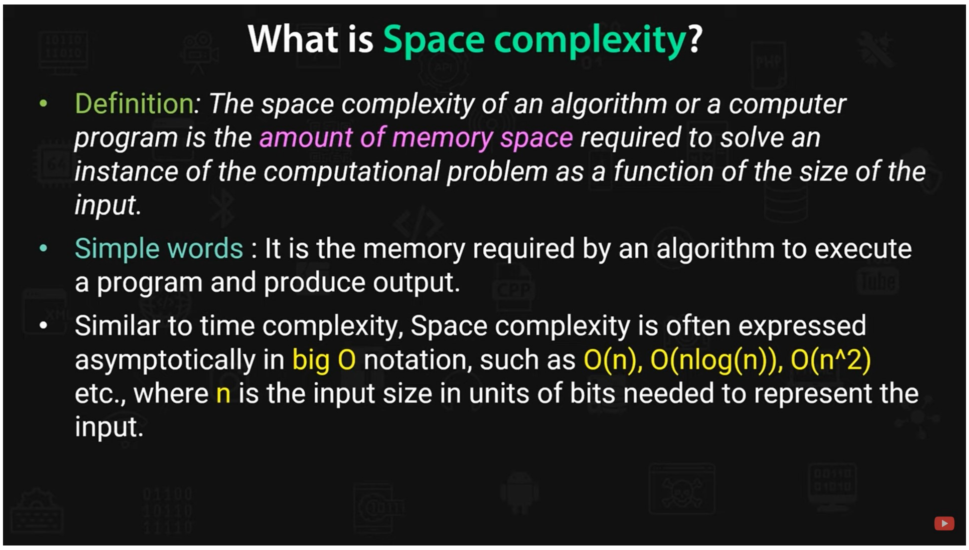
What is Asymptotic Algorithm Analysis ?

- Definition: *In mathematical analysis, asymptotic analysis of algorithm is a method of defining the mathematical boundaries of its* ***run-time*** *performance.*
- Using the asymptotic analysis, we can easily estimate about the average case, best case and worst case scenario of an algorithm.

Simple words: It is used to mathematically calculate the running time of any operation inside an algorithm.

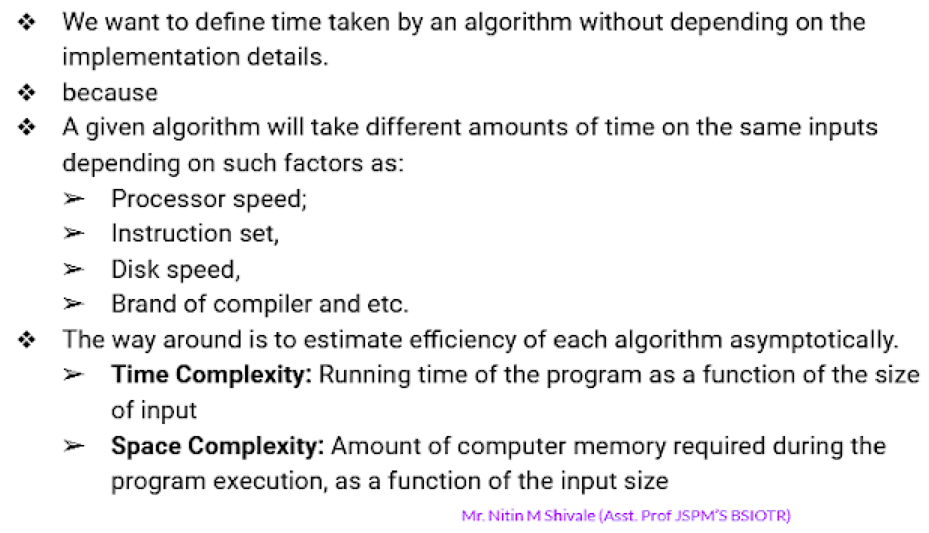
Asymptotic Algorithm analysis is to estimate the time complexity function for arbitrarily large input.

Time Complexity : is a computational way to show ***how(behavior)*** *runtime of a program increases as the size of its input increases.*



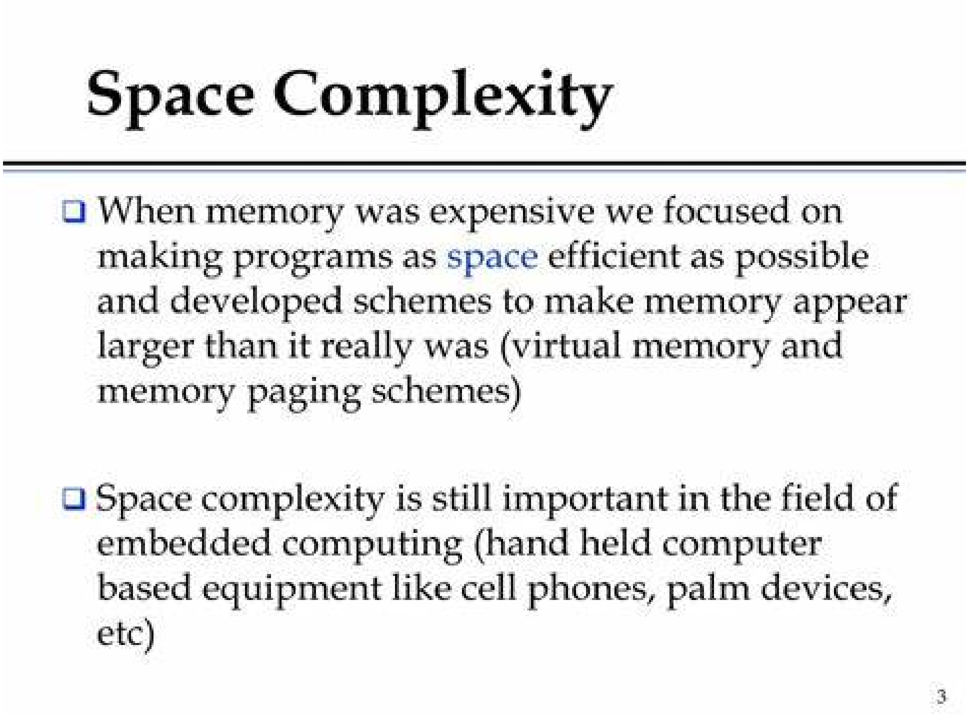
What is Space complexity?

- Definition: *The space complexity of an algorithm or a computer program is the amount of memory space required to solve an instance of the computational problem as a function of the size of the input.*
- Simple words : It is the memory required by an algorithm to execute a program and produce output.
- Similar to time complexity, Space complexity is often expressed asymptotically in big O notation, such as O(n), O(nlog(n)), O(n^2) etc., where n is the input size in units of bits needed to represent the input.



- ❖ We want to define time taken by an algorithm without depending on the implementation details.
- ❖ because
- ❖ A given algorithm will take different amounts of time on the same inputs depending on such factors as:
  - ➢ Processor speed;
  - ➢ Instruction set,
  - ➢ Disk speed,
  - ➢ Brand of compiler and etc.
- ❖ The way around is to estimate efficiency of each algorithm asymptotically.
  - ➢ **Time Complexity:** Running time of the program as a function of the size of input
  - ➢ **Space Complexity:** Amount of computer memory required during the program execution, as a function of the input size

Mr. Nitin M Shivale (Asst. Prof JSPM'S BSIOTR)

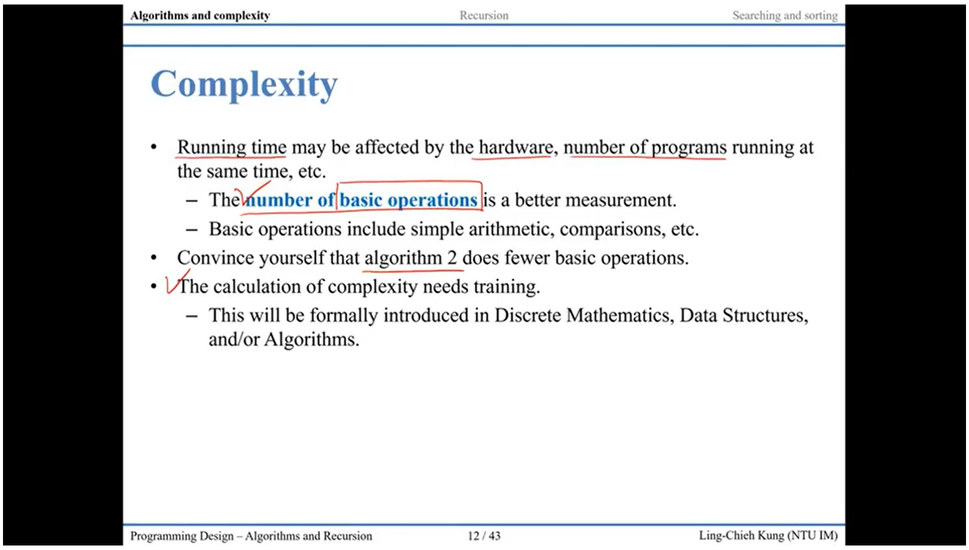


Space Complexity

- ❑ When memory was expensive we focused on making programs as space efficient as possible and developed schemes to make memory appear larger than it really was (virtual memory and memory paging schemes)

- ❑ Space complexity is still important in the field of embedded computing (hand held computer based equipment like cell phones, palm devices, etc)

3

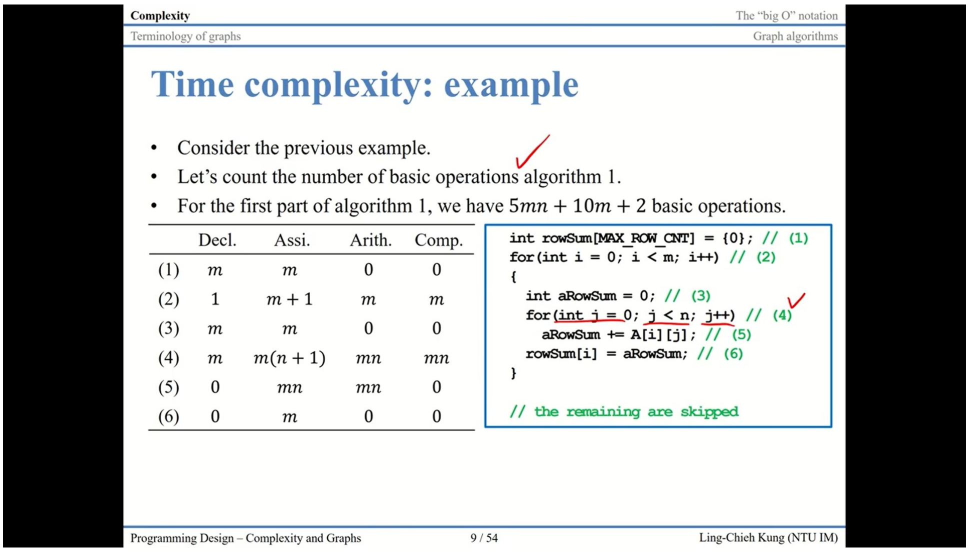


Algorithms and complexity Recursion Searching and sorting

Complexity

- Running time may be affected by the hardware, number of programs running at the same time, etc.
  - – The number of basic operations is a better measurement.
  - – Basic operations include simple arithmetic, comparisons, etc.
- Convince yourself that algorithm 2 does fewer basic operations.
- The calculation of complexity needs training.
  - – This will be formally introduced in Discrete Mathematics, Data Structures, and/or Algorithms.

Programming Design – Algorithms and Recursion 12 / 43 Ling-Chieh Kung (NTU IM)

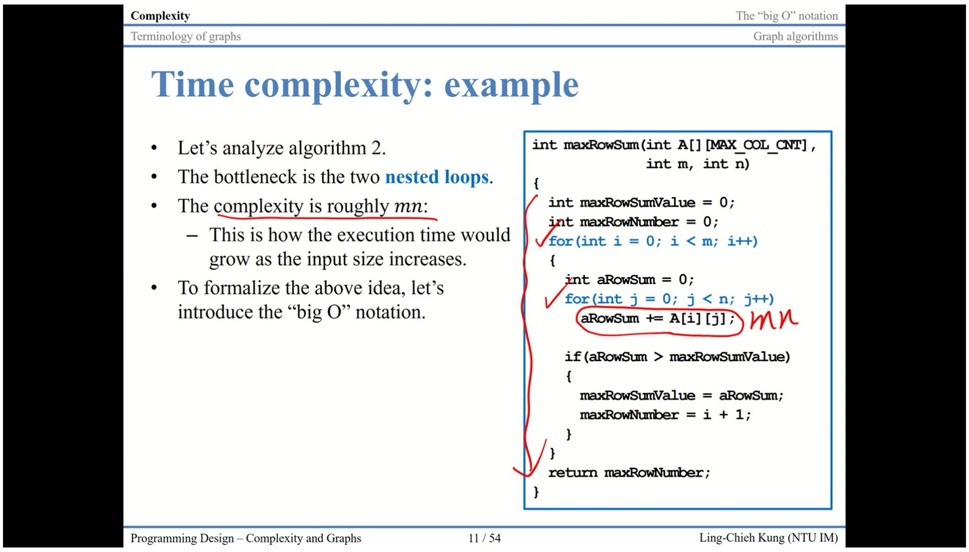


Time complexity: example

- Consider the previous example.
- Let's count the number of basic operations algorithm 1.
- For the first part of algorithm 1, we have $5mn + 10m + 2$ basic operations.

	Decl.	Assi.	Arith.	Comp.
(1)	m	m	0	0
(2)	1	$m+1$	m	m
(3)	m	m	0	0
(4)	m	$m(n+1)$	mn	mn
(5)	0	mn	mn	0
(6)	0	m	0	0

```
int rowSum[MAX_ROW_CNT] = {0}; // (1)
for(int i = 0; i < m; i++) // (2)
{
  int aRowSum = 0; // (3)
  for(int j = 0; j < n; j++) // (4)
    aRowSum += A[i][j]; // (5)
  rowSum[i] = aRowSum; // (6)
}

// the remaining are skipped
```

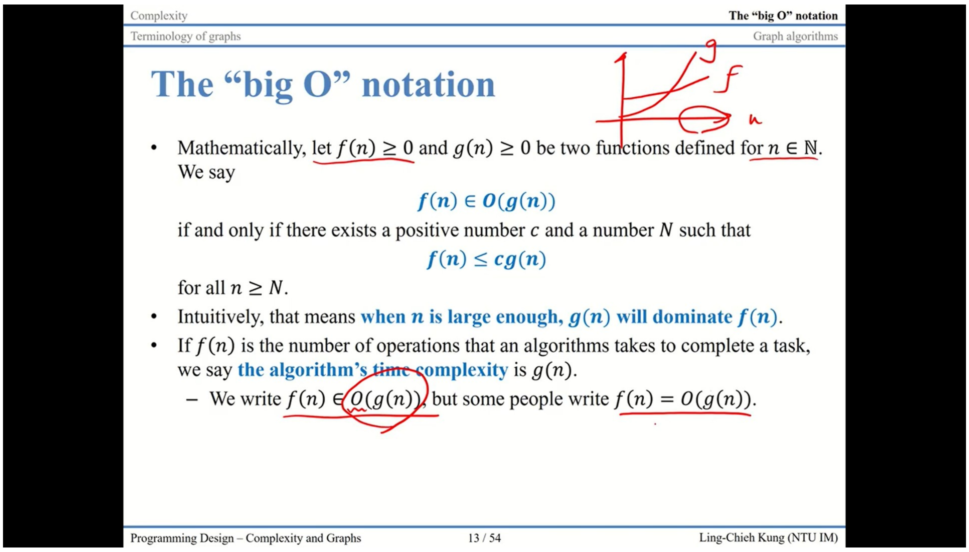


Time complexity: example

- Let's analyze algorithm 2.
- The bottleneck is the two **nested loops**.
- The complexity is roughly mn:
  - This is how the execution time would grow as the input size increases.
- To formalize the above idea, let's introduce the "big O" notation.

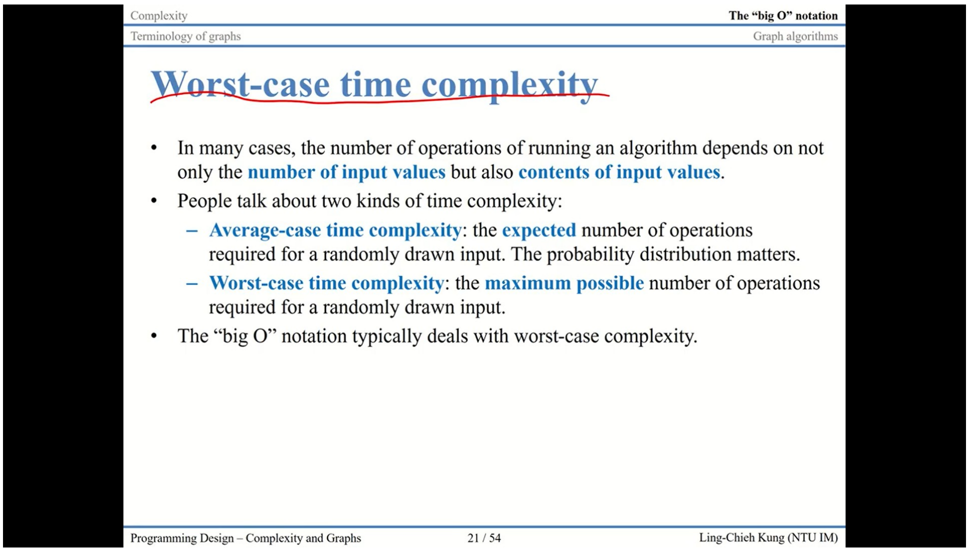
```
int maxRowSum(int A[][MAX_COL_CNT],
              int m, int n)
{
  int maxRowSumValue = 0;
  int maxRowNumber = 0;
  for(int i = 0; i < m; i++)
  {
    int aRowSum = 0;
    for(int j = 0; j < n; j++)
      aRowSum += A[i][j]; mn

    if(aRowSum > maxRowSumValue)
    {
      maxRowSumValue = aRowSum;
      maxRowNumber = i + 1;
    }
  }
  return maxRowNumber;
}
```

Big "O" Notation



The "big O" notation

- Mathematically, let $f(n) \geq 0$ and $g(n) \geq 0$ be two functions defined for $n \in \mathbb{N}$. We say
  $$f(n) \in O(g(n))$$
  if and only if there exists a positive number c and a number N such that
  $$f(n) \leq cg(n)$$
  for all $n \geq N$.
- Intuitively, that means **when n is large enough, $g(n)$ will dominate $f(n)$**.
- If $f(n)$ is the number of operations that an algorithms takes to complete a task, we say **the algorithm's time complexity** is $g(n)$.
  - We write $f(n) \in O(g(n))$, but some people write $f(n) = O(g(n))$.

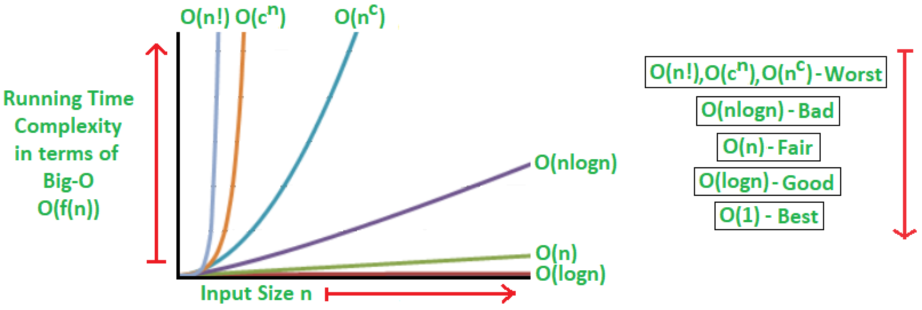


Complexity | The "big O" notation
Terminology of graphs | Graph algorithms

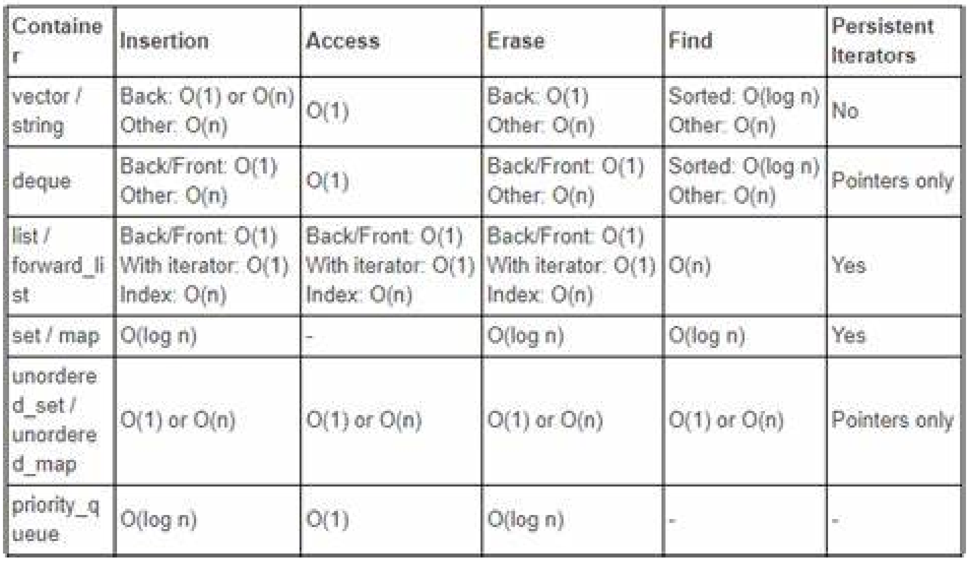
Worst-case time complexity

- In many cases, the number of operations of running an algorithm depends on not only the **number of input values** but also **contents of input values**.
- People talk about two kinds of time complexity:
  - **Average-case time complexity**: the **expected** number of operations required for a randomly drawn input. The probability distribution matters.
  - **Worst-case time complexity**: the **maximum possible** number of operations required for a randomly drawn input.
- The "big O" notation typically deals with worst-case complexity.

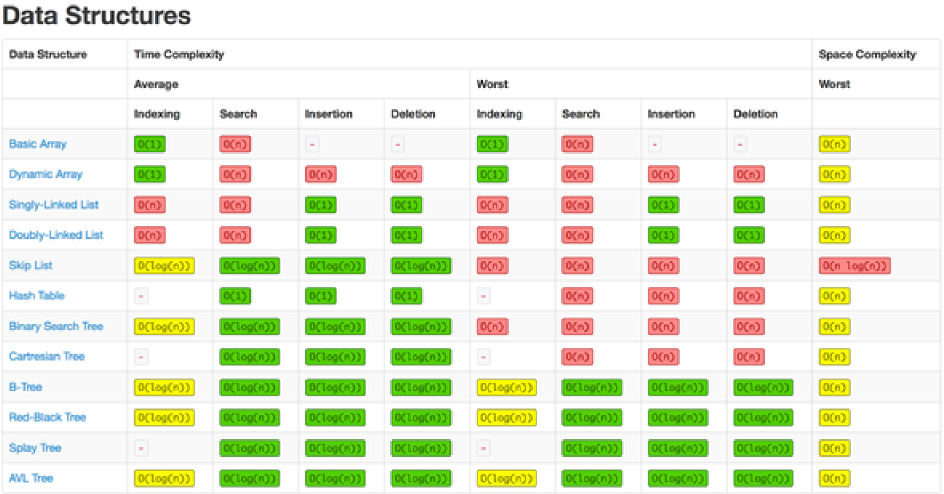
Programming Design – Complexity and Graphs 21 / 54 Ling-Chieh Kung (NTU IM)



$O(n!)$,$O(c^n)$,$O(n^c)$ - Worst
O(nlogn) - Bad
O(n) - Fair
O(logn) - Good
O(1) - Best



Container	Insertion	Access	Erase	Find	Persistent Iterators
vector / string	Back: O(1) or O(n) Other: O(n)	O(1)	Back: O(1) Other: O(n)	Sorted: O(log n) Other: O(n)	No
deque	Back/Front: O(1) Other: O(n)	O(1)	Back/Front: O(1) Other: O(n)	Sorted: O(log n) Other: O(n)	Pointers only
list / forward_list	Back/Front: O(1) With iterator: O(1) Index: O(n)	Back/Front: O(1) With iterator: O(1) Index: O(n)	Back/Front: O(1) With iterator: O(1) Index: O(n)	O(n)	Yes
set / map	O(log n)	-	O(log n)	O(log n)	Yes
unordered_set / unordered_map	O(1) or O(n)	O(1) or O(n)	O(1) or O(n)	O(1) or O(n)	Pointers only
priority_queue	O(log n)	O(1)	O(log n)	-	-



Data Structures

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Indexing	Search	Insertion	Deletion	Indexing	Search	Insertion	Deletion	
Basic Array	O(1)	O(n)	-	-	O(1)	O(n)	-	-	O(n)
Dynamic Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Singly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	-	O(1)	O(1)	O(1)	-	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	-	O(log(n))	O(log(n))	O(log(n))	-	O(n)	O(n)	O(n)	O(n)
B-Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	-	O(log(n))	O(log(n))	O(log(n))	-	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)

C^{++} Type of Data Structure

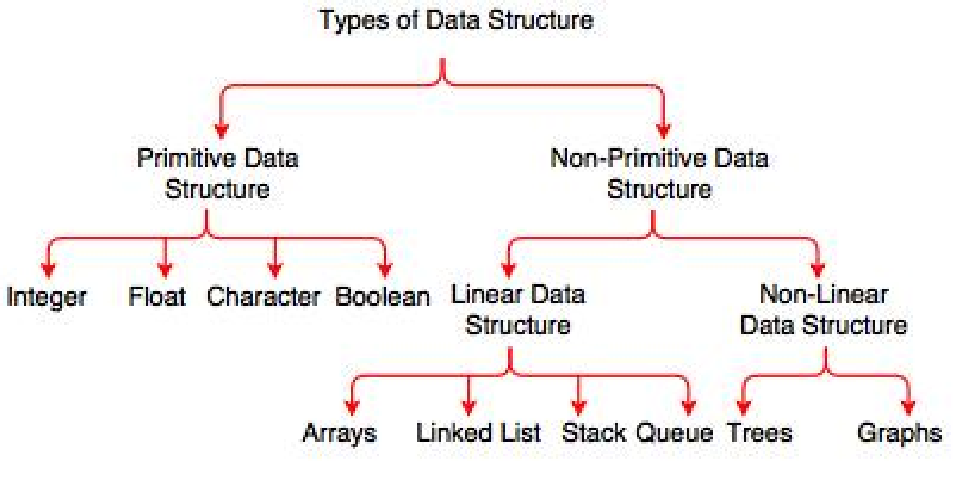
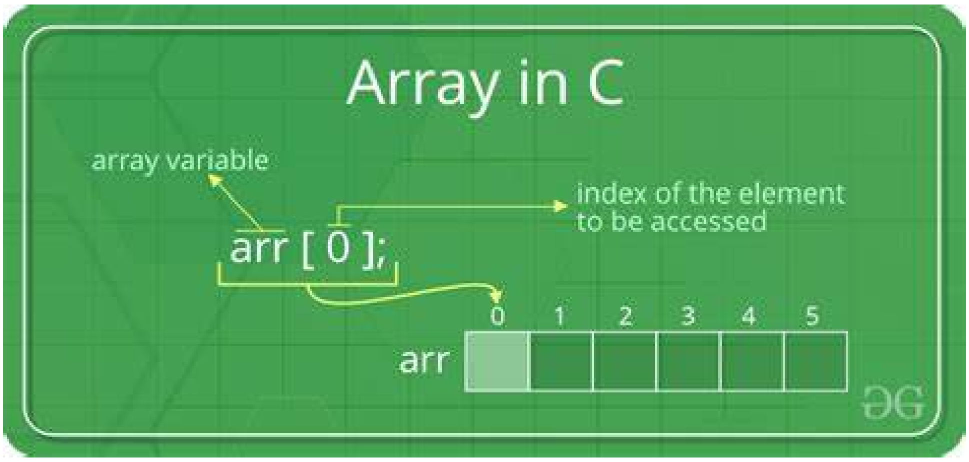
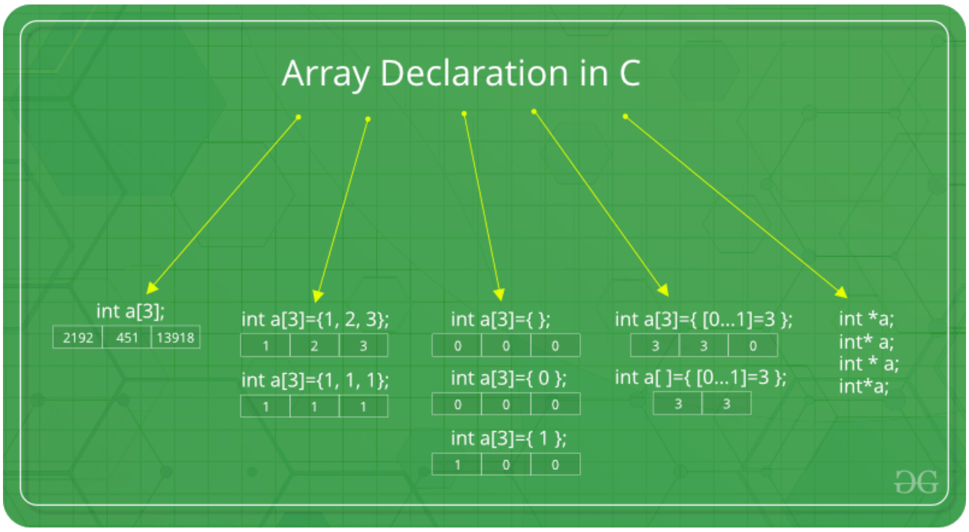


Fig. Types of Data Structure

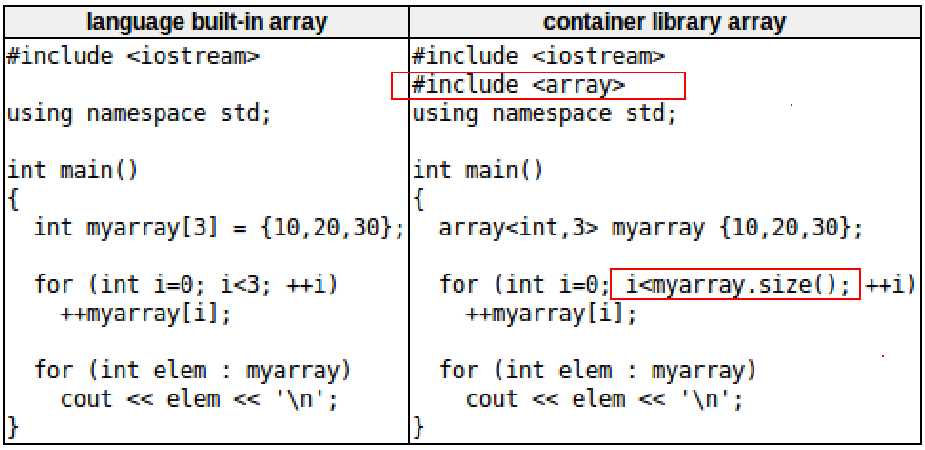
Linear Data Structue

Array





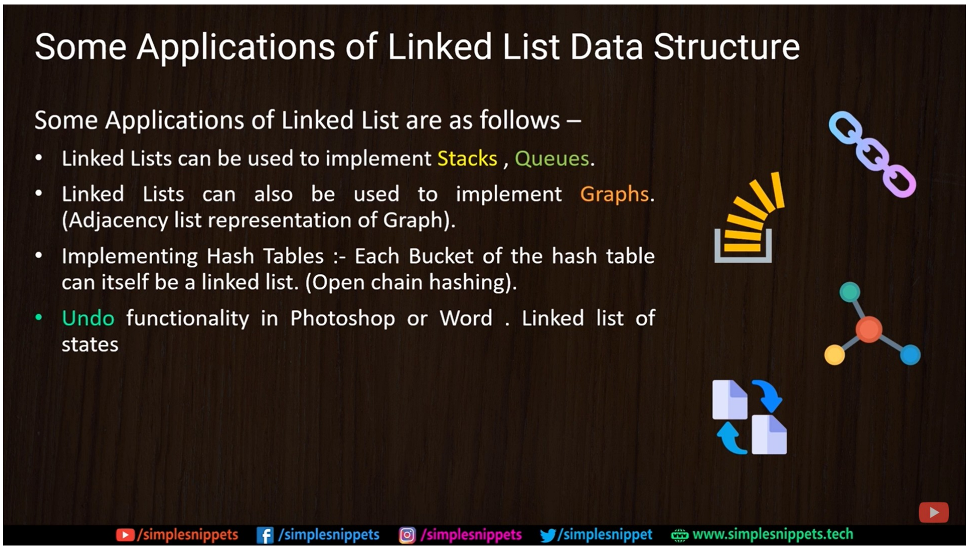
Operations in Array



language built-in array	container library array

`#include <iostream>`
 	
`using namespace std;`
 	
`int main()`
`{`
`  int myarray[3] = {10,20,30};`
 	
`  for (int i=0; i<3; ++i)`
`    ++myarray[i];`
 	
`  for (int elem : myarray)`
`    cout << elem << '\n';`
`}`
	`#include <iostream>`
	`#include <array>`
	`using namespace std;`
	
	`int main()`
	`{`
	`  array<int,3> myarray {10,20,30};`
	
	`  for (int i=0; i<myarray.size(); ++i)`
	`    ++myarray[i];`
	
	`  for (int elem : myarray)`
	`    cout << elem << '\n';`
	`}`

Linked List



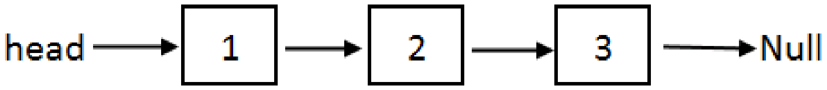
Some Applications of Linked List Data Structure

Some Applications of Linked List are as follows –

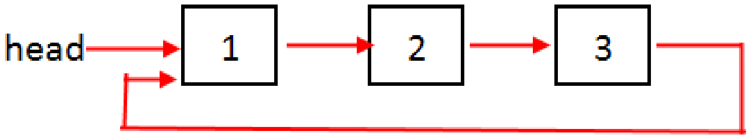
- Linked Lists can be used to implement Stacks , Queues.
- Linked Lists can also be used to implement Graphs. (Adjacency list representation of Graph).
- Implementing Hash Tables :- Each Bucket of the hash table can itself be a linked list. (Open chain hashing).
- Undo functionality in Photoshop or Word . Linked list of states

/simplesnippets /simplesnippets /simplesnippets /simplesnippet www.simplesnippets.tech

CLL

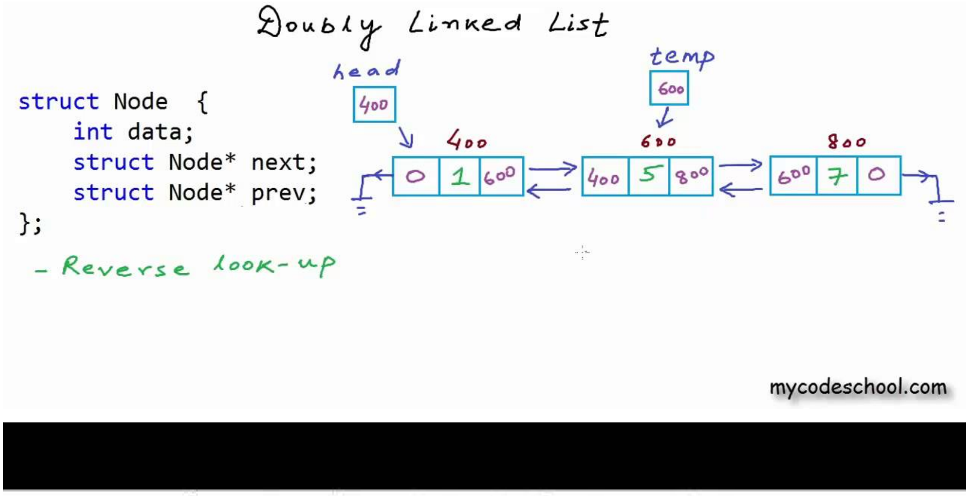


Singly Linked List



Circular Linked List

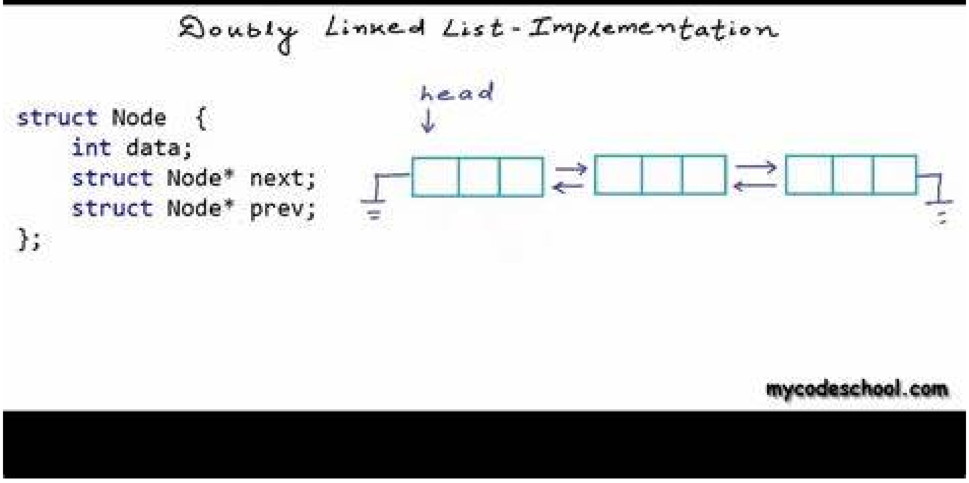
DLL



Doubly Linked List

```
struct Node  {
    int data;
    struct Node* next;
    struct Node* prev;
};
```

- Reverse look-up

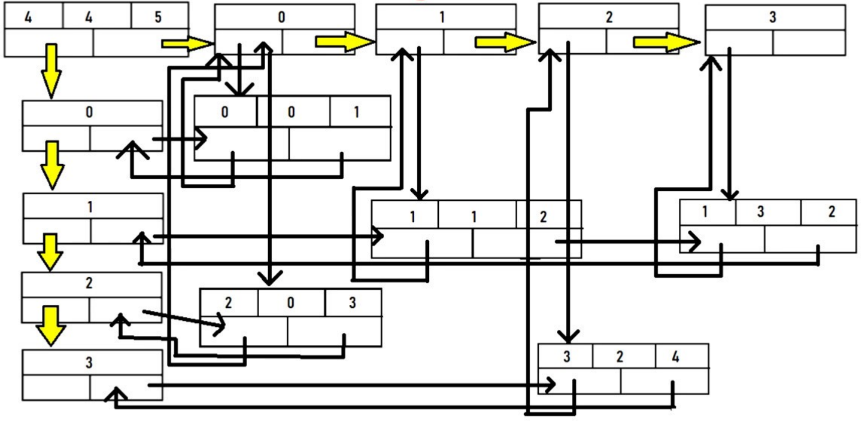


Doubly Linked List - Implementation

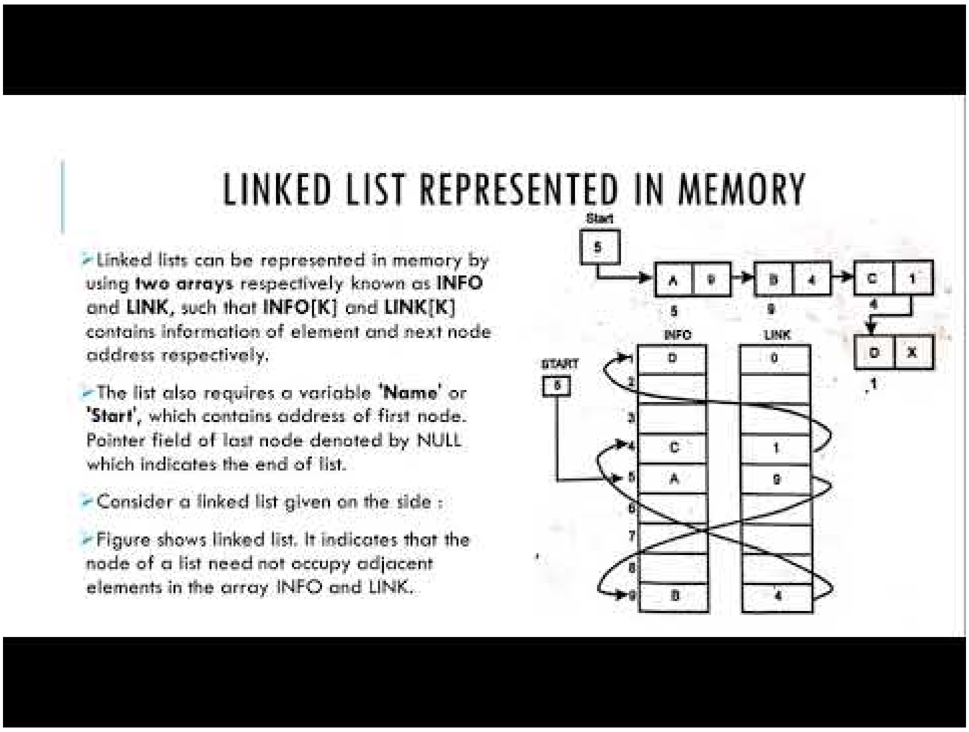
```
struct Node  {
    int data;
    struct Node* next;
    struct Node* prev;
};
```

Matrix

This is the final representation of sparse matrix using linked list

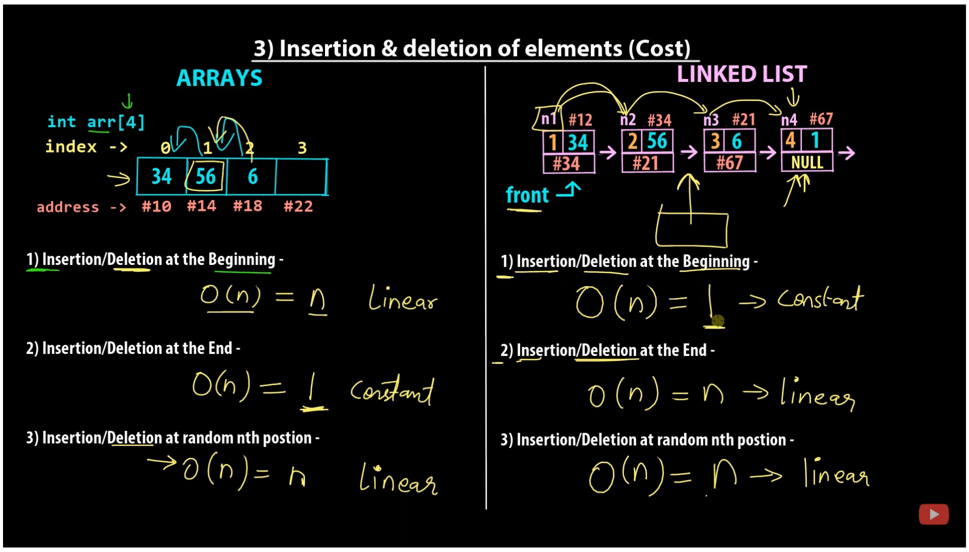


Linked List in Memory



LINKED LIST REPRESENTED IN MEMORY

- Linked lists can be represented in memory by using **two arrays** respectively known as **INFO** and **LINK**, such that **INFO[K]** and **LINK[K]** contains information of element and next node address respectively.
- The list also requires a variable **'Name'** or **'Start'**, which contains address of first node. Pointer field of last node denoted by NULL which indicates the end of list.
- Consider a linked list given on the side :
- Figure shows linked list. It indicates that the node of a list need not occupy adjacent elements in the array INFO and LINK.



3) Insertion & deletion of elements (Cost)

ARRAYS

```
int arr[4]
index ->   0   1   2   3
         | 34 | 56 | 6 |   |
address -> #10 #14 #18 #22
```

1) Insertion/Deletion at the Beginning -

$O(n) = n$ Linear

2) Insertion/Deletion at the End -

$O(n) = 1$ Constant

3) Insertion/Deletion at random nth postion -

$\rightarrow O(n) = n$ Linear

LINKED LIST

1) Insertion/Deletion at the Beginning -

$O(n) = 1 \rightarrow$ Constant

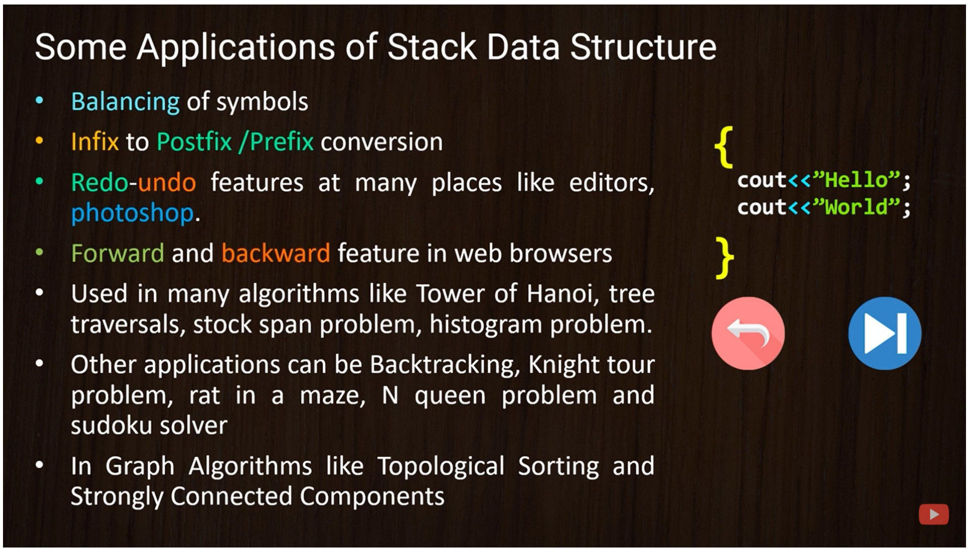
2) Insertion/Deletion at the End -

$O(n) = n \rightarrow$ linear

3) Insertion/Deletion at random nth postion -

$O(n) = n \rightarrow$ linear

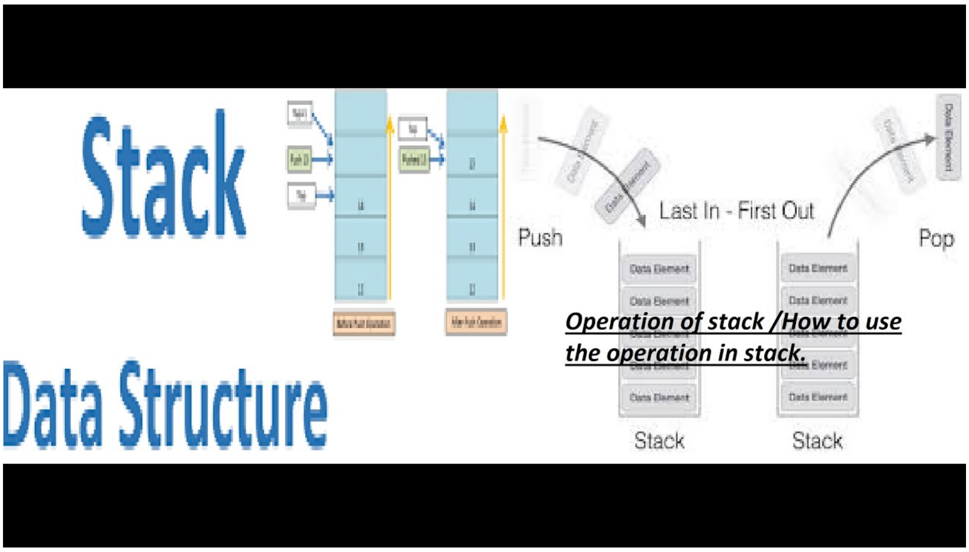
Stack



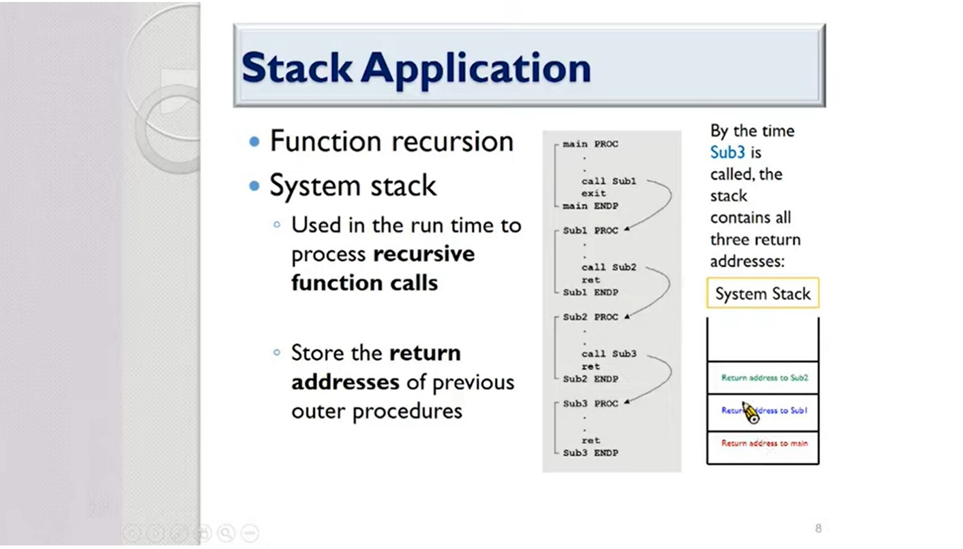
Some Applications of Stack Data Structure

- Balancing of symbols
- Infix to Postfix /Prefix conversion
- Redo-undo features at many places like editors, photoshop.
- Forward and backward feature in web browsers
- Used in many algorithms like Tower of Hanoi, tree traversals, stock span problem, histogram problem.
- Other applications can be Backtracking, Knight tour problem, rat in a maze, N queen problem and sudoku solver
- In Graph Algorithms like Topological Sorting and Strongly Connected Components

```
{
  cout<<"Hello";
  cout<<"World";
}
```



Operation of stack /How to use the operation in stack.



Stack Application

- Function recursion
- System stack
  - Used in the run time to process **recursive function calls**
  - Store the **return addresses** of previous outer procedures

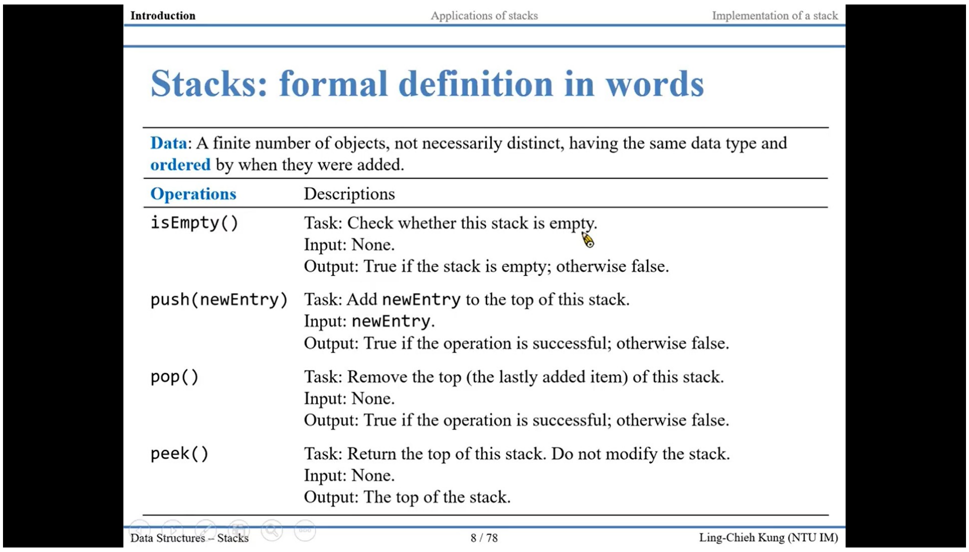
```
main PROC
   .
   .
   call Sub1
   exit
main ENDP

Sub1 PROC
   .
   .
   call Sub2
   ret
Sub1 ENDP

Sub2 PROC
   .
   .
   call Sub3
   ret
Sub2 ENDP

Sub3 PROC
   .
   .
   ret
Sub3 ENDP
```

By the time Sub3 is called, the stack contains all three return addresses:

System Stack

Return address to Sub2
Return address to Sub1
Return address to main

8



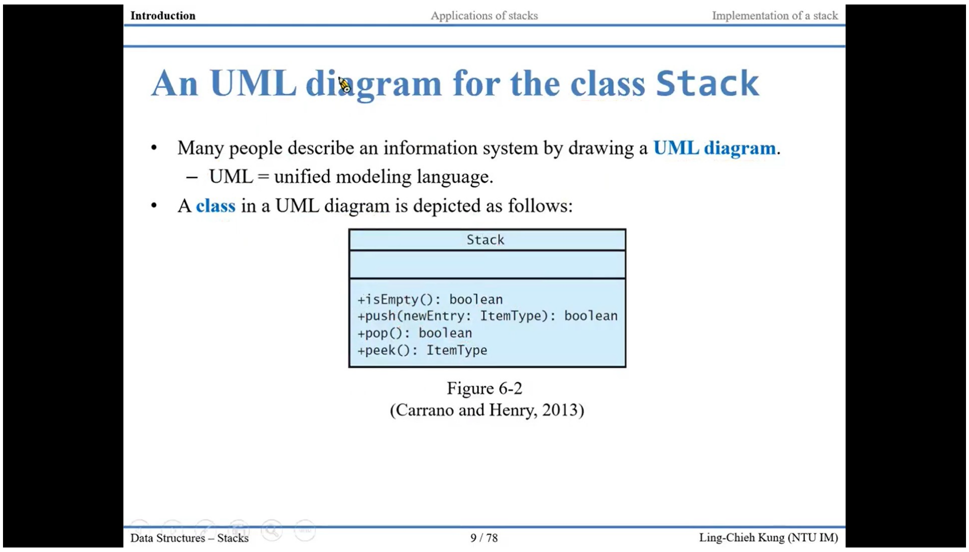
Introduction Applications of stacks Implementation of a stack

Stacks: formal definition in words

Data: A finite number of objects, not necessarily distinct, having the same data type and **ordered** by when they were added.

Operations	Descriptions
`isEmpty()`	Task: Check whether this stack is empty. Input: None. Output: True if the stack is empty; otherwise false.
`push(newEntry)`	Task: Add `newEntry` to the top of this stack. Input: `newEntry`. Output: True if the operation is successful; otherwise false.
`pop()`	Task: Remove the top (the lastly added item) of this stack. Input: None. Output: True if the operation is successful; otherwise false.
`peek()`	Task: Return the top of this stack. Do not modify the stack. Input: None. Output: The top of the stack.

Data Structures – Stacks 8 / 78 Ling-Chieh Kung (NTU IM)



Introduction Applications of stacks Implementation of a stack

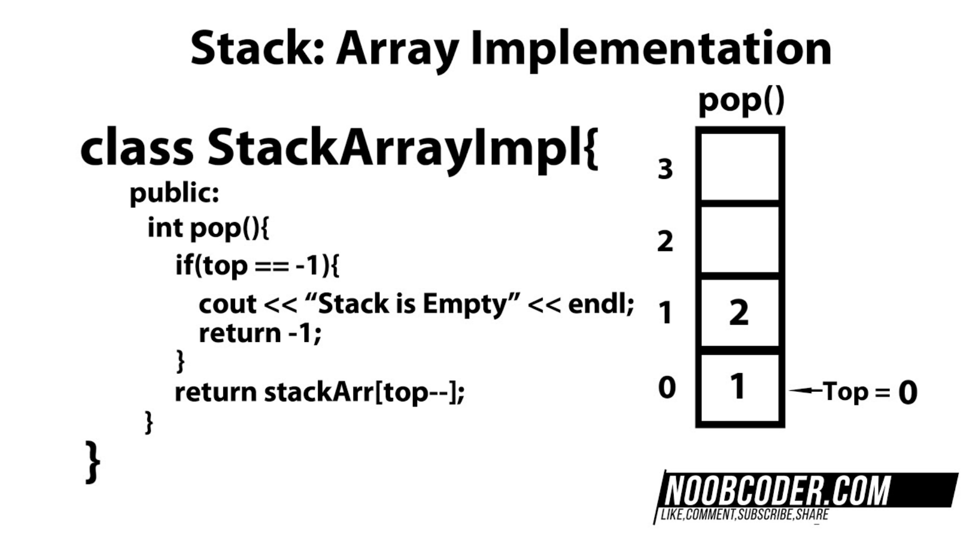
An UML diagram for the class Stack

- Many people describe an information system by drawing a **UML diagram**.
  - UML = unified modeling language.
- A **class** in a UML diagram is depicted as follows:

```
Stack

+isEmpty(): boolean
+push(newEntry: ItemType): boolean
+pop(): boolean
+peek(): ItemType
```

Figure 6-2
(Carrano and Henry, 2013)

Data Structures – Stacks 9 / 78 Ling-Chieh Kung (NTU IM)



Stack: Array Implementation

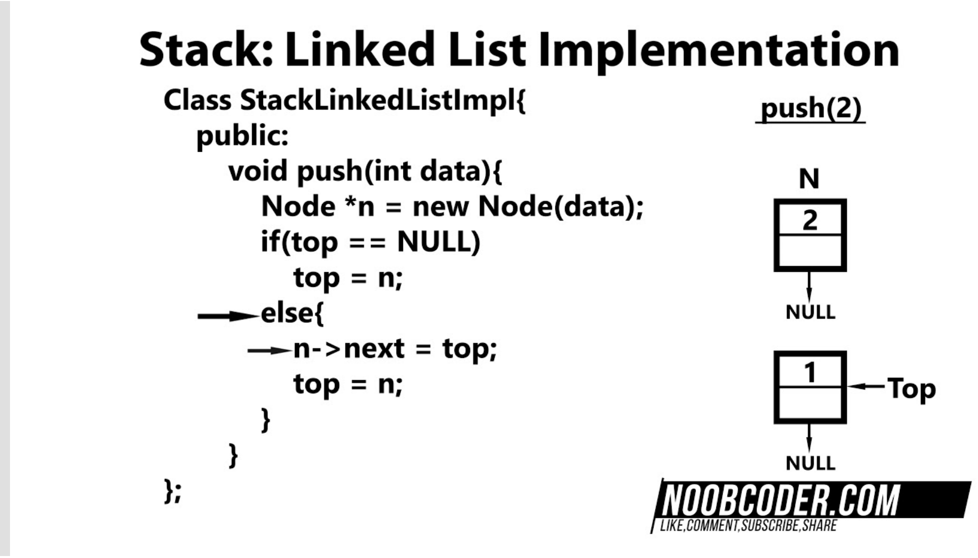
```
class StackArrayImpl{
    public:
      int pop(){
        if(top == -1){
          cout << “Stack is Empty” << endl;
          return -1;
        }
        return stackArr[top--];
      }
}
```

In [ ]:

```
#include <iostream>
#include <stdexcept>
using namespace std;

const int MAX_STACK = 999;


template<typename ItemType>
```

```
class StackInterface {
public:
    virtual bool isEmpty() const = 0; //pure virtual function
    virtual bool push(const ItemType& newEntry) = 0; //pure virtual function
    virtual bool pop() = 0; //pure virtual function
    virtual ItemType peek() const = 0; //pure virtual function
};


template<typename ItemType>
class ArrayStack : public StackInterface<ItemType>{
private:
    ItemType items[MAX_STACK];
    int top;
public:
    ArrayStack(); //Default constructor
    bool isEmpty() const;
    bool push(const ItemType& newEntry);
    bool pop();
    ItemType peek() const;
};

template<typename ItemType>
ArrayStack<ItemType>::ArrayStack() : top(-1){}

template<typename ItemType>
bool ArrayStack<ItemType>::isEmpty() const{
    return top < 0;
}

template<typename ItemType>
bool ArrayStack<ItemType>::push(const ItemType& newEntry) {
    bool result = false;
    if(this->top < MAX_STACK - 1){
        top++;
        this->items[top] = newEntry;
        result = true;
    }
    return result;
}

template<typename ItemType>
bool ArrayStack<ItemType>::pop(){
    bool result = false;
    if(!this->isEmpty()){
        top--;
        return true;
    }
    return result;
}

template<typename ItemType>
ItemType ArrayStack<ItemType>::peek() const{
    if(!this->isEmpty()){
        return items[top];
    }
    else
        throw logic_error("...");
}
```

```
int main(){
    StackInterface<int>* as = new ArrayStack<int>();
    as->push(4);
    as->push(11);
    as->pop();
    try {
        cout << as->peek() << endl;
        as->pop();
        cout << as->peek() << endl;
    }
    catch (logic_error e){
        cout << "empty!" << endl;
    }


    return 0;
}
```



Stack: Linked List Implementation

```
Class StackLinkedListImpl{
    public:
        void push(int data){
            Node *n = new Node(data);
            if(top == NULL)
                top = n;
    ——►else{
         ——►n->next = top;
                top = n;
            }
        }
};
```

In [ ]:

```
#include <iostream>
#include <stdexcept>
using namespace std;


template<typename ItemType>
class Node {
private:
    ItemType item;
    Node<ItemType>* next;
public:
    Node();
    Node(const ItemType& anItem);
    Node(const ItemType& anItem, Node<ItemType>* nextNodePtr);
    void setItem(const ItemType& anItem);
    void setNext(Node<ItemType>* nextNodePtr);
    ItemType getItem() const;
    Node<ItemType>* getNext() const;
};

template<typename ItemType>
Node<ItemType>::Node() : next(nullptr) {}
```

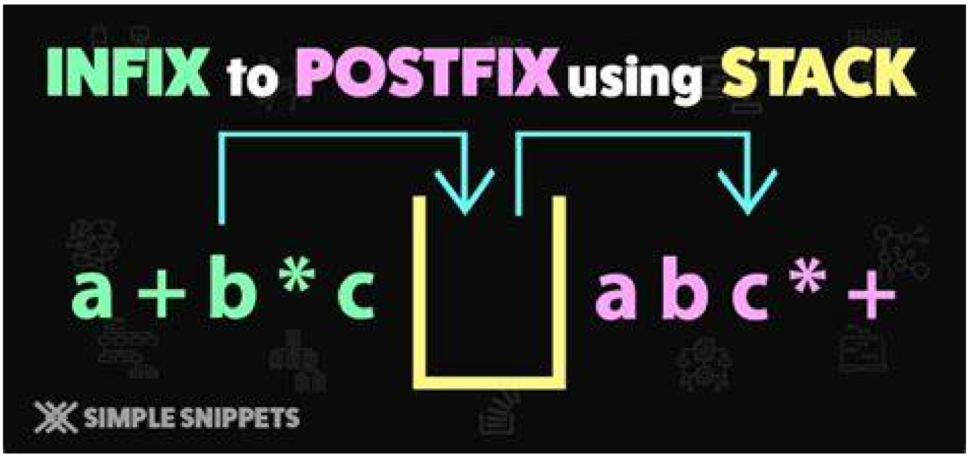
```
template<typename ItemType>
Node<ItemType>:: Node(const ItemType& anItem)
    : item(anItem), next(nullptr) {}

template<typename ItemType>
Node<ItemType>::Node(const ItemType& anItem, Node<ItemType>* nextNodePtr)
    : item(anItem), next(nextNodePtr) {}

template<typename ItemType>
void Node<ItemType>::setItem(const ItemType& anItem){
    item = anItem;
}

template<typename ItemType>
void Node<ItemType>::setNext(Node<ItemType>* nextNodePtr){
    next = nextNodePtr;
}

template<typename ItemType>
ItemType Node<ItemType>::getItem() const {
    return item;
}

template<typename ItemType>
Node<ItemType>* Node<ItemType>::getNext() const{
    return next;
}


template<typename ItemType>
class stackInterface {
public:
    virtual bool isEmpty() const = 0;
    virtual bool push(const ItemType& newItem) = 0;
    virtual bool pop() = 0;
    virtual ItemType peek() const = 0;
};


template<typename ItemType>
class LinkedStack : public stackInterface<ItemType>{
private:
    Node<ItemType>* topPtr;
public:
    //Constructor and Destructor
    LinkedStack(); // Default constructor
    LinkedStack(const LinkedStack<ItemType>& aStack); //Copy constructor
    virtual ~LinkedStack(); // Destructor
    // Stack Operations: Interface
    bool isEmpty() const;
    bool push(const ItemType& newItem);
    bool pop();
    ItemType peek() const;
};


template<typename ItemType>
LinkedStack<ItemType>::LinkedStack() : topPtr(nullptr) {}

template<typename ItemType>
LinkedStack<ItemType>::~LinkedStack() {
    while(!isEmpty()){ // Pop until stack is empty
        pop();
    }
}

template<typename ItemType>
LinkedStack<ItemType>::LinkedStack(const LinkedStack<ItemType>& aStack){
    //Point to nodes in the original chain
    Node<ItemType>* origChainPtr = aStack.topPtr;

    if(origChainPtr == nullptr){
        this->topPtr = nullptr; // Original Stack is empty
    }
    else {
        //Copy the first node
        topPtr = new Node<ItemType>();
        topPtr->setItem(origChainPtr->getItem());
        Node<ItemType>* newChainPtr = topPtr;

        //Copy remaining nodes
        while(origChainPtr->getNext() != nullptr){
            origChainPtr = origChainPtr->getNext();
            ItemType nextItem = origChainPtr->getItem();
            Node<ItemType>* newNodePtr = new Node<ItemType>(nextItem);
            newChainPtr->setNext(newNodePtr);
            newChainPtr = newChainPtr->getNext();
        }
        newChainPtr->setNext(nullptr); //Mark the bottom of stack
    }
}


template<typename ItemType>
bool LinkedStack<ItemType>::isEmpty() const {
    return topPtr == nullptr;
}

template<typename ItemType>
bool LinkedStack<ItemType>::push(const ItemType& newItem){
    Node<ItemType>* newNodePtr = new Node<ItemType>(newItem, topPtr);
    topPtr = newNodePtr;
    return true;
}

template<typename ItemType>
bool LinkedStack<ItemType>::pop(){
    bool result = false;
    if(!isEmpty()){
        Node<ItemType>* nodeToDeletePtr = topPtr;
        topPtr = topPtr->getNext();
        delete nodeToDeletePtr;
        result = true;
    }
    return result;
}

template<typename ItemType>
ItemType LinkedStack<ItemType>::peek() const {
    if(!this->isEmpty()){
        return topPtr->getItem();
    }
    else {
        throw logic_error("...");
    }

}


int main(){

    LinkedStack<int> ls;
    ls.push(4);
    ls.push(11);
    LinkedStack<int> lsCopy(ls);
    lsCopy.pop();
    try {
        cout << lsCopy.peek() << endl;
        cout << ls.peek() << endl;
    }
    catch(logic_error e) {
        cout << "empty!" << endl;
    }

    return 0;
}
```

In [ ]:

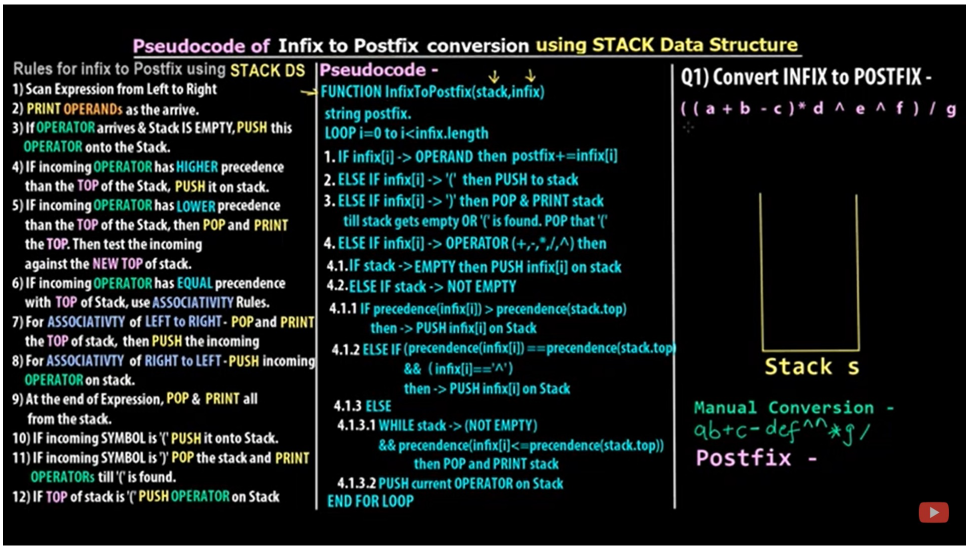
```
#include<iostream>
using namespace std;

class Node{
public:
    int data;
    Node* next;

    Node();
    Node(int d);
};

Node::Node(){
    data = 0;
    next = nullptr;
}

Node::Node(int d){
    data = d;
    next = nullptr;
}

class Stack{
public:
    Node* top;

    Stack();
    bool isEmpty();
    bool checkIfNodeExist(Node* newNode);
    void push(Node* newNode);
    Node* pop();
    Node* peek();
    int count();
    void display();
};
```

```
Stack::Stack(){
    top = nullptr;
}

bool Stack::isEmpty(){
    if(top==nullptr)
        return true;
    else
        return false;
}

bool Stack::checkIfNodeExist(Node* newNode){
    Node* temp = top;
    while(temp!=nullptr){
        if(temp->data==newNode->data)
            return true;
        else {
            temp = temp->next;
        }
    }
    return false;
}

void Stack::push(Node* newNode){
    if(top==nullptr){
        top = newNode;
        cout << "Pushed First Node Successfully" << endl << endl;
    }
    else if(checkIfNodeExist(newNode)){
        cout << "Node with value already exist" << endl;
        cout << "Please enter different key and value" << endl << endl;
    }
    else {
        newNode->next = top;
        top = newNode;
        cout << "Pushed Successfully" << endl;
    }
}

Node* Stack::pop(){
    if(isEmpty()){
        return nullptr;
    }
    else {
        Node* temp = top;
        top = top->next;
        return temp;
    }
}

Node* Stack::peek(){
    if(isEmpty()){
        return nullptr;
    }
    else {
        return top;
    }
}

int Stack::count(){
    int count = 0;
    Node* ptr = top;
    while(ptr!=nullptr){
        count++;
        ptr = ptr->next;
    }
    return count;
}


void Stack::display(){
    if(isEmpty()){
        cout << "Stack is empty";
    }
    else {
        cout << "Stack: " << endl;
        Node* ptr = top;
        while(ptr!=nullptr){
            cout << "Value: " << ptr->data << endl;
            ptr = ptr->next;
        }
    }
    cout << endl;
}



int main(){
    Stack s;
    int option, data;

    do {
        cout << "What operation do you want to perform?" << endl;
        cout << "Select Option Number or 0 to Exit" << endl << endl;
        cout << "1. Push" << endl;
        cout << "2. Pop" << endl;
        cout << "3. Peek" << endl;
        cout << "4. isEmpty" << endl;
        cout << "5. Count" << endl;
        cout << "6. Display" << endl;
        cout << "9. Clear Screen" << endl;
        cout << "0. Exit" << endl << endl;

        cout << "Enter Your Option Number: ";
        cin >> option;

        Node* newNode = new Node();

        switch(option){
            case 0:
                break;
            case 1:
                cout << "Enter Value of Node" << endl;
                cin >> data;
                newNode->data = data;
                s.push(newNode);
                break;
            case 2:
                cout << "Pop Function Called." << endl;
                newNode = s.pop();
                if(newNode==nullptr){
                    cout << "Stack Underflow" << endl << endl;
                }
                else{
                    cout << "Pop Value: " << newNode->data << endl << endl;
                }

                delete newNode;

                break;
            case 3:
                cout << "Peek Function Called." << endl;
                newNode = s.peek();
                cout << "The Top Value of Stack: " << newNode->data << endl << endl
                break;
            case 4:
                if(s.isEmpty()){
                    cout << "Stack is Empty" << endl;
                }
                else {
                    cout << "Stack is not Empty" << endl;
                }
                break;
            case 5:
                cout << "Count Function Called." << endl;
                cout << "No of Node in the Stack: " << s.count() << endl << endl;
                break;
            case 6:
                s.display();
                cout << endl;
                break;
            case 9:
                system("cls");
                break;
            default:
                cout << "Please Enter the Proper Option Number" << endl;
                break;
        }





    } while(option!=0);


    return 0;
}
```

Infix Prefix & Postfix Expressions Using Stack



INFIX to POSTFIX using STACK



Pseudocode of Infix to Postfix conversion using STACK Data Structure

Rules for infix to Postfix using STACK DS

1) Scan Expression from Left to Right
2) PRINT OPERANDs as the arrive.
3) If OPERATOR arrives & Stack IS EMPTY, PUSH this OPERATOR onto the Stack.
4) IF incoming OPERATOR has HIGHER precedence than the TOP of the Stack, PUSH it on stack.
5) IF incoming OPERATOR has LOWER precedence than the TOP of the Stack, then POP and PRINT the TOP. Then test the incoming against the NEW TOP of stack.
6) IF incoming OPERATOR has EQUAL precendence with TOP of Stack, use ASSOCIATIVITY Rules.
7) For ASSOCIATIVTY of LEFT to RIGHT - POP and PRINT the TOP of stack, then PUSH the incoming
8) For ASSOCIATIVTY of RIGHT to LEFT - PUSH incoming OPERATOR on stack.
9) At the end of Expression, POP & PRINT all from the stack.
10) IF incoming SYMBOL is '(' PUSH it onto Stack.
11) IF incoming SYMBOL is ')' POP the stack and PRINT OPERATORs till '(' is found.
12) IF TOP of stack is '(' PUSH OPERATOR on Stack

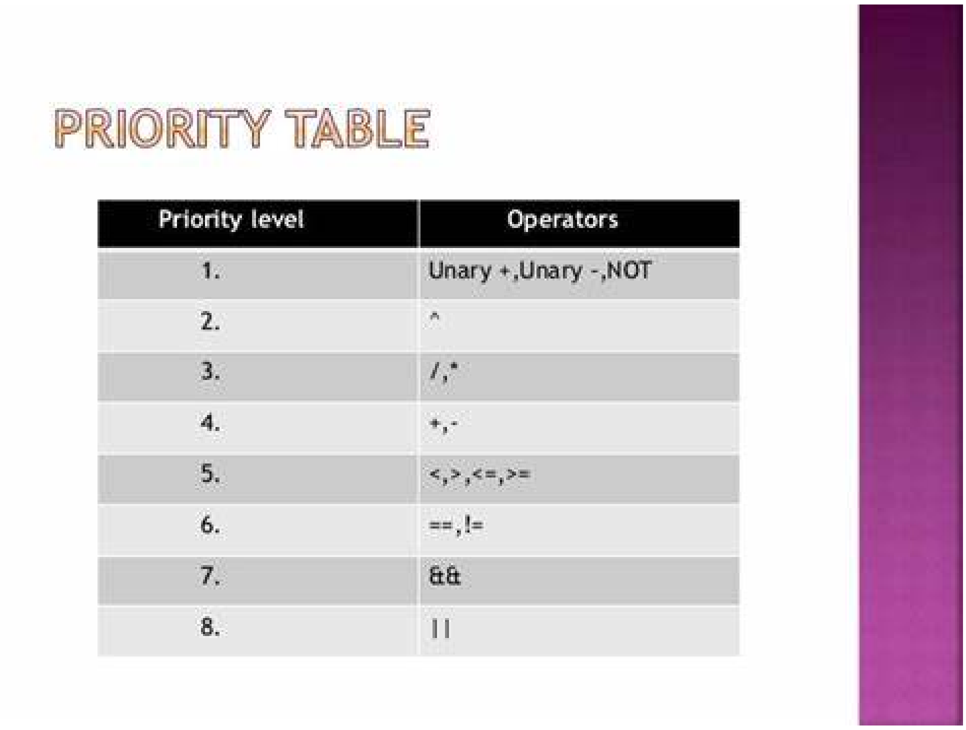
Pseudocode -

```
FUNCTION InfixToPostfix(stack,infix)
 string postfix.
 LOOP i=0 to i<infix.length
 1. IF infix[i] -> OPERAND then postfix+=infix[i]
 2. ELSE IF infix[i] -> '(' then PUSH to stack
 3. ELSE IF infix[i] -> ')' then POP & PRINT stack
      till stack gets empty OR '(' is found. POP that '('
 4. ELSE IF infix[i] -> OPERATOR (+,-,*,/,^) then
  4.1.IF stack -> EMPTY then PUSH infix[i] on stack
  4.2.ELSE IF stack -> NOT EMPTY
   4.1.1 IF precedence(infix[i]) > precendence(stack.top)
           then -> PUSH infix[i] on Stack
   4.1.2 ELSE IF (precendence(infix[i]) ==precendence(stack.top)
                 && (infix[i]=='^')
                 then -> PUSH infix[i] on Stack
   4.1.3 ELSE
    4.1.3.1 WHILE stack -> (NOT EMPTY)
              && precendence(infix[i]<=precendence(stack.top))
                 then POP and PRINT stack
    4.1.3.2 PUSH current OPERATOR on Stack
 END FOR LOOP
```

Q1) Convert INFIX to POSTFIX -

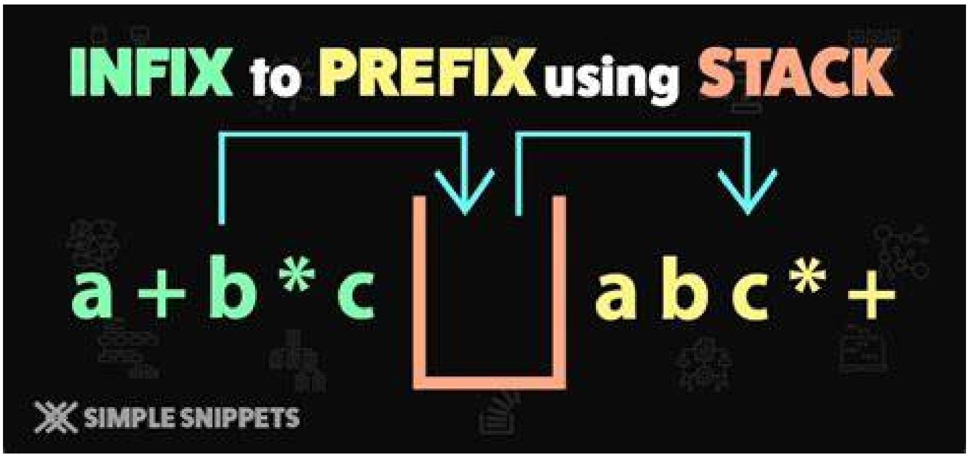
((a + b - c) * d ^ e ^ f) / g

Manual Conversion -
ab+c-def^^*g/
Postfix -

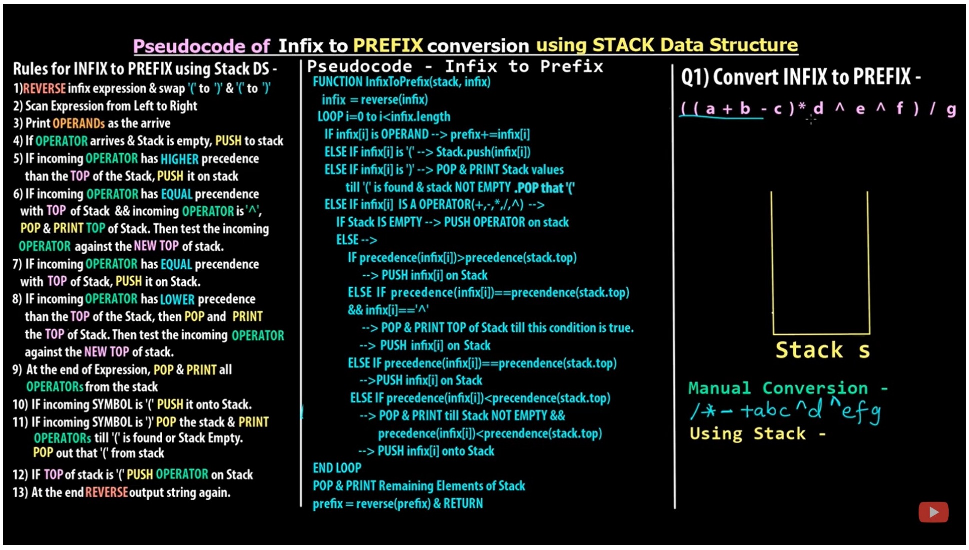


PRIORITY TABLE

Priority level	Operators
1.	Unary +,Unary -,NOT
2.	^
3.	/,*
4.	+,-
5.	<,>,<=,>=
6.	==,!=
7.	&&
8.	||



INFIX to PREFIX using STACK



Pseudocode of Infix to PREFIX conversion using STACK Data Structure

Rules for INFIX to PREFIX using Stack DS -

1)REVERSE infix expression & swap '(' to ')' & '(' to ')'
2) Scan Expression from Left to Right
3) Print OPERANDs as the arrive
4) If OPERATOR arrives & Stack is empty, PUSH to stack
5) IF incoming OPERATOR has HIGHER precedence than the TOP of the Stack, PUSH it on stack
6) IF incoming OPERATOR has EQUAL precendence with TOP of Stack && incoming OPERATOR is '^', POP & PRINT TOP of Stack. Then test the incoming OPERATOR against the NEW TOP of stack.
7) IF incoming OPERATOR has EQUAL precendence with TOP of Stack, PUSH it on Stack.
8) IF incoming OPERATOR has LOWER precedence than the TOP of the Stack, then POP and PRINT the TOP of Stack. Then test the incoming OPERATOR against the NEW TOP of stack.
9) At the end of Expression, POP & PRINT all OPERATORs from the stack
10) IF incoming SYMBOL is '(' PUSH it onto Stack.
11) IF incoming SYMBOL is ')' POP the stack & PRINT OPERATORs till '(' is found or Stack Empty. POP out that '(' from stack
12) IF TOP of stack is '(' PUSH OPERATOR on Stack
13) At the end REVERSE output string again.

Pseudocode - Infix to Prefix

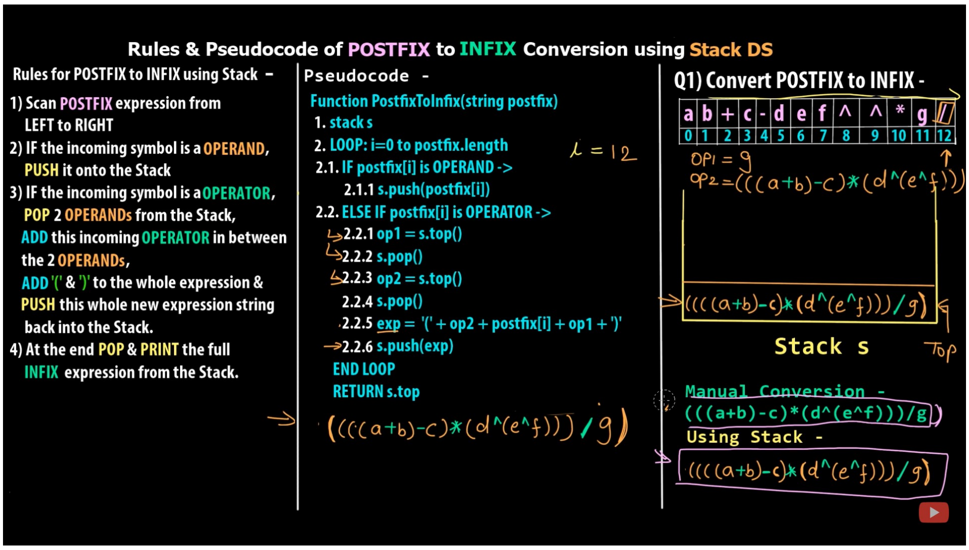
```
FUNCTION InfixToPrefix(stack, infix)
  infix = reverse(infix)
  LOOP i=0 to i<infix.length
    IF infix[i] is OPERAND --> prefix+=infix[i]
    ELSE IF infix[i] is '(' --> Stack.push(infix[i])
    ELSE IF infix[i] is ')' --> POP & PRINT Stack values
          till '(' is found & stack NOT EMPTY .POP that '('
    ELSE IF infix[i]  IS A OPERATOR(+,-,*,/,^) -->
      IF Stack IS EMPTY --> PUSH OPERATOR on stack
      ELSE -->
        IF precedence(infix[i])>precedence(stack.top)
            --> PUSH infix[i] on Stack
        ELSE IF  precedence(infix[i])==precendence(stack.top)
        && infix[i]=='^'
            --> POP & PRINT TOP of Stack till this condition is true.
            --> PUSH  infix[i] on Stack
        ELSE IF precedence(infix[i])==precendence(stack.top)
            -->PUSH infix[i] on Stack
        ELSE IF precedence(infix[i])<precendence(stack.top)
            --> POP & PRINT till Stack NOT EMPTY &&
                precedence(infix[i])<precendence(stack.top)
            --> PUSH infix[i] onto Stack
END LOOP
POP & PRINT Remaining Elements of Stack
prefix = reverse(prefix) & RETURN
```

Q1) Convert INFIX to PREFIX -

((a + b - c) * d ^ e ^ f) / g

Stack s

Manual Conversion -
/*-+abc^d^efg
Using Stack -



Rules & Pseudocode of POSTFIX to INFIX Conversion using Stack DS

Rules for POSTFIX to INFIX using Stack -

1) Scan POSTFIX expression from LEFT to RIGHT
2) IF the incoming symbol is a OPERAND, PUSH it onto the Stack
3) IF the incoming symbol is a OPERATOR, POP 2 OPERANDs from the Stack, ADD this incoming OPERATOR in between the 2 OPERANDs, ADD '(' & ')' to the whole expression & PUSH this whole new expression string back into the Stack.
4) At the end POP & PRINT the full INFIX expression from the Stack.

Pseudocode -

```
Function PostfixToInfix(string postfix)
 1. stack s
 2. LOOP: i=0 to postfix.length                i = 12
 2.1. IF postfix[i] is OPERAND ->
      2.1.1 s.push(postfix[i])
 2.2. ELSE IF postfix[i] is OPERATOR ->
      2.2.1 op1 = s.top()
      2.2.2 s.pop()
      2.2.3 op2 = s.top()
      2.2.4 s.pop()
      2.2.5 exp = '(' + op2 + postfix[i] + op1 + ')'
      2.2.6 s.push(exp)
   END LOOP
   RETURN s.top
```

→ ((((a+b)-c)*(d^(e^f)))/g)

Q1) Convert POSTFIX to INFIX -

a	b	+	c	-	d	e	f	^	^	*	g	/
0	1	2	3	4	5	6	7	8	9	10	11	12

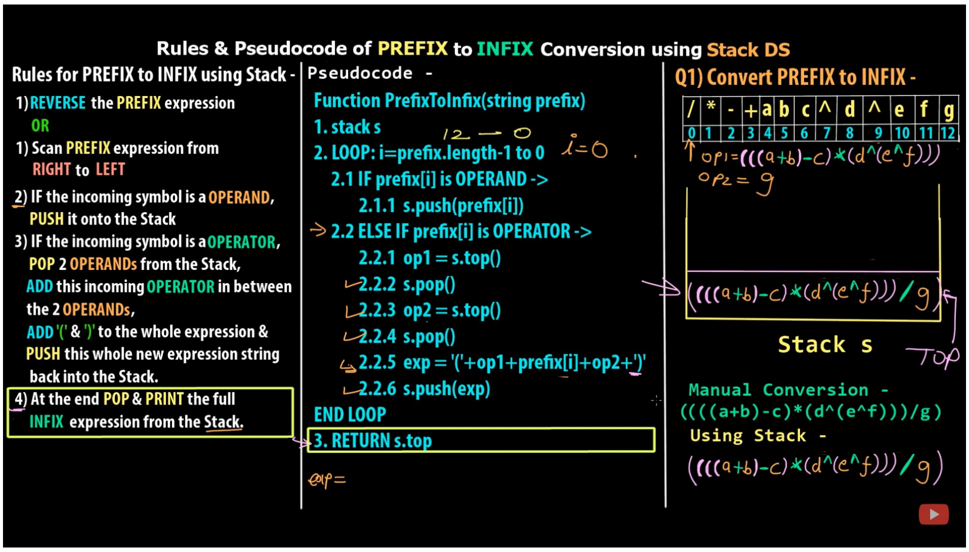
op1 = g
op2 = (((a+b)-c)*(d^(e^f)))

→ ((((a+b)-c)*(d^(e^f)))/g)

Stack s

Top

Manual Conversion -
(((a+b)-c)*(d^(e^f)))/g)
Using Stack -
→ ((((a+b)-c)*(d^(e^f)))/g)



Rules & Pseudocode of PREFIX to INFIX Conversion using Stack DS

Rules for PREFIX to INFIX using Stack -

1) REVERSE the PREFIX expression
   OR
1) Scan PREFIX expression from RIGHT to LEFT
2) IF the incoming symbol is a OPERAND, PUSH it onto the Stack
3) IF the incoming symbol is a OPERATOR, POP 2 OPERANDs from the Stack, ADD this incoming OPERATOR in between the 2 OPERANDs, ADD '(' & ')' to the whole expression & PUSH this whole new expression string back into the Stack.
4) At the end POP & PRINT the full INFIX expression from the Stack.

Pseudocode -

```
Function PrefixToInfix(string prefix)
1. stack s                              12 → 0
2. LOOP: i=prefix.length-1 to 0         i=0
   2.1 IF prefix[i] is OPERAND ->
       2.1.1 s.push(prefix[i])
   2.2 ELSE IF prefix[i] is OPERATOR ->
       2.2.1 op1 = s.top()
       2.2.2 s.pop()
       2.2.3 op2 = s.top()
       2.2.4 s.pop()
       2.2.5 exp = '('+op1+prefix[i]+op2+')'
       2.2.6 s.push(exp)
END LOOP
3. RETURN s.top
exp=
```

Q1) Convert PREFIX to INFIX -

/	*	-	+	a	b	c	^	d	^	e	f	g
0	1	2	3	4	5	6	7	8	9	10	11	12

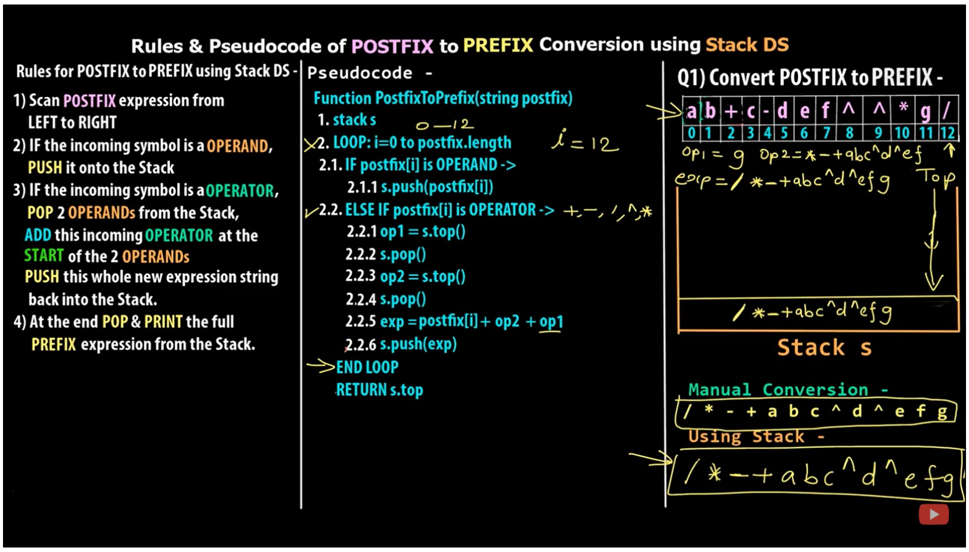
op1=(((a+b)-c)*(d^(e^f)))
op2= g

((((a+b)-c)*(d^(e^f)))/g)

Stack s

Top

Manual Conversion -
((((a+b)-c)*(d^(e^f)))/g)
Using Stack -
((((a+b)-c)*(d^(e^f)))/g)



Rules & Pseudocode of POSTFIX to PREFIX Conversion using Stack DS

Rules for POSTFIX to PREFIX using Stack DS -

1) Scan POSTFIX expression from LEFT to RIGHT
2) IF the incoming symbol is a OPERAND, PUSH it onto the Stack
3) IF the incoming symbol is a OPERATOR, POP 2 OPERANDs from the Stack, ADD this incoming OPERATOR at the START of the 2 OPERANDs PUSH this whole new expression string back into the Stack.
4) At the end POP & PRINT the full PREFIX expression from the Stack.

Pseudocode -

```
Function PostfixToPrefix(string postfix)
 1. stack s                             0—12
 2. LOOP: i=0 to postfix.length         i = 12
 2.1. IF postfix[i] is OPERAND ->
      2.1.1 s.push(postfix[i])
 2.2. ELSE IF postfix[i] is OPERATOR -> +, −, /, ^, *
      2.2.1 op1 = s.top()
      2.2.2 s.pop()
      2.2.3 op2 = s.top()
      2.2.4 s.pop()
      2.2.5 exp = postfix[i] + op2 + op1
      2.2.6 s.push(exp)
 END LOOP
 RETURN s.top
```

Q1) Convert POSTFIX to PREFIX -

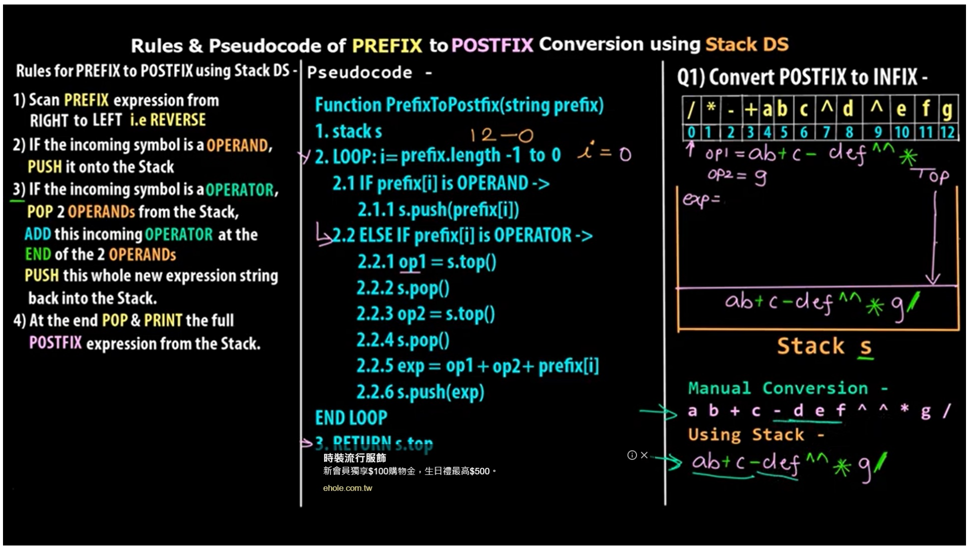
a	b	+	c	-	d	e	f	^	^	*	g	/
0	1	2	3	4	5	6	7	8	9	10	11	12

op1 = g op2 = *−+abc^d^ef
exp = /*−+abc^d^efg Top

/*−+abc^d^efg

Stack s

Manual Conversion -
/ * - + a b c ^ d ^ e f g
Using Stack -
/*−+abc^d^efg



Rules & Pseudocode of PREFIX to POSTFIX Conversion using Stack DS

Rules for PREFIX to POSTFIX using Stack DS -

1) Scan PREFIX expression from RIGHT to LEFT i.e REVERSE
2) IF the incoming symbol is a OPERAND, PUSH it onto the Stack
3) IF the incoming symbol is a OPERATOR, POP 2 OPERANDs from the Stack, ADD this incoming OPERATOR at the END of the 2 OPERANDs PUSH this whole new expression string back into the Stack.
4) At the end POP & PRINT the full POSTFIX expression from the Stack.

Pseudocode -

```
Function PrefixToPostfix(string prefix)
1. stack s                              12—0
2. LOOP: i= prefix.length -1 to 0       i = 0
   2.1 IF prefix[i] is OPERAND ->
       2.1.1 s.push(prefix[i])
   2.2 ELSE IF prefix[i] is OPERATOR ->
       2.2.1 op1 = s.top()
       2.2.2 s.pop()
       2.2.3 op2 = s.top()
       2.2.4 s.pop()
       2.2.5 exp = op1 + op2+ prefix[i]
       2.2.6 s.push(exp)
END LOOP
3. RETURN s.top
```

Q1) Convert POSTFIX to INFIX -

/	*	-	+	a	b	c	^	d	^	e	f	g
0	1	2	3	4	5	6	7	8	9	10	11	12

op1 = ab+c− def^^*
op2 = g
Top
exp=

ab+c−def^^*g/

Stack s

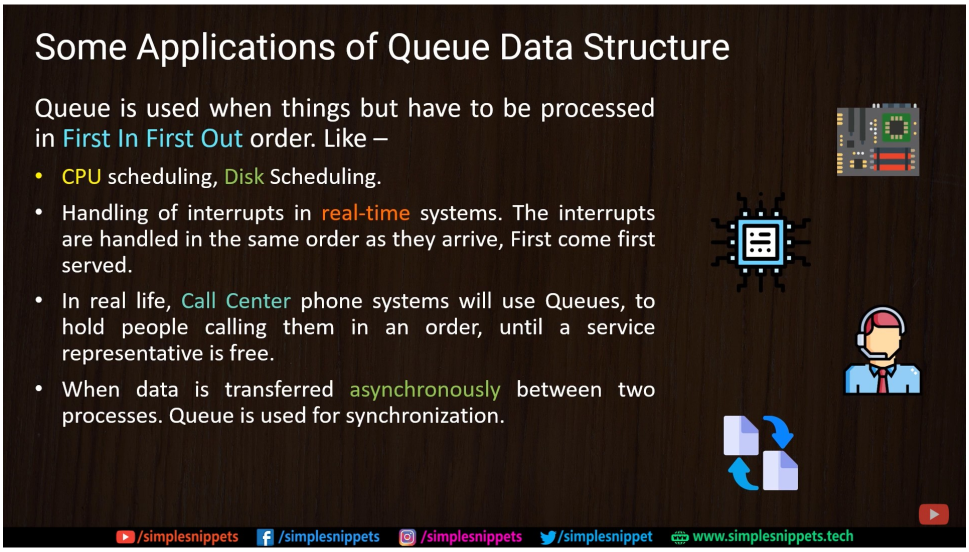
Manual Conversion -
a b + c - d e f ^ ^ * g /
Using Stack -
ab+c−def^^*g/

In [ ]:

```
#include<iostream>
#include<string>
#include<stack>
#include<algorithm>
using namespace std;


bool isOperator(char c){
    if(c=='+' || c=='-' || c=='*' || c=='/' || c=='^')
```

```
        return true;
    else
        return false;
}

bool isOperand(char c){
    if((c>='A' && c<='Z') || (c>='a' && c<='z'))
        return true;
    else
        return false;
}



int precedence(char c){
    if(c=='^')
        return 3;
    else if(c=='*' || c=='/')
        return 2;
    else if(c=='+' || c=='-')
        return 1;
    else
        return -1;
}

string infixToPostfix(string infix){
    stack<char> s;
    string postfix;

    for(int i=0; i<infix.length(); i++){
        // check if oprand
        if((infix[i]>='A' && infix[i]<='Z')
           || (infix[i]>='a' && infix[i]<='z')){
            postfix += infix[i];
        }
        else if(infix[i]=='(' ){
            s.push(infix[i]);
        }
        else if(infix[i]==')'){
            while((s.top()!='(') && (!s.empty())){
                postfix += s.top();
                s.pop();
            }
            if(s.top()=='('){
                s.pop();
               }
        }
        else if(isOperator(infix[i])){
            if(s.empty()){
                s.push(infix[i]);
            }
            else{
                if(precedence(infix[i])>precedence(s.top())){
                    s.push(infix[i]);
                }
                else if((precedence(infix[i])==precedence(s.top())) && (infix[i]==
                    s.push(infix[i]);
                }
                else{
                    while((!s.empty()) && (precedence(infix[i])<=precedence(s.top(
                        postfix += s.top();
                        s.pop();
                    }
                    s.push(infix[i]);
                }
            }
        }
    }

    while(!s.empty()){
        postfix += s.top();
        s.pop();
    }

    return postfix;
}


string infixToPrefix(string infix){
    stack<char> s;
    string prefix;

    reverse(infix.begin(), infix.end()); // include<algorithn>

    for(int i=0; i<infix.length(); i++){
        if(infix[i]=='(')
            infix[i] = ')';
        else if(infix[i]==')')
            infix[i] ='(';
    }

    cout << "\nReverse Infix: " << infix << endl;

    for(int i=0; i<infix.length(); i++){
        if((infix[i]>='A' && infix[i]<='Z') || (infix[i]>='a' && infix[i]<='z')){
            prefix += infix[i];
        }
        else if(infix[i]=='('){
            s.push(infix[i]);
        }
        else if(infix[i]==')'){
            while((s.top()!='(') && (!s.empty())){
                    prefix += s.top();
                    s.pop();
            }
            if(s.top()=='('){
                s.pop();
            }
        }
        else if(isOperator(infix[i])){
            if(s.empty()){
                s.push(infix[i]);
            }
            else {
                if(precedence(infix[i]) > precedence(s.top())){
                    s.push(infix[i]);
                }
                else if((precedence(infix[i])==precedence(s.top())) && (infix[i]==
                    while((precedence(infix[i])==precedence(s.top())) &&(infix[i]==
                        prefix += s.top();
                        s.pop();
                    }
                    s.push(infix[i]);
                }
                else if(precedence(infix[i])==precedence(s.top())){
                    s.push(infix[i]);
                }
                else {
                    while((!s.empty()) && (precedence(infix[i])<precedence(s.top()
                        prefix += s.top();
                        s.pop();
                    }
                    s.push(infix[i]);
                }
            }
        }
    }

    while((!s.empty())){
        prefix += s.top();
        s.pop();
    }

    reverse(prefix.begin(), prefix.end());

    return prefix;
}

string postToInfix(string postfix){
    stack<string> s;

    for(int i=0; i<postfix.length(); i++){
        if(isOperand(postfix[i])){
            string op(1, postfix[i]);
            s.push(op);
        }
        else {
            string op1 = s.top();
            s.pop();
            string op2 = s.top();
            s.pop();
            s.push('(' + op2 + postfix[i] + op1 + ')');
        }
    }

    return s.top();
}

string preToInfix(string prefix){
    stack<string> s;

    for(int i=prefix.length()-1; i>=0; i--){
        if(isOperand(prefix[i])){
            string op(1, prefix[i]);
            s.push(op);
        }
        else {
            string op1 = s.top();
            s.pop();
            string op2 = s.top();
            s.pop();
            s.push('(' + op1 + prefix[i] + op2 + ')');
        }
    }

    return s.top();
}

string postToPrefix(string postfix){
    stack<string> s;

    for(int i=0; i<postfix.length(); i++){
        if(isOperand(postfix[i])){
            string op(1, postfix[i]);
            s.push(op);
        }
        else {
            string op1 = s.top();
            s.pop();
            string op2 = s.top();
            s.pop();
            s.push(postfix[i] + op2 + op1);
        }
    }

    return s.top();
}

string preToPostfix(string prefix){
    stack<string> s;

    for(int i=prefix.length()-1; i>=0; i--){
        if(isOperand(prefix[i])){
            string op(1, prefix[i]);
            s.push(op);
        }
        else {
            string op1 = s.top();
            s.pop();
            string op2 = s.top();
            s.pop();
            s.push(op1 + op2 + prefix[i]);
        }
    }

    return s.top();
}



int main(){

    string infix_exp, postfix_exp, prefix_exp;
    string post_in_exp, pre_in_exp, post_pre_exp, pre_post_exp;
    cout << "Enter a Infix Expression: " << endl;
    cin >> infix_exp;


    cout << "\nInfix Express: " << infix_exp << endl;

    postfix_exp = infixToPostfix(infix_exp);
    cout << "\nPostfix Expression: " << postfix_exp << endl;

    prefix_exp = infixToPrefix(infix_exp);
    cout << "\nPrefix Expression: " << prefix_exp << endl;
    cout << endl;

    post_in_exp = postToInfix(postfix_exp);
    cout << "\nPostfix to Infix: " << post_in_exp << endl;
    cout << endl;

    pre_in_exp = preToInfix(prefix_exp);
    cout << "\nPrefix to Infix: " << pre_in_exp << endl;
    cout << endl;

    post_pre_exp = postToPrefix(postfix_exp);
    cout << "\nPostfix to Prefix: " << post_pre_exp << endl;
    cout << endl;

    pre_post_exp = preToPostfix(prefix_exp);
    cout << "\nPrefix to Postfix: " << pre_post_exp << endl;
    cout << endl;

    return 0;
}
```

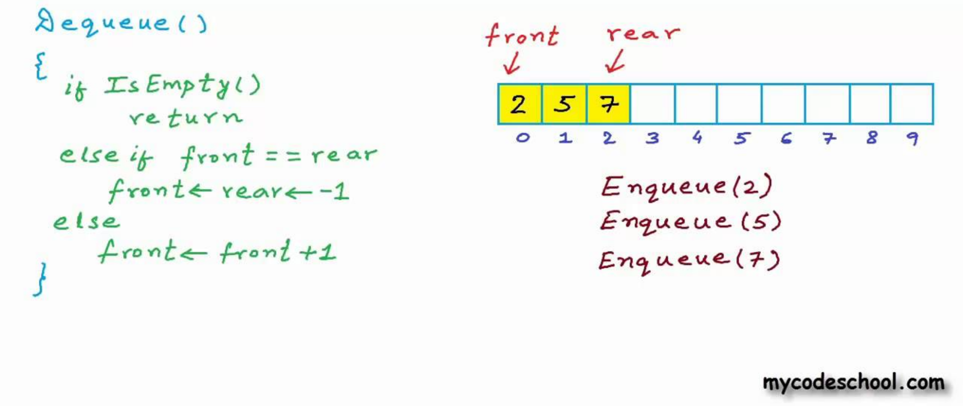
Queue



Some Applications of Queue Data Structure

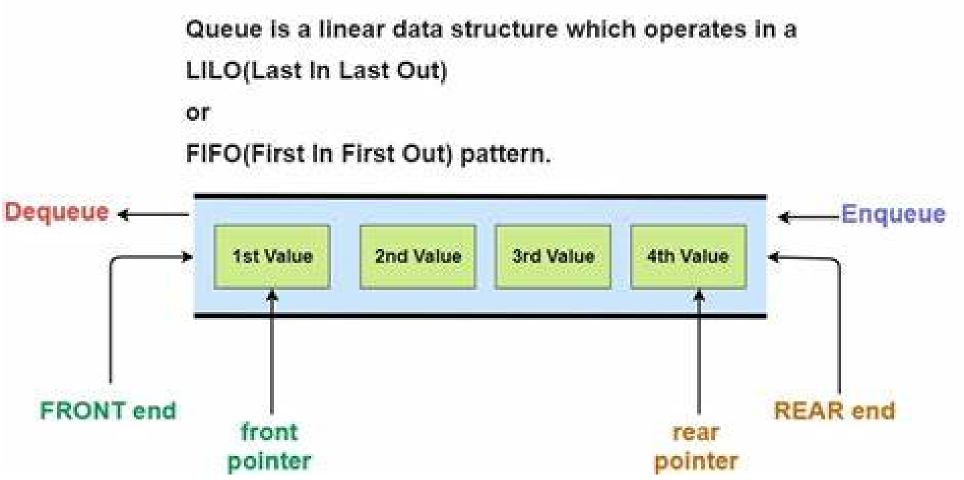
Queue is used when things but have to be processed in First In First Out order. Like –

- CPU scheduling, Disk Scheduling.
- Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive, First come first served.
- In real life, Call Center phone systems will use Queues, to hold people calling them in an order, until a service representative is free.
- When data is transferred asynchronously between two processes. Queue is used for synchronization.

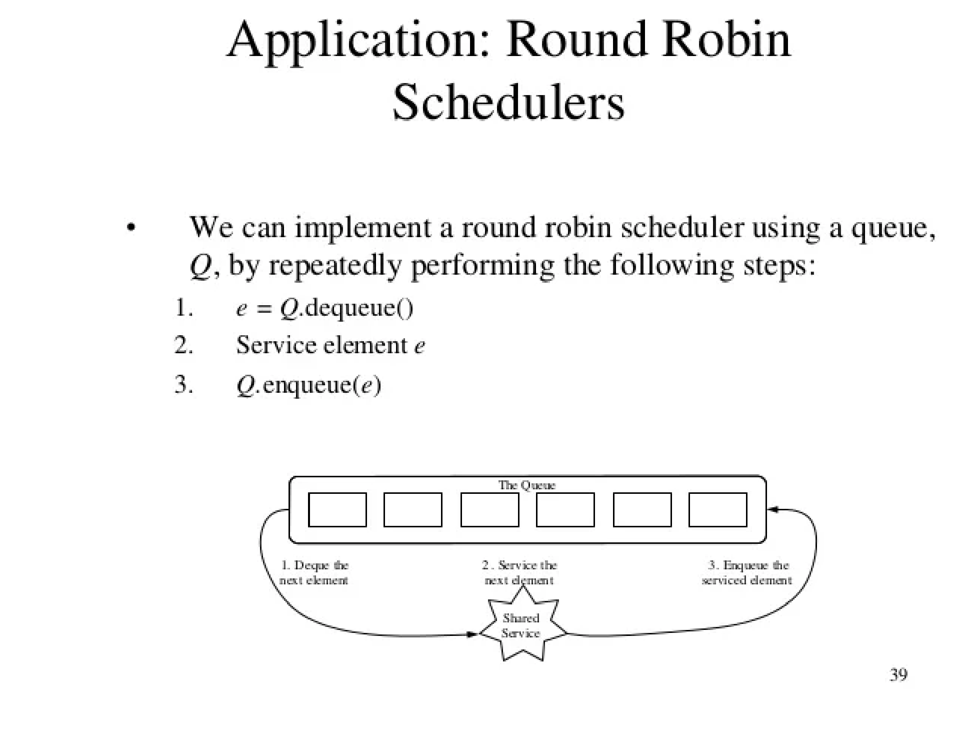


```
Dequeue()
{
   if IsEmpty()
        return
   else if front == rear
        front ← rear ← -1
   else
        front ← front +1
}
```

Enqueue(2)
Enqueue(5)
Enqueue(7)



Queue is a linear data structure which operates in a LILO(Last In Last Out) or FIFO(First In First Out) pattern.



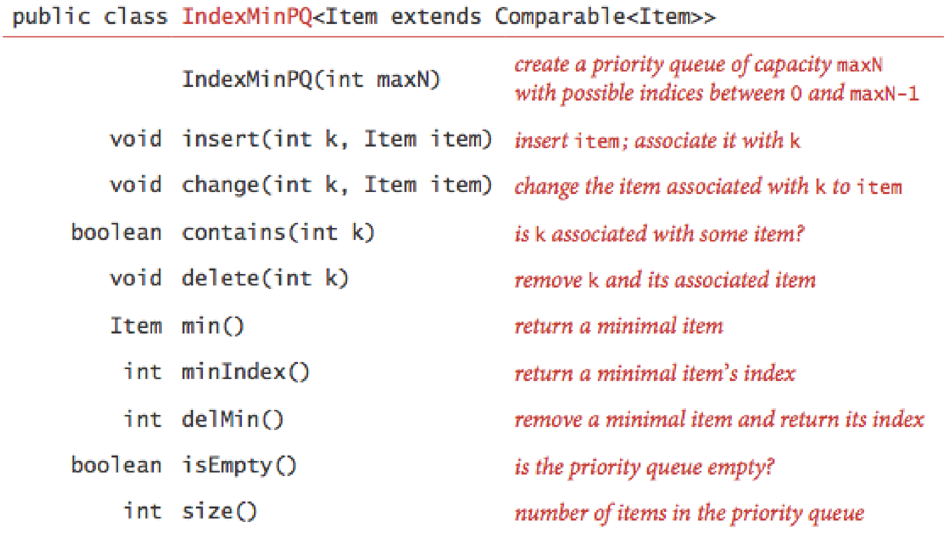
Application: Round Robin Schedulers

- We can implement a round robin scheduler using a queue, Q, by repeatedly performing the following steps:

1. $e = Q$.dequeue()
2. Service element e
3. Q.enqueue(e)

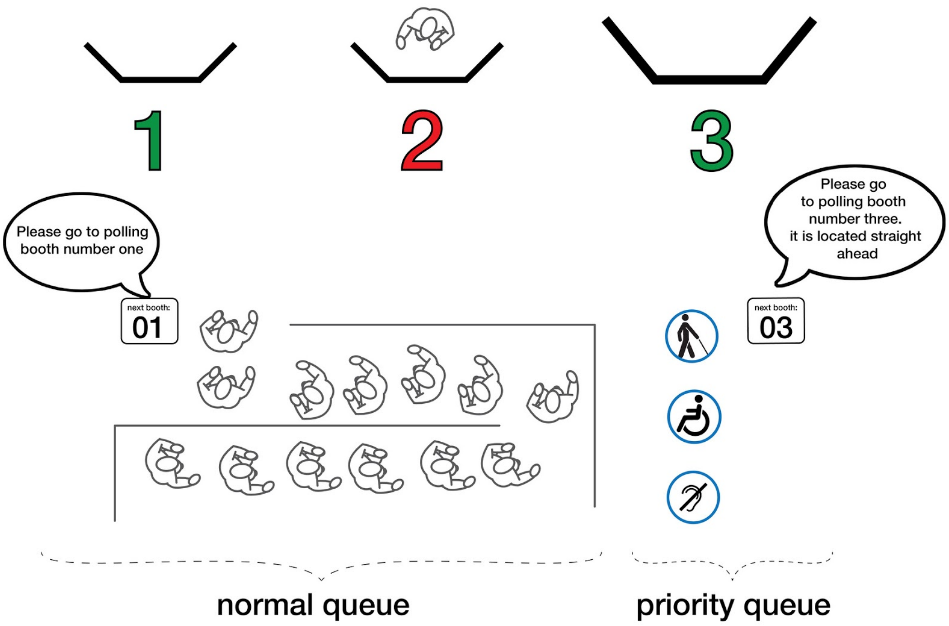
The Queue
1. Deque the next element
2 . Service the next element
3. Enqueue the serviced element
Shared Service

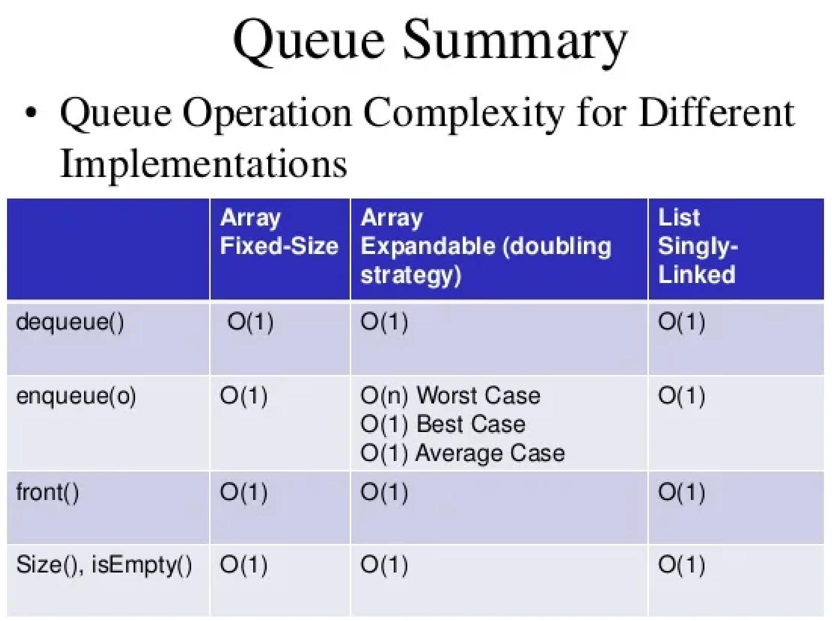
Priority Queue



`public class IndexMinPQ<Item extends Comparable<Item>>`

	`IndexMinPQ(int maxN)`	*create a priority queue of capacity* `maxN` *with possible indices between* `0` *and* `maxN-1`
`void`	`insert(int k, Item item)`	*insert* `item`*; associate it with* `k`
`void`	`change(int k, Item item)`	*change the item associated with* `k` *to* `item`
`boolean`	`contains(int k)`	*is* `k` *associated with some item?*
`void`	`delete(int k)`	*remove* `k` *and its associated item*
`Item`	`min()`	*return a minimal item*
`int`	`minIndex()`	*return a minimal item's index*
`int`	`delMin()`	*remove a minimal item and return its index*
`boolean`	`isEmpty()`	*is the priority queue empty?*
`int`	`size()`	*number of items in the priority queue*

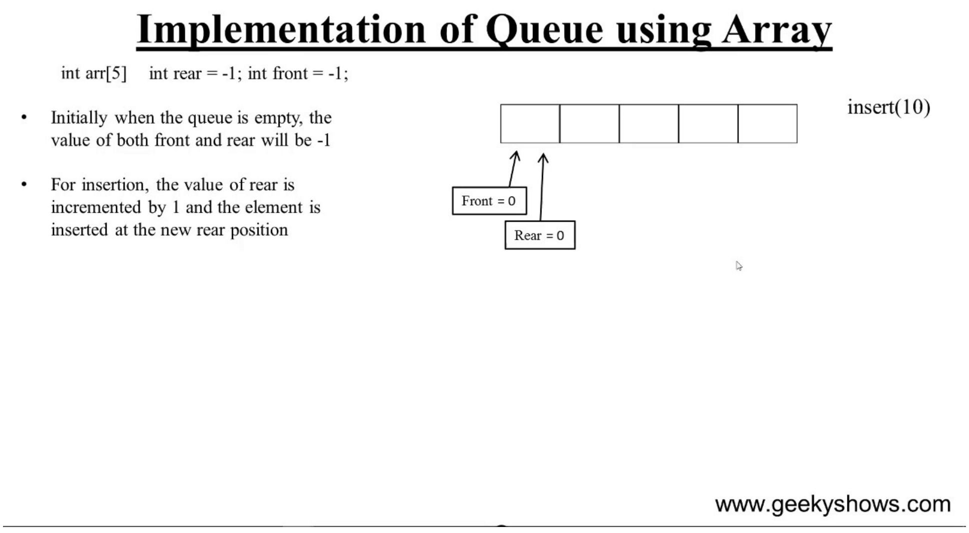

1
2
3
Please go to polling booth number one
Please go to polling booth number three. it is located straight ahead
next booth: 01
next booth: 03
normal queue
priority queue



Queue Summary

- Queue Operation Complexity for Different Implementations

	Array Fixed-Size	Array Expandable (doubling strategy)	List Singly-Linked
dequeue()	O(1)	O(1)	O(1)
enqueue(o)	O(1)	O(n) Worst Case O(1) Best Case O(1) Average Case	O(1)
front()	O(1)	O(1)	O(1)
Size(), isEmpty()	O(1)	O(1)	O(1)



Implementation of Queue using Array

int arr[5]   int rear = -1; int front = -1;

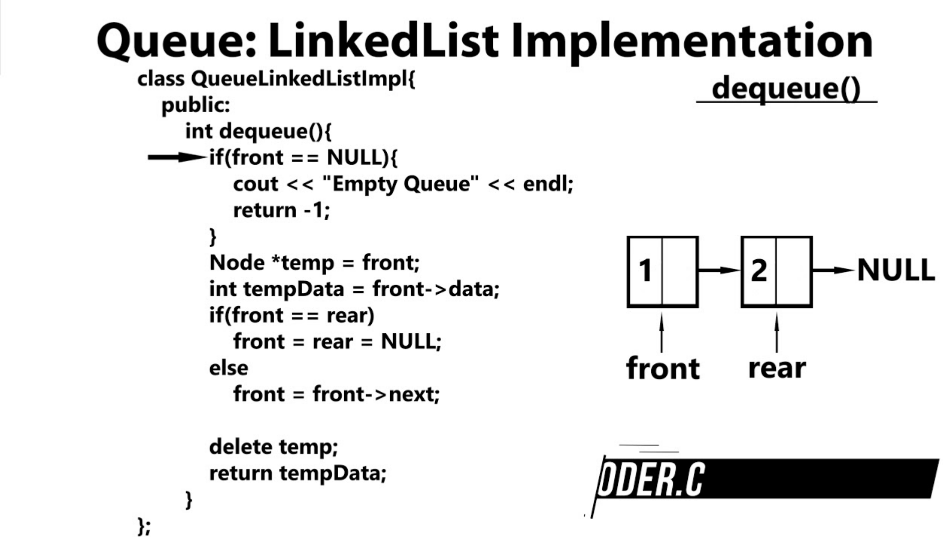
- Initially when the queue is empty, the value of both front and rear will be -1
- For insertion, the value of rear is incremented by 1 and the element is inserted at the new rear position

insert(10)

www.geekyshows.com

In [ ]:

```
#include <iostream>
#include <string>
using namespace std;

const int MAX_QUEUE = 5;


template<typename ItemType>
class Queue {
private:
    ItemType items[MAX_QUEUE];
    int front;
    int rear;
public:
    Queue();//default constructor
    ~Queue();
    bool isEmpty() const;
    bool isFull() const;
    void enqueue(const ItemType& newEntry);
    ItemType dequeue();
    int size();
    void display();
};


template<typename ItemType>
Queue<ItemType>::Queue() {
    front = -1;
    rear = -1;

    for (int i = 0; i < MAX_QUEUE; i++){
        items[i] = -1;
    }
}

template<typename ItemType>
Queue<ItemType>::~Queue() {
    delete [] items;
}

template<typename ItemType>
bool Queue<ItemType>::isEmpty() const{
    return ((front < 0) && (rear < 0));
};

template<typename ItemType>
bool Queue<ItemType>::isFull() const {
    return ((this->front == 0) && (this->rear == (MAX_QUEUE - 1)));
};

template<typename ItemType>
void Queue<ItemType>::enqueue(const ItemType& newEntry){
    if(this->rear == MAX_QUEUE - 1){
        cout << "Can not Enqueue." << endl;
        return;
    }
    if(this->isEmpty()){
        this->front = 0;
        this->rear = 0;
        this->items[rear] = newEntry;
    }
    else {
        rear++;
        this->items[rear] = newEntry;
    }
}

template<typename ItemType>
ItemType Queue<ItemType>::dequeue(){
    int result = 0;

    if(this->isEmpty()){
        return -1;
    }
    if(this->front == this->rear){
        result = this->items[front];
        this->items[front] = -1;
        rear = -1;
        front = -1;
        return result;
    }
    else {
        result = this->items[front];
        this->items[front] = -1;
        front++;
        return result;
    }
};

template<typename ItemType>
int Queue<ItemType>::size(){
    if(!this->isEmpty())
        return (this->rear - this->front + 1);
    else
        return 0;
}

template<typename ItemType>
void Queue<ItemType>::display(){
    cout << "The elements in Queue are as below: " << endl;
    for (int i = 0; i < MAX_QUEUE; i++){
        cout << items[i] << " ";
    }
    cout << endl << endl << endl;
}


int main(){

    Queue<int> q1;//Call default constructor

    int option, item;

    do {
        cout << "1. Enqueue" << endl;
        cout << "2. Dequeue" << endl;
        cout << "3. isEmpty" << endl;
        cout << "4. isFull" << endl;
        cout << "5. Count" << endl;
        cout << "6. Display" << endl;
        cout << "7. Clear Screen" << endl;
        cout << "0. Exit" << endl;
        cout << endl;

        cout << "Select Your Option Number: ";
        cin >> option;

        switch(option){
            cout << endl;
            case 0:
                break;
            case 1:
                cout << "Equeue your number(>=0): ";
                cin >> item;
                q1.enqueue(item);
                cout << endl << endl;
                break;
            case 2:
                cout << "Dequeue..." << endl;
                item = q1.dequeue();
                if (item == -1){
                    cout << "Queue is empty" << endl << endl;
                }
                else {
                    cout << "Dequeued element: " << item << endl << endl;
                }
                break;
            case 3:
                if(q1.isEmpty())
                    cout << "Queue is empty" << endl << endl;
                else
                    cout << "Queue is not empty" << endl << endl;
                break;
            case 4:
                if(q1.isFull())
                    cout << "Queue is full" << endl << endl;
                else
                    cout << "Queue is not full" << endl << endl;
                break;
            case 5:
                cout << "Count of queue: " << q1.size() << endl << endl;
                break;
            case 6:
                q1.display();
                break;
            case 7:
                system("cls");
                break;
            default:
                cout << "Please enter the proper option" << endl;
                break;
        }


    } while (option != 0);

    return 0;
}
```



Queue: LinkedList Implementation

dequeue()

```
class QueueLinkedListImpl{
  public:
    int dequeue(){
      if(front == NULL){
        cout << "Empty Queue" << endl;
        return -1;
      }
      Node *temp = front;
      int tempData = front->data;
      if(front == rear)
        front = rear = NULL;
      else
        front = front->next;

      delete temp;
      return tempData;
    }
};
```

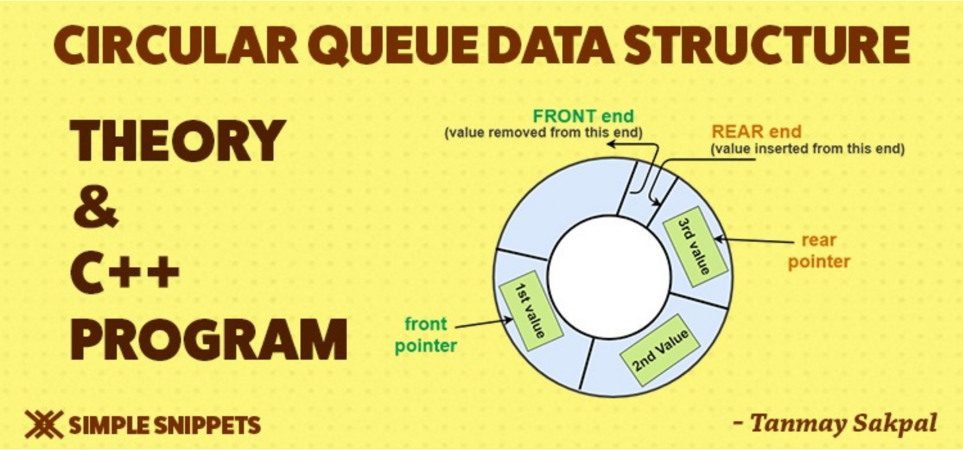
In [ ]:

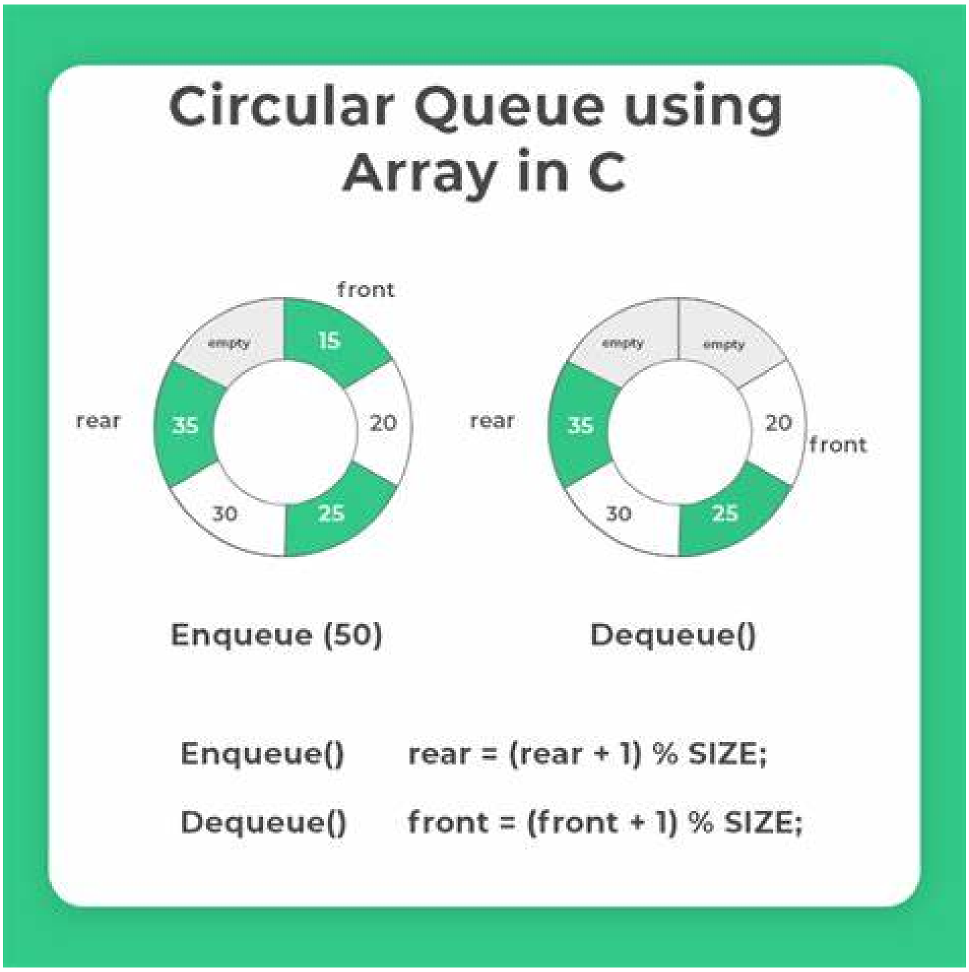
```
#include <iostream>
using namespace std;

class Node{
public:
    int data;
    Node* next;

    Node();
    Node(int d);
};

Node::Node(){
    data = 0;
    next = nullptr;
}

Node::Node(int d){
    data = d;
    next = nullptr;
}

class Queue{
public:
    Node* front;
    Node* rear;

    Queue();
    bool isEmpty();
    bool checkIfNodeExist(Node* newNode);
    void enqueue(Node* newNode);
    Node* dequeue();
    int count();
    void display();
};

Queue::Queue() : front(nullptr), rear(nullptr) {}

bool Queue::isEmpty(){
    if(front==nullptr && rear==nullptr)
        return true;
    else
```

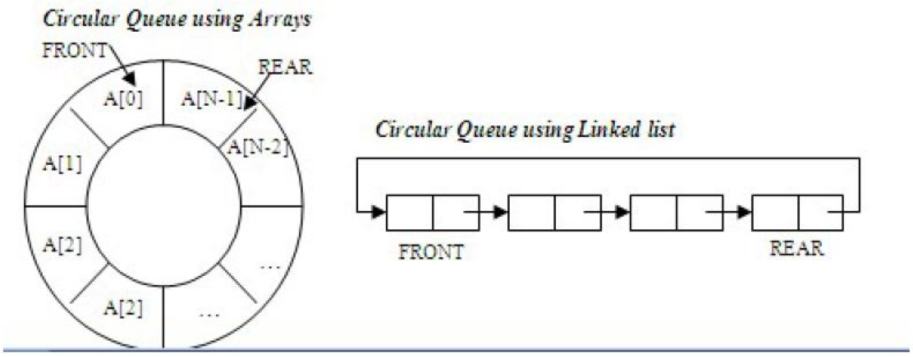
```
        return false;
}

bool Queue::checkIfNodeExist(Node* newNode){
    Node* ptr = front;
    while(ptr!=nullptr){
        if(ptr->data==newNode->data)
            return true;
        else
            ptr = ptr->next;
    }
    return false;
}

void Queue::enqueue(Node* newNode){
    if(checkIfNodeExist(newNode)){
        cout << "Node with value: " << newNode->data << " already exist" << endl;
        cout << "Enter different value" << endl << endl;
    }
    if(isEmpty()){
        front = newNode;
        rear = newNode;
        cout << "Enqueue first node successfully." << endl << endl;
    }
    else {
        rear->next = newNode;
        rear = newNode;
        cout << "Enqueue successfully" << endl << endl;
    }
}

Node* Queue::dequeue(){
    if(isEmpty()){
        cout << "Queue is empty" << endl << endl;
    }
    else if(front==rear){
        cout << "Dequeue Value: " << front->data << endl << endl;
        front = nullptr;
        rear = nullptr;
    }
    else {
        cout << "Dequeue Value: " << front->data << endl << endl;
        front = front->next;
    }
    return front;
}

int Queue::count(){
    if(isEmpty())
        return 0;
    else {
        int count = 0;
        Node* ptr = front;
        while(ptr!=nullptr){
            count++;
            ptr = ptr->next;
        }
        return count;
    }
}
```

```
void Queue::display(){
    if(isEmpty()){
        cout << "Queue is empty" << endl << endl;
    }
    else {
        cout << "Queue: " << endl;
        Node* ptr = front;
        while(ptr!=nullptr){
            cout << ptr->data << " -> ";
            ptr = ptr->next;
        }
    }
    cout << endl << endl;
}


int main(){
    Queue q;
    int option, data;

    do {
        cout << "What operation do you want to perform?" << endl;
        cout << "Select Option Number or 0 to Exit" << endl << endl;
        cout << "1. Enqueue" << endl;
        cout << "2. Dequeue" << endl;
        cout << "3. isEmpty" << endl;
        cout << "4. Count" << endl;
        cout << "8. Display" << endl;
        cout << "9. Clear Screen" << endl;
        cout << "0. Exit" << endl << endl;

        cout << "Enter Your Option Number: ";
        cout << endl;
        cin >> option;

        Node* newNode = new Node();

        switch(option){
            case 0:
                break;
            case 1:
                cout << "Enqueue Function Called." << endl;
                cout << "Enter Value of Node: ";
                cin >> data;
                newNode->data = data;
                q.enqueue(newNode);
                cout << endl;
                break;
            case 2:
                cout << "Dequeue Function Called." << endl;
                newNode = q.dequeue();
                delete newNode;
                cout << endl;
                break;
            case 3:
                if(q.isEmpty()){
                    cout << "Queue is Empty" << endl;
                }
                else {
                    cout << "Queue is not Empty" << endl;
                }
                break;
            case 4:
                cout << "Count Function Called." << endl;
                cout << "No of Node in the Queue: " << q.count() << endl << endl;
                break;
            case 8:
                q.display();
                cout << endl;
                break;
            case 9:
                system("cls");
                break;
            default:
                cout << "Please Enter the Proper Option Number" << endl;
                break;
        }




    } while(option!=0);


    return 0;
}
```

Circular Queue




Circular Queue using Array in C
front
empty
15
rear
35
20
30
25
empty
empty
rear
35
20
front
30
25
Enqueue (50)
Dequeue()
Enqueue() rear = (rear + 1) % SIZE;
Dequeue() front = (front + 1) % SIZE;

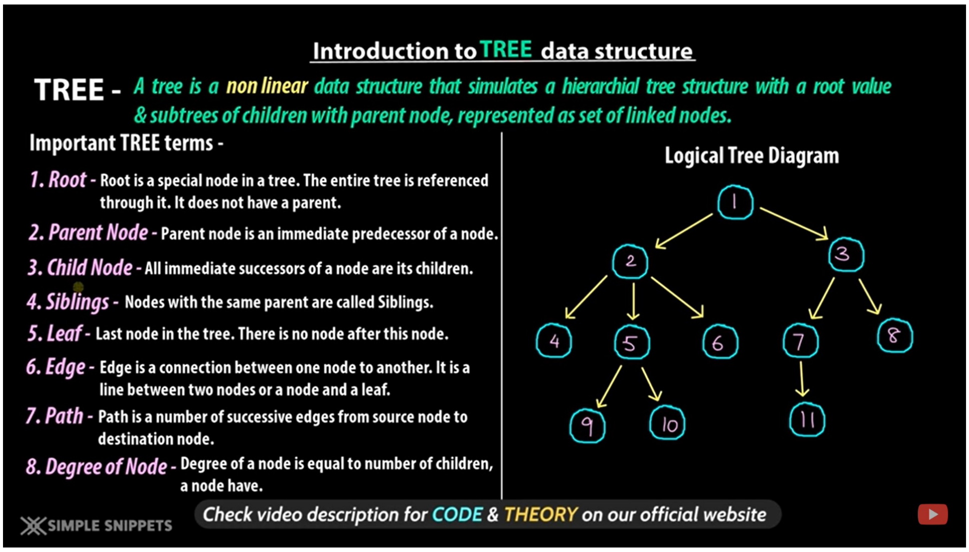


In [ ]:

```
#include <iostream>
#include <string>
#include <cmath>
using namespace std;

const int MAX_QUEUE = 5;


template<typename ItemType>
class CircularQueue {
private:
    ItemType items[MAX_QUEUE];
    int front;
    int rear;
    int itemCount;
public:
    CircularQueue();//default constructor
    ~CircularQueue();
    bool isEmpty() const;
    bool isFull() const;
    void enqueue(const ItemType& newEntry);
    ItemType dequeue();
    int size();
    void display();
};


template<typename ItemType>
CircularQueue<ItemType>::CircularQueue() {
    front = -1;
    rear = -1;
    itemCount = 0;

    for (int i = 0; i < MAX_QUEUE; i++){
        items[i] = -1;
    }
}

template<typename ItemType>
CircularQueue<ItemType>::~CircularQueue() {
    delete [] this->items;
}

template<typename ItemType>
bool CircularQueue<ItemType>::isEmpty() const{
    return ((front < 0) && (rear < 0));
};

template<typename ItemType>
bool CircularQueue<ItemType>::isFull() const {
    return (((this->rear + 1) % MAX_QUEUE) == this->front);
};

template<typename ItemType>
void CircularQueue<ItemType>::enqueue(const ItemType& newEntry){
    if(this->isFull()){
        cout << "Queue is full" << endl << endl;
        return;
    }
    if(this->isEmpty()){
        this->front = 0;
        this->rear = 0;
        this->items[rear] = newEntry;
    }
    else {
        rear = (rear + 1) % MAX_QUEUE;
        this->items[rear] = newEntry;
    }
    itemCount++;
}

template<typename ItemType>
ItemType CircularQueue<ItemType>::dequeue(){
    int result = 0;

    if(this->isEmpty()){
        return -1;
    }
    if(this->front == this->rear){
        result = this->items[front];
        this->items[front] = -1;
        rear = -1;
        front = -1;
        itemCount--;
        return result;
    }
    else {
        result = this->items[front];
        this->items[front] = -1;
        front = (front + 1) % MAX_QUEUE;
        itemCount--;
        return result;
    }
};

template<typename ItemType>
int CircularQueue<ItemType>::size(){
    if(!this->isEmpty())
        return itemCount;
    else
        return 0;
}

template<typename ItemType>
void CircularQueue<ItemType>::display(){
    cout << "The elements in Queue are as below: " << endl;
    for (int i = 0; i < MAX_QUEUE; i++){
        cout << items[i] << " ";
    }
    cout << endl << endl << endl;
}


int main(){

    CircularQueue<int> q1;//Call default constructor

    int option, item;

    do {
        cout << "1. Enqueue" << endl;
        cout << "2. Dequeue" << endl;
        cout << "3. isEmpty" << endl;
        cout << "4. isFull" << endl;
        cout << "5. Count" << endl;
        cout << "6. Display" << endl;
        cout << "7. Clear Screen" << endl;
        cout << "0. Exit" << endl;
        cout << endl;

        cout << "Select Your Option Number: ";
        cin >> option;

        switch(option){
            cout << endl;
            case 0:
                break;
            case 1:
                cout << "Equeue your number(>=0): ";
                cin >> item;
                q1.enqueue(item);
                cout << endl << endl;
                break;
            case 2:
                cout << "Dequeue..." << endl;
                item = q1.dequeue();
                if (item == -1){
                    cout << "Queue is empty" << endl << endl;
                }
                else {
                    cout << "Dequeued element: " << item << endl << endl;
                }
                break;
            case 3:
                if(q1.isEmpty())
                    cout << "Queue is empty" << endl << endl;
                else
                    cout << "Queue is not empty" << endl << endl;
                break;
            case 4:
                if(q1.isFull())
                    cout << "Queue is full" << endl << endl;
                else
                    cout << "Queue is not full" << endl << endl;
                break;
            case 5:
                cout << "Count of queue: " << q1.size() << endl << endl;
                break;
            case 6:
                q1.display();
                break;
            case 7:
                system("cls");
                break;
            default:
                cout << "Please enter the proper option" << endl;
                break;
        }


    } while (option != 0);

    return 0;
}
```

Non-Linear Data Structue

Tree



Introduction to TREE data structure

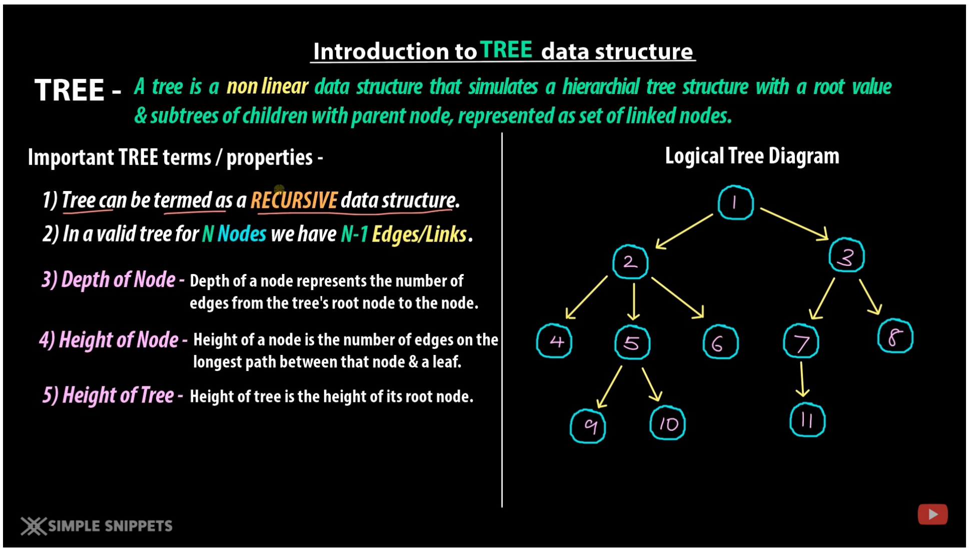
TREE - A tree is a non linear data structure that simulates a hierarchial tree structure with a root value & subtrees of children with parent node, represented as set of linked nodes.

Important TREE terms -

1. Root - Root is a special node in a tree. The entire tree is referenced through it. It does not have a parent.
2. Parent Node - Parent node is an immediate predecessor of a node.
3. Child Node - All immediate successors of a node are its children.
4. Siblings - Nodes with the same parent are called Siblings.
5. Leaf - Last node in the tree. There is no node after this node.
6. Edge - Edge is a connection between one node to another. It is a line between two nodes or a node and a leaf.
7. Path - Path is a number of successive edges from source node to destination node.
8. Degree of Node - Degree of a node is equal to number of children, a node have.

Logical Tree Diagram

Check video description for CODE & THEORY on our official website



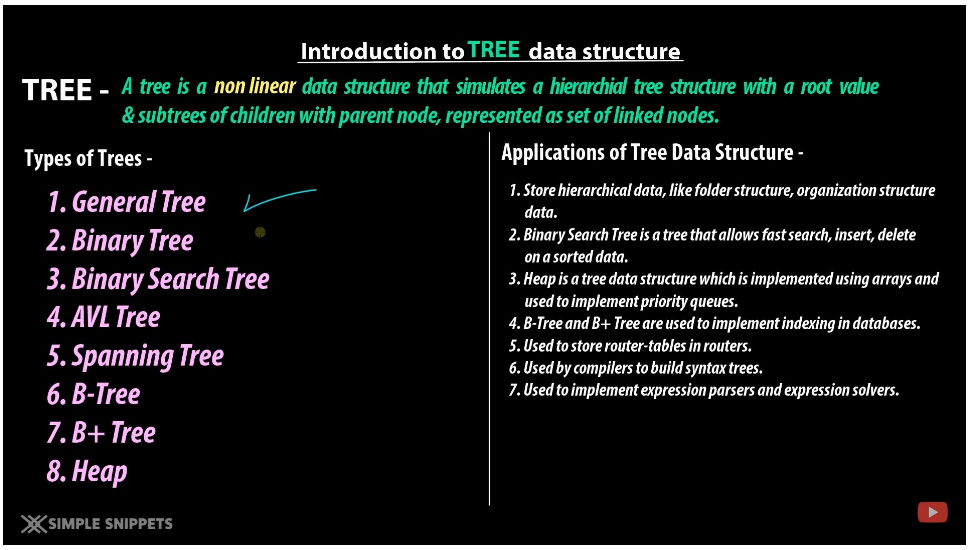
Introduction to TREE data structure

TREE - A tree is a non linear data structure that simulates a hierarchial tree structure with a root value & subtrees of children with parent node, represented as set of linked nodes.

Important TREE terms / properties -

1) Tree can be termed as a RECURSIVE data structure.
2) In a valid tree for N Nodes we have N-1 Edges/Links.
3) Depth of Node - Depth of a node represents the number of edges from the tree's root node to the node.
4) Height of Node - Height of a node is the number of edges on the longest path between that node & a leaf.
5) Height of Tree - Height of tree is the height of its root node.

Logical Tree Diagram



Introduction to TREE data structure

TREE - A tree is a non linear data structure that simulates a hierarchial tree structure with a root value & subtrees of children with parent node, represented as set of linked nodes.

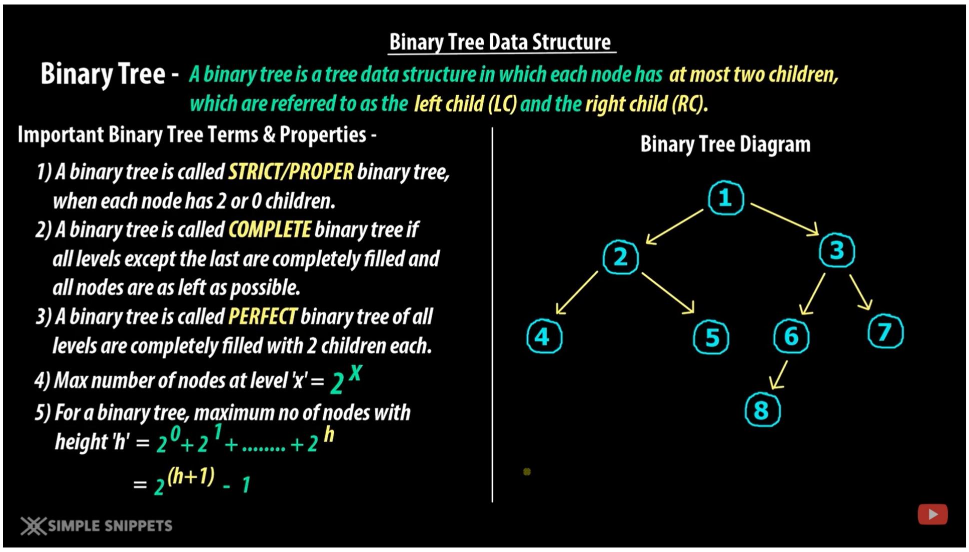
Types of Trees -

1. General Tree
2. Binary Tree
3. Binary Search Tree
4. AVL Tree
5. Spanning Tree
6. B-Tree
7. B+ Tree
8. Heap

Applications of Tree Data Structure -

1. Store hierarchical data, like folder structure, organization structure data.
2. Binary Search Tree is a tree that allows fast search, insert, delete on a sorted data.
3. Heap is a tree data structure which is implemented using arrays and used to implement priority queues.
4. B-Tree and B+ Tree are used to implement indexing in databases.
5. Used to store router-tables in routers.
6. Used by compilers to build syntax trees.
7. Used to implement expression parsers and expression solvers.

Binary Tree



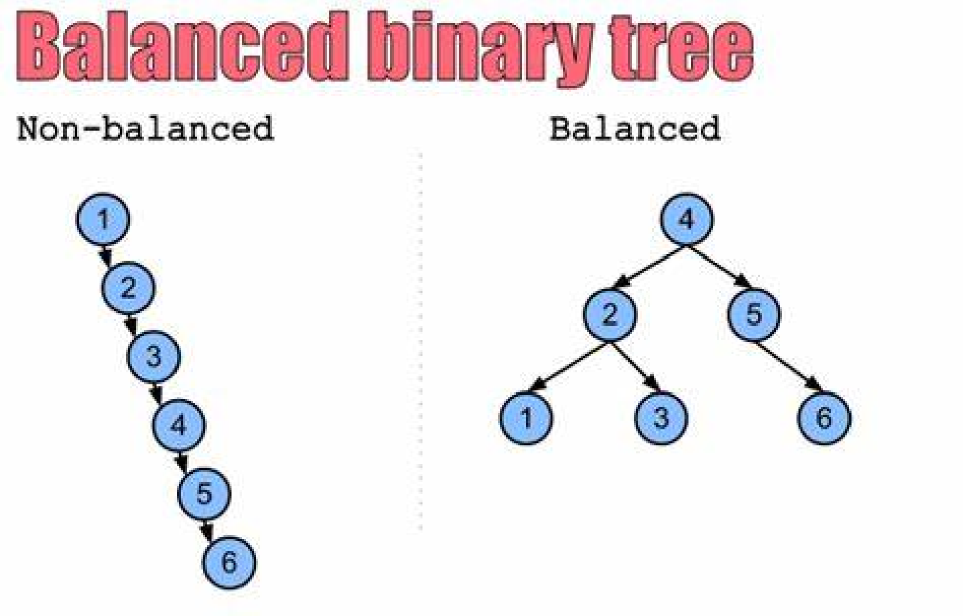
Binary Tree Data Structure

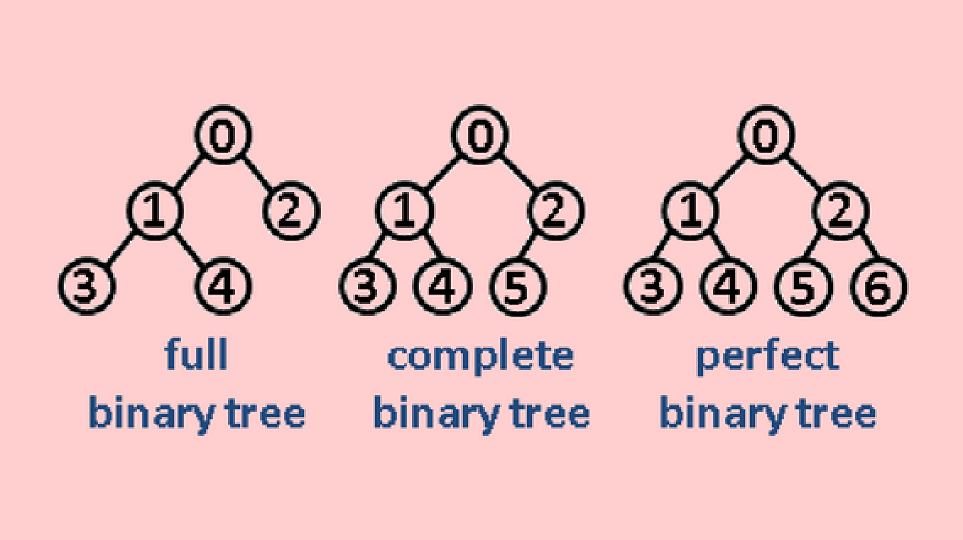
Binary Tree - *A binary tree is a tree data structure in which each node has at most two children, which are referred to as the left child (LC) and the right child (RC).*

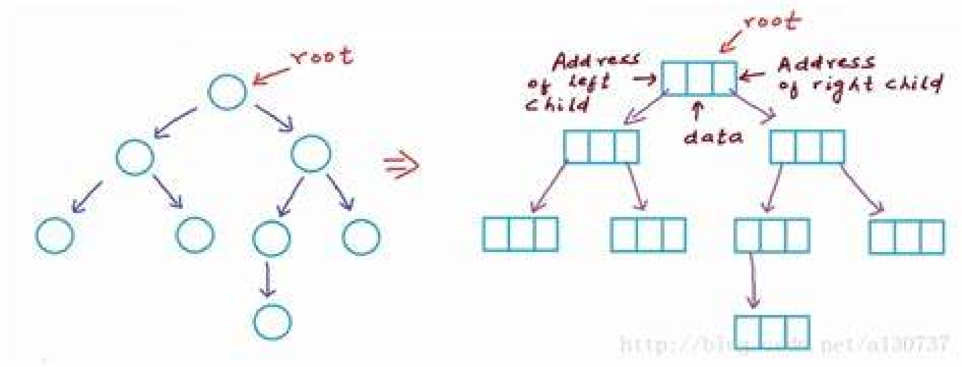
Important Binary Tree Terms & Properties -

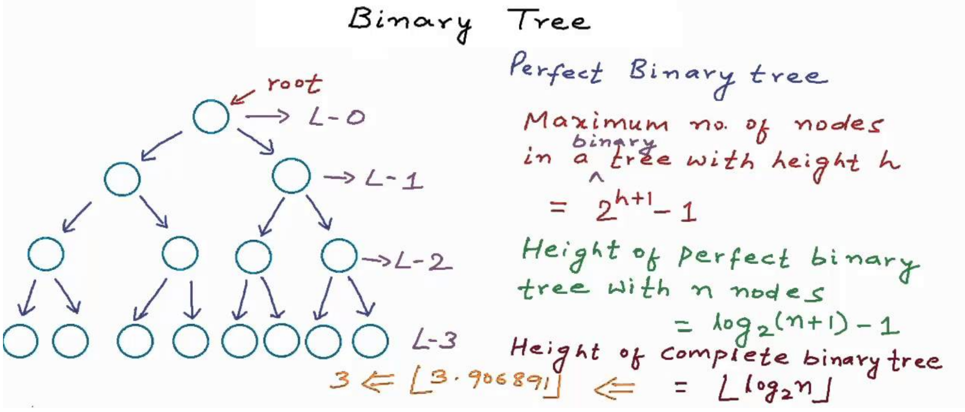
1) *A binary tree is called STRICT/PROPER binary tree, when each node has 2 or 0 children.*
2) *A binary tree is called COMPLETE binary tree if all levels except the last are completely filled and all nodes are as left as possible.*
3) *A binary tree is called PERFECT binary tree of all levels are completely filled with 2 children each.*
4) *Max number of nodes at level 'x' =* 2^x
5) *For a binary tree, maximum no of nodes with height 'h' =* $2^0 + 2^1 + \ldots\ldots + 2^h$

$= 2^{(h+1)} - 1$







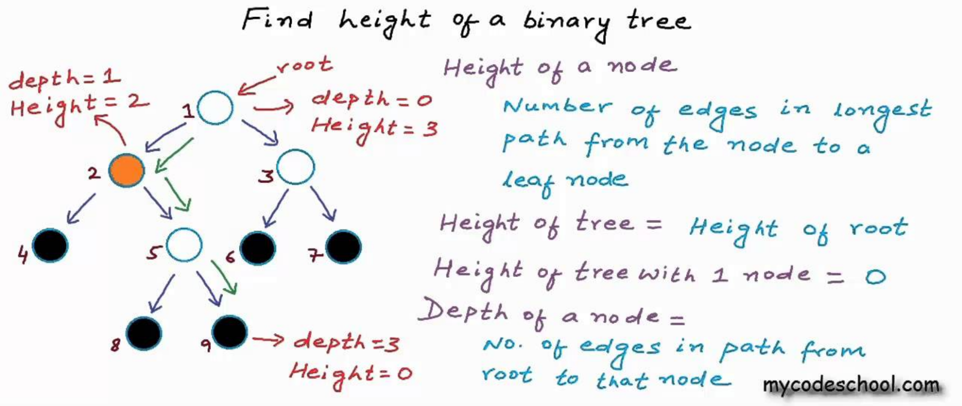

Binary Tree

Perfect Binary tree

Maximum no. of nodes in a binary tree with height h $= 2^{h+1} - 1$

Height of perfect binary tree with n nodes $= \log_2(n+1) - 1$

Height of complete binary tree $= \lfloor \log_2 n \rfloor$


Find height of a binary tree

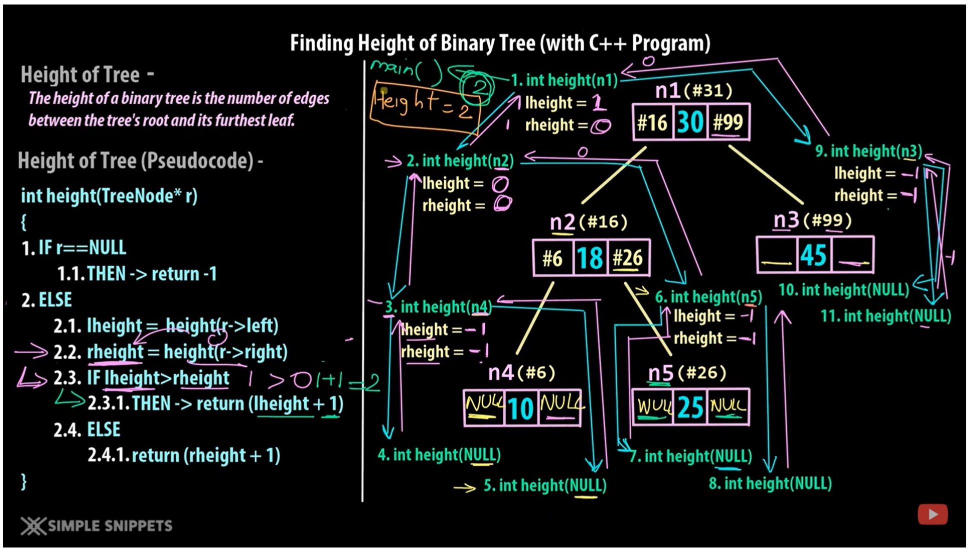
Height of a node

Number of edges in longest path from the node to a leaf node

Height of tree = Height of root

Height of tree with 1 node = 0

Depth of a node = No. of edges in path from root to that node

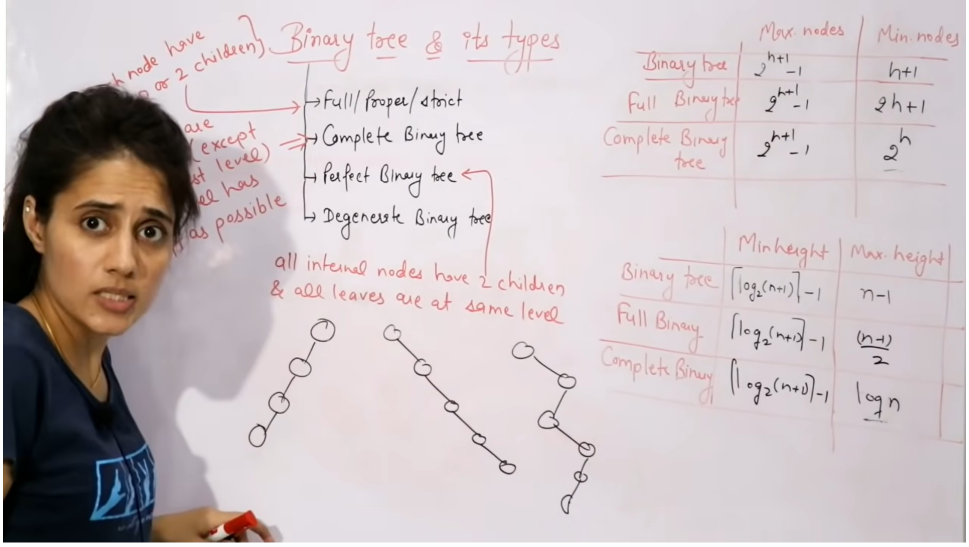

Finding Height of Binary Tree (with C++ Program)

Height of Tree -

The height of a binary tree is the number of edges between the tree's root and its furthest leaf.

Height of Tree (Pseudocode) -

```
int height(TreeNode* r)
{
1. IF r==NULL
      1.1. THEN -> return -1
2. ELSE
      2.1. lheight = height(r->left)
      2.2. rheight = height(r->right)
      2.3. IF lheight>rheight      1 > 0   1+1 = 2
            2.3.1. THEN -> return (lheight + 1)
      2.4. ELSE
            2.4.1. return (rheight + 1)
}
```



h node have
o or 2 children}

Binary tree & its types
- Full/Proper/strict
- Complete Binary tree
- Perfect Binary tree
- Degenerate Binary tree

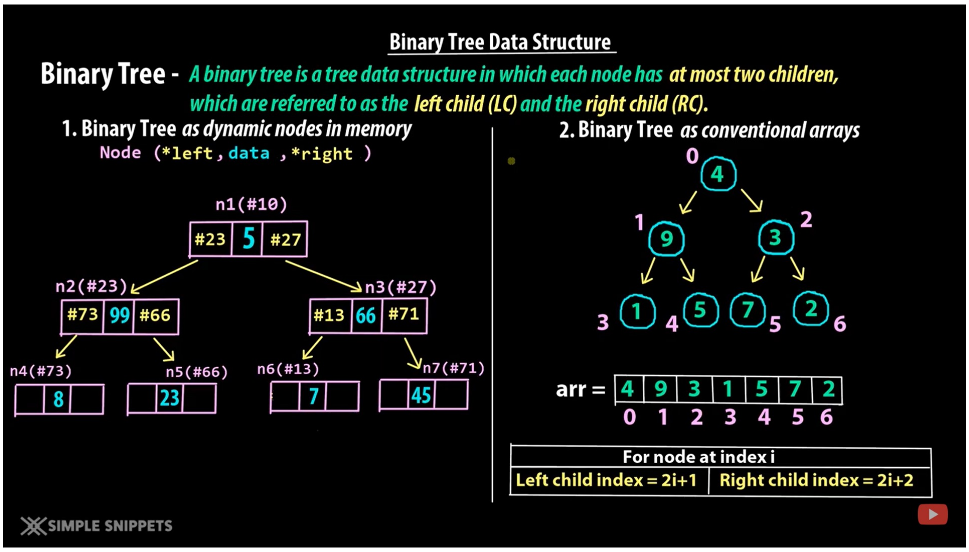
are
(except
st level)
el has
as possible

all internal nodes have 2 children
& all leaves are at same level

	Max. nodes	Min. nodes
Binary tree	$2^{h+1}-1$	$h+1$
Full Binary tree	$2^{h+1}-1$	$2h+1$
Complete Binary tree	$2^{h+1}-1$	2^h

	Min height	Max. height
Binary tree	$\lceil \log_2(n+1) \rceil - 1$	$n-1$
Full Binary	$\lceil \log_2(n+1) \rceil - 1$	$\frac{(n-1)}{2}$
Complete Binary	$\lceil \log_2(n+1) \rceil - 1$	$\log n$

BT Implementation



Binary Tree Data Structure

Binary Tree - *A binary tree is a tree data structure in which each node has at most two children, which are referred to as the left child (LC) and the right child (RC).*

1. Binary Tree *as dynamic nodes in memory*

```
Node (*left, data, *right )
```

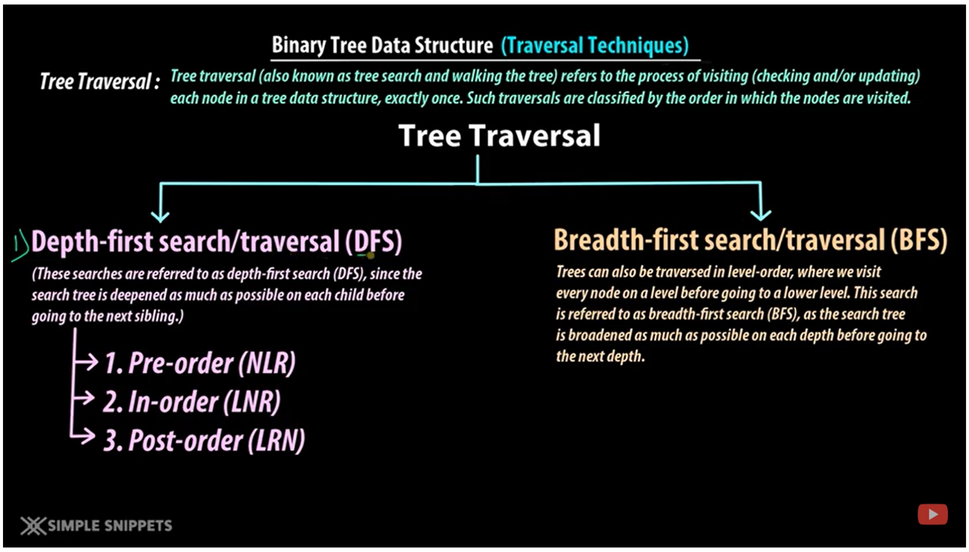
2. Binary Tree *as conventional arrays*

arr = | 4 | 9 | 3 | 1 | 5 | 7 | 2 |
0 1 2 3 4 5 6

For node at index i	
Left child index = 2i+1	Right child index = 2i+2

SIMPLE SNIPPETS

BT Traversal



Binary Tree Data Structure (Traversal Techniques)

Tree Traversal : *Tree traversal (also known as tree search and walking the tree) refers to the process of visiting (checking and/or updating) each node in a tree data structure, exactly once. Such traversals are classified by the order in which the nodes are visited.*

Tree Traversal

Depth-first search/traversal (DFS)

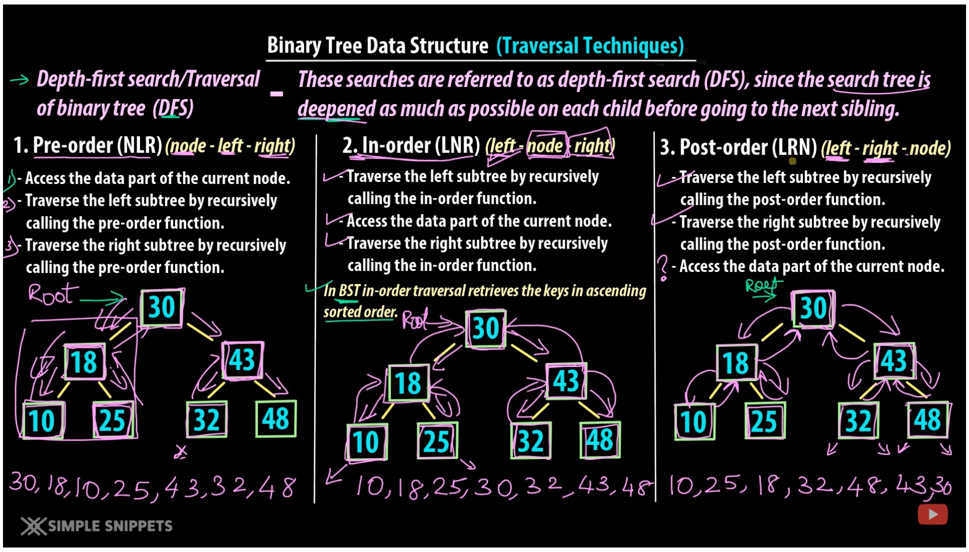
(These searches are referred to as depth-first search (DFS), since the search tree is deepened as much as possible on each child before going to the next sibling.)

- *1. Pre-order (NLR)*
- *2. In-order (LNR)*
- *3. Post-order (LRN)*

Breadth-first search/traversal (BFS)

Trees can also be traversed in level-order, where we visit every node on a level before going to a lower level. This search is referred to as breadth-first search (BFS), as the search tree is broadened as much as possible on each depth before going to the next depth.

SIMPLE SNIPPETS



Binary Tree Data Structure (Traversal Techniques)

→ *Depth-first search/Traversal of binary tree (DFS)* - *These searches are referred to as depth-first search (DFS), since the search tree is deepened as much as possible on each child before going to the next sibling.*

1. Pre-order (NLR) *(node - left - right)*

1) - Access the data part of the current node.
2) - Traverse the left subtree by recursively calling the pre-order function.
3) - Traverse the right subtree by recursively calling the pre-order function.

30, 18, 10, 25, 43, 32, 48

2. In-order (LNR) *(left - node - right)*

- Traverse the left subtree by recursively calling the in-order function.
- Access the data part of the current node.
- Traverse the right subtree by recursively calling the in-order function.

In BST in-order traversal retrieves the keys in ascending sorted order.

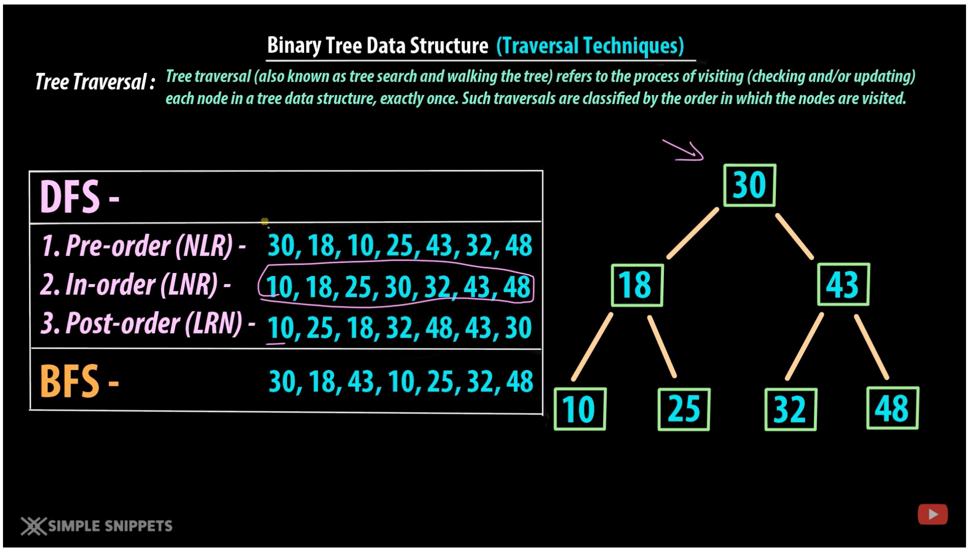
10, 18, 25, 30, 32, 43, 48

3. Post-order (LRN) *(left - right - node)*

- Traverse the left subtree by recursively calling the post-order function.
- Traverse the right subtree by recursively calling the post-order function.
? - Access the data part of the current node.

10, 25, 18, 32, 48, 43, 30

SIMPLE SNIPPETS

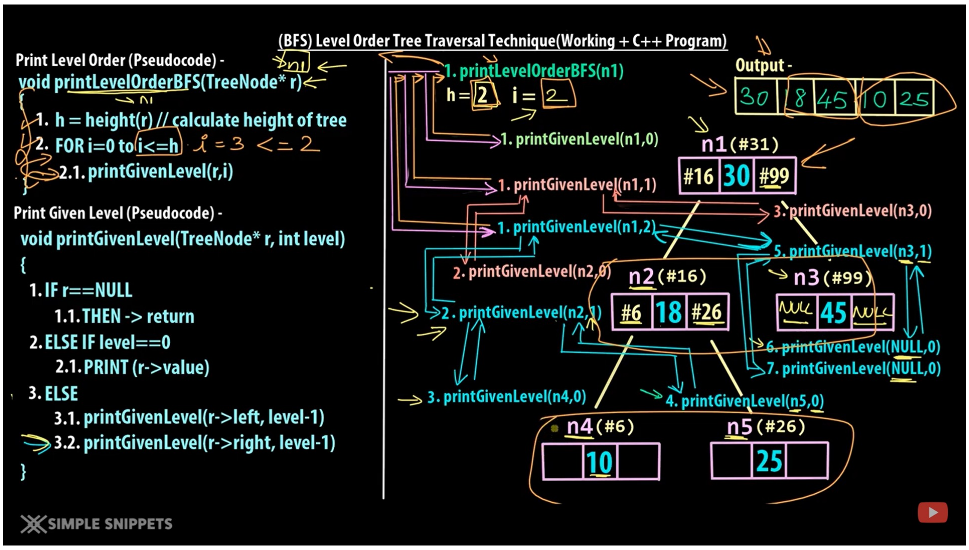


Binary Tree Data Structure (Traversal Techniques)

Tree Traversal : *Tree traversal (also known as tree search and walking the tree) refers to the process of visiting (checking and/or updating) each node in a tree data structure, exactly once. Such traversals are classified by the order in which the nodes are visited.*

DFS -	
1. Pre-order (NLR) -	30, 18, 10, 25, 43, 32, 48
2. In-order (LNR) -	10, 18, 25, 30, 32, 43, 48
3. Post-order (LRN) -	10, 25, 18, 32, 48, 43, 30
BFS -	30, 18, 43, 10, 25, 32, 48

SIMPLE SNIPPETS



(BFS) Level Order Tree Traversal Technique(Working + C++ Program)

Print Level Order (Pseudocode) -

```
void printLevelOrderBFS(TreeNode* r)
{
  1. h = height(r) // calculate height of tree
  2. FOR i=0 to i<=h
      2.1. printGivenLevel(r,i)
}
```

n1

i = 3 <= 2

Print Given Level (Pseudocode) -

```
void printGivenLevel(TreeNode* r, int level)
{
  1. IF r==NULL
      1.1. THEN -> return
  2. ELSE IF level==0
      2.1. PRINT (r->value)
  3. ELSE
      3.1. printGivenLevel(r->left, level-1)
      3.2. printGivenLevel(r->right, level-1)
}
```

1. printLevelOrderBFS(n1)
h = 2 i = 2

1. printGivenLevel(n1,0)

1. printGivenLevel(n1,1)

1. printGivenLevel(n1,2)

2. printGivenLevel(n2,0)

2. printGivenLevel(n2,1)

3. printGivenLevel(n4,0)

4. printGivenLevel(n5,0)

3. printGivenLevel(n3,0)

5. printGivenLevel(n3,1)

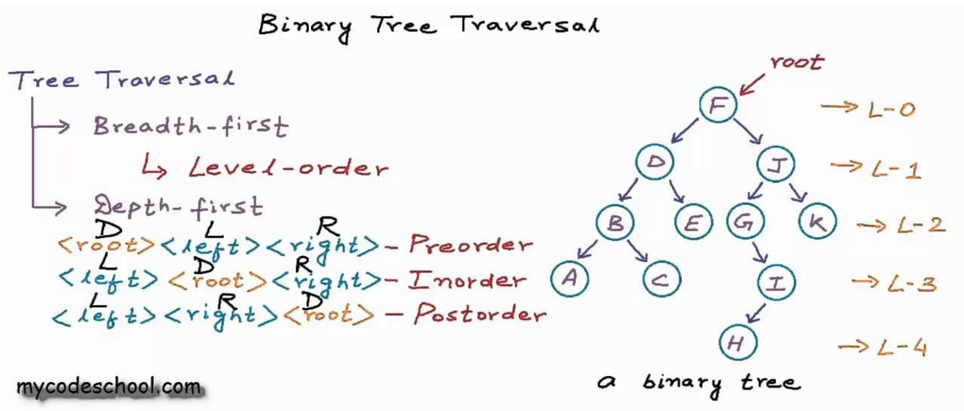
6. printGivenLevel(NULL,0)

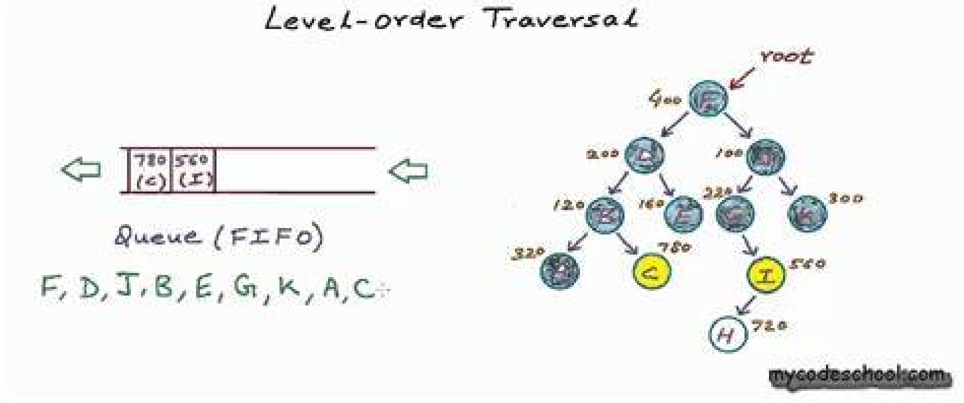
7. printGivenLevel(NULL,0)

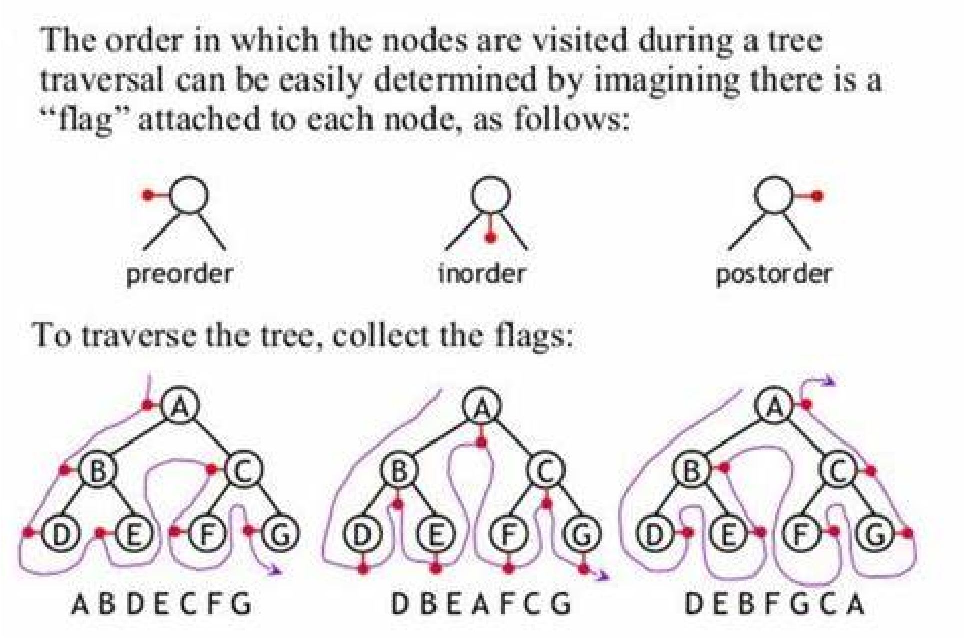
Output -

30	18	45	10	25

SIMPLE SNIPPETS

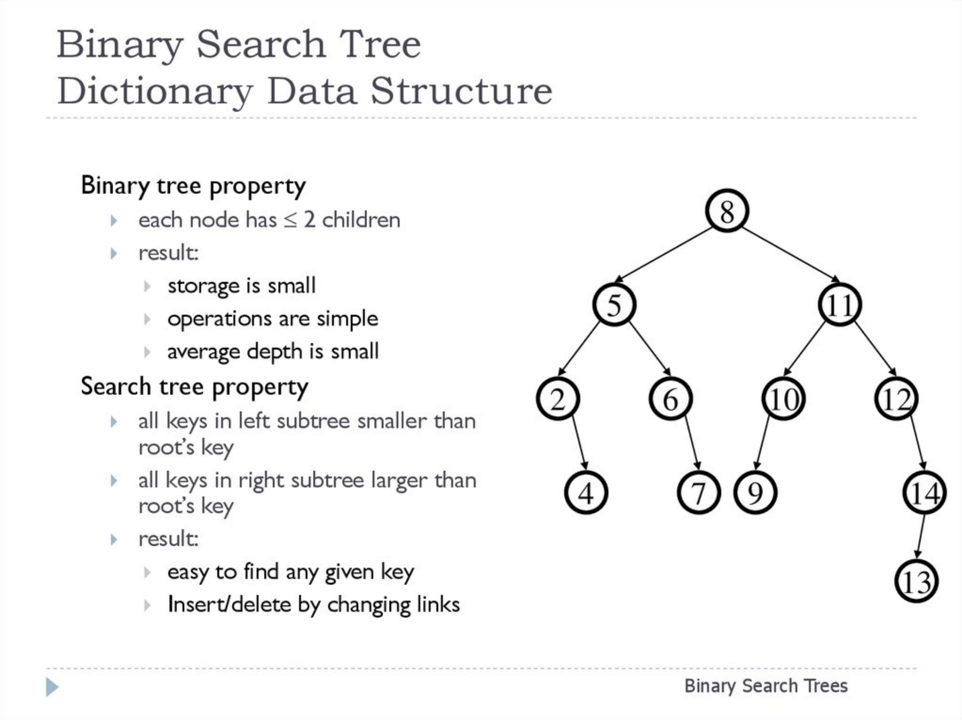





The order in which the nodes are visited during a tree traversal can be easily determined by imagining there is a "flag" attached to each node, as follows:

To traverse the tree, collect the flags:

BST vs BT



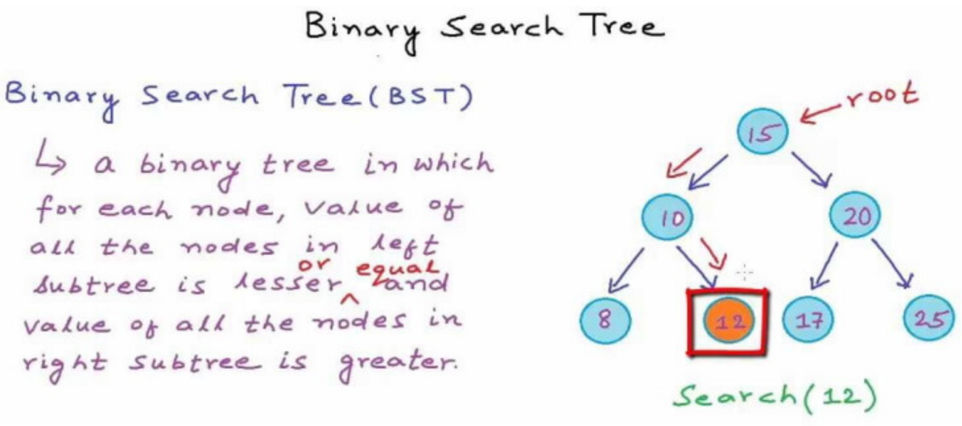
Binary Search Tree
Dictionary Data Structure

Binary tree property
- each node has ≤ 2 children
- result:
  - storage is small
  - operations are simple
  - average depth is small

Search tree property
- all keys in left subtree smaller than root's key
- all keys in right subtree larger than root's key
- result:
  - easy to find any given key
  - Insert/delete by changing links

Binary Search Trees

Binary Search Tree/BST

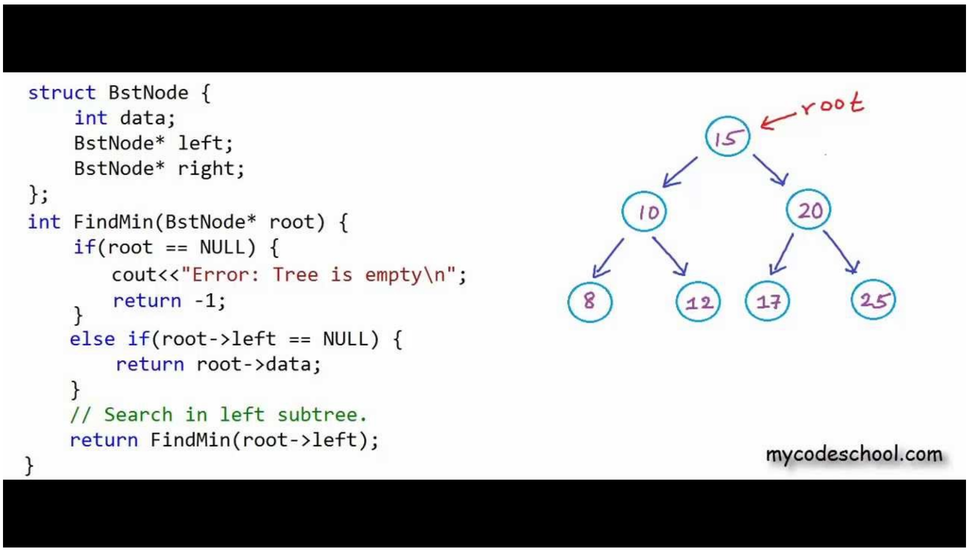


Binary Search Tree

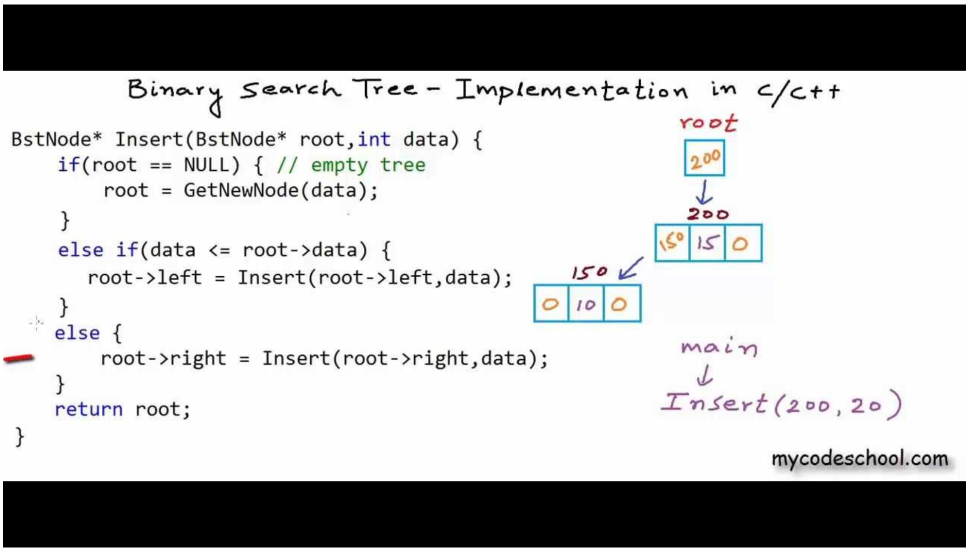
Binary Search Tree(BST)

↳ a binary tree in which for each node, value of all the nodes in left subtree is lesser or equal and value of all the nodes in right subtree is greater.

Search(12)

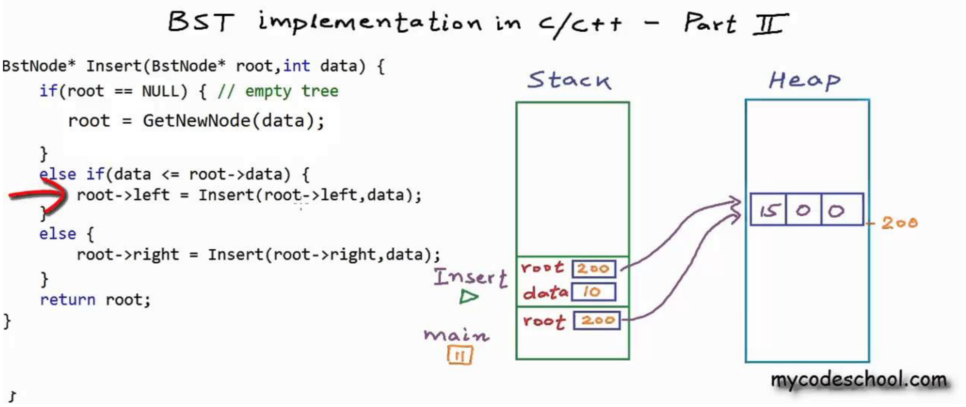


```
struct BstNode {
    int data;
    BstNode* left;
    BstNode* right;
};
int FindMin(BstNode* root) {
    if(root == NULL) {
        cout<<"Error: Tree is empty\n";
        return -1;
    }
    else if(root->left == NULL) {
        return root->data;
    }
    // Search in left subtree.
    return FindMin(root->left);
}
```



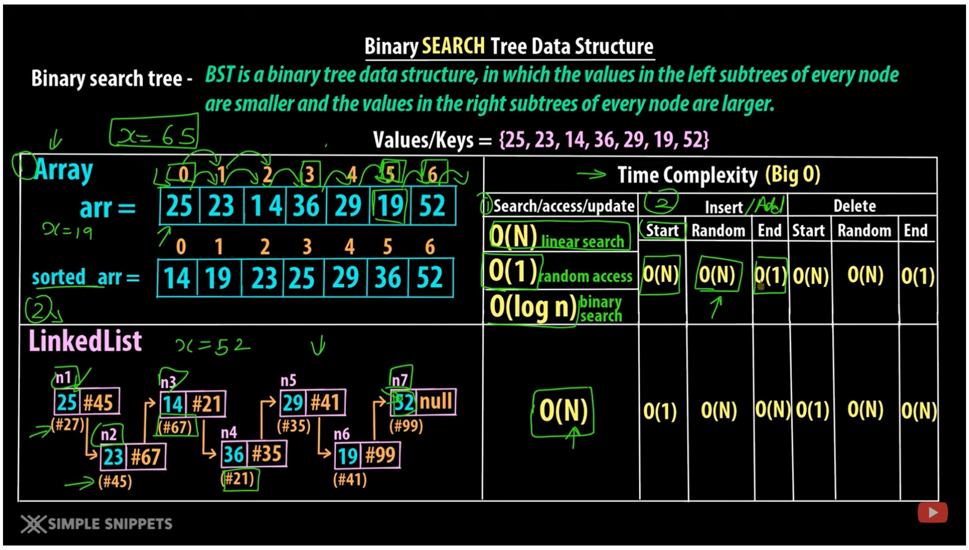
Binary Search Tree - Implementation in c/c++

```
BstNode* Insert(BstNode* root,int data) {
    if(root == NULL) { // empty tree
        root = GetNewNode(data);
    }
    else if(data <= root->data) {
      root->left = Insert(root->left,data);
    }
    else {
        root->right = Insert(root->right,data);
    }
    return root;
}
```



BST implementation in c/c++ - Part II

```
BstNode* Insert(BstNode* root,int data) {
    if(root == NULL) { // empty tree
      root = GetNewNode(data);
    }
    else if(data <= root->data) {
        root->left = Insert(root->left,data);
    }
    else {
        root->right = Insert(root->right,data);
    }
    return root;
}
```



Binary SEARCH Tree Data Structure

Binary search tree - *BST is a binary tree data structure, in which the values in the left subtrees of every node are smaller and the values in the right subtrees of every node are larger.*

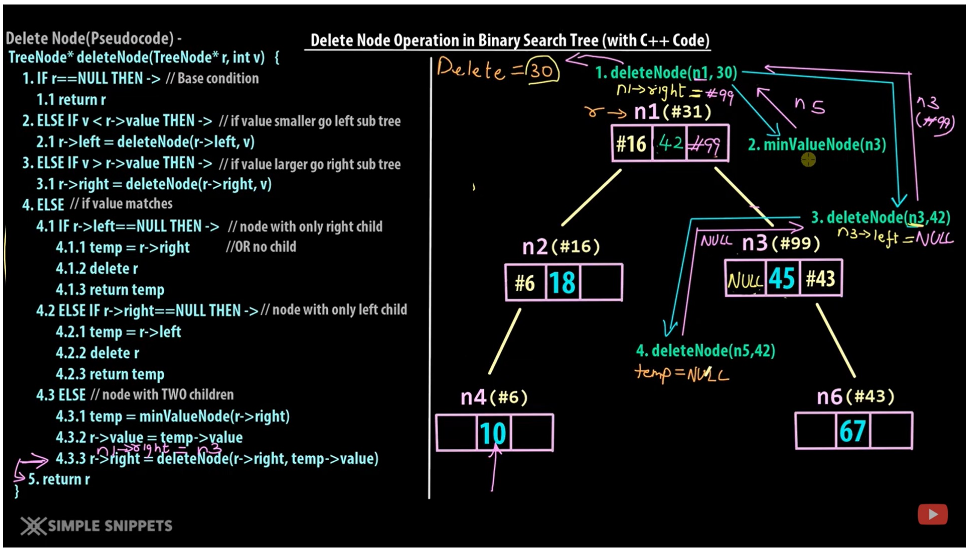
Values/Keys = {25, 23, 14, 36, 29, 19, 52}

	0	1	2	3	4	5	6
arr =	25	23	14	36	29	19	52
sorted_arr =	14	19	23	25	29	36	52

Time Complexity (Big 0)

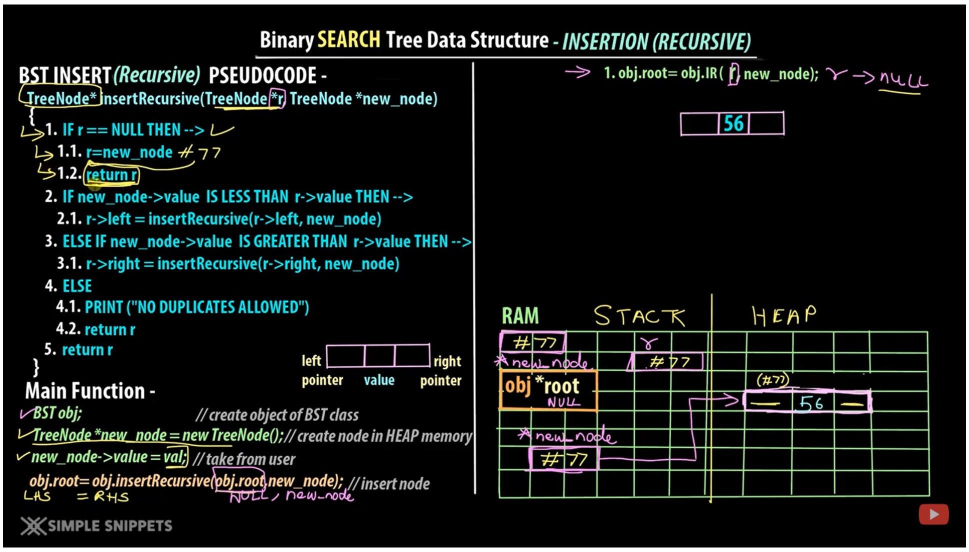
	Search/access/update	Insert/Add			Delete		
		Start	Random	End	Start	Random	End
Array	O(N) linear search O(1) random access O(log n) binary search	O(N)	O(N)	O(1)	O(N)	O(N)	O(1)
LinkedList	O(N)	O(1)	O(N)	O(N)	O(1)	O(N)	O(N)

Insert Node - Recursive



Delete Node(Pseudocode) -

```
TreeNode* deleteNode(TreeNode* r, int v)  {
  1. IF r==NULL THEN -> // Base condition
     1.1 return r
  2. ELSE IF v < r->value THEN -> // if value smaller go left sub tree
     2.1 r->left = deleteNode(r->left, v)
  3. ELSE IF v > r->value THEN -> // if value larger go right sub tree
     3.1 r->right = deleteNode(r->right, v)
  4. ELSE // if value matches
     4.1 IF r->left==NULL THEN -> // node with only right child
        4.1.1 temp = r->right        //OR no child
        4.1.2 delete r
        4.1.3 return temp
     4.2 ELSE IF r->right==NULL THEN -> // node with only left child
        4.2.1 temp = r->left
        4.2.2 delete r
        4.2.3 return temp
     4.3 ELSE // node with TWO children
        4.3.1 temp = minValueNode(r->right)
        4.3.2 r->value = temp->value
              n1->right = n3
        4.3.3 r->right = deleteNode(r->right, temp->value)
  5. return r
}
```



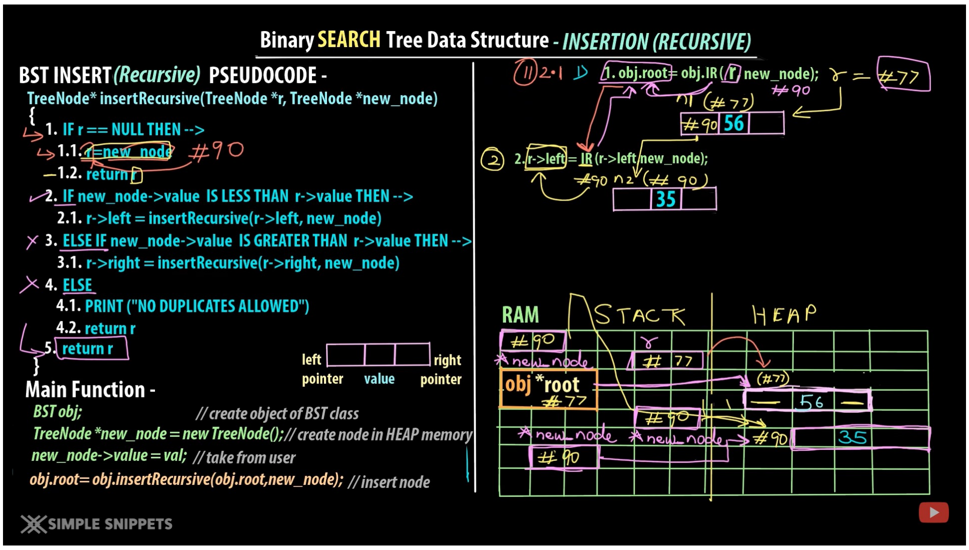
Binary SEARCH Tree Data Structure - INSERTION (RECURSIVE)

BST INSERT (Recursive) PSEUDOCODE -

```
TreeNode* insertRecursive(TreeNode *r, TreeNode *new_node)
{
  1. IF r == NULL THEN -->
     1.1. r=new_node #77
     1.2. return r
  2. IF new_node->value IS LESS THAN r->value THEN -->
     2.1. r->left = insertRecursive(r->left, new_node)
  3. ELSE IF new_node->value IS GREATER THAN r->value THEN -->
     3.1. r->right = insertRecursive(r->right, new_node)
  4. ELSE
     4.1. PRINT ("NO DUPLICATES ALLOWED")
     4.2. return r
  5. return r
}
```

Main Function -

```
BST obj;                                   // create object of BST class
TreeNode *new_node = new TreeNode();// create node in HEAP memory
new_node->value = val;  // take from user
obj.root= obj.insertRecursive(obj.root,new_node); // insert node
LHS     = RHS           NULL , new_node
```



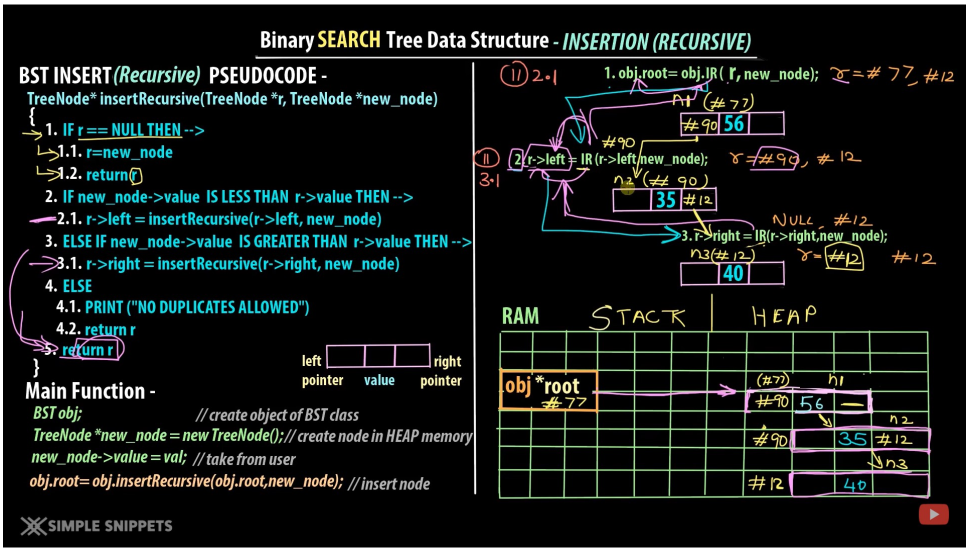
Binary SEARCH Tree Data Structure - INSERTION (RECURSIVE)

BST INSERT (Recursive) PSEUDOCODE -

```
TreeNode* insertRecursive(TreeNode *r, TreeNode *new_node)
{
  1. IF r == NULL THEN -->
     1.1. r=new_node #90
     1.2. return r
  2. IF new_node->value IS LESS THAN r->value THEN -->
     2.1. r->left = insertRecursive(r->left, new_node)
  3. ELSE IF new_node->value IS GREATER THAN r->value THEN -->
     3.1. r->right = insertRecursive(r->right, new_node)
  4. ELSE
     4.1. PRINT ("NO DUPLICATES ALLOWED")
     4.2. return r
  5. return r
}
```

Main Function -

```
BST obj;                                   // create object of BST class
TreeNode *new_node = new TreeNode();// create node in HEAP memory
new_node->value = val;  // take from user
obj.root= obj.insertRecursive(obj.root,new_node); // insert node
```



Binary SEARCH Tree Data Structure - *INSERTION (RECURSIVE)*

BST INSERT *(Recursive)* PSEUDOCODE -

```
TreeNode* insertRecursive(TreeNode *r, TreeNode *new_node)
{
  1. IF r == NULL THEN -->
    1.1. r=new_node
    1.2. return r
  2. IF new_node->value IS LESS THAN r->value THEN -->
    2.1. r->left = insertRecursive(r->left, new_node)
  3. ELSE IF new_node->value IS GREATER THAN r->value THEN -->
    3.1. r->right = insertRecursive(r->right, new_node)
  4. ELSE
    4.1. PRINT ("NO DUPLICATES ALLOWED")
    4.2. return r
  5. return r
}
```

Main Function -

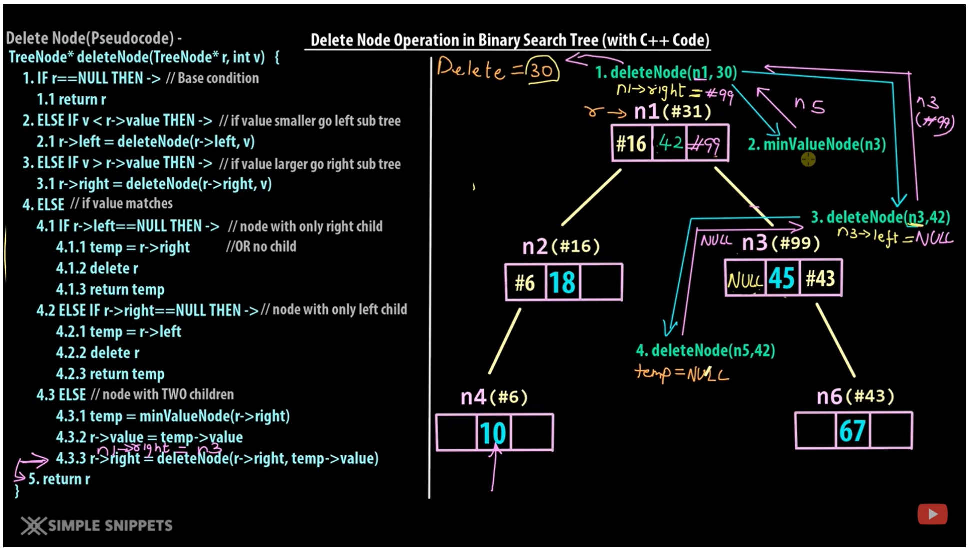
```
BST obj;                       // create object of BST class
TreeNode *new_node = new TreeNode();// create node in HEAP memory
new_node->value = val;  // take from user
obj.root= obj.insertRecursive(obj.root,new_node);  // insert node
```

1. obj.root= obj.IR(r, new_node);

2. r->left = IR (r->left, new_node);

3. r->right = IR(r->right,new_node);

Delete Node



Delete Node(Pseudocode) -

```
TreeNode* deleteNode(TreeNode* r, int v)  {
  1. IF r==NULL THEN -> // Base condition
    1.1 return r
  2. ELSE IF v < r->value THEN -> // if value smaller go left sub tree
    2.1 r->left = deleteNode(r->left, v)
  3. ELSE IF v > r->value THEN -> // if value larger go right sub tree
    3.1 r->right = deleteNode(r->right, v)
  4. ELSE // if value matches
    4.1 IF r->left==NULL THEN -> // node with only right child
      4.1.1 temp = r->right        //OR no child
      4.1.2 delete r
      4.1.3 return temp
    4.2 ELSE IF r->right==NULL THEN -> // node with only left child
      4.2.1 temp = r->left
      4.2.2 delete r
      4.2.3 return temp
    4.3 ELSE // node with TWO children
      4.3.1 temp = minValueNode(r->right)
      4.3.2 r->value = temp->value
      4.3.3 r->right = deleteNode(r->right, temp->value)
  5. return r
}
```

Delete Node Operation in Binary Search Tree (with C++ Code)

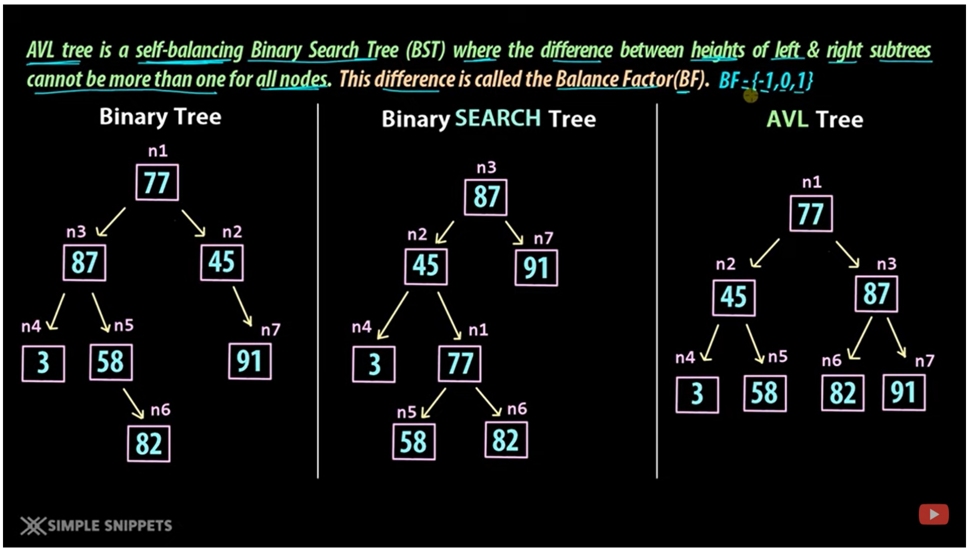
1. deleteNode(n1, 30)

2. minValueNode(n3)

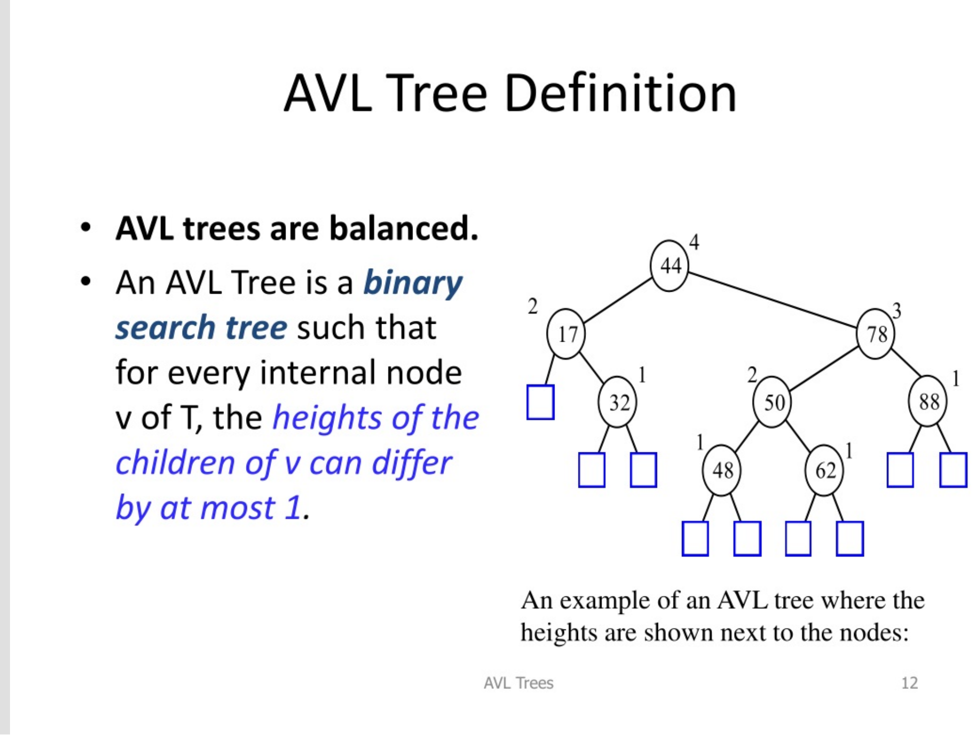
3. deleteNode(n3,42)

4. deleteNode(n5,42)

AVL Tree



AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left & right subtrees cannot be more than one for all nodes. This difference is called the Balance Factor(BF). BF = {-1,0,1}

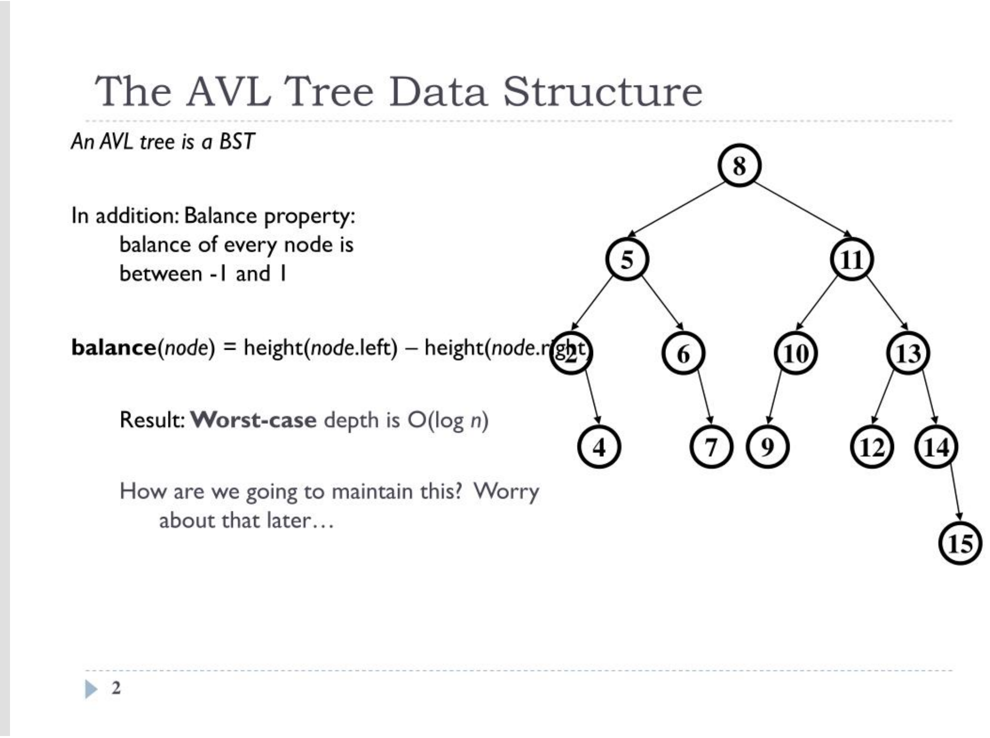


AVL Tree Definition

- **AVL trees are balanced.**
- An AVL Tree is a ***binary search tree*** such that for every internal node v of T, the *heights of the children of v can differ by at most 1*.

An example of an AVL tree where the heights are shown next to the nodes:

AVL Trees 12



The AVL Tree Data Structure

An AVL tree is a BST

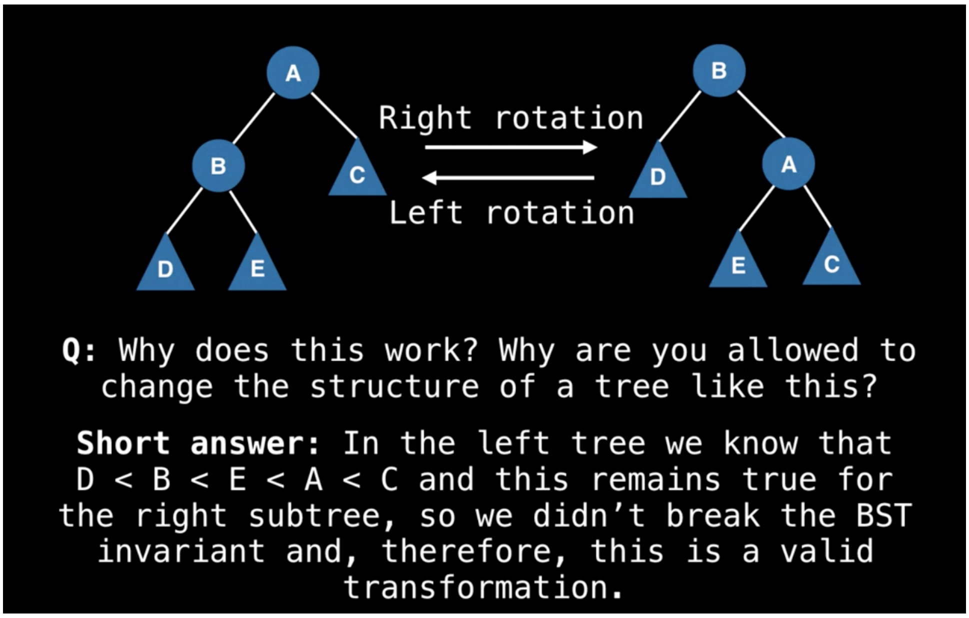
In addition: Balance property:
balance of every node is
between -1 and 1

balance(*node*) = height(*node*.left) – height(*node*.right)

Result: **Worst-case** depth is O(log *n*)

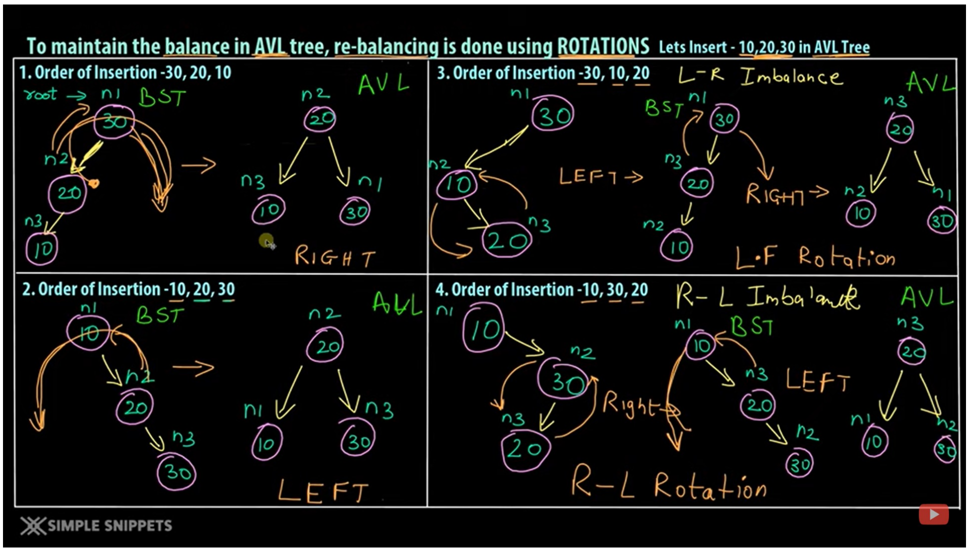
How are we going to maintain this? Worry about that later…

2



Q: Why does this work? Why are you allowed to change the structure of a tree like this?

Short answer: In the left tree we know that D < B < E < A < C and this remains true for the right subtree, so we didn't break the BST invariant and, therefore, this is a valid transformation.



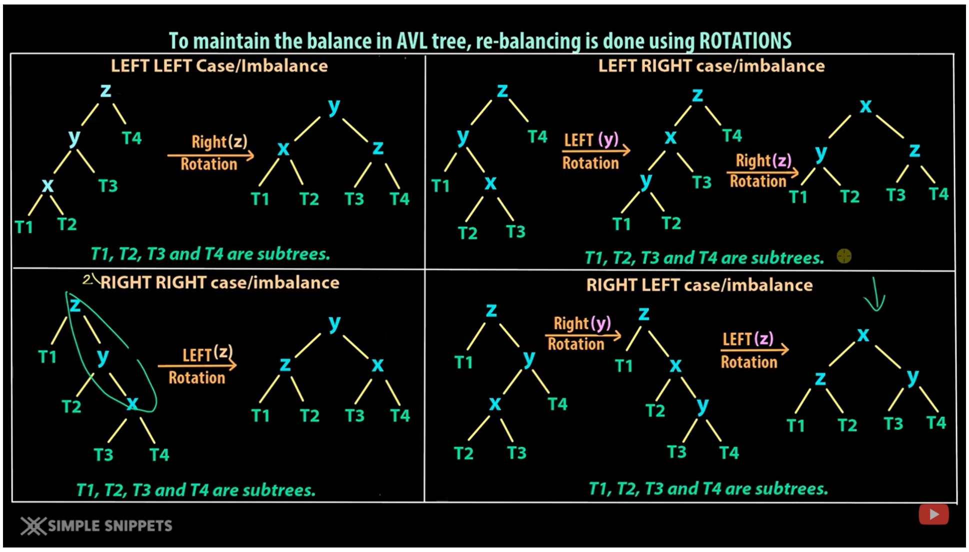
To maintain the balance in AVL tree, re-balancing is done using ROTATIONS Lets Insert - 10,20,30 in AVL Tree

1. Order of Insertion -30, 20, 10

3. Order of Insertion -30, 10, 20

2. Order of Insertion -10, 20, 30

4. Order of Insertion -10, 30, 20



To maintain the balance in AVL tree, re-balancing is done using ROTATIONS

LEFT LEFT Case/Imbalance

T1, T2, T3 and T4 are subtrees.

LEFT RIGHT case/imbalance

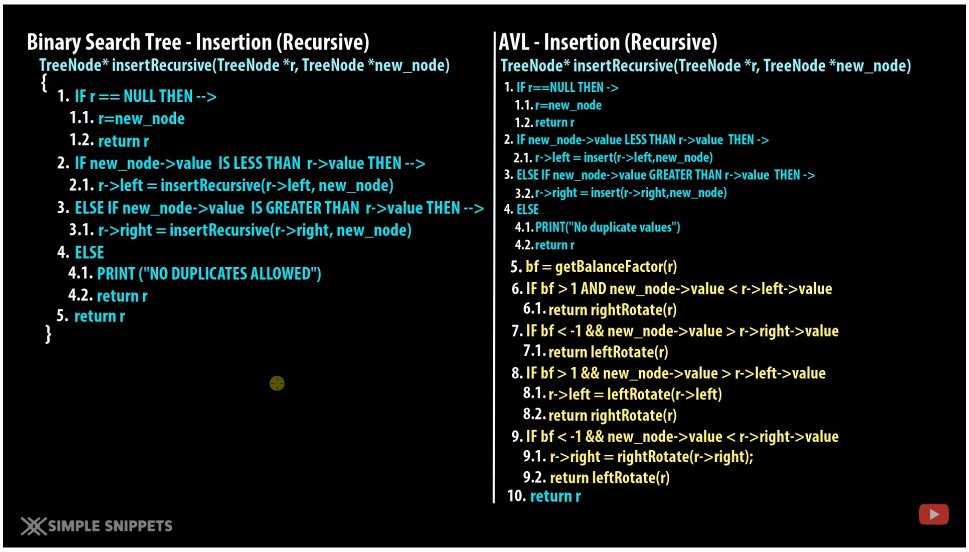
T1, T2, T3 and T4 are subtrees.

2.RIGHT RIGHT case/imbalance

T1, T2, T3 and T4 are subtrees.

RIGHT LEFT case/imbalance

T1, T2, T3 and T4 are subtrees.

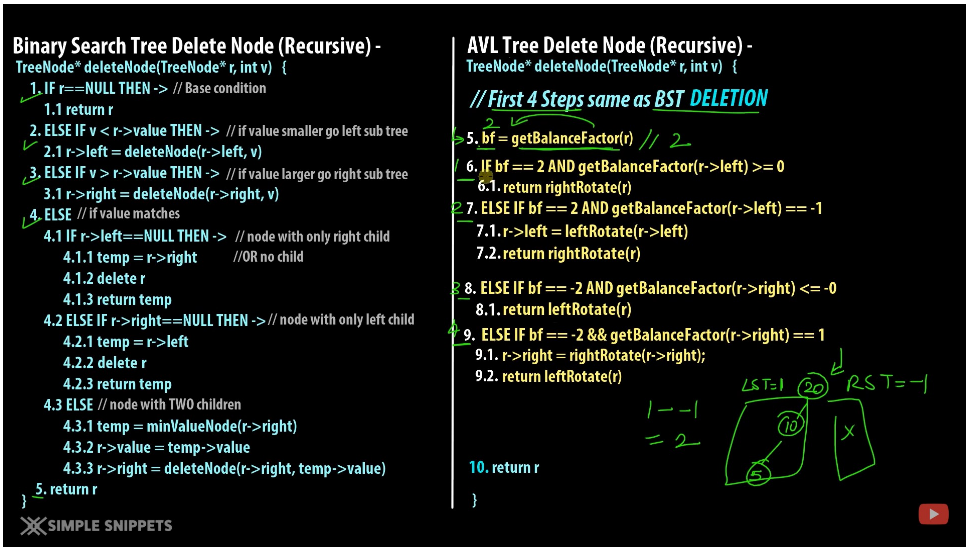


Binary Search Tree - Insertion (Recursive)

```
TreeNode* insertRecursive(TreeNode *r, TreeNode *new_node)
{
    1. IF r == NULL THEN -->
        1.1. r=new_node
        1.2. return r
    2. IF new_node->value  IS LESS THAN  r->value THEN -->
        2.1. r->left = insertRecursive(r->left, new_node)
    3. ELSE IF new_node->value  IS GREATER THAN  r->value THEN -->
        3.1. r->right = insertRecursive(r->right, new_node)
    4. ELSE
        4.1. PRINT ("NO DUPLICATES ALLOWED")
        4.2. return r
    5. return r
}
```

AVL - Insertion (Recursive)

```
TreeNode* insertRecursive(TreeNode *r, TreeNode *new_node)
1. IF r==NULL THEN ->
    1.1. r=new_node
    1.2. return r
2. IF new_node->value LESS THAN r->value  THEN ->
    2.1. r->left = insert(r->left,new_node)
3. ELSE IF new_node->value GREATER THAN r->value  THEN ->
    3.2. r->right = insert(r->right,new_node)
4. ELSE
    4.1. PRINT("No duplicate values")
    4.2. return r
5. bf = getBalanceFactor(r)
6. IF bf > 1 AND new_node->value < r->left->value
    6.1. return rightRotate(r)
7. IF bf < -1 && new_node->value > r->right->value
    7.1. return leftRotate(r)
8. IF bf > 1 && new_node->value > r->left->value
    8.1. r->left = leftRotate(r->left)
    8.2. return rightRotate(r)
9. IF bf < -1 && new_node->value < r->right->value
    9.1. r->right = rightRotate(r->right);
    9.2. return leftRotate(r)
10. return r
```



Binary Search Tree Delete Node (Recursive) -

```
TreeNode* deleteNode(TreeNode* r, int v)  {
    1. IF r==NULL THEN -> // Base condition
        1.1 return r
    2. ELSE IF v < r->value THEN ->  // if value smaller go left sub tree
        2.1 r->left = deleteNode(r->left, v)
    3. ELSE IF v > r->value THEN ->  // if value larger go right sub tree
        3.1 r->right = deleteNode(r->right, v)
    4. ELSE  // if value matches
        4.1 IF r->left==NULL THEN ->  // node with only right child
            4.1.1 temp = r->right        //OR no child
            4.1.2 delete r
            4.1.3 return temp
        4.2 ELSE IF r->right==NULL THEN -> // node with only left child
            4.2.1 temp = r->left
            4.2.2 delete r
            4.2.3 return temp
        4.3 ELSE // node with TWO children
            4.3.1 temp = minValueNode(r->right)
            4.3.2 r->value = temp->value
            4.3.3 r->right = deleteNode(r->right, temp->value)
    5. return r
}
```

AVL Tree Delete Node (Recursive) -

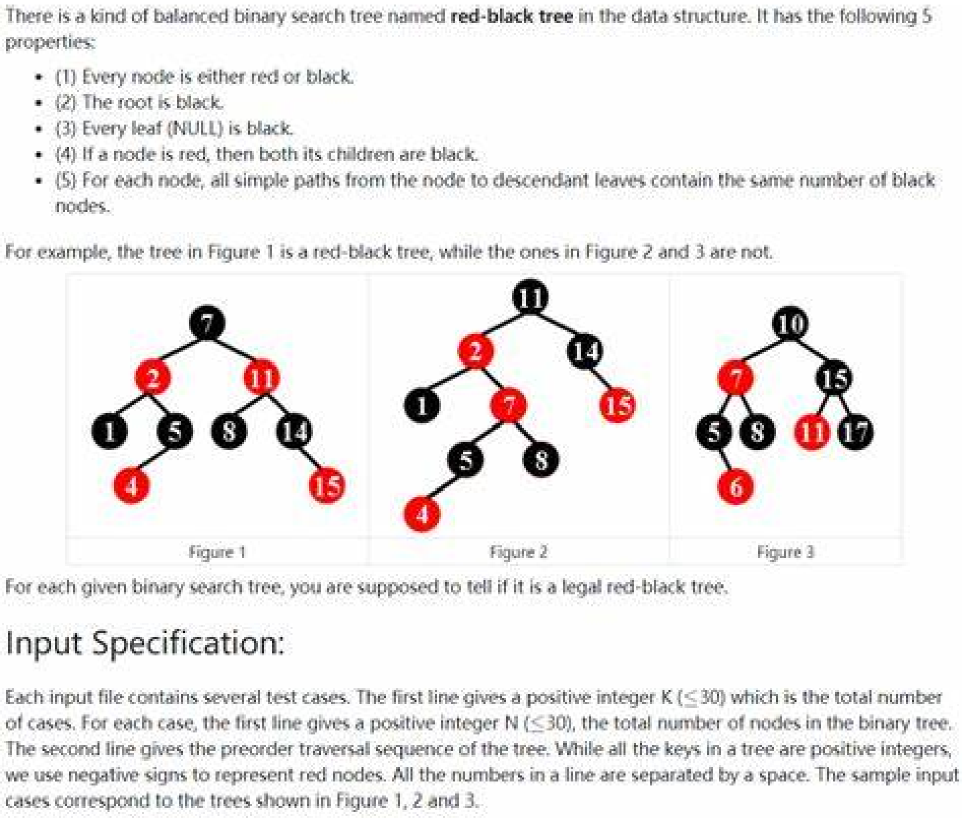
```
TreeNode* deleteNode(TreeNode* r, int v)  {

    // First 4 Steps same as BST DELETION
          2
1   5. bf = getBalanceFactor(r)  // 2
    6. IF bf == 2 AND getBalanceFactor(r->left) >= 0
        6.1. return rightRotate(r)
2   7. ELSE IF bf == 2 AND getBalanceFactor(r->left) == -1
        7.1. r->left = leftRotate(r->left)
        7.2. return rightRotate(r)

3   8. ELSE IF bf == -2 AND getBalanceFactor(r->right) <= -0
        8.1. return leftRotate(r)
4   9. ELSE IF bf == -2 && getBalanceFactor(r->right) == 1
        9.1. r->right = rightRotate(r->right);
        9.2. return leftRotate(r)

                                1 — -1
                                = 2

    10. return r

}
```

```
In [ ]: #include <iostream>
        #define SPACE 5
        using namespace std;

        class TreeNode{
        public:
            int value;
            TreeNode* left;
            TreeNode* right;

            TreeNode();
            TreeNode(int v);
        };

        TreeNode::TreeNode() : value(0), left(nullptr), right(nullptr){}

        TreeNode::TreeNode(int v){
            value = v;
            left = nullptr;
            right = nullptr;
        }


        class BST{
        public:
            TreeNode* root;
```

```
    BST();

    bool isEmpty();
    void insertNode(TreeNode* newNode, int v);
    TreeNode* searchNode(int v);
    int height(TreeNode* root);
    TreeNode* minValueNode(TreeNode* root);
    TreeNode* deleteNode(TreeNode* root, int v);
    void print2D(TreeNode* root, int space);
    void printPreorder(TreeNode* root);
    void printInorder(TreeNode* root);
    void printPostorder(TreeNode* root);
    void printLevelOrder(TreeNode* root);
    void printGivenLevel(TreeNode* root, int level);
};

BST::BST() : root(nullptr){}

bool BST::isEmpty(){
    if(root==nullptr)
        return true;
    else
        return false;
}

void BST::insertNode(TreeNode* newNode, int v){
    newNode->value = v;
    if(root==nullptr){
        root = newNode;
        cout << "Value Inserted as Root Node!" << endl;
    }
    else {
        TreeNode* ptr = root;
        while(ptr!=nullptr){
            if(newNode->value == ptr->value){
                cout << "Value Already exist. Insert Another Value." << endl;
                return;
            }
            else if((newNode->value < ptr->value) && (ptr->left==nullptr)){
                ptr->left = newNode;
                cout << "Value Inserted in the left" << endl;
                break;
            }
            else if(newNode->value < ptr->value){
                ptr = ptr->left;
            }
            else if((newNode->value > ptr->value) && (ptr->right==nullptr)){
                ptr->right = newNode;
                cout << "Value Inserted in the right" << endl;
                break;
            }
            else {
                ptr = ptr->right;
            }
        }
    }
}


TreeNode* BST::searchNode(int v){
    if(root==nullptr)
        return root;
    TreeNode* ptr = root;
    while (ptr!=nullptr){
        if(ptr->value == v)
            return ptr;
        else if(ptr->value > v){
            ptr = ptr->left;
        }
        else {
            ptr = ptr->right;
        }
    }
    return nullptr;
}

int BST::height(TreeNode* root){
    if(root==nullptr)
        return -1;
    else {
        int leftHeight = height(root->left);
        int rightHeight = height(root->right);

        if(leftHeight > rightHeight)
            return (leftHeight + 1);
        else
            return (rightHeight + 1);
    }
}

TreeNode* BST::minValueNode(TreeNode* root){
    TreeNode* ptr = root;
    while(ptr->left!=nullptr){
        ptr = ptr->left;
    }
    return ptr;
}

TreeNode* BST::deleteNode(TreeNode* root, int v){

    if(root==nullptr)
        return root;
    TreeNode* ptr = root;
    if(ptr->value > v){
        ptr->left = deleteNode(ptr->left, v);
    }
    else if(ptr->value < v){
        ptr->right = deleteNode(ptr->right, v);
    }
    else{ // find the node with v
        if(ptr->left==nullptr){ // maybe have right child or no right child when pt
             TreeNode* temp = ptr->right;
            delete ptr;
            return temp;
        }
        else if(ptr->right==nullptr){ // only left child
            TreeNode* temp = ptr->left;
            delete ptr;
            return temp;
        }
        else { // have two children
```

```
            TreeNode* temp = minValueNode(ptr->right);
            ptr->value = temp->value;
            ptr->right = deleteNode(ptr->right, temp->value);
        }
    }
    return ptr;
}

void BST::print2D(TreeNode* root, int space){
    if(root == nullptr)
        return;
    space += SPACE; // increase distance between level
    print2D(root->right, space);
    cout << endl;
    for(int i=SPACE; i<space; i++){
        cout << " ";
    }
    cout << root->value << endl;
    print2D(root->left, space);
}

 void BST::printPreorder(TreeNode* root){
    if(root==nullptr){
        return;
    }
    cout << root->value << " ";
    printPreorder(root->left);
    printPreorder(root->right);
 }

 void BST::printInorder(TreeNode* root){
    if(root==nullptr){
        return;
    }
    printInorder(root->left);
    cout << root->value << " ";
    printInorder(root->right);
 }

 void BST::printPostorder(TreeNode* root){
    if(root==nullptr){
        return;
    }
    printPostorder(root->left);
    printPostorder(root->right);
    cout << root->value << " ";
 }

void BST::printGivenLevel(TreeNode* root, int level){
    if(root==nullptr){
        return;
    }
    else if (level==0)
        cout << root->value << " ";
    else {
        printGivenLevel(root->left, level - 1);
        printGivenLevel(root->right, level - 1);
    }
}

void BST::printLevelOrder(TreeNode* root){
    int h = height(root);
    for(int i=0; i<=h; i++){
        printGivenLevel(root, i);
    }
 }


class AVLTree : public BST {
public:

    AVLTree();

    int getBalanceFactor(TreeNode* ptrNode);
    TreeNode* insertRecursive(TreeNode* root, TreeNode* newNode, int v);
    TreeNode* rotateRecursive(TreeNode* root);
    TreeNode* insertAndRotate(TreeNode* root, TreeNode* newNode, int v);
    TreeNode* deleteAndRotate(TreeNode* root, int v);
    TreeNode* rightRotate(TreeNode* ptrNode);
    TreeNode* leftRotate(TreeNode* ptrNode);
    TreeNode* recusiveSearch(TreeNode* root, int v);
};

AVLTree::AVLTree(){
    root = nullptr;
}

TreeNode* AVLTree::insertRecursive(TreeNode* root, TreeNode* newNode, int v){
    newNode->value = v;
    if(root==nullptr){
        root = newNode;
        cout << "Value inserted successfully" << endl;
        return root;
    }
    if(newNode->value < root->value){
        root->left = insertAndRotate(root->left, newNode, v);
    }
    else if(newNode->value > root->value){
        root->right = insertAndRotate(root->right,  newNode, v);
    }
    else {
        cout << "No duplicate value allowed" << endl;
        return root;
    }
    return root;
}

int AVLTree::getBalanceFactor(TreeNode* root){
        if (root==nullptr)
            return -1;
        return (height(root->left) - height(root->right));
    }

TreeNode* AVLTree::rightRotate(TreeNode* root){
    TreeNode* ptrNode = root->left;
    ptrNode->right = root;
    root->left = nullptr;

    return ptrNode;
}

TreeNode* AVLTree::leftRotate(TreeNode* root){
    TreeNode* ptrNode = root->right;
    ptrNode->left = root;
    root->right = nullptr;
    return ptrNode;
}
/*
TreeNode* AVLTree::insertAndRotate(TreeNode* root, TreeNode* newNode, int v){
    newNode->value = v;
    if(root==nullptr){
        root = newNode;
        cout << "Value inserted successfully" << endl;
        return root;
    }
    if(newNode->value < root->value){
        root->left = insertAndRotate(root->left, newNode, v);
    }
    else if(newNode->value > root->value){
        root->right = insertAndRotate(root->right,  newNode, v);
    }
    else {
        cout << "No duplicate value allowed" << endl;
        return root;
    }

    // Transform to AVL Tree
    int balanceFactor = getBalanceFactor(root);

    if((balanceFactor>1) && (newNode->value < root->left->value))
        return rightRotate(root);

    if((balanceFactor>1) && (newNode->value > root->left->value)){
        root->left = leftRotate(root->left);
        return rightRotate(root);
    }

    if((balanceFactor<-1) && (newNode->value > root->right->value))
        return leftRotate(root);

    if((balanceFactor<-1) && (newNode->value < root->right->value)){
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }
    return root;
}
*/


TreeNode* AVLTree::rotateRecursive(TreeNode* root){
    int balanceFactor = getBalanceFactor(root);

    if((balanceFactor>1) && (getBalanceFactor(root->left)>=0))
        return rightRotate(root);

    if((balanceFactor>1) && (getBalanceFactor(root->left)==-1)){
        root->left = leftRotate(root->left);
        return rightRotate(root);
    }

    if((balanceFactor<-1) && (getBalanceFactor(root->right)<=0))
        return leftRotate(root);
    if((balanceFactor<-1) && (getBalanceFactor(root->right)==1)){
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }
    return root;
}

TreeNode* AVLTree::insertAndRotate(TreeNode* root, TreeNode* newNode, int v){

    root = insertRecursive(root, newNode, v);
    root = rotateRecursive(root);

    return root;
}

TreeNode* AVLTree::deleteAndRotate(TreeNode* root, int v){

    root = deleteNode(root, v);
    root = rotateRecursive(root);

    return root;
}

TreeNode* AVLTree::recusiveSearch(TreeNode* root, int v){
    if(root==nullptr || root->value==v)
        return root;
    else if(v < root->value)
        return recusiveSearch(root->left, v);
    else
        return recusiveSearch(root->right, v);
}


int main(){
    int option, value, height;
    int space = 2;
    BST obj;
    AVLTree balanceObj;

    do{
        cout << "Welcome to learning Data Structure and Algorithm." << endl << endl

        cout << "Select Option Number or Enter 0 to exit." << endl;
        cout << "1. Insert Node" << endl;
        cout << "2. Search Node" << endl;
        cout << "3. Delete Node" << endl;
        cout << "4. Height of Tree" << endl;
        cout << "5. Print Values in 2D" << endl;
        cout << "6. Insert Node to AVL Tree" << endl;
        cout << "7. Delete Node in AVL Tree" << endl;
        cout << "8. Search Node in AVL Tree" << endl;
        cout << "9. Print AVL Tree" << endl;
        cout << "10. Clear Screen" << endl;
        cout << "0. Exit Program" << endl;
        cout << endl;

        cin >> option;

        TreeNode* newNode = new TreeNode(); // in heap memory

        switch(option){
            case 0:
                break;
            case 1:
                cout << "Insert" << endl;
                cout << "Enter Value of Tree Node to Insert in BST: ";
                cin >> value;
                obj.insertNode(newNode, value);
                cout << endl << endl;
                break;
            case 2:
                cout << "Search" << endl;
                cout << "Enter Value of Tree Node to Search: ";
                cin >> value;
                newNode = obj.searchNode(value);
                if(newNode!=nullptr){
                    cout << "Value " << value << " Found" << endl;
                }
                else {
                    cout << "Value " << value << " Not Found" << endl;
                }
                break;
            case 3:
                cout << "Delete" << endl;
                cout << "Enter Value of Tree Node to Delete: ";
                cin >> value;
                if(obj.searchNode(value)!=nullptr){
                    obj.deleteNode(obj.root, value);
                    cout << "Value Deleted" << endl << endl;

                }
                else {
                    cout << "Value " << value << " not Found" << endl << endl;
                }

                break;
            case 4:
                cout << "Tree Height" << endl;
                height = obj.height(obj.root);
                cout << "Height: " << height << endl;
                cout << endl << endl;
                break;
            case 5:
                cout << "2D Print" << endl;
                obj.print2D(obj.root, space);
                cout << endl << endl;
                cout << "Preorder: " << endl;
                obj.printPreorder(obj.root);
                cout << endl << endl;
                cout << "Inorder: " << endl;
                obj.printInorder(obj.root);
                cout << endl << endl;
                cout << "Postorder: " << endl;
                obj.printPostorder(obj.root);
                cout << endl << endl;
                cout << "Levelorder: " << endl;
                obj.printLevelOrder(obj.root);
                cout << endl << endl;
                break;
            case 6:
                cout << "Insert Node to AVL Tree" << endl;
                cout << "Enter Value: ";
                cin >> value;
                balanceObj.root = balanceObj.insertAndRotate(balanceObj.root, newNo
                cout << endl << endl;
                break;
            case 7:
                cout << "Delete Node in AVL Tree" << endl;
                cout << "Enter Value: ";
                cin >> value;
                balanceObj.root = balanceObj.deleteAndRotate(balanceObj.root, value
                cout << endl << endl;
                break;
            case 8:
                cout << "Search Node in AVL Tree" << endl;
                cout << "Enter Value to Search: ";
                cin >> value;
                newNode = balanceObj.searchNode(value);
                if(newNode!=nullptr){
                    cout << "Value " << value << " Found" << endl;
                }
                else {
                    cout << "Value " << value << " Not Found" << endl;
                }
                break;
            case 9:
                cout << "Print AVL Tree" << endl;
                balanceObj.print2D(balanceObj.root, space);
                cout << endl << endl;
                break;
            case 10:
                system("cls");
                break;
            default:
                cout << "Please Enter the Proper Option." << endl;
                break;
        }
    } while (option !=0);


    return 0;
}
```

Red Black Tree



There is a kind of balanced binary search tree named **red-black tree** in the data structure. It has the following 5 properties:

- (1) Every node is either red or black.
- (2) The root is black.
- (3) Every leaf (NULL) is black.
- (4) If a node is red, then both its children are black.
- (5) For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

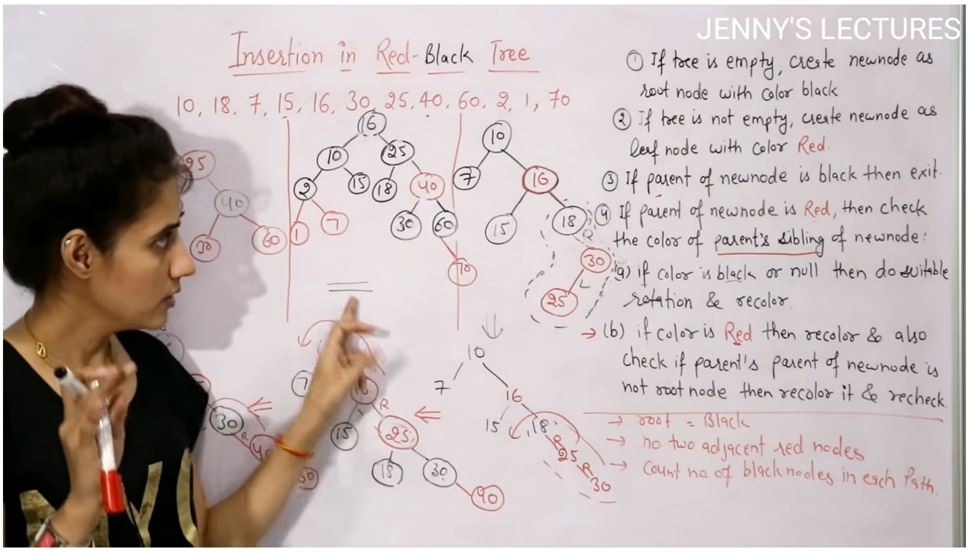
For example, the tree in Figure 1 is a red-black tree, while the ones in Figure 2 and 3 are not.

Figure 1 Figure 2 Figure 3

For each given binary search tree, you are supposed to tell if it is a legal red-black tree.

Input Specification:

Each input file contains several test cases. The first line gives a positive integer K (≤30) which is the total number of cases. For each case, the first line gives a positive integer N (≤30), the total number of nodes in the binary tree. The second line gives the preorder traversal sequence of the tree. While all the keys in a tree are positive integers, we use negative signs to represent red nodes. All the numbers in a line are separated by a space. The sample input cases correspond to the trees shown in Figure 1, 2 and 3.



JENNY'S LECTURES

Insertion in Red-Black Tree

10, 18, 7, 15, 16, 30, 25, 40, 60, 2, 1, 70

① If tree is empty, create newnode as root node with color black

② If tree is not empty, create newnode as leaf node with color Red.

③ If parent of newnode is black then exit.

④ If parent of newnode is Red, then check the color of parent's sibling of newnode:

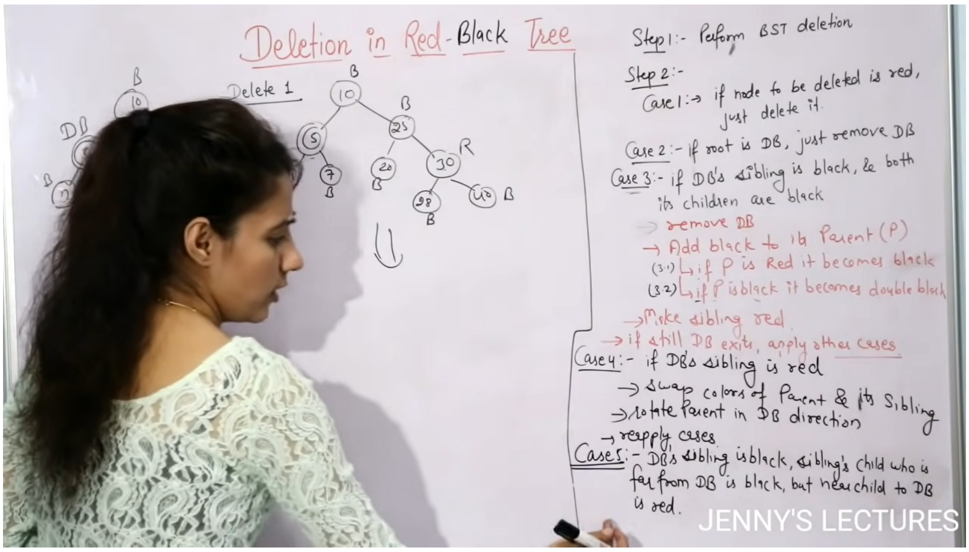
(a) if color is black or null then do suitable rotation & recolor

→ (b) if color is Red then recolor & also check if parent's parent of newnode is not root node then recolor it & recheck

→ root = Black

→ no two adjacent red nodes

→ Count no of black nodes in each Path.



Deletion in Red-Black Tree

Delete 1

Step 1:- Perform BST deletion

Step 2:-

Case 1:→ if node to be deleted is red, just delete it.

Case 2:- if root is DB, just remove DB

Case 3:- if DB's sibling is black, & both its children are black

→ remove DB

→ Add black to its Parent (P)

(3.1) ↳ if P is Red it becomes black

(3.2) ↳ if P is black it becomes double black

→ Make sibling red

→ if still DB exits, apply other cases

Case 4:- if DB's sibling is red

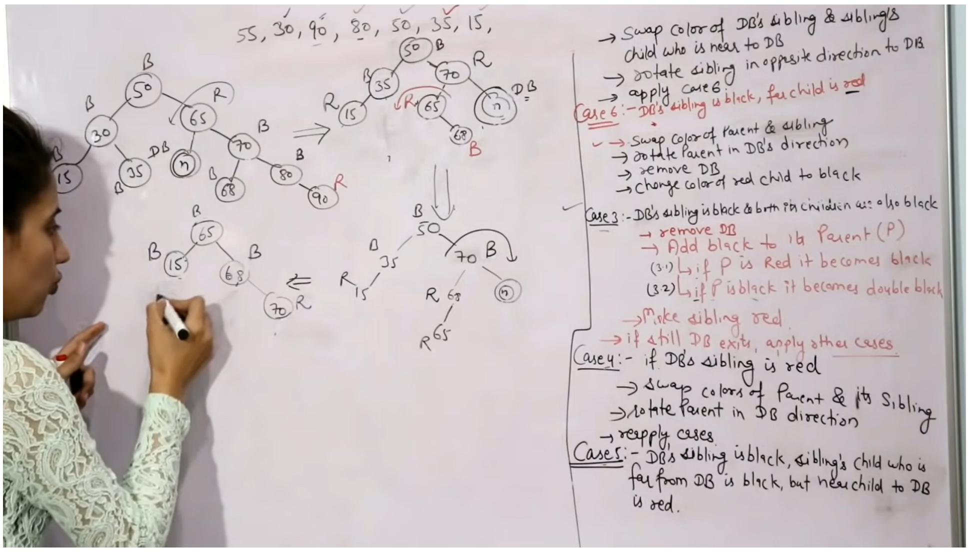
→ swap colors of Parent & its Sibling

→ rotate Parent in DB direction

→ reapply cases

Case 5:- DB's sibling is black, sibling's child who is far from DB is black, but near child to DB is red.

JENNY'S LECTURES



→ Swap color of DB's sibling & sibling's child who is near to DB
→ rotate sibling in opposite direction to DB
→ apply case 6

Case 6 :- DB's sibling is black, far child is red

→ Swap color of Parent & sibling
→ rotate Parent in DB's direction
→ remove DB
→ change color of red child to black

Case 3 :- DB's sibling is black & both its children are also black

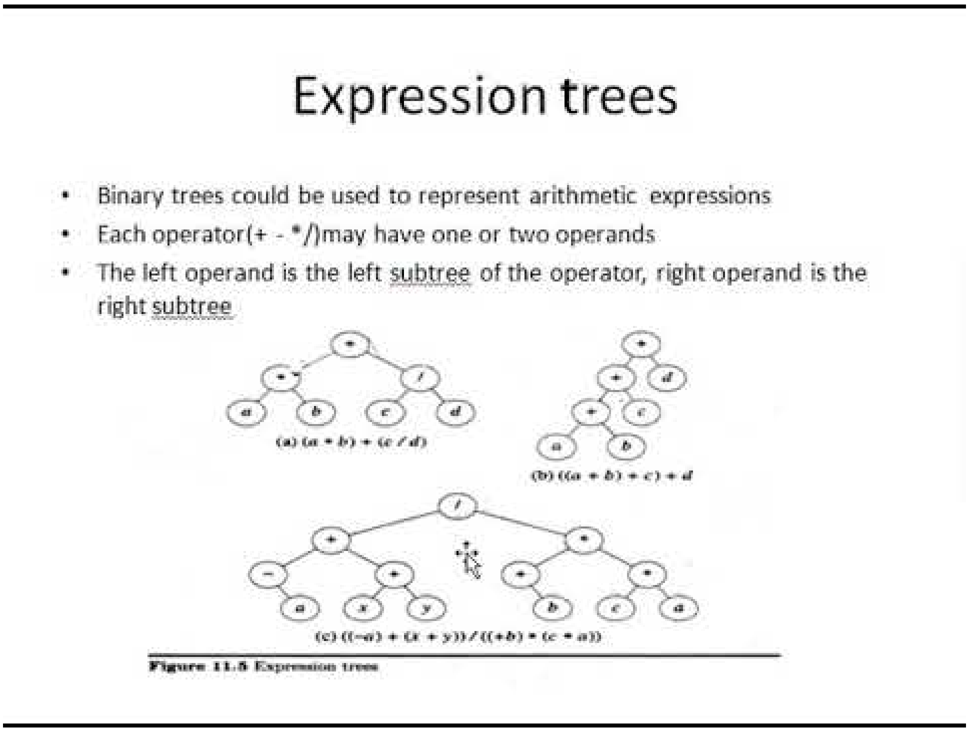
→ remove DB
→ Add black to its Parent (P)
(3.1) ↳ if P is Red it becomes black
(3.2) ↳ if P is black it becomes double black
→ Make sibling red
→ if still DB exits, apply other cases

Case 4 :- if DB's sibling is red

→ swap colors of Parent & its Sibling
→ rotate Parent in DB direction
→ reapply cases

Case 5 :- DB's sibling is black, sibling's child who is far from DB is black, but near child to DB is red.

Expression Tree



Expression trees

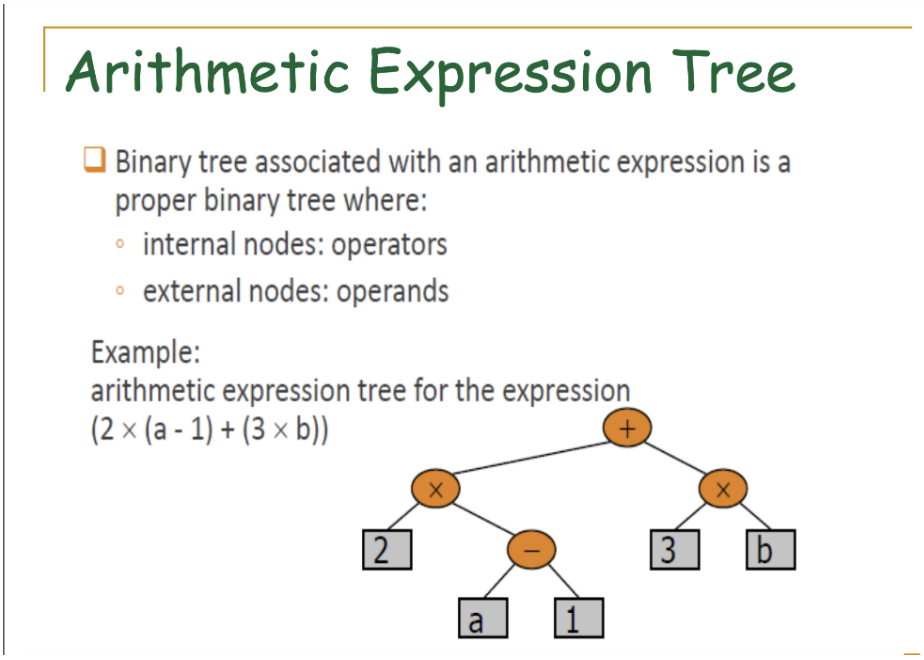
- Binary trees could be used to represent arithmetic expressions
- Each operator(+ - */)may have one or two operands
- The left operand is the left subtree of the operator, right operand is the right subtree

(a) (a * b) + (c / d)

(b) ((a + b) + c) + d

(c) ((−a) + (x + y)) / ((+b) * (c * a))

Figure 11.5 Expression trees

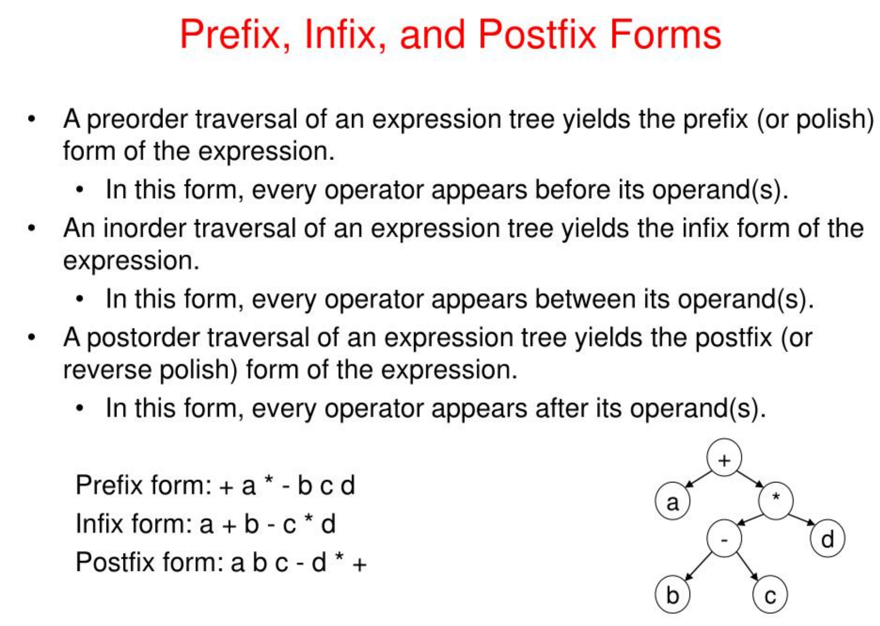


Arithmetic Expression Tree

- Binary tree associated with an arithmetic expression is a proper binary tree where:
  - internal nodes: operators
  - external nodes: operands

Example:
arithmetic expression tree for the expression
(2 × (a - 1) + (3 × b))

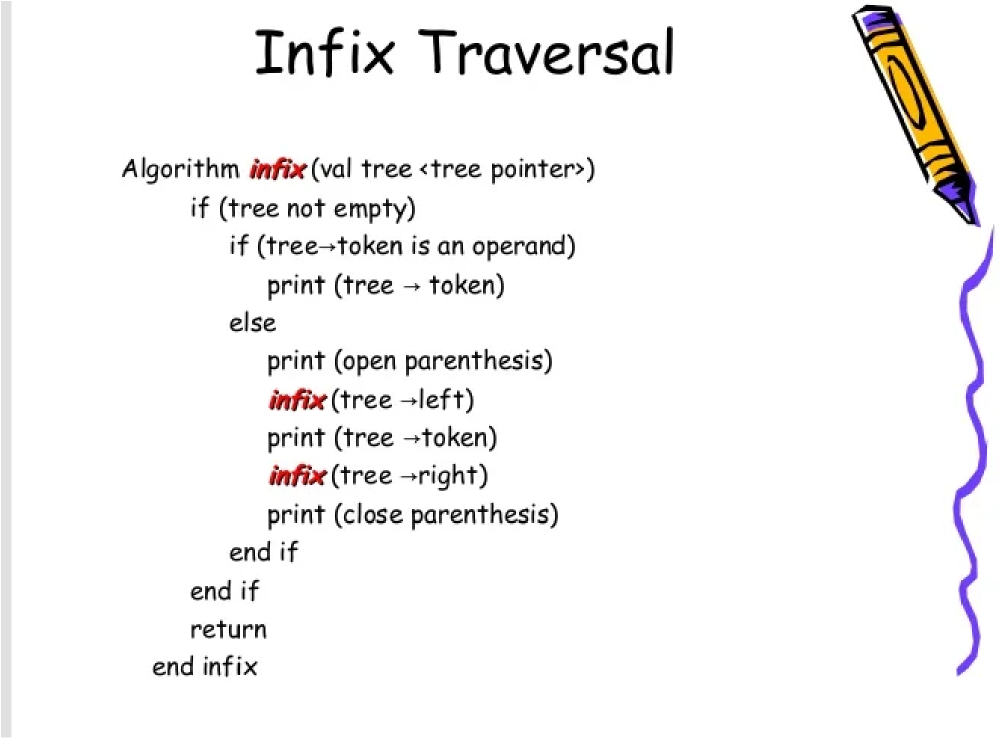
Prefix, Postfix, Infix



Prefix, Infix, and Postfix Forms

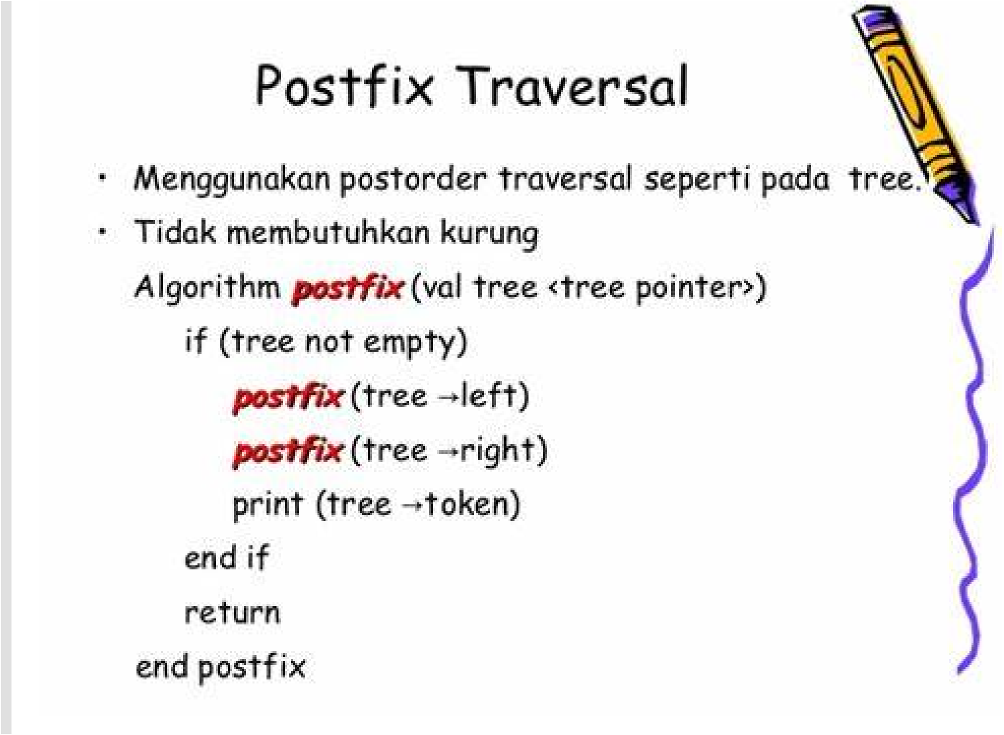
- A preorder traversal of an expression tree yields the prefix (or polish) form of the expression.
  - In this form, every operator appears before its operand(s).
- An inorder traversal of an expression tree yields the infix form of the expression.
  - In this form, every operator appears between its operand(s).
- A postorder traversal of an expression tree yields the postfix (or reverse polish) form of the expression.
  - In this form, every operator appears after its operand(s).

Prefix form: + a * - b c d
Infix form: a + b - c * d
Postfix form: a b c - d * +



Infix Traversal

```
Algorithm infix (val tree <tree pointer>)
    if (tree not empty)
        if (tree→token is an operand)
            print (tree → token)
        else
            print (open parenthesis)
            infix (tree →left)
            print (tree →token)
            infix (tree →right)
            print (close parenthesis)
        end if
    end if
    return
  end infix
```

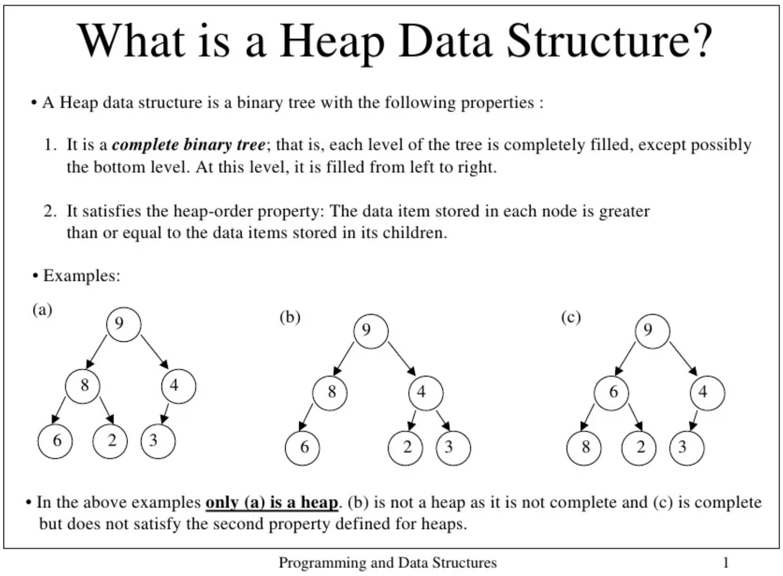


Postfix Traversal

- Menggunakan postorder traversal seperti pada tree.
- Tidak membutuhkan kurung

```
Algorithm postfix (val tree <tree pointer>)
    if (tree not empty)
        postfix (tree →left)
        postfix (tree →right)
        print (tree →token)
    end if
    return
end postfix
```

Heap



What is a Heap Data Structure?

- A Heap data structure is a binary tree with the following properties :

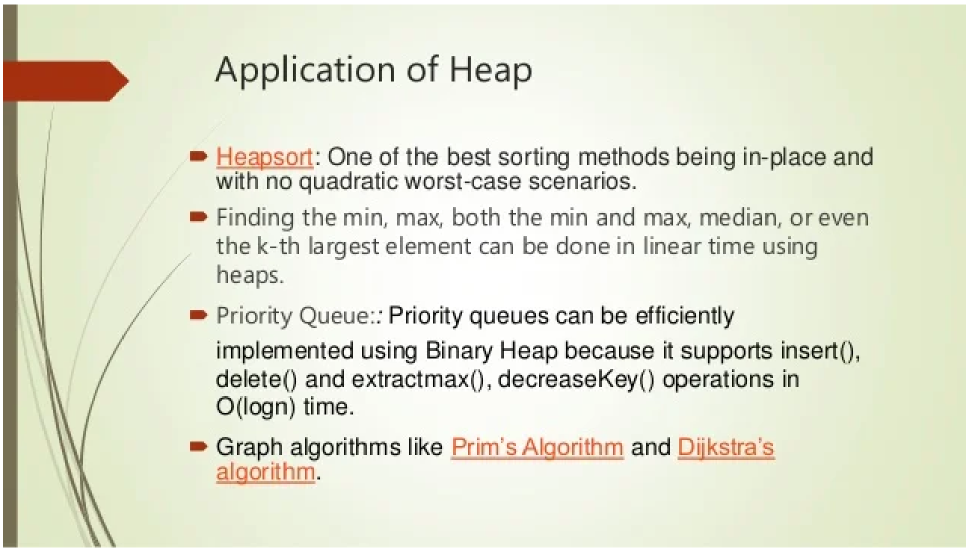
  1. It is a ***complete binary tree***; that is, each level of the tree is completely filled, except possibly the bottom level. At this level, it is filled from left to right.

  2. It satisfies the heap-order property: The data item stored in each node is greater than or equal to the data items stored in its children.

- Examples:

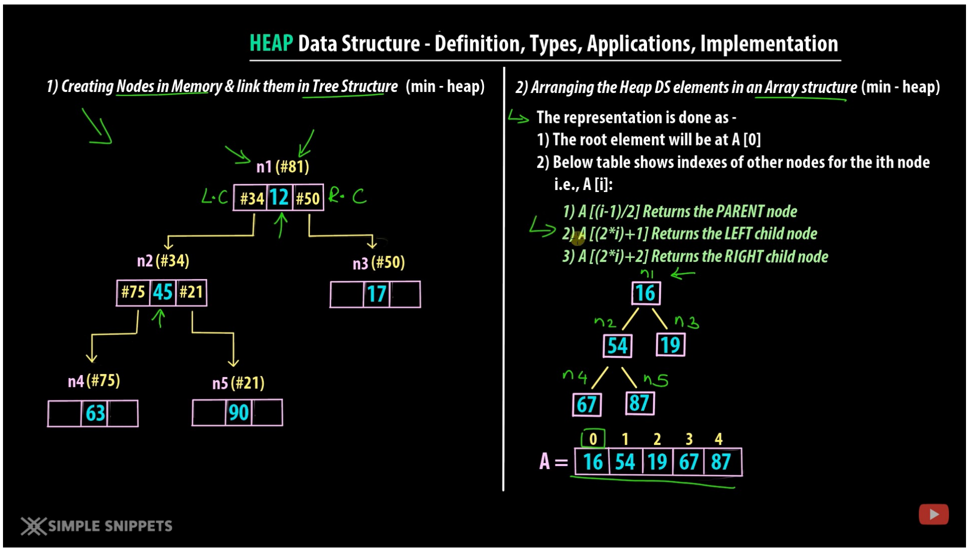
- In the above examples **only (a) is a heap**. (b) is not a heap as it is not complete and (c) is complete but does not satisfy the second property defined for heaps.

Programming and Data Structures 1



Application of Heap

- Heapsort: One of the best sorting methods being in-place and with no quadratic worst-case scenarios.
- Finding the min, max, both the min and max, median, or even the k-th largest element can be done in linear time using heaps.
- Priority Queue:: Priority queues can be efficiently implemented using Binary Heap because it supports insert(), delete() and extractmax(), decreaseKey() operations in O(logn) time.
- Graph algorithms like Prim's Algorithm and Dijkstra's algorithm.



HEAP Data Structure - Definition, Types, Applications, Implementation

1) Creating Nodes in Memory & link them in Tree Structure (min - heap)

2) Arranging the Heap DS elements in an Array structure (min - heap)

The representation is done as -

1) The root element will be at A [0]

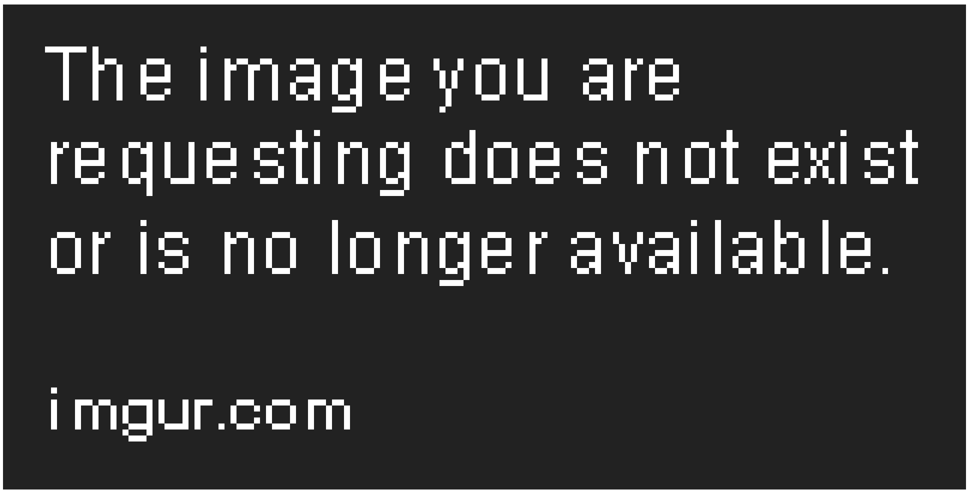
2) Below table shows indexes of other nodes for the ith node i.e., A [i]:

1) A [(i-1)/2] Returns the PARENT node

2) A [(2*i)+1] Returns the LEFT child node

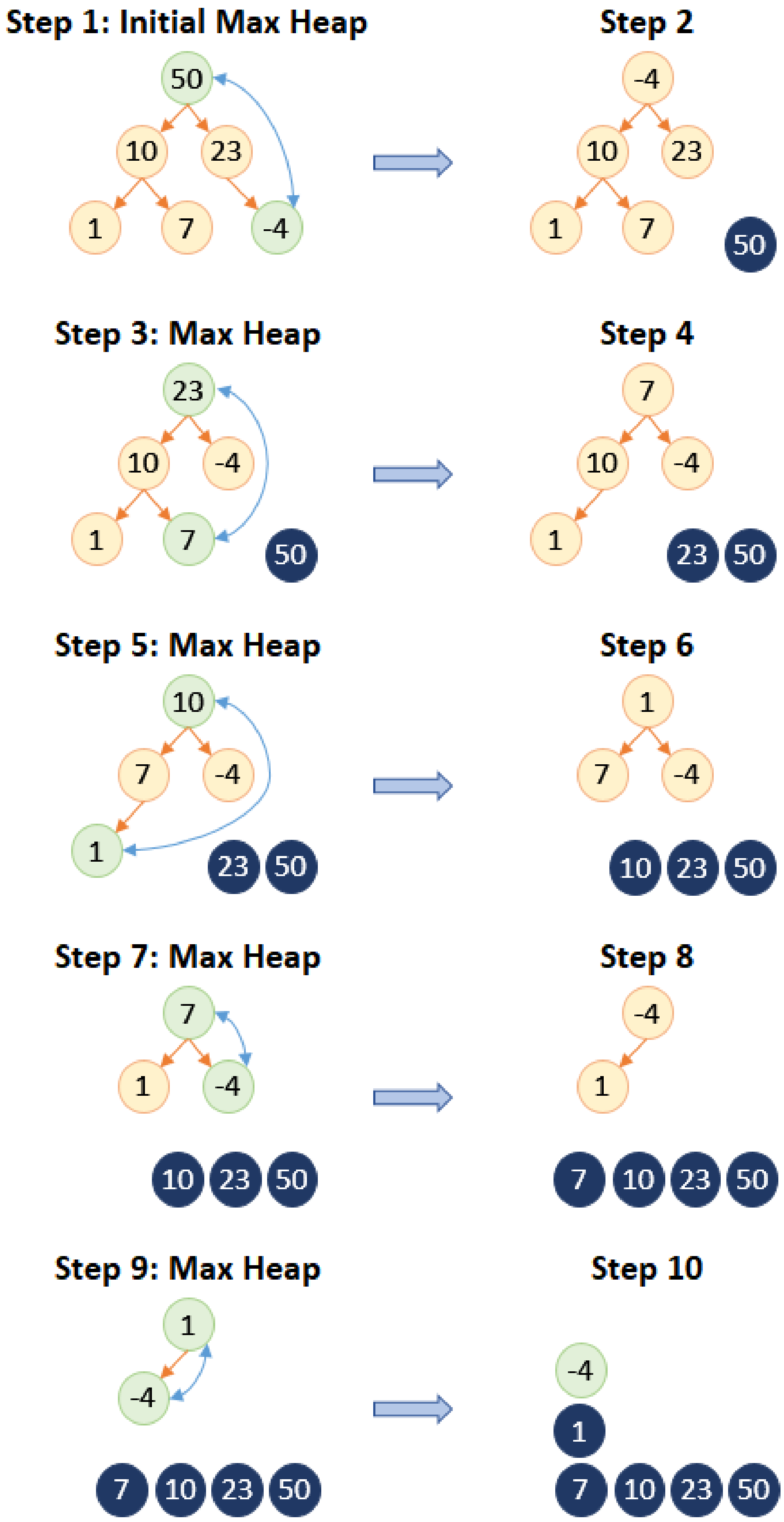
3) A [(2*i)+2] Returns the RIGHT child node

	0	1	2	3	4
A =	16	54	19	67	87

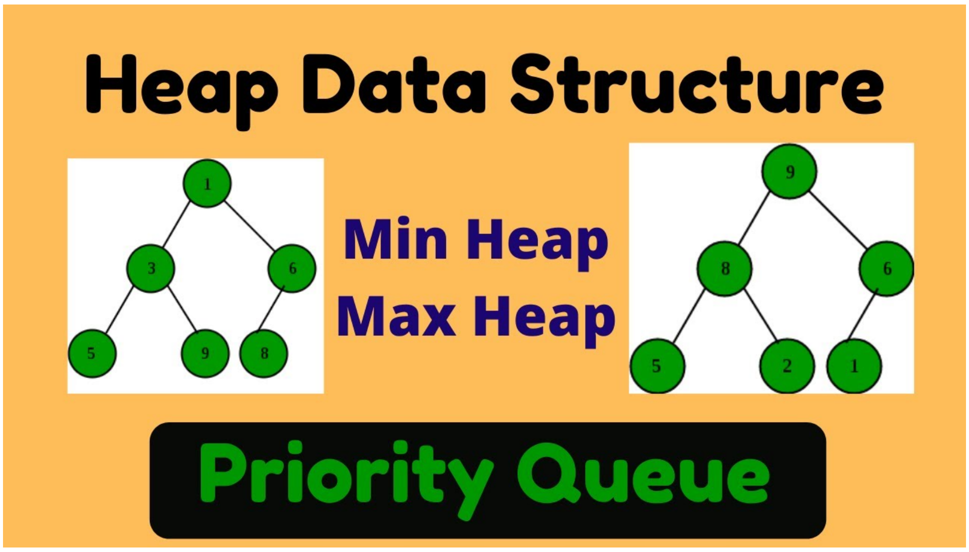


The image you are requesting does not exist or is no longer available.

imgur.com



Priority Queue

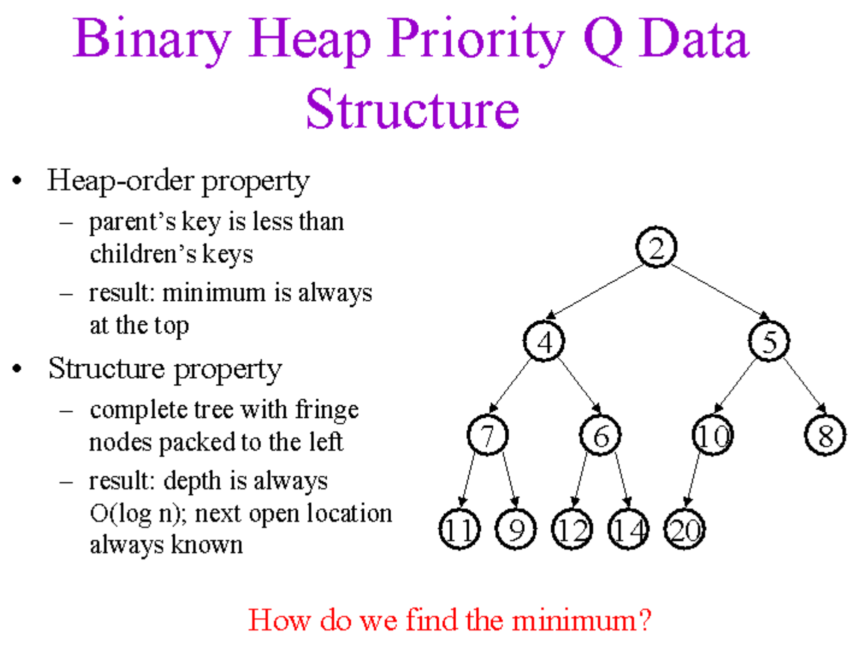


Heap Data Structure

Min Heap

Max Heap

Priority Queue



Binary Heap Priority Q Data Structure

- Heap-order property
  - parent's key is less than children's keys
  - result: minimum is always at the top
- Structure property
  - complete tree with fringe nodes packed to the left
  - result: depth is always O(log n); next open location always known

How do we find the minimum?

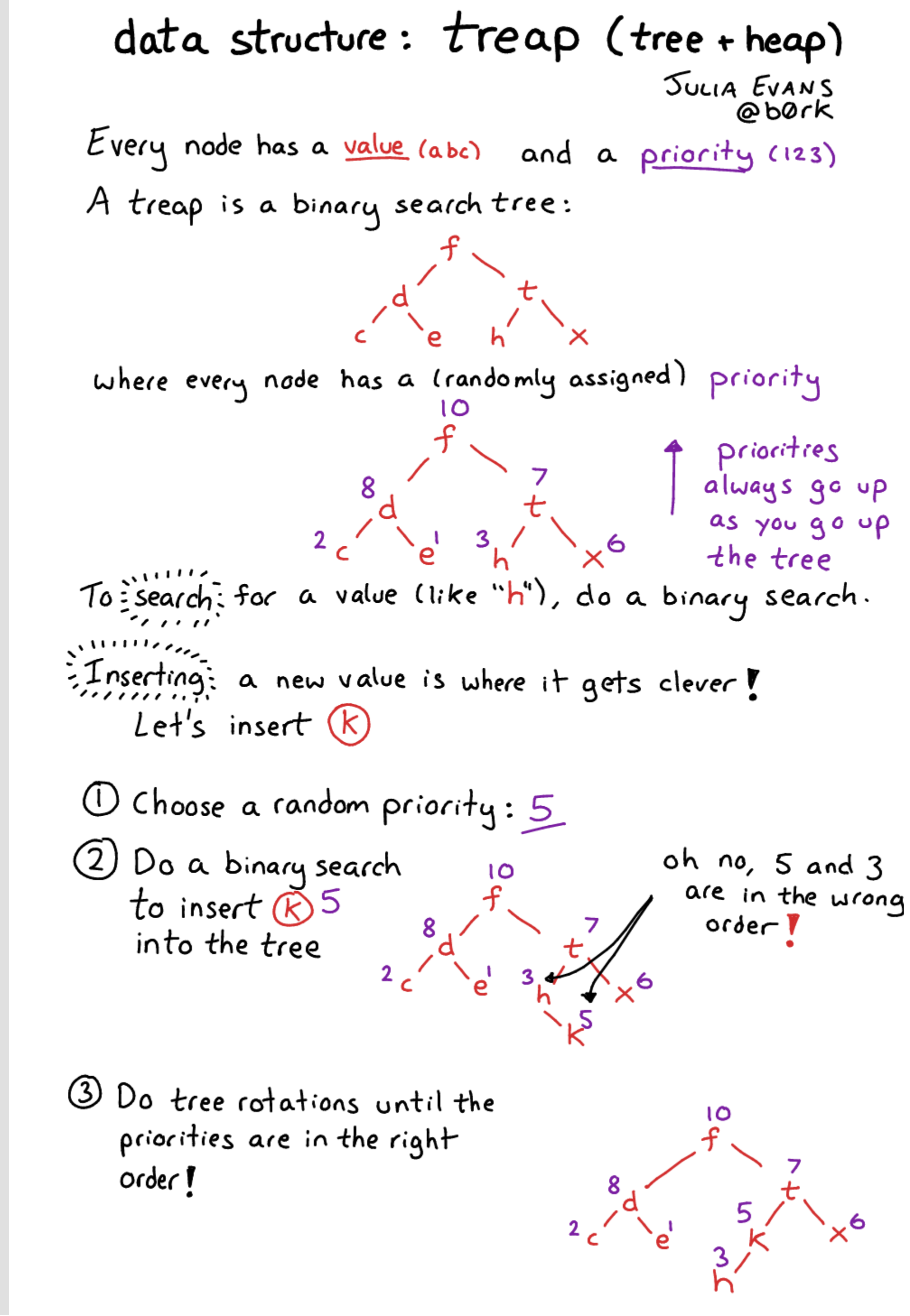
2
4
5
7
6
10
8
11
9
12
14
20

In [ ]:

```
#include <iostream>
#include <math.h>
using namespace std;


/*
swap(int &x, int &y){
    int temp = x;
    x = y;
    y = temp;
}
*/

class MinHeap{
public:
    int* heapArray;
    int capacity;
    int heapSize;

    MinHeap(int cap);
    int linearSearch(int val);
    int parent(int i);
    int leftchild(int i);
    int rightchild(int i);
    void insertion(int val);
    void minHeapify(int i);
    void deletion(int val);
    void decreaseKey(int i, int newVal);
    int getMin();
    int extractMin();
    void heapSort();
    int height();
    void printHeapArray();
};

MinHeap::MinHeap(int cap){
    capacity = cap;
    heapSize = 0;
    heapArray = new int[cap];
}

int MinHeap::linearSearch(int val){
    for(int i=0; i<heapSize; i++){
        if(heapArray[i]==val){
            return i;
        }
    }
    return -1;
}

int MinHeap::parent(int i) {
    return (i-1)/2;
}
int MinHeap::leftchild(int i){
    return (2*i+1);
}
int MinHeap::rightchild(int i){
    return (2*i+2);
}

void MinHeap::insertion(int val){
    if(heapSize == capacity){
        cout << "\nMin Heap Overflow" << endl;
        return;
    }
    for(int i=0; i<heapSize; i++){
        if(heapArray[i]==val){
            cout << "Value Already exist. Insert Another Value." << endl;
            return;
        }
    }
    heapSize++;
    int i = heapSize - 1;
    heapArray[i] = val;

    while(i!=0){
        if(heapArray[parent(i)] > heapArray[i]){
            swap(heapArray[parent(i)], heapArray[i]);
            i = parent(i); // continue to compare until i==0
        }
        else
```

```
            return;
    }
}

void MinHeap::minHeapify(int i){
    int left = leftchild(i);
    int right = rightchild(i);
    int smallest = i;
    if ((left<heapSize) && (heapArray[left]<heapArray[smallest])){
        smallest = left;
    }
    if((right<heapSize) && (heapArray[right]<heapArray[smallest])){
        smallest = right;
    }
    if(smallest!=i){
        swap(heapArray[i], heapArray[smallest]);
        minHeapify(smallest);
    }
}

void MinHeap::decreaseKey(int i, int newVal){
    heapArray[i] = newVal;
    while(i>=0 && (heapArray[parent(i)] > heapArray[i])){
        swap(heapArray[parent(i)], heapArray[i]);
        i = parent(i); // continue to compare until i==0
    }
}

void MinHeap::deletion(int val){
    int index = linearSearch(val);
    if(index==-1){
        cout << "Value " << val << " Not Found";
    }
    else {
        decreaseKey(index, INT_MIN);
        extractMin();
        cout << "Delete Value " << val << endl;
    }
}

int MinHeap::getMin(){
    return heapArray[0];
}

int MinHeap::extractMin(){
    if(heapSize <=0)
        return INT_MAX;
    if(heapSize==1){
        heapSize--;
        return heapArray[0];
    }
    int min = heapArray[0];
    heapArray[0] = heapArray[heapSize-1];
    heapSize--;
    minHeapify(0);
    return min;
}


void MinHeap::heapSort(){
    for(int i=0; i<capacity; i++){
        cout << extractMin() << " ";
    }
}

int MinHeap::height(){
    return ceil(log2(heapSize+1)) - 1;
}

void MinHeap::printHeapArray(){
    for(int i=0; i<heapSize; i++){
        cout << heapArray[i] << " ";
    }
    cout << endl << endl;
}



int main(){

    int cap;
    cout << "Enter the Size of Min Heap: ";
    cin >> cap;
    MinHeap obj(cap);
    cout << "Min Heap Created" << endl << endl;

    int option, val;

    do{
        cout << "What operation do you to perform?" << endl << endl;
        cout << "1. Insert" << endl;
        cout << "2. Search" << endl;
        cout << "3. Delete" << endl;
        cout << "4. Get Min" << endl;
        cout << "5. Extract Min" << endl;
        cout << "6. Height of Heap" << endl;
        cout << "7. Min Heap Sort" << endl;
        cout << "8. Print the Heap" << endl;
        cout << "9. Clear Screen" << endl;
        cout << "0. Exit Program" << endl;

        cout << "\nSelect Option Number or 0 to Exit." << endl;
        cin >> option;

        switch(option){
            case 0:
                break;
            case 1:
                cout << "Insert Operation" << endl;
                cout << "Enter Value to Insert in Heap: ";
                cin >> val;
                obj.insertion(val);
                cout << endl << endl;
                break;
            case 2:
                cout << "Search Operation" << endl;
                cout << "Enter Value to Search in Heap: ";
                cin >> val;
                if(obj.linearSearch(val)==-1){
                    cout << "Not Found";
                }
                else {
                    cout << val << " Found at index " << obj.linearSearch(val);
                }
                cout << endl << endl;
                break;
            case 3:
                cout << "Delete Operation" << endl;
                cout << "Enter Value to Delete in Heap: ";
                cin >> val;
                obj.deletion(val);
                cout << endl << endl;
                break;
            case 4:
                cout << "Get Min Operation" << endl;
                cout << "Min Value: " << endl;
                cout << obj.getMin() << endl;
                cout << endl << endl;
                break;
            case 5:
                cout << "Extract Min Value: ";
                cout << obj.extractMin() << endl;
                cout << endl << endl;
                break;
            case 6:
                cout << "The Height of Heap: ";
                cout << obj.height() << endl;
                cout << endl << endl;
                break;
            case 7:
                cout << "\nUnsorted Array: " << endl;
                obj.printHeapArray();
                cout << "Sorted Array: " << endl;
                obj.heapSort();
                cout << endl << endl;
                break;
            case 8:
                cout << "Print Heap Array: " << endl;
                obj.printHeapArray();
                cout << endl << endl;
                break;
            case 9:
                system("cls");
                break;
            default:
                cout << "Please Enter the Proper Option.";
                break;
        }
    } while(option!=0);
    return 0;
}
```

Treep



data structure: treap (tree + heap)

JULIA EVANS
@b0rk

Every node has a value (abc) and a priority (123)

A treap is a binary search tree:

where every node has a (randomly assigned) priority

priorities always go up as you go up the tree

To search for a value (like "h"), do a binary search.

Inserting a new value is where it gets clever!

Let's insert (k)

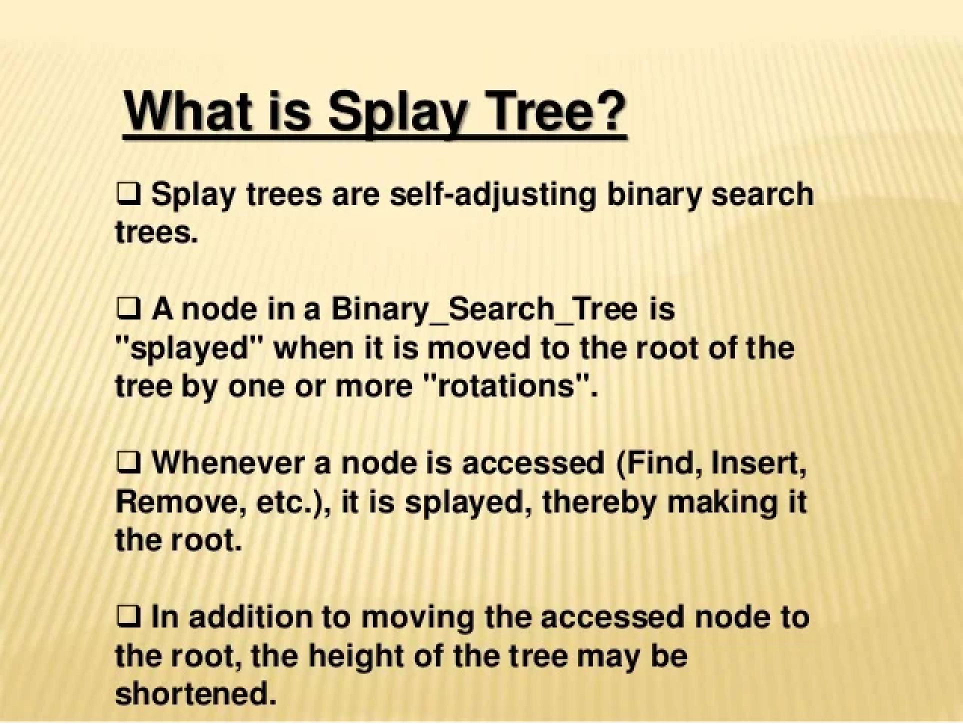
① Choose a random priority: 5

② Do a binary search to insert (k) 5 into the tree

oh no, 5 and 3 are in the wrong order!

③ Do tree rotations until the priorities are in the right order!

Splay Tree



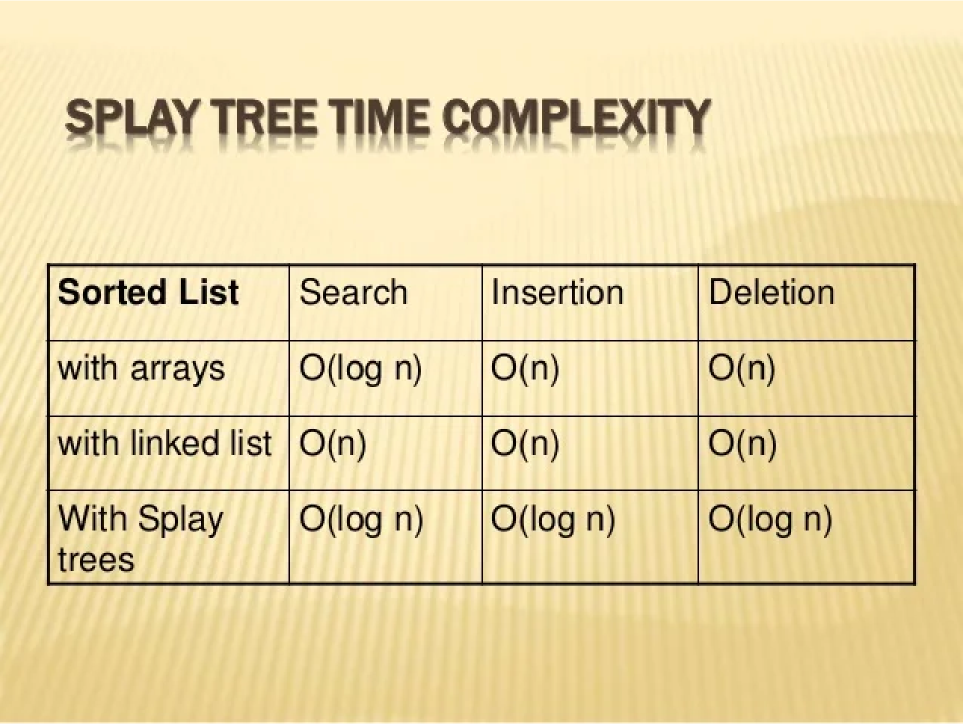
What is Splay Tree?

❑ Splay trees are self-adjusting binary search trees.

❑ A node in a Binary_Search_Tree is "splayed" when it is moved to the root of the tree by one or more "rotations".

❑ Whenever a node is accessed (Find, Insert, Remove, etc.), it is splayed, thereby making it the root.

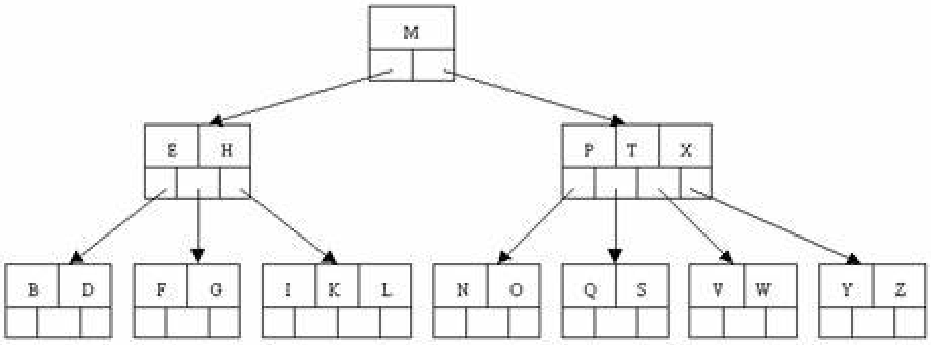
❑ In addition to moving the accessed node to the root, the height of the tree may be shortened.



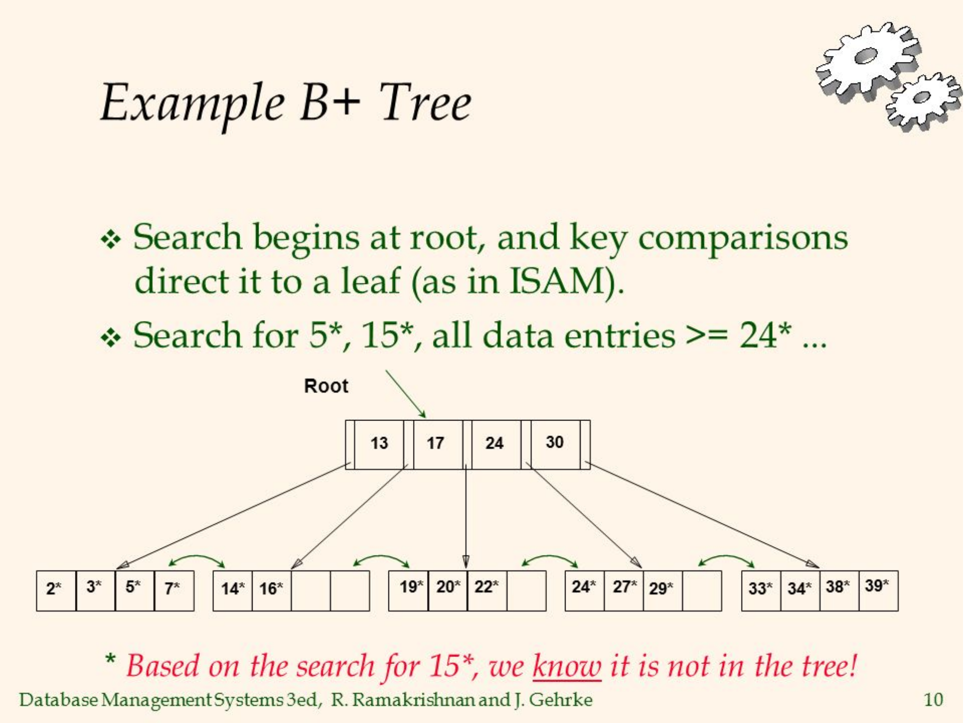
SPLAY TREE TIME COMPLEXITY

Sorted List	Search	Insertion	Deletion
with arrays	O(log n)	O(n)	O(n)
with linked list	O(n)	O(n)	O(n)
With Splay trees	O(log n)	O(log n)	O(log n)

B Tree



B^+ Tree



Example B+ Tree

- Search begins at root, and key comparisons direct it to a leaf (as in ISAM).
- Search for 5*, 15*, all data entries >= 24* ...

Root

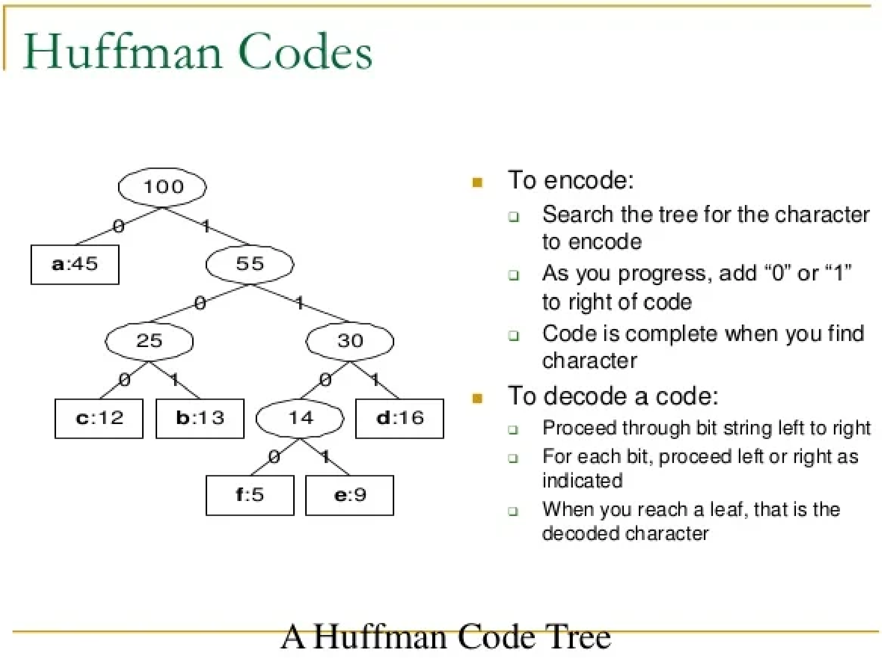
13 | 17 | 24 | 30

2* 3* 5* 7* | 14* 16* | 19* 20* 22* | 24* 27* 29* | 33* 34* 38* 39*

* *Based on the search for 15*, we know it is not in the tree!*

Database Management Systems 3ed, R. Ramakrishnan and J. Gehrke 10

Huffman Code

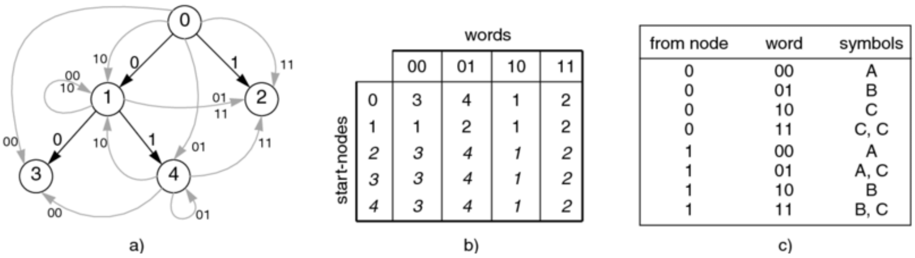


Huffman Codes

100
0
1
a:45
55
0
1
25
30
0
1
0
1
c:12
b:13
14
d:16
0
1
f:5
e:9

- To encode:
  - Search the tree for the character to encode
  - As you progress, add "0" or "1" to right of code
  - Code is complete when you find character
- To decode a code:
  - Proceed through bit string left to right
  - For each bit, proceed left or right as indicated
  - When you reach a leaf, that is the decoded character

A Huffman Code Tree



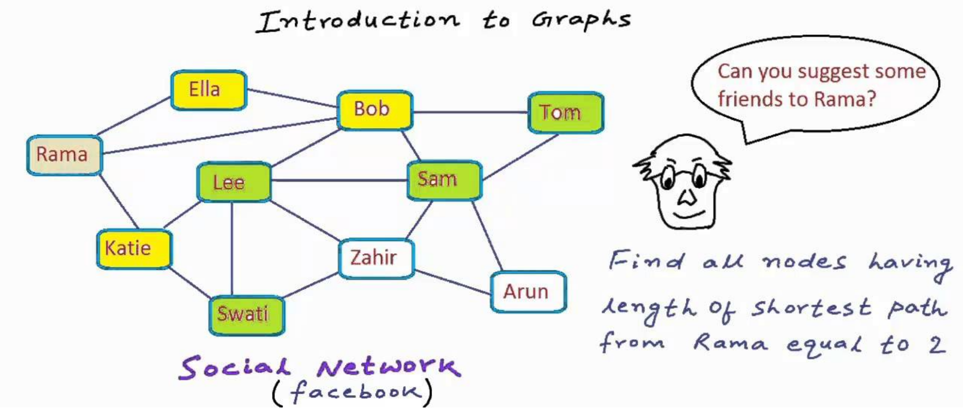
b)

start-nodes	00	01	10	11
0	3	4	1	2
1	1	2	1	2
2	3	4	1	2
3	3	4	1	2
4	3	4	1	2

c)

from node	word	symbols
0	00	A
0	01	B
0	10	C
0	11	C, C
1	00	A
1	01	A, C
1	10	B
1	11	B, C

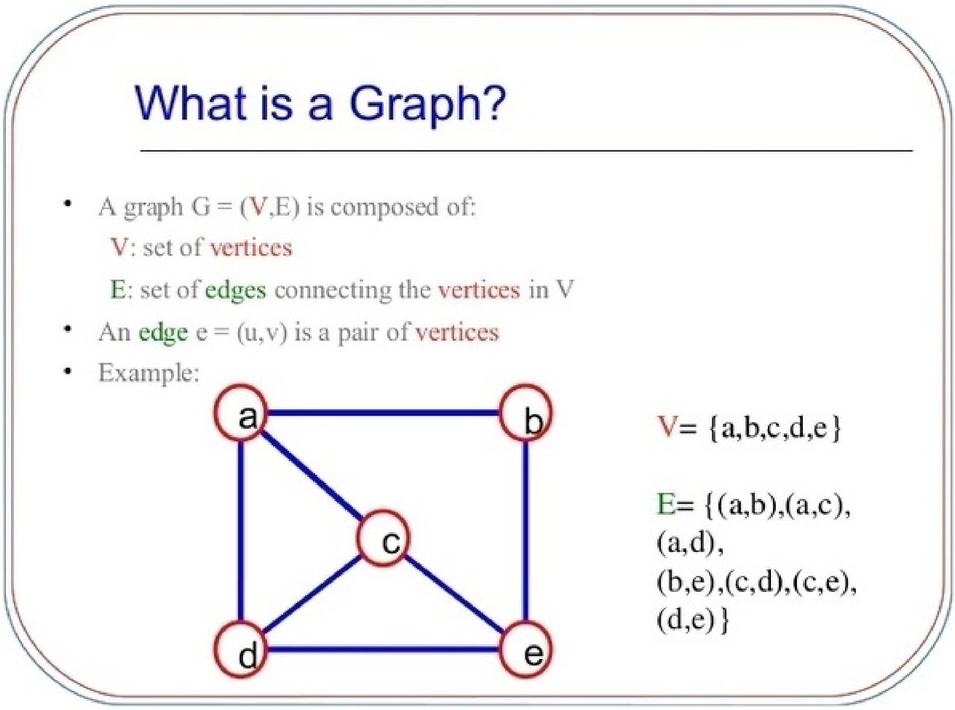
Graph



Introduction to Graphs

Social Network (facebook)

Find all nodes having length of shortest path from Rama equal to 2

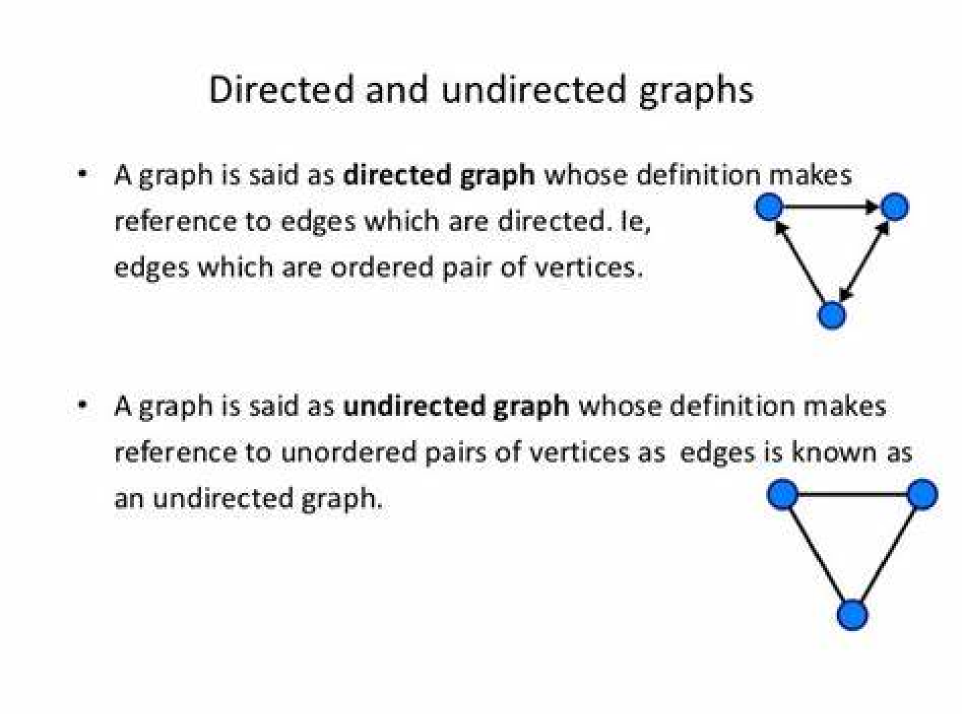


What is a Graph?

- A graph G = (V,E) is composed of:
  V: set of vertices
  E: set of edges connecting the vertices in V
- An edge e = (u,v) is a pair of vertices
- Example:

V= {a,b,c,d,e}

E= {(a,b),(a,c),
(a,d),
(b,e),(c,d),(c,e),
(d,e)}



Directed and undirected graphs

- A graph is said as **directed graph** whose definition makes reference to edges which are directed. Ie, edges which are ordered pair of vertices.
- A graph is said as **undirected graph** whose definition makes reference to unordered pairs of vertices as edges is known as an undirected graph.

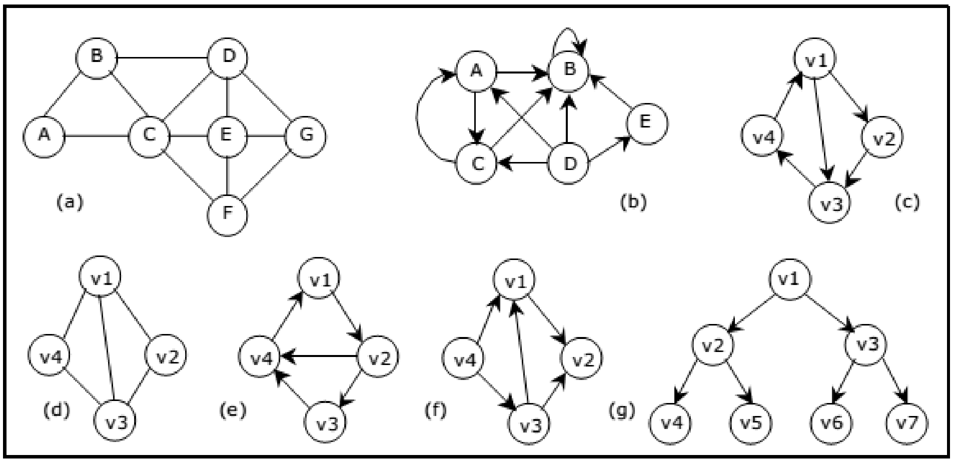
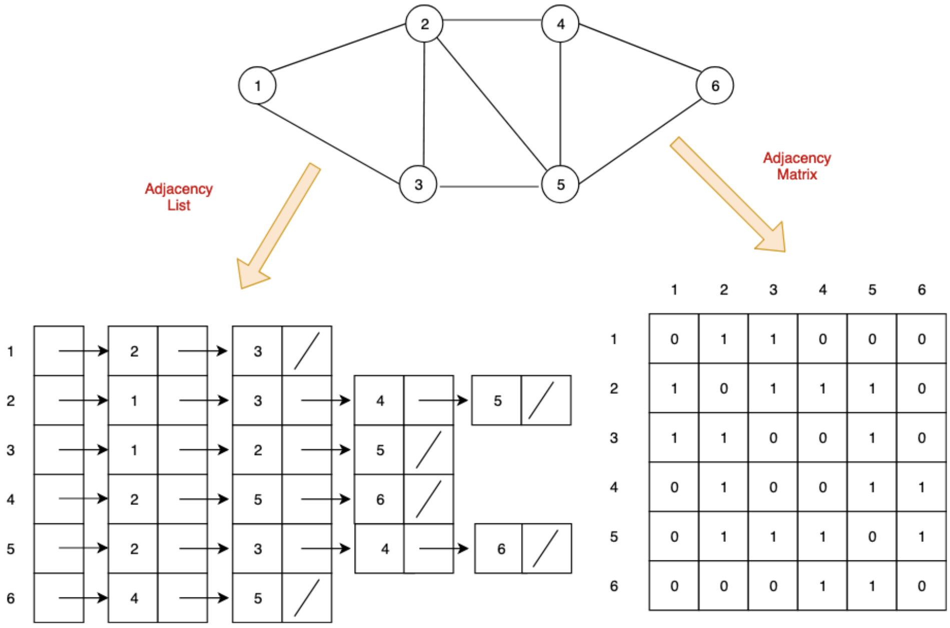
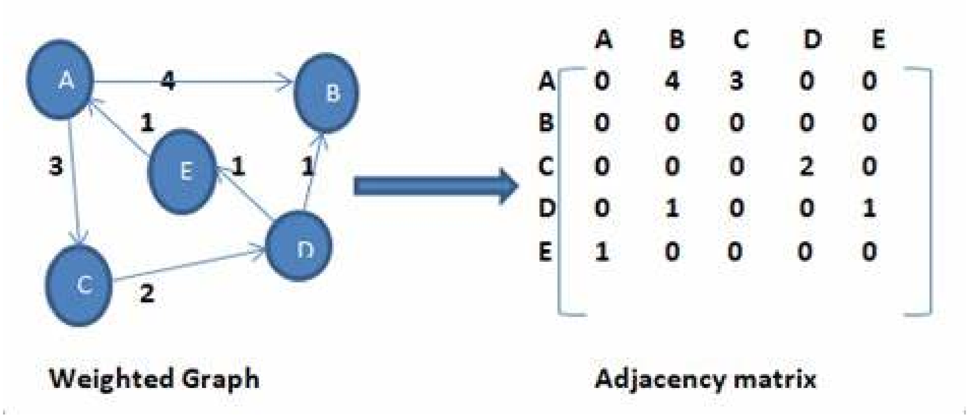


Figure 6.5.1 Various Graphs

Graph Presentation



	1	2	3	4	5	6
1	0	1	1	0	0	0
2	1	0	1	1	1	0
3	1	1	0	0	1	0
4	0	1	0	0	1	1
5	0	1	1	1	0	1
6	0	0	0	1	1	0



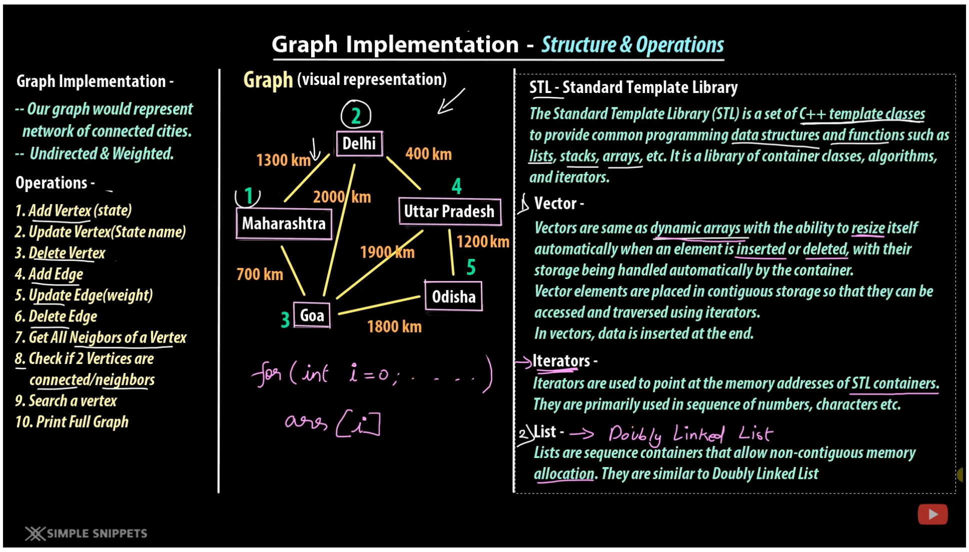
$$\begin{array}{c|ccccc} & A & B & C & D & E \\ A & 0 & 4 & 3 & 0 & 0 \\ B & 0 & 0 & 0 & 0 & 0 \\ C & 0 & 0 & 0 & 2 & 0 \\ D & 0 & 1 & 0 & 0 & 1 \\ E & 1 & 0 & 0 & 0 & 0 \end{array}$$

Weighted Graph

Adjacency matrix

In [ ]:

```
#include <iostream>
using namespace std;

const int MAX_NODE_CNT = 10;

void distFromSource(
    const bool adjacenet[][MAX_NODE_CNT],
    int dist[],
    int nodeCnt,
    int source
);

int main(){
    int nodeCnt = 5;
    bool adjacent[MAX_NODE_CNT][MAX_NODE_CNT] = {
        {1, 1, 0, 0, 1},
        {1, 1, 1, 0, 0},
        {0, 1, 1, 1, 0},
        {0, 0, 1, 1, 1},
        {1, 0, 0, 1, 1}
    };
    int dist[MAX_NODE_CNT] = {0};
    int source = 0;

    distFromSource(adjacent, dist, nodeCnt, source);

    cout << "\nThe complete result: \n";
    for (int i = 0; i < nodeCnt; i++){
        cout << dist[i] << " ";
    }

    return 0;
}

void distFromSource(const bool adjacent[][MAX_NODE_CNT],
    int dist[], int nodeCnt, int source)
{
    for (int i = 0; i< nodeCnt; i++){
        dist[i] = nodeCnt;
    }

    dist[source] = 0;
    int curDist = 1;
    int complete = 1;

    while (complete < nodeCnt){
        for (int i = 0; i < nodeCnt; i++){
            if(dist[i] == curDist - 1){
                for (int j = 0; j < nodeCnt; j++){
                    if((adjacent[i][j] == true)
                       && (dist[j] == nodeCnt)){
                           dist[j] = curDist;
                           complete++;
                    }
                }
            }
        }
        curDist++;
    }
}
```



Graph Implementation - *Structure & Operations*

Graph Implementation -

-- *Our graph would represent network of connected cities.*

-- *Undirected & Weighted.*

Operations -

1. *Add Vertex (state)*
2. *Update Vertex(State name)*
3. *Delete Vertex*
4. *Add Edge*
5. *Update Edge(weight)*
6. *Delete Edge*
7. *Get All Neigbors of a Vertex*
8. *Check if 2 Vertices are connected/neighbors*
9. *Search a vertex*
10. *Print Full Graph*

Graph (visual representation)

STL - Standard Template Library

The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as lists, stacks, arrays, etc. It is a library of container classes, algorithms, and iterators.

1) Vector -

Vectors are same as dynamic arrays with the ability to resize itself automatically when an element is inserted or deleted, with their storage being handled automatically by the container.

Vector elements are placed in contiguous storage so that they can be accessed and traversed using iterators.

In vectors, data is inserted at the end.

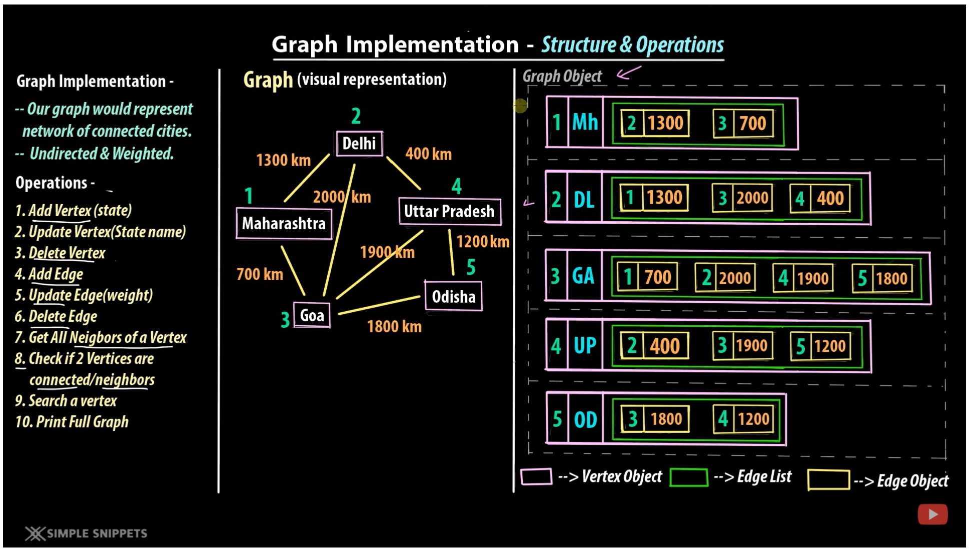
→ Iterators -

Iterators are used to point at the memory addresses of STL containers. They are primarily used in sequence of numbers, characters etc.

2) List - → Doubly Linked List

Lists are sequence containers that allow non-contiguous memory allocation. They are similar to Doubly Linked List

SIMPLE SNIPPETS



Graph Implementation - *Structure & Operations*

Graph Implementation -

-- *Our graph would represent network of connected cities.*

-- *Undirected & Weighted.*

Operations -

1. *Add Vertex (state)*
2. *Update Vertex(State name)*
3. *Delete Vertex*
4. *Add Edge*
5. *Update Edge(weight)*
6. *Delete Edge*
7. *Get All Neigbors of a Vertex*
8. *Check if 2 Vertices are connected/neighbors*
9. *Search a vertex*
10. *Print Full Graph*

Graph (visual representation)

Graph Object

1 Mh | 2 1300 | 3 700

2 DL | 1 1300 | 3 2000 | 4 400

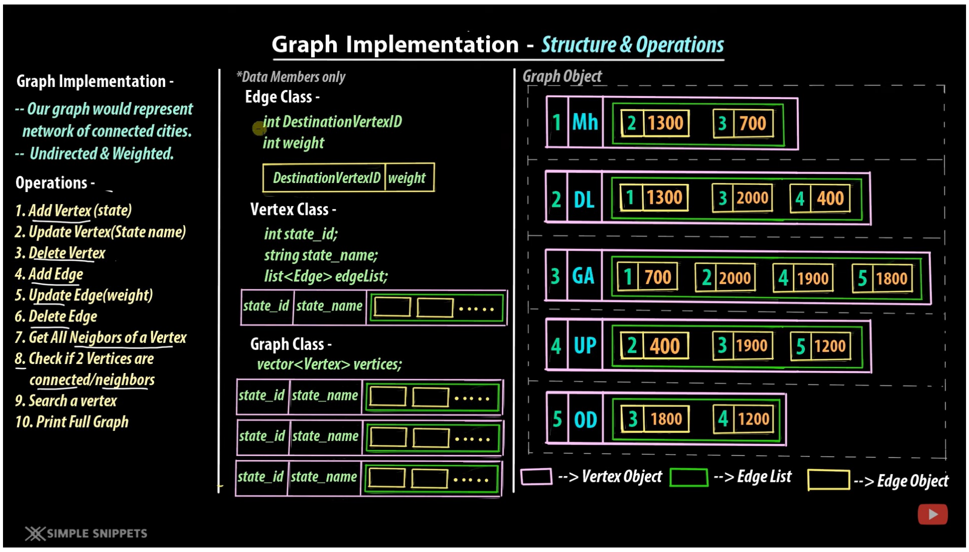
3 GA | 1 700 | 2 2000 | 4 1900 | 5 1800

4 UP | 2 400 | 3 1900 | 5 1200

5 OD | 3 1800 | 4 1200

▭ --> *Vertex Object* 🟩 --> *Edge List* 🟨 --> *Edge Object*

SIMPLE SNIPPETS



Graph Implementation - *Structure & Operations*

Graph Implementation -

-- *Our graph would represent network of connected cities.*

-- *Undirected & Weighted.*

Operations -

1. *Add Vertex (state)*
2. *Update Vertex(State name)*
3. *Delete Vertex*
4. *Add Edge*
5. *Update Edge(weight)*
6. *Delete Edge*
7. *Get All Neigbors of a Vertex*
8. *Check if 2 Vertices are connected/neighbors*
9. *Search a vertex*
10. *Print Full Graph*

**Data Members only*

Edge Class -

int DestinationVertexID
int weight

Vertex Class -

int state_id;
string state_name;
list<Edge> edgeList;

Graph Class -

vector<Vertex> vertices;

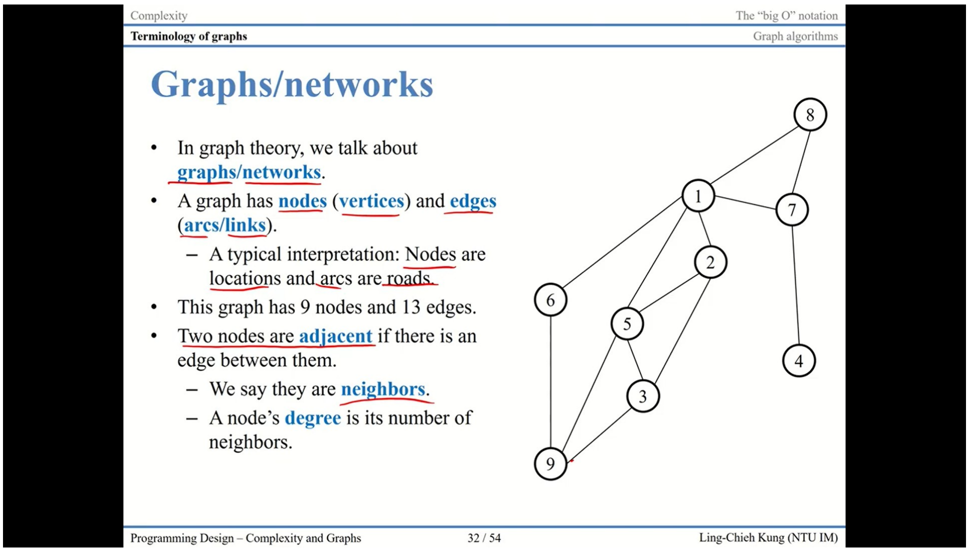
DestinationVertexID | weight
state_id | state_name | .....
state_id | state_name | .....
state_id | state_name | .....
state_id | state_name | .....
Graph Object
1 Mh | 2 1300 | 3 700
2 DL | 1 1300 | 3 2000 | 4 400
3 GA | 1 700 | 2 2000 | 4 1900 | 5 1800
4 UP | 2 400 | 3 1900 | 5 1200
5 OD | 3 1800 | 4 1200
--> Vertex Object
--> Edge List
--> Edge Object
SIMPLE SNIPPETS

In [ ]:

```cpp
#include <iostream>
#include <vector>
#include <list>
#include <iterator>
using namespace std;


class Edge {
private:
    int DestinationVertexID;
    int weight;
public:
    Edge(); // constructor
    Edge(int destVID, int w);
    void setEdgeValues(int destVID, int w);
    void setWeight(int w);
    int getDestinationVertexID();
    int getWeight();
};

Edge::Edge() : DestinationVertexID(0), weight(0) {}

Edge::Edge(int destVID, int w){
    DestinationVertexID = destVID;
    weight = w;
}

void Edge::setEdgeValues(int destVID, int w){
    DestinationVertexID = destVID;
    weight = w;
}

void Edge::setWeight(int w){
    weight = w;
}

int Edge::getDestinationVertexID(){
    return DestinationVertexID;
}

int Edge::getWeight(){
    return weight;
}
```

```
class Vertex {
private:
    int state_id;
    string state_name;
public:
    Vertex();
    Vertex(int id, string name);
    int getStateID();
    string getStateName();
    void setStateID(int id);
    void setStateName(string sname);
    list<Edge> edgeList;
    list<Edge> getEdgeList();
    void printEdgeList();
};

Vertex::Vertex() : state_id(0), state_name(""){}

Vertex::Vertex(int id, string sname){
    state_id = id;
    state_name = sname;
}

int Vertex::getStateID(){
    return state_id;
}

string Vertex::getStateName(){
    return state_name;
}


void Vertex::setStateID(int id){
    state_id = id;
}

void Vertex::setStateName(string sname){
    state_name = sname;
}

list<Edge> Vertex::getEdgeList(){
    return edgeList;
}

void Vertex::printEdgeList(){
    cout << ("[");
    for(auto it=edgeList.begin(); it!=edgeList.end(); it++){
        cout << it->getDestinationVertexID()<< "(" << it->getWeight() << ") -->";

    }
    cout << ("]") << endl;
}


class Graph {
public:
    vector<Vertex> vertices;
    bool checkIfVertexExistByID(int vid);
    bool checkIfEdgeExistByID(int fromVertex, int toVertex);
    Vertex getVertexByID(int vid);
    void addVertex(Vertex newVertex);
    void addEdgeByID(int fromVertex, int toVertex, int weight);
    void updateEdgeByID(int fromVertex, int toVertex, int newWeight);
    void deleteEdgeByID(int fromVertex, int toVertex);
    void deleteVertexByID(int vid);
    void updateVertexByID(int vid, string sname);
    void printGraph();
};


Vertex Graph::getVertexByID(int vid){
    Vertex temp;
    for(int i=0; i<vertices.size(); i++){
        temp = vertices.at(i);
        if(temp.getStateID() == vid){
            return temp;
        }
    }
    return temp;
}

bool Graph::checkIfVertexExistByID(int vid){
    for(int i=0; i<vertices.size(); i++){
        if(vertices.at(i).getStateID() == vid)
            return true;
    }
    return false;
}

void Graph::addVertex(Vertex newVertex){
    bool check = checkIfVertexExistByID(newVertex.getStateID());
    if(check){
        cout << "Vertex with this ID already exist" << endl << endl << endl;
    }
    else {
        vertices.push_back(newVertex);
        cout << "New Vertex Added Successfully" << endl << endl << endl;
    }
}

bool Graph::checkIfEdgeExistByID(int fromVertex, int toVertex){
    Vertex v = getVertexByID(fromVertex);
    list<Edge> e;
    e = v.getEdgeList();
    for(auto it = e.begin(); it != e.end(); it++){
        if(it->getDestinationVertexID() == toVertex){
            return true;
        }
    }
    return false;
}

void Graph::addEdgeByID(int fromVertex, int toVertex, int weight){
    bool check1 = checkIfVertexExistByID(fromVertex);
    bool check2 = checkIfVertexExistByID(toVertex);
    if (check1 && check2){
        bool check3 = checkIfEdgeExistByID(fromVertex, toVertex);
        if(check3){
            cout << "Edge between" << getVertexByID(fromVertex).getStateName()
            << "(" << fromVertex << ") and " << getVertexByID(fromVertex).getStateN
            << "(" << toVertex << ") already existed";
        }
        else {
            for(int i=0; i<vertices.size(); i++){
                if(vertices.at(i).getStateID() == fromVertex){
                    Edge e(toVertex, weight);
                    vertices.at(i).edgeList.push_back(e);
                }
                else if(vertices.at(i).getStateID() == toVertex){
                    Edge e(fromVertex, weight);
                    vertices.at(i).edgeList.push_back(e);
                }
            }
        }
        cout << "Edge between " << fromVertex << " and "
        << toVertex << "added edge successfully" << endl << endl;
    }
    cout << "Invalid Vertex ID entered." << endl << endl;
}

void Graph::updateEdgeByID(int fromVertex, int toVertex, int newWeight){
    bool check = checkIfEdgeExistByID(fromVertex, toVertex);
    if(check){
        for(int i=0; i<vertices.size(); i++){
            if(vertices.at(i).getStateID() == fromVertex){
                for(auto it=vertices.at(i).edgeList.begin(); it!=vertices.at(i).edg
                    if(it->getDestinationVertexID() == toVertex){
                        it->setWeight(newWeight);
                        break;
                    }
                }
            }
            else if(vertices.at(i).getStateID() == toVertex){
                for(auto it=vertices.at(i).edgeList.begin(); it!=vertices.at(i).edg
                    if(it->getDestinationVertexID() == fromVertex){
                        it->setWeight(newWeight);
                        break;
                    }
                }
            }
        }
        cout << "Edge Weight Updated Successfully" << endl << endl;
    }
    else {
        cout << "Edge Between " << getVertexByID(fromVertex).getStateID()
        << "and " << getVertexByID(fromVertex).getStateID() << " not existed";
    }
}

void Graph::deleteEdgeByID(int fromVertex, int toVertex){
    bool check = checkIfEdgeExistByID(fromVertex, toVertex);
    if(check){
        for(int i=0; i<vertices.size(); i++){
            if(vertices.at(i).getStateID() == fromVertex){
                for(auto it=vertices.at(i).edgeList.begin(); it!=vertices.at(i).edg
                    if(it->getDestinationVertexID() == toVertex){
                        vertices.at(i).edgeList.erase(it);
                        break;
                    }
                }
            }
            else if(vertices.at(i).getStateID() == toVertex){
                for(auto it=vertices.at(i).edgeList.begin(); it!=vertices.at(i).edg
                    if(it->getDestinationVertexID() == fromVertex){
                        vertices.at(i).edgeList.erase(it);
                        break;
                    }
                }
            }
        }
        cout << "Edge Deleted Successfully" << endl << endl;
    }
    else {
        cout << "Edge does not exist" << endl << endl;
    }
}

void Graph::deleteVertexByID(int vid){
    bool check = checkIfVertexExistByID(vid);
    if(check){
        int vertexIndex = 0;
        for(int i=0; i<vertices.size(); i++){
            if(vertices.at(i).getStateID() == vid){
                vertexIndex = i;
            }
        }
        for(auto it=vertices.at(vertexIndex).edgeList.begin(); it!=vertices.at(vert
            deleteEdgeByID(it->getDestinationVertexID(), vid);
        }
        vertices.erase(vertices.begin() + vertexIndex);
        cout << "Vertex Deleted Successfully" << endl;
    }
    else{
        cout << "Invalid Vertex ID entered." << endl;
    }

}

void Graph::updateVertexByID(int vid, string sname){
    bool check = checkIfVertexExistByID(vid);
    if(check){
        for(int i=0; i<vertices.size(); i++){
            if(vertices.at(i).getStateID() == vid){
                vertices.at(i).setStateName(sname);
                break;
            }
        }
        cout << "Vertex(State) Updated Successfully" << endl << endl;
    }
    else {
        cout << "Invalid Vertex ID entered." << endl;
    }
}


void Graph::printGraph(){
    for(int i=0; i<vertices.size(); i++){
        Vertex temp;
        temp = vertices.at(i);
        cout << temp.getStateName() << "(" << temp.getStateID() << ") --> ";
        temp.printEdgeList();
        cout << endl << endl;
    }
}




int main(){

    int option;
    int id, id1, id2, weight;
    Graph g;
    string sname;
    Vertex v;

    do {
        cout << "\nWelcome to Learning Data Structure and Algorithm" << endl;
        cout << "Topic: Graph" << endl << endl;

        cout << "What operation do you want to perform? "
        << "Select Option Number or Enter 0 to exit." << endl << endl;

        cout << "1. Add Vertex" << endl;
        cout << "2. Update Vertex" << endl;
        cout << "3. Delete Vertex" <<  endl;
        cout << "4. Add Edge" << endl;
        cout << "5. Update Edge" << endl;
        cout << "6. Delete Edge" << endl;
        cout << "7. Check if 2 Vertices are Neighbors" << endl;
        cout << "8. Print All Neighbors of a Vertex" << endl;
        cout << "9. Print Graph" << endl;
        cout << "10. Clear Screen" << endl;
        cout << "0. Exit Program" << endl;
        cout << endl;
        cin >> option;

        cout << endl;

        switch (option) {
            case 0:
                break;
            case 1:
                cout << "Add Vertex" << endl;
                cout << "Enter State ID: ";
                cin >> id;
                cout << "Enter State Name: ";
                cin >> sname;
                v.setStateID(id);
                v.setStateName(sname);
                g.addVertex(v);
                break;
            case 2:
                cout << "Update Vertex" << endl;
                cout << "Enter ID of Vertex(State) to Update: ";
                cin >> id;
                cout << "Enter the new State Name: ";
                cin >> sname;
                g.updateVertexByID(id, sname);
                break;
            case 3:
                cout << "Delete Vertex" <<  endl;
                cout << "Enter ID of Vertex(State) to Delete: ";
                cin >> id;
                g.deleteVertexByID(id);
                break;
            case 4:
                cout << "Add Edge" << endl;
                cout << "Enter ID of Source Vertex(State): ";
                cin >> id1;
                cout << "Enter ID of Destination Vertex(State): ";
                cin >> id2;
                cout << "Enter Weight of Edge: ";
                cin >> weight;
                g.addEdgeByID(id1, id2, weight);
                break;
            case 5:
                cout << "Update Edge" << endl;
                cout << "Enter ID of Source Vertex(State): ";
                cin >> id1;
                cout << "Enter ID of Destination Vertex(State): ";
                cin >> id2;
                cout << "Enter Updated Weight of Edge: ";
                cin >> weight;
                g.updateEdgeByID(id1, id2, weight);
                break;
            case 6:
                cout << "Delete Edge" << endl;
                cout << "Enter ID of Source Vertex(State): ";
                cin >> id1;
                cout << "Enter ID of Destination Vertex(State): ";
                cin >> id2;
                g.deleteEdgeByID(id1, id2);
                break;
            case 7:
                cout << "Check if 2 Vertices are Neighbors" << endl;
                break;
            case 8:
                cout << "Print All Neighbors of a Vertex " << endl;
                break;
            case 9:
                cout << "Print Graph" << endl;
                g.printGraph();
            case 10:
                break;
            default:
                cout << "Please enter the proper option" << endl;
                break;
        }
    } while ( option != 0);

    return 0;
}
```

Graph Algorithm



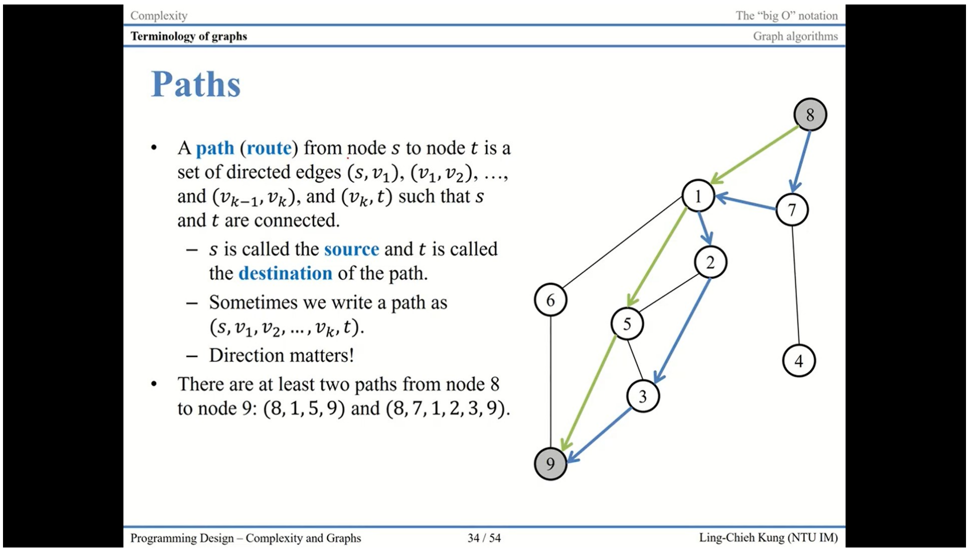
Complexity | The "big O" notation

Terminology of graphs | Graph algorithms

Graphs/networks

- In graph theory, we talk about **graphs/networks**.
- A graph has **nodes** (**vertices**) and **edges** (**arcs/links**).
  - A typical interpretation: Nodes are locations and arcs are roads.
- This graph has 9 nodes and 13 edges.
- Two nodes are **adjacent** if there is an edge between them.
  - We say they are **neighbors**.
  - A node's **degree** is its number of neighbors.

Programming Design – Complexity and Graphs 32 / 54 Ling-Chieh Kung (NTU IM)



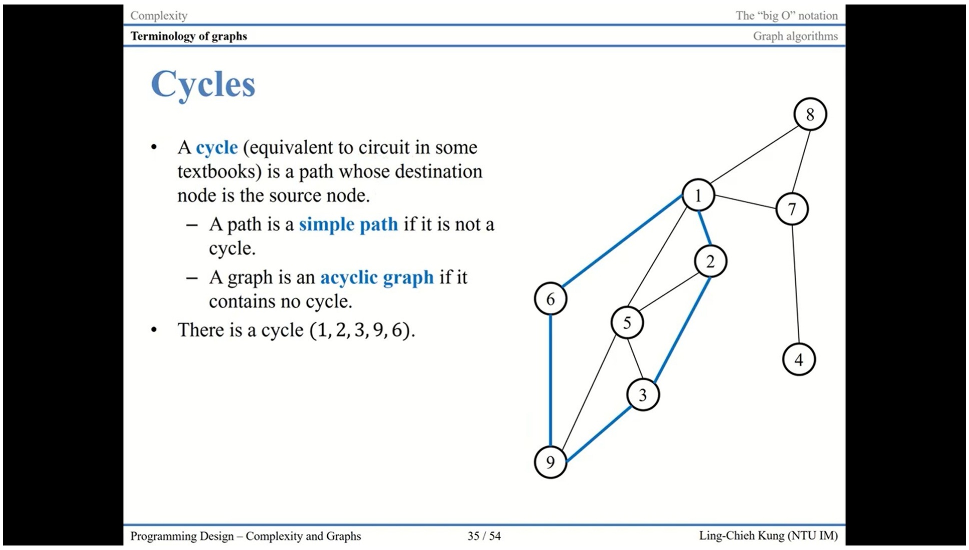
Complexity | The "big O" notation

Terminology of graphs | Graph algorithms

Paths

- A **path** (**route**) from node s to node t is a set of directed edges (s, v_1), (v_1, v_2), …, and (v_{k-1}, v_k), and (v_k, t) such that s and t are connected.
  - s is called the **source** and t is called the **destination** of the path.
  - Sometimes we write a path as $(s, v_1, v_2, \ldots, v_k, t)$.
  - Direction matters!
- There are at least two paths from node 8 to node 9: $(8, 1, 5, 9)$ and $(8, 7, 1, 2, 3, 9)$.

Programming Design – Complexity and Graphs 34 / 54 Ling-Chieh Kung (NTU IM)



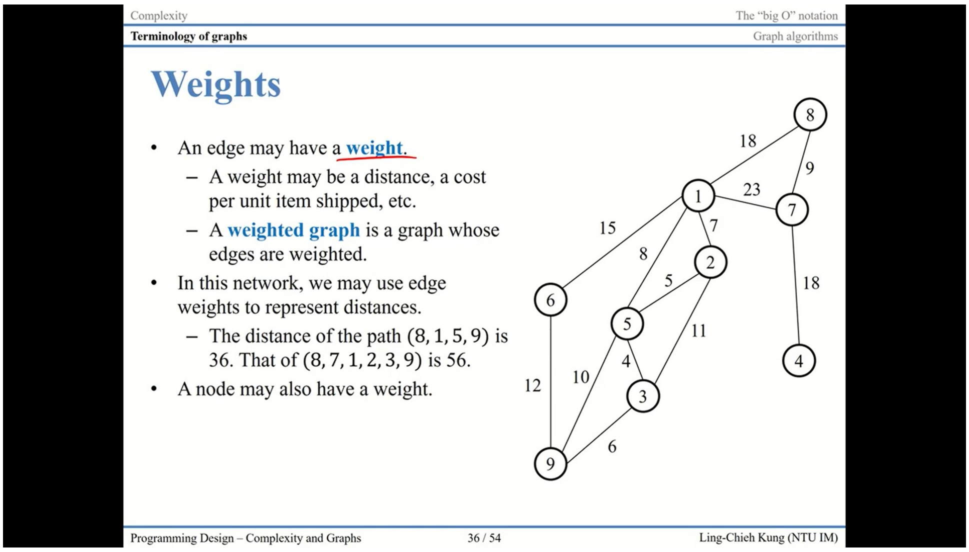
Complexity | The "big O" notation

Terminology of graphs | Graph algorithms

Cycles

- A **cycle** (equivalent to circuit in some textbooks) is a path whose destination node is the source node.
  - A path is a **simple path** if it is not a cycle.
  - A graph is an **acyclic graph** if it contains no cycle.
- There is a cycle $(1, 2, 3, 9, 6)$.

Programming Design – Complexity and Graphs 35 / 54 Ling-Chieh Kung (NTU IM)



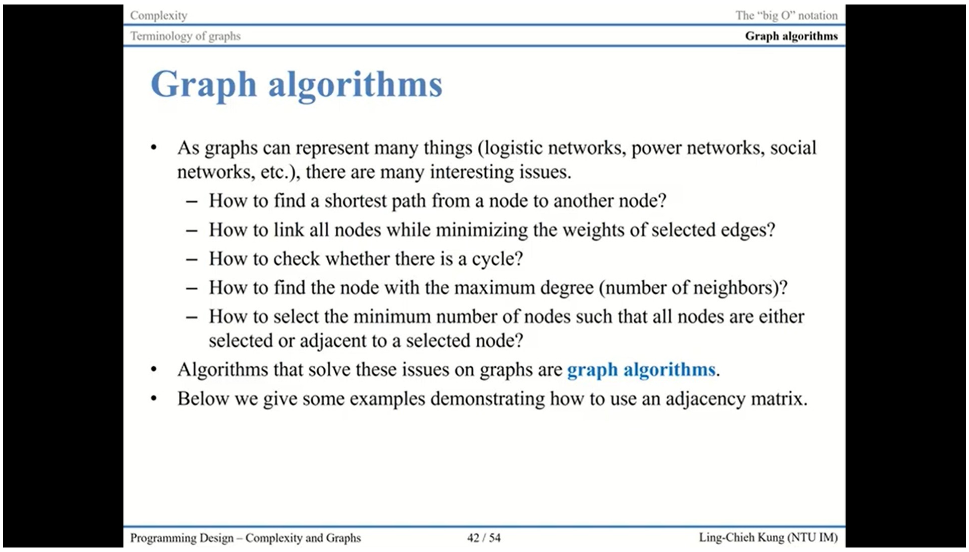
Complexity The "big O" notation

Terminology of graphs Graph algorithms

Weights

- An edge may have a **weight.**
  - A weight may be a distance, a cost per unit item shipped, etc.
  - A **weighted graph** is a graph whose edges are weighted.
- In this network, we may use edge weights to represent distances.
  - The distance of the path (8, 1, 5, 9) is 36. That of (8, 7, 1, 2, 3, 9) is 56.
- A node may also have a weight.

Programming Design – Complexity and Graphs 36 / 54 Ling-Chieh Kung (NTU IM)



Complexity The "big O" notation

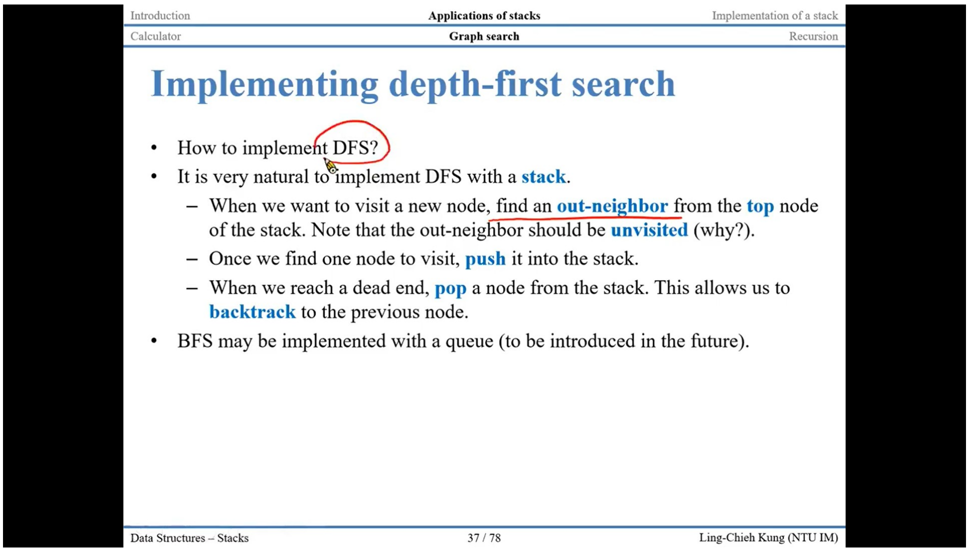
Terminology of graphs Graph algorithms

Graph algorithms

- As graphs can represent many things (logistic networks, power networks, social networks, etc.), there are many interesting issues.
  - How to find a shortest path from a node to another node?
  - How to link all nodes while minimizing the weights of selected edges?
  - How to check whether there is a cycle?
  - How to find the node with the maximum degree (number of neighbors)?
  - How to select the minimum number of nodes such that all nodes are either selected or adjacent to a selected node?
- Algorithms that solve these issues on graphs are **graph algorithms**.
- Below we give some examples demonstrating how to use an adjacency matrix.

Programming Design – Complexity and Graphs 42 / 54 Ling-Chieh Kung (NTU IM)

DFS



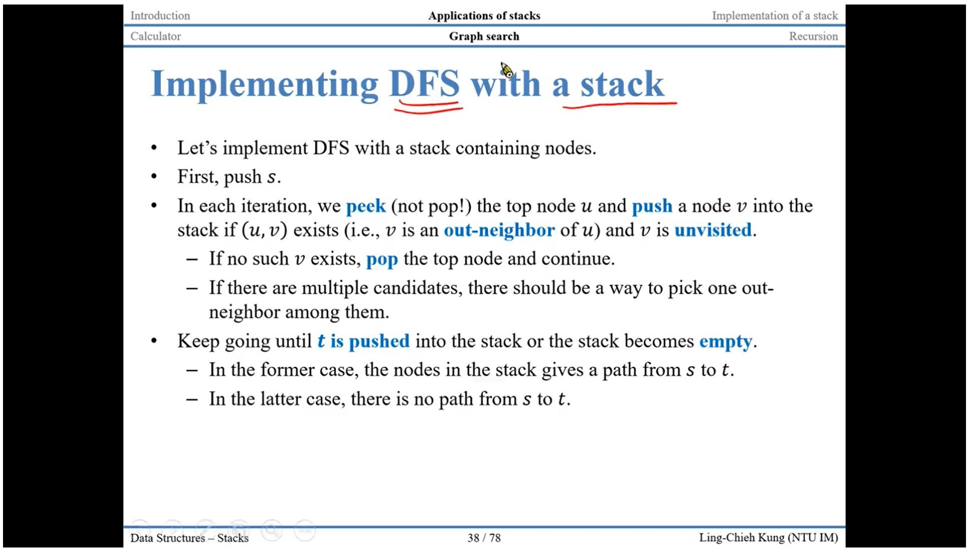
Introduction Applications of stacks Implementation of a stack

Calculator Graph search Recursion

Implementing depth-first search

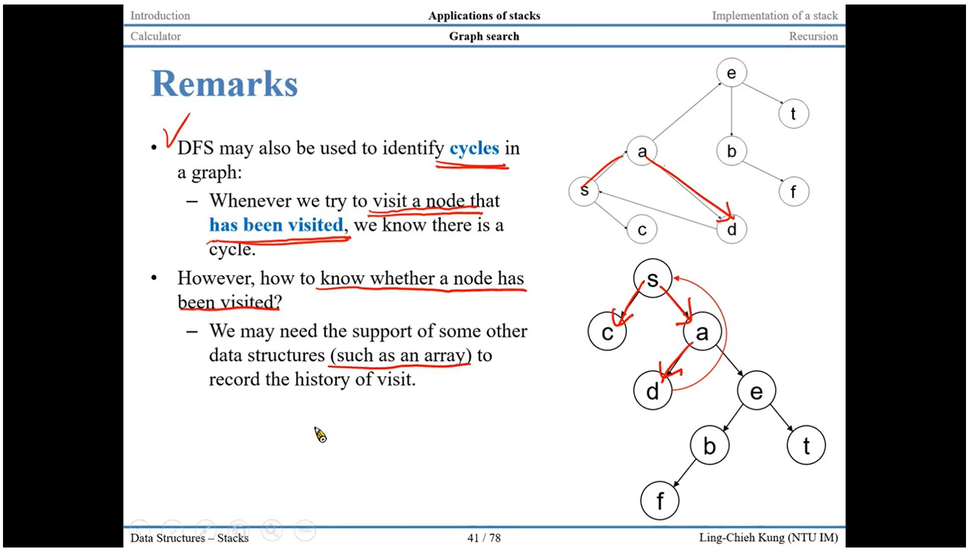
- How to implement DFS?
- It is very natural to implement DFS with a **stack**.
  - When we want to visit a new node, find an **out-neighbor** from the **top** node of the stack. Note that the out-neighbor should be **unvisited** (why?).
  - Once we find one node to visit, **push** it into the stack.
  - When we reach a dead end, **pop** a node from the stack. This allows us to **backtrack** to the previous node.
- BFS may be implemented with a queue (to be introduced in the future).

Data Structures – Stacks 37 / 78 Ling-Chieh Kung (NTU IM)



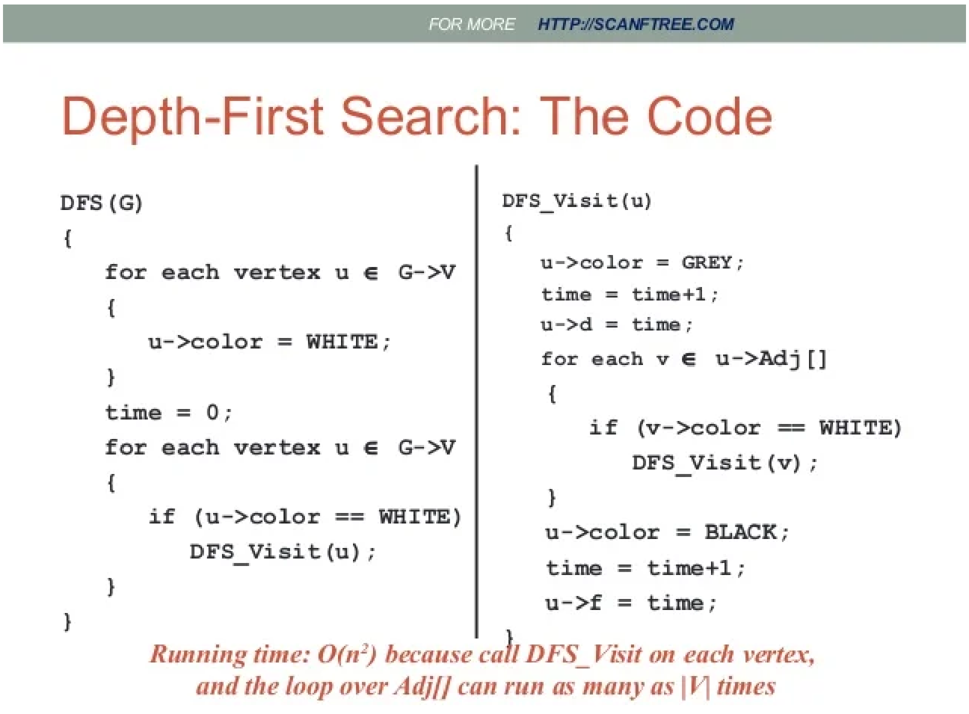
Implementing DFS with a stack

- Let's implement DFS with a stack containing nodes.
- First, push s.
- In each iteration, we **peek** (not pop!) the top node u and **push** a node v into the stack if (u, v) exists (i.e., v is an **out-neighbor** of u) and v is **unvisited**.
  - If no such v exists, **pop** the top node and continue.
  - If there are multiple candidates, there should be a way to pick one out-neighbor among them.
- Keep going until ***t* is pushed** into the stack or the stack becomes **empty**.
  - In the former case, the nodes in the stack gives a path from s to t.
  - In the latter case, there is no path from s to t.



Remarks

- ✓ DFS may also be used to identify **cycles** in a graph:
  - Whenever we try to visit a node that **has been visited**, we know there is a cycle.
- However, how to know whether a node has been visited?
  - We may need the support of some other data structures (such as an array) to record the history of visit.



Depth-First Search: The Code

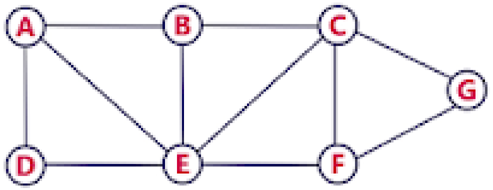
```
DFS(G)
{
    for each vertex u ∈ G->V
    {
        u->color = WHITE;
    }
    time = 0;
    for each vertex u ∈ G->V
    {
        if (u->color == WHITE)
            DFS_Visit(u);
    }
}
```

```
DFS_Visit(u)
{
    u->color = GREY;
    time = time+1;
    u->d = time;
    for each v ∈ u->Adj[]
    {
        if (v->color == WHITE)
            DFS_Visit(v);
    }
    u->color = BLACK;
    time = time+1;
    u->f = time;
}
```

Running time: $O(n^2)$ because call DFS_Visit on each vertex, and the loop over Adj[] can run as many as |V| times

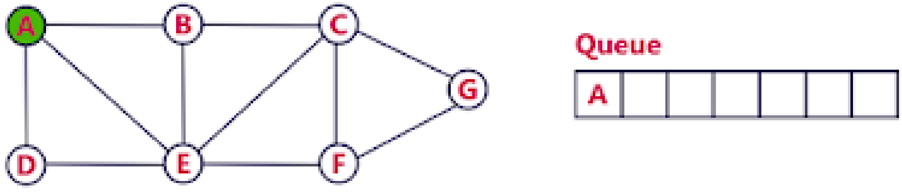
BFS

Consider the following example graph to perform BFS traversal



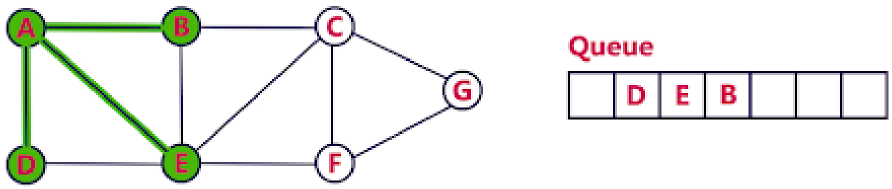
Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.



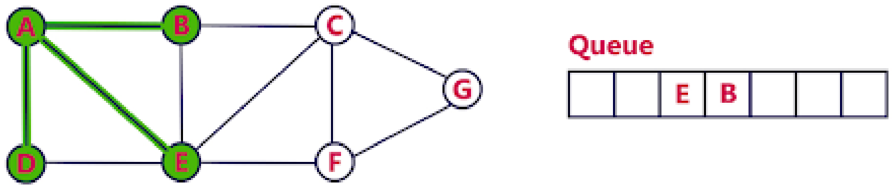
Step 2:

- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..



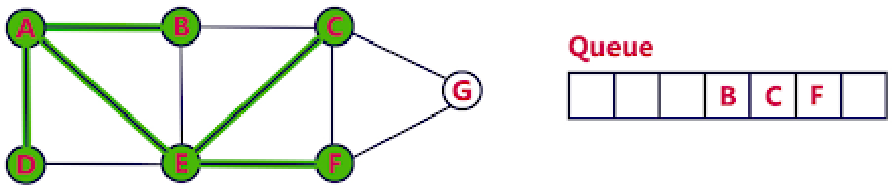
Step 3:

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.



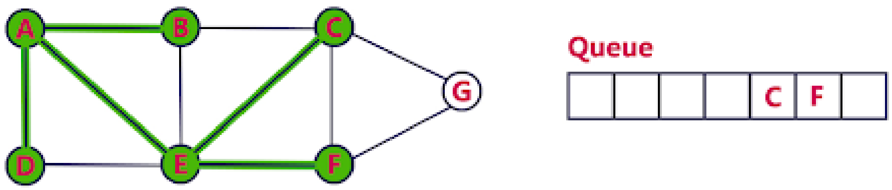
Step 4:

- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.



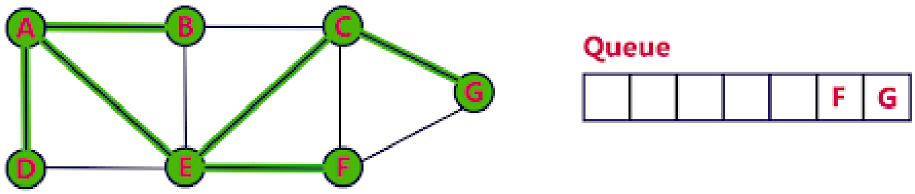
Step 5:

- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.



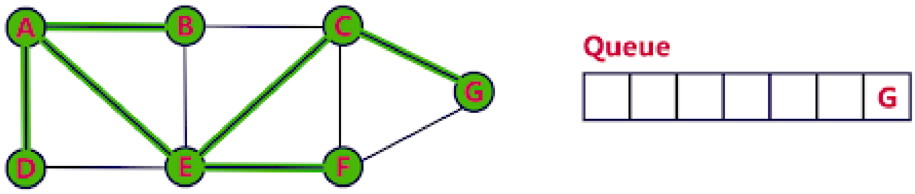
Step 6:

- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.



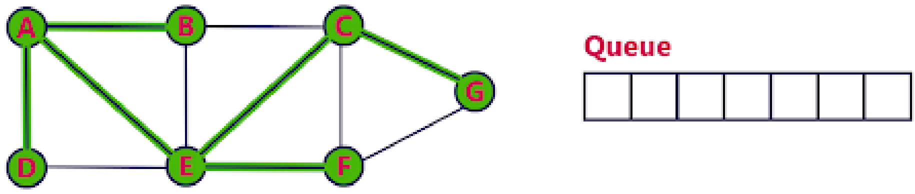
Step 7:

- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.

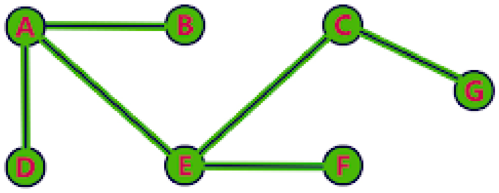


Step 8:

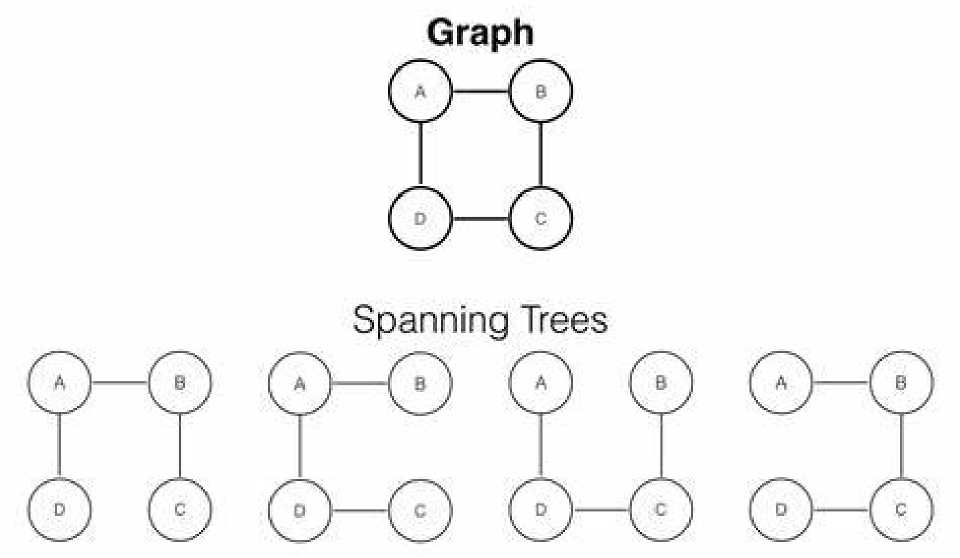
- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.

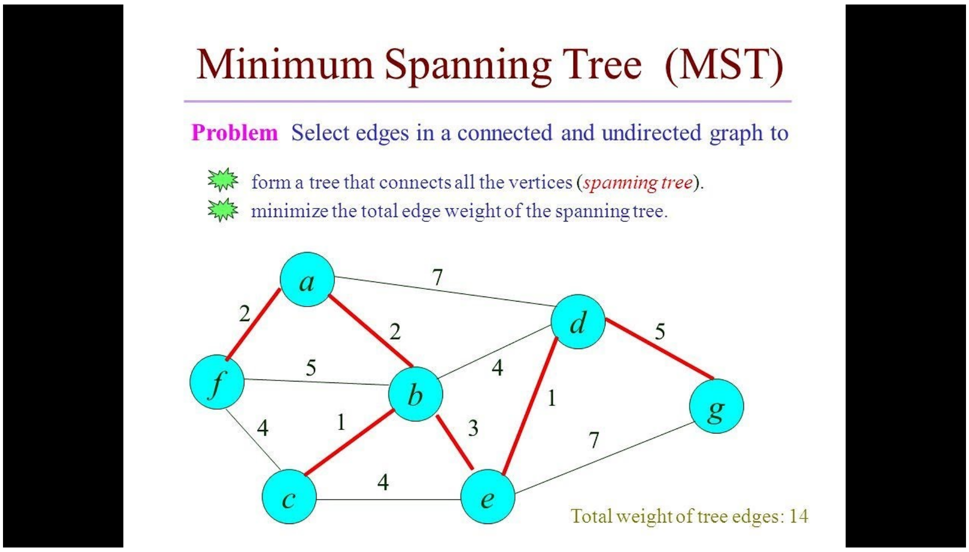


- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...



Graph - Spanning Tree



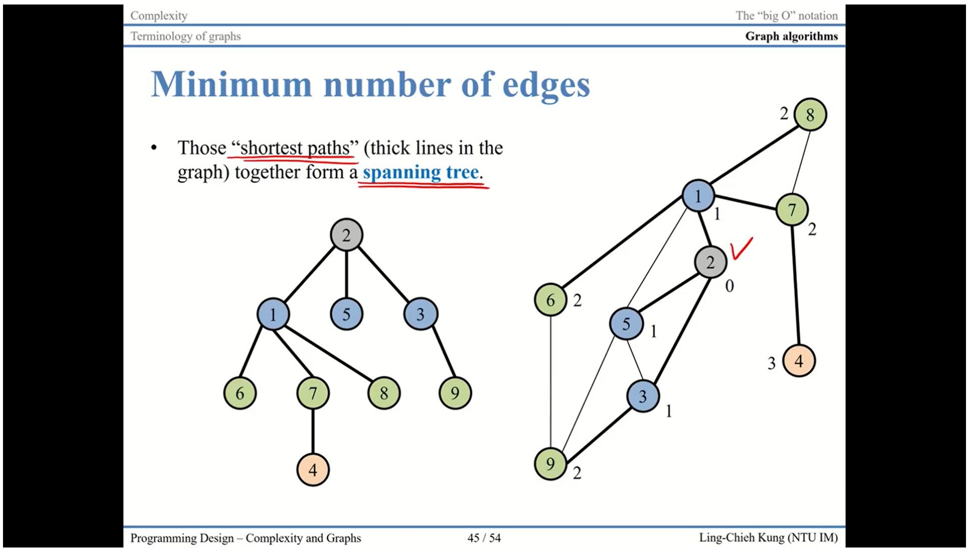


Minimum Spanning Tree (MST)

Problem Select edges in a connected and undirected graph to

- form a tree that connects all the vertices (*spanning tree*).
- minimize the total edge weight of the spanning tree.

Total weight of tree edges: 14



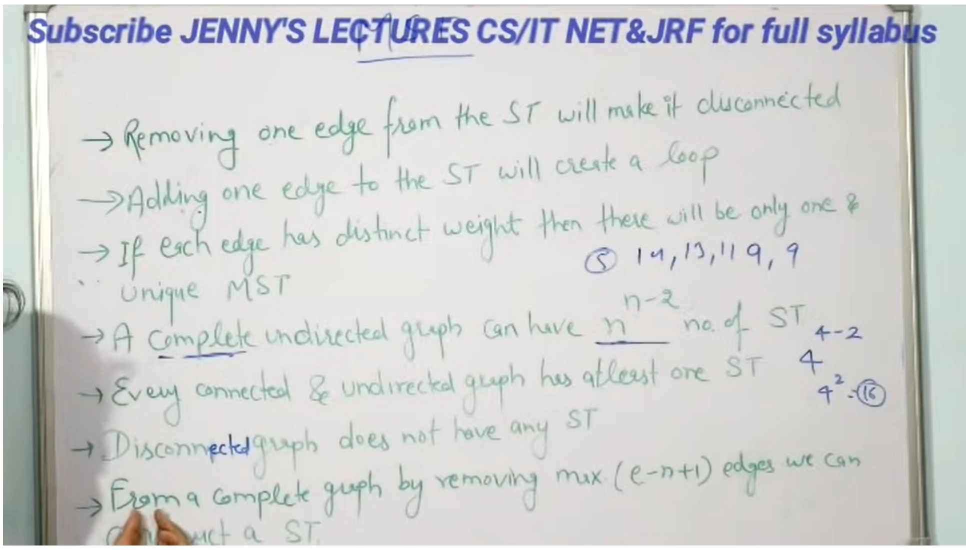
Complexity | The "big O" notation

Terminology of graphs | **Graph algorithms**

Minimum number of edges

- Those "shortest paths" (thick lines in the graph) together form a **spanning tree**.

Programming Design – Complexity and Graphs 45 / 54 Ling-Chieh Kung (NTU IM)



Subscribe JENNY'S LECTURES CS/IT NET&JRF for full syllabus

→ Removing one edge from the ST will make it disconnected

→ Adding one edge to the ST will create a loop

→ If each edge has distinct weight then there will be only one & unique MST

5 14, 13, 11 9, 9

→ A complete undirected graph can have n^{n-2} no. of ST

4^{4-2}

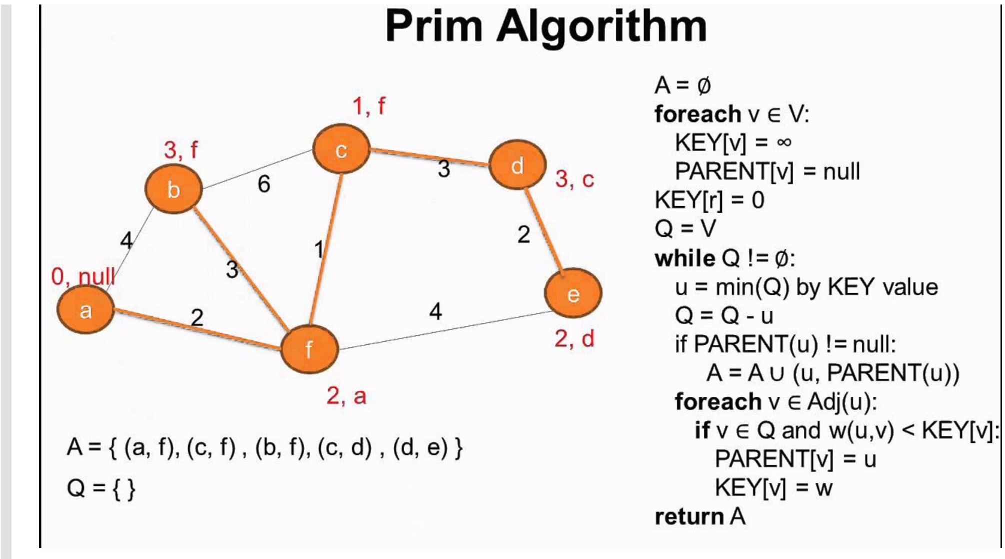
$4^2 = 16$

→ Every connected & undirected graph has atleast one ST

→ Disconnected graph does not have any ST

→ From a complete graph by removing max (e-n+1) edges we can construct a ST.

Prim's Algorithm for Minimum Spanning Tree (MST)



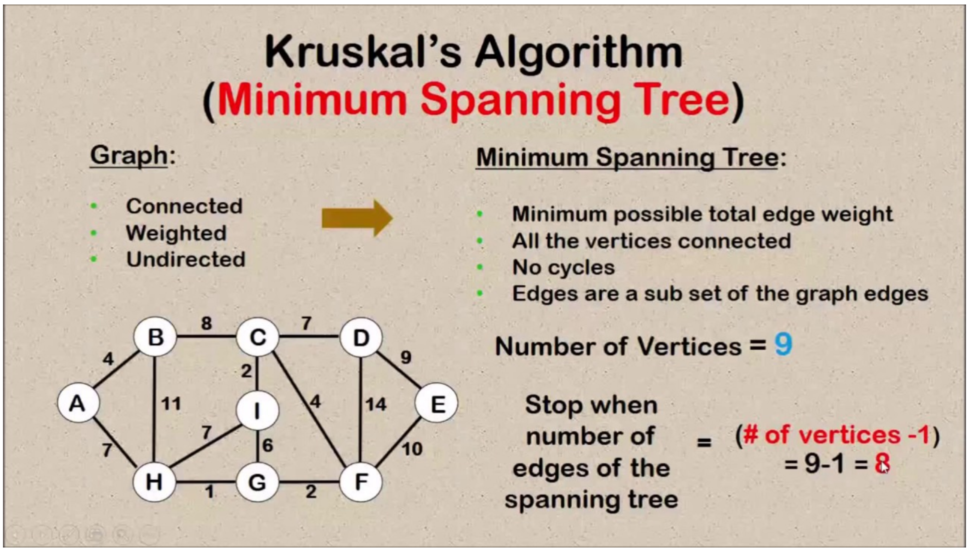
Prim Algorithm

A = { (a, f), (c, f) , (b, f), (c, d) , (d, e) }

Q = { }

```
A = Ø
foreach v ∈ V:
  KEY[v] = ∞
  PARENT[v] = null
KEY[r] = 0
Q = V
while Q != Ø:
  u = min(Q) by KEY value
  Q = Q - u
  if PARENT(u) != null:
     A = A ∪ (u, PARENT(u))
  foreach v ∈ Adj(u):
    if v ∈ Q and w(u,v) < KEY[v]:
      PARENT[v] = u
      KEY[v] = w
return A
```

Kruskals Algorithm for Minimum Spanning Tree- Greedy method



Kruskal's Algorithm (Minimum Spanning Tree)

Graph:

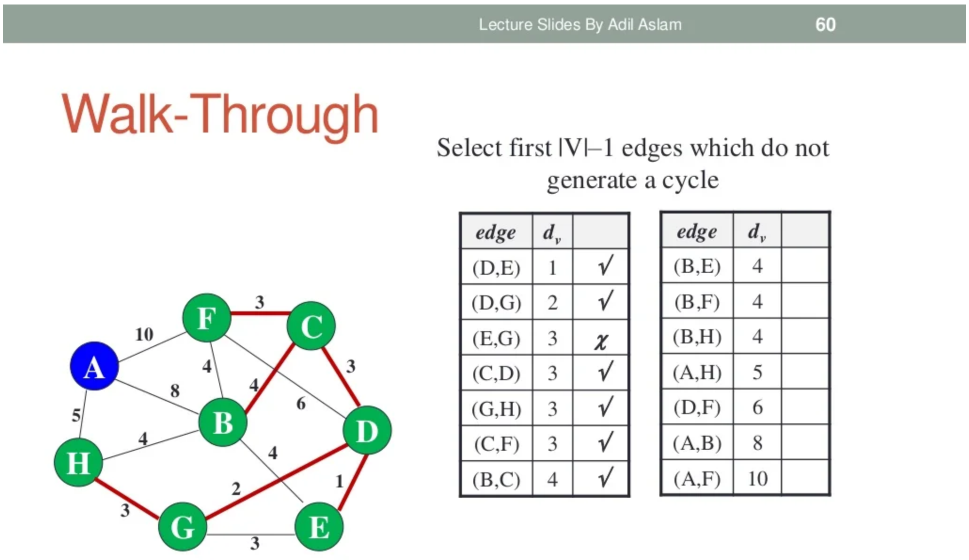
- Connected
- Weighted
- Undirected

Minimum Spanning Tree:

- Minimum possible total edge weight
- All the vertices connected
- No cycles
- Edges are a sub set of the graph edges

Number of Vertices = 9

Stop when number of edges of the spanning tree = (# of vertices -1) = 9-1 = 8



Lecture Slides By Adil Aslam 60

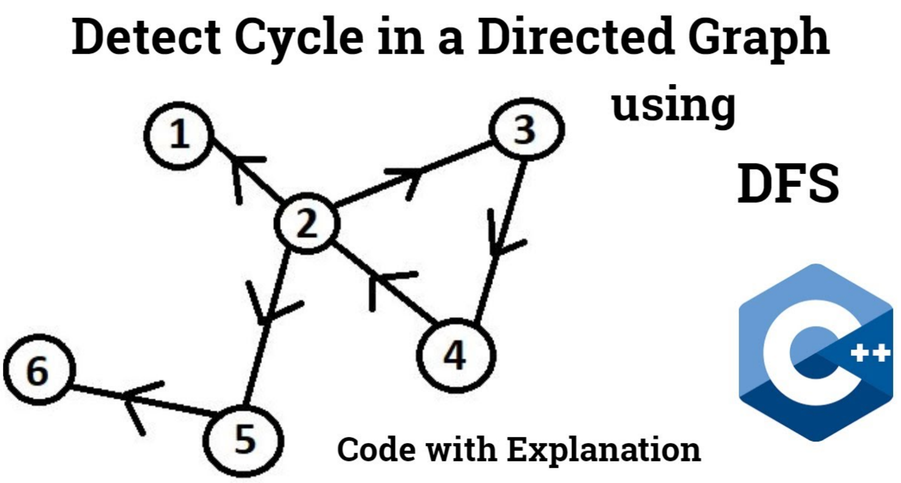
Walk-Through

Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

Detect Cycle in Directed Graph

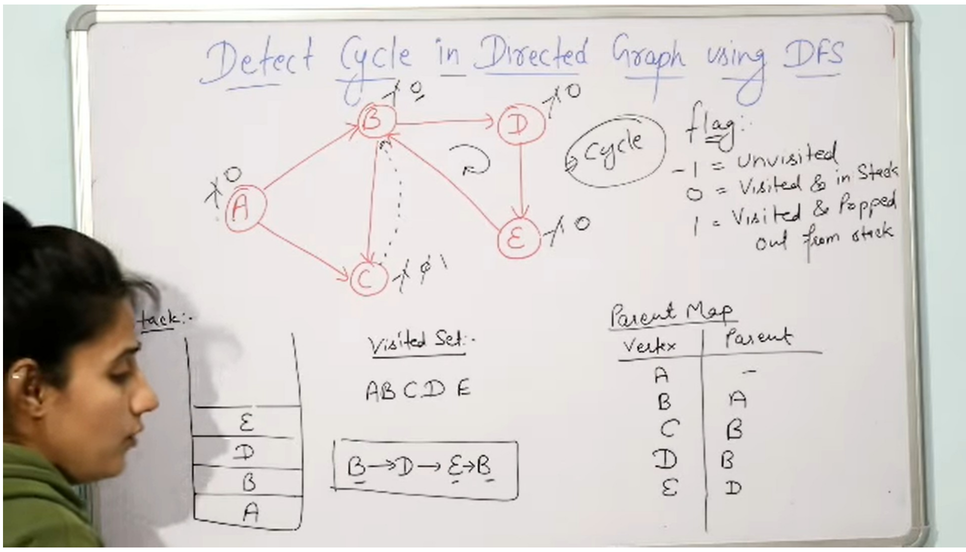


Detect Cycle in a Directed Graph

using

DFS

Code with Explanation



Detect Cycle in Directed Graph using DFS

flag:

-1 = Unvisited

0 = Visited & in Stack

1 = Visited & Popped out from stack

Cycle

Stack:

E
D
B
A

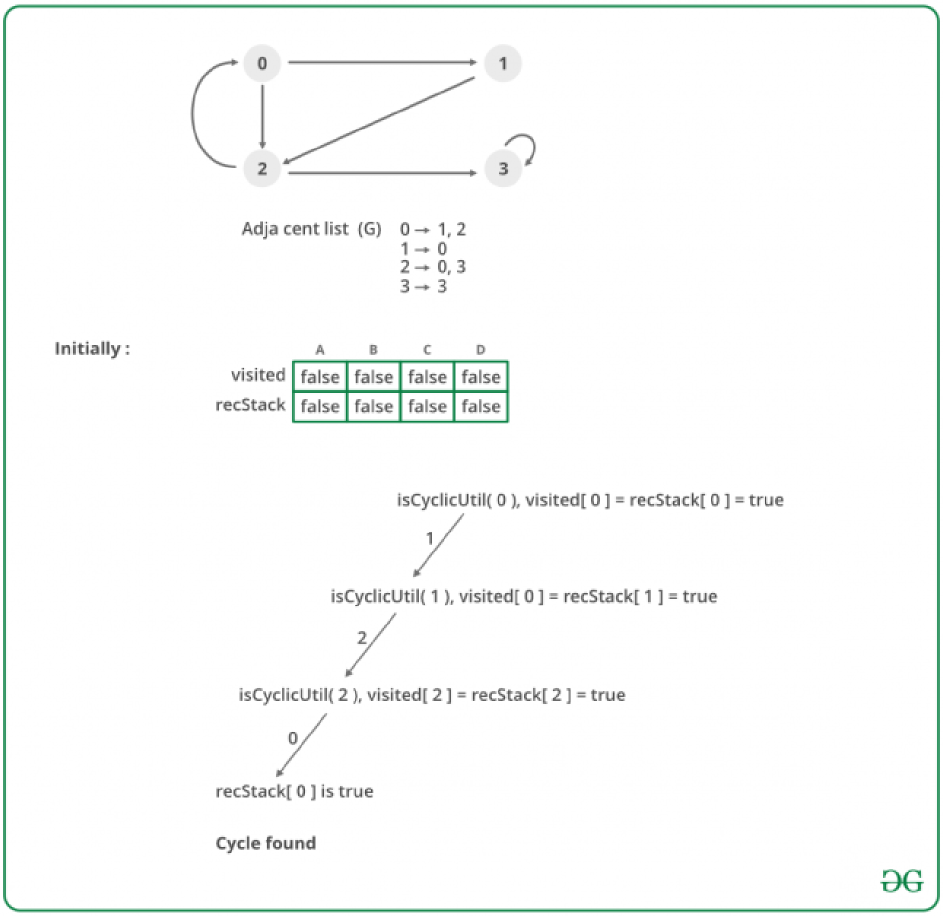
Visited Set:

A B C D E

B→D→E→B

Parent Map

Vertex	Parent
A	-
B	A
C	B
D	B
E	D



Adja cent list (G) 0 → 1, 2
1 → 0
2 → 0, 3
3 → 3

Initially :

	A	B	C	D
visited	false	false	false	false
recStack	false	false	false	false

isCyclicUtil(0), visited[0] = recStack[0] = true

1

isCyclicUtil(1), visited[0] = recStack[1] = true

2

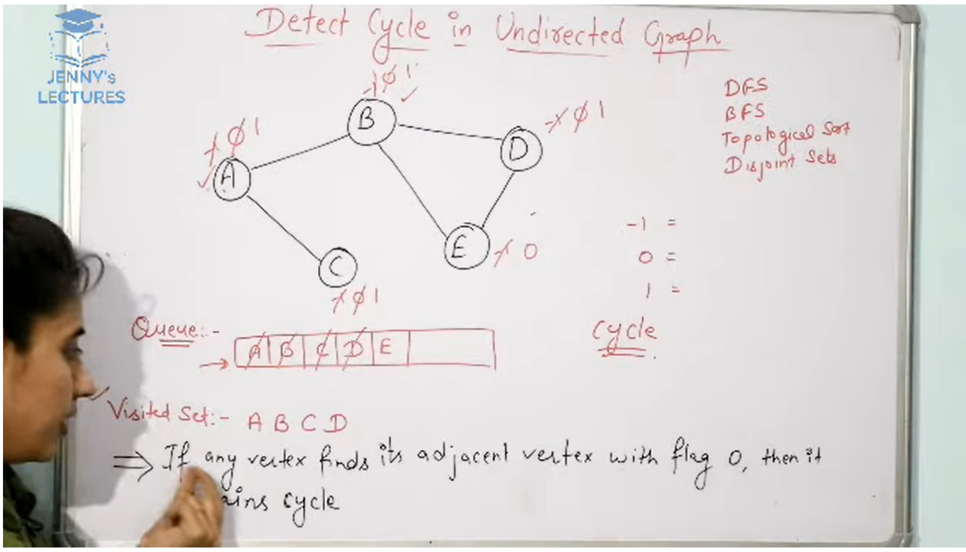
isCyclicUtil(2), visited[2] = recStack[2] = true

0

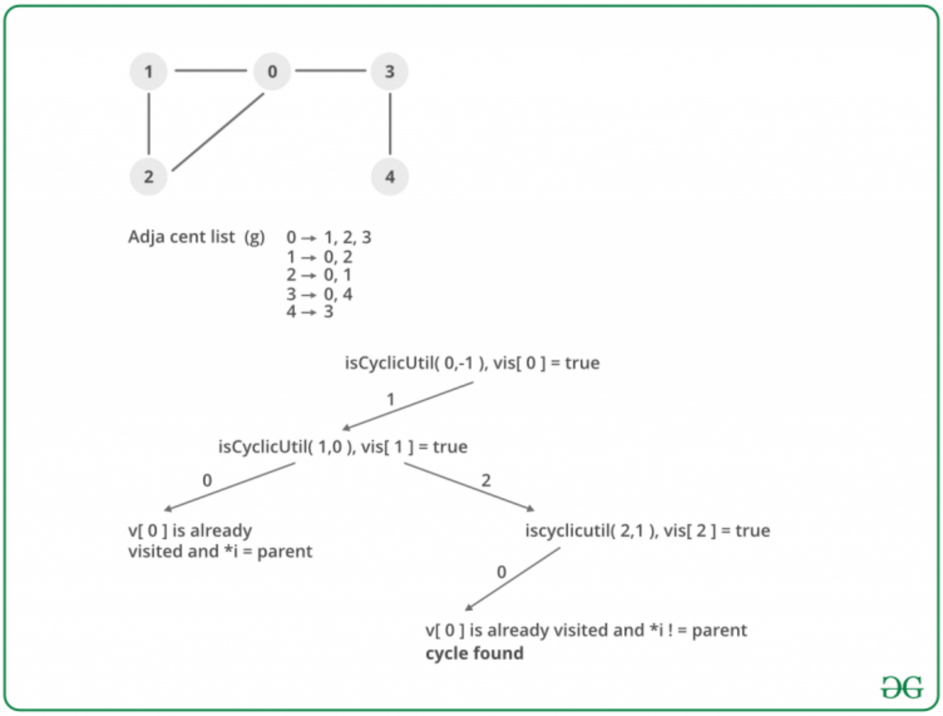
recStack[0] is true

Cycle found

Detect Cycle in Undirected Graph



⇒ If any vertex finds its adjacent vertex with flag 0, then it [illegible]ains cycle



Adja cent list (g) 0 → 1, 2, 3
1 → 0, 2
2 → 0, 1
3 → 0, 4
4 → 3

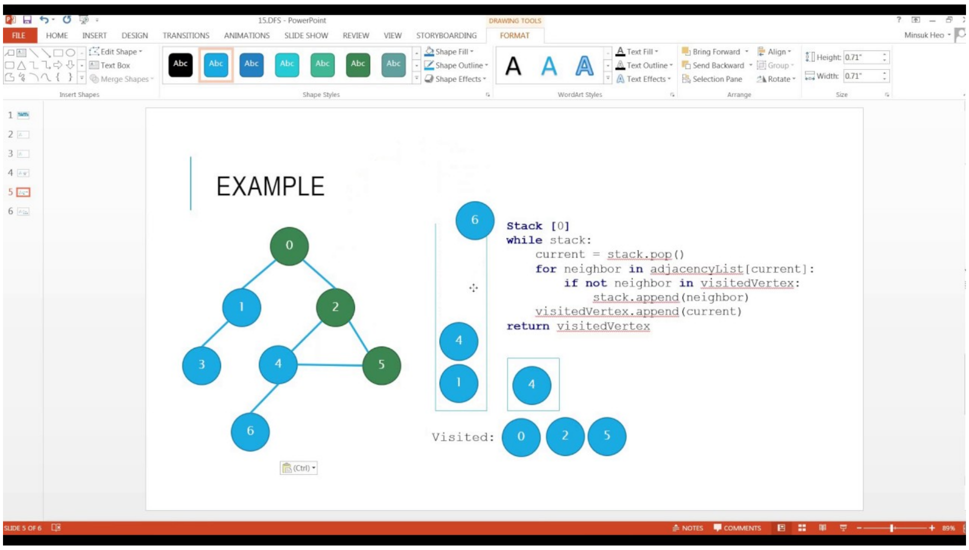
isCyclicUtil(0,-1), vis[0] = true

isCyclicUtil(1,0), vis[1] = true

v[0] is already visited and *i = parent

iscyclicutil(2,1), vis[2] = true

v[0] is already visited and *i ! = parent
cycle found

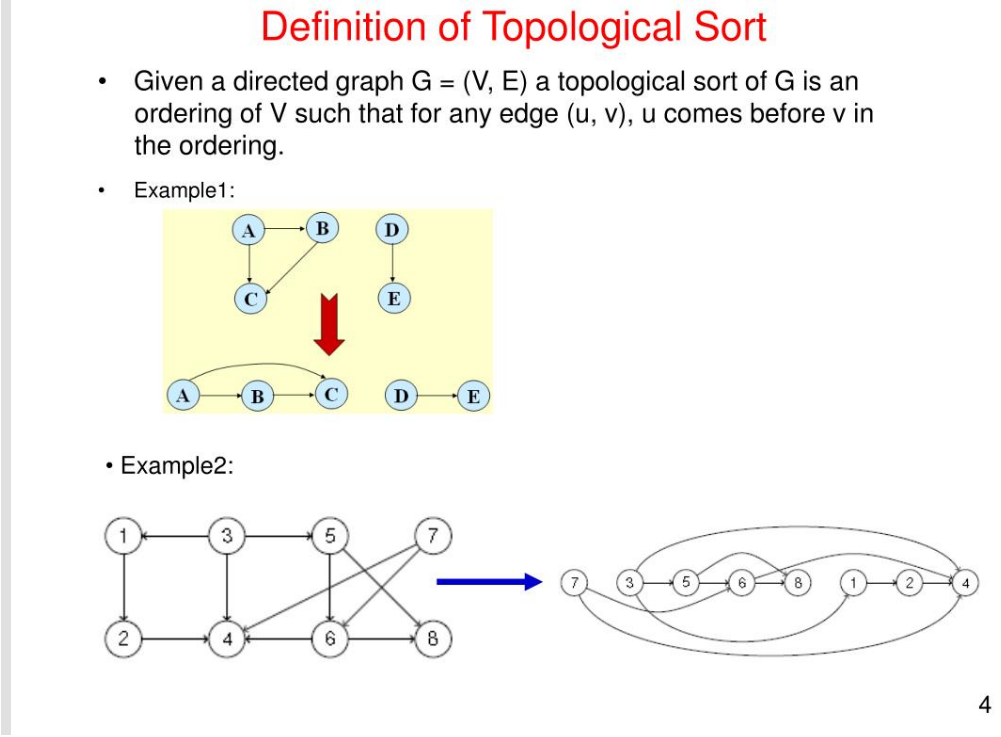


EXAMPLE

```
Stack [0]
while stack:
    current = stack.pop()
    for neighbor in adjacencyList[current]:
        if not neighbor in visitedVertex:
            stack.append(neighbor)
    visitedVertex.append(current)
return visitedVertex
```

Visited:

Topological Sorting for Graph

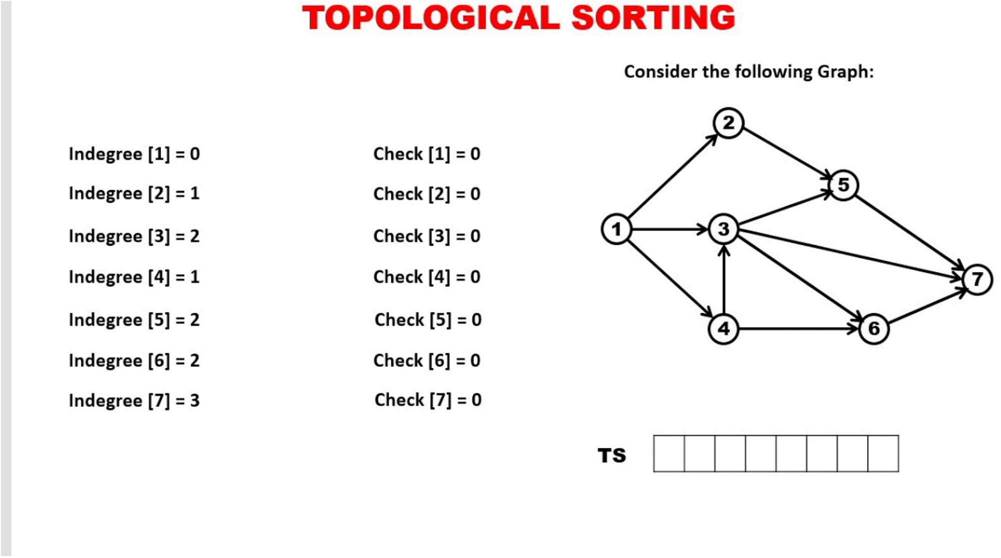


Definition of Topological Sort

- Given a directed graph G = (V, E) a topological sort of G is an ordering of V such that for any edge (u, v), u comes before v in the ordering.
- Example1:

- Example2:

4



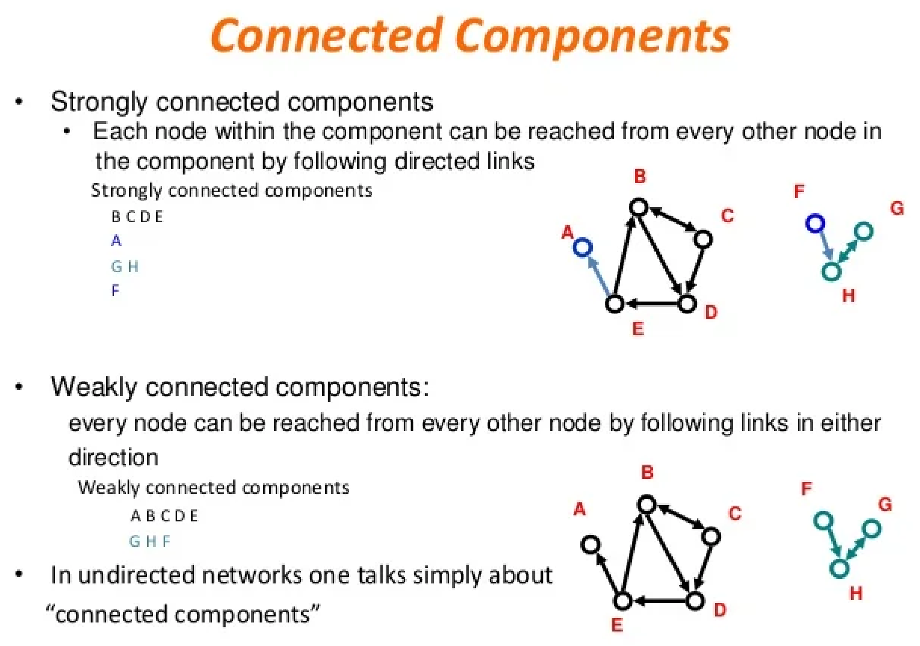
TOPOLOGICAL SORTING

Consider the following Graph:

Indegree [1] = 0	Check [1] = 0
Indegree [2] = 1	Check [2] = 0
Indegree [3] = 2	Check [3] = 0
Indegree [4] = 1	Check [4] = 0
Indegree [5] = 2	Check [5] = 0
Indegree [6] = 2	Check [6] = 0
Indegree [7] = 3	Check [7] = 0

TS

Connected Components



Connected Components

- Strongly connected components
  - Each node within the component can be reached from every other node in the component by following directed links

Strongly connected components

B C D E

A

G H

F

- Weakly connected components:

every node can be reached from every other node by following links in either direction

Weakly connected components

A B C D E

G H F

- In undirected networks one talks simply about "connected components"

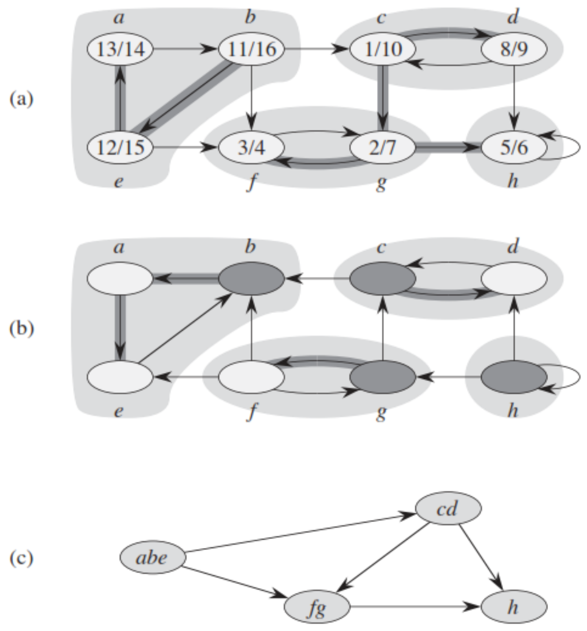
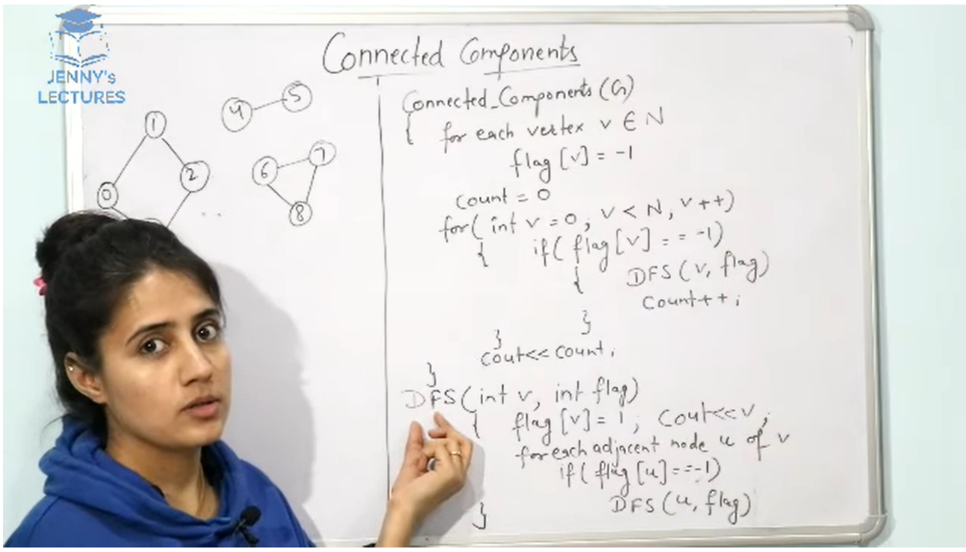
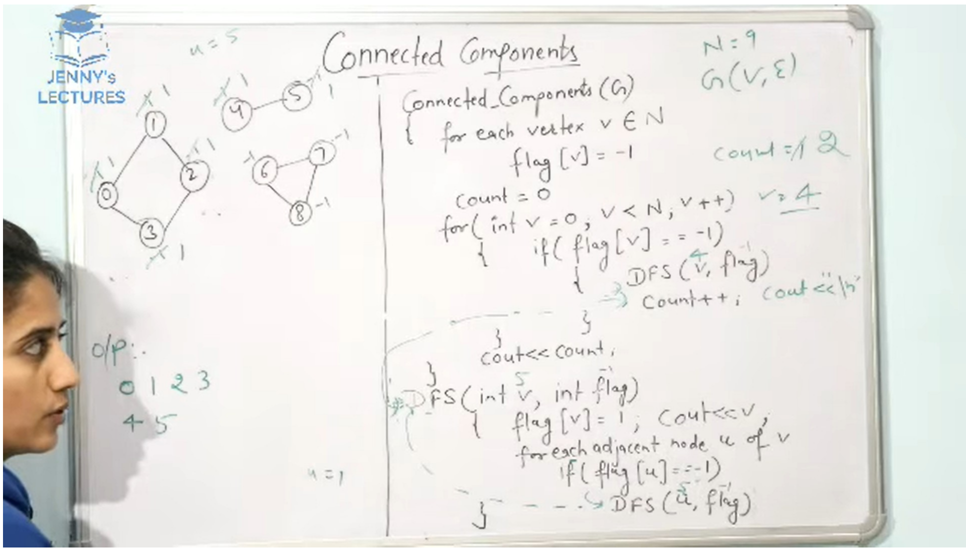


Figure 22.9 **(a)** A directed graph G. Each shaded region is a strongly connected component of G. Each vertex is labeled with its discovery and finishing times in a depth-first search, and tree edges are shaded. **(b)** The graph G^T, the transpose of G, with the depth-first forest computed in line 3 of STRONGLY-CONNECTED-COMPONENTS shown and tree edges shaded. Each strongly connected component corresponds to one depth-first tree. Vertices b, c, g, and h, which are heavily shaded, are the roots of the depth-first trees produced by the depth-first search of G^T. **(c)** The acyclic component graph G^{SCC} obtained by contracting all edges within each strongly connected component of G so that only a single vertex remains in each component.



Connected Components

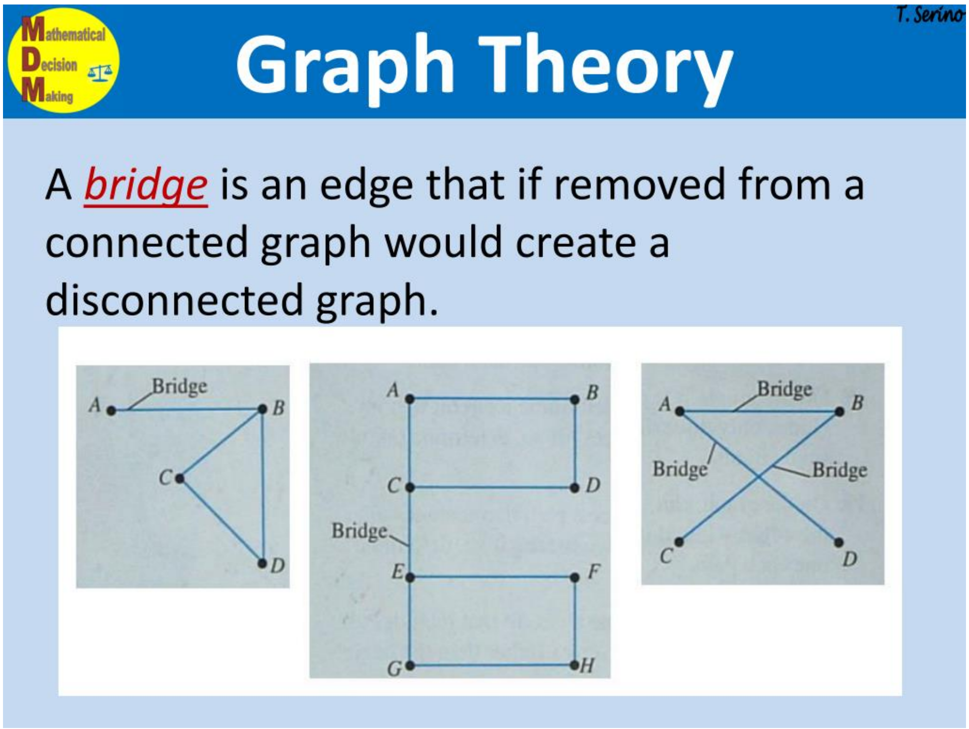
```
Connected_Components (G)
{  for each vertex v ∈ N
        flag [v] = -1
   count = 0
   for( int v = 0, v < N, v++)
   {      if( flag [v] = = -1)
              DFS (v, flag)
          {   Count++;
          }
   }
   cout<< count;
}
DFS(int v, int flag)
{   flag [v] = 1; Cout<<v;
    foreach adjacent node u of v
        if( flag [u] = = -1)
            DFS (u, flag)
}
```



Connected Components

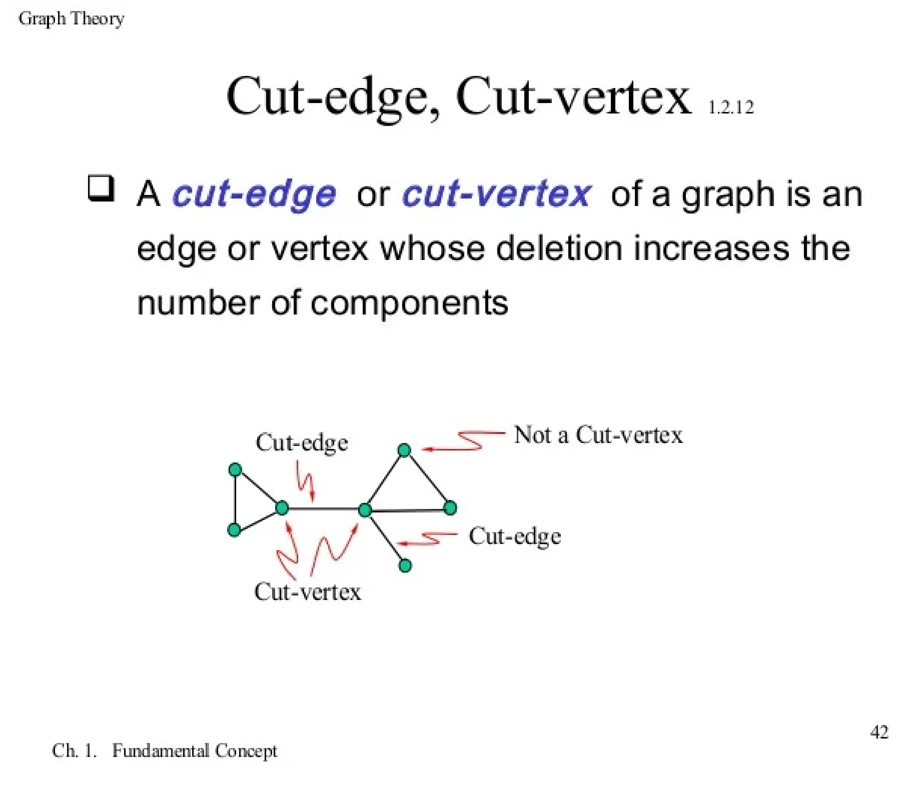
```
Connected_Components (G)
{  for each vertex v ∈ N
        flag [v] = -1
   count = 0
   for( int v = 0, v < N, v++)
   {      if( flag [v] = = -1)
              DFS (v, flag)
          {   Count++;  cout<<"\n"
          }
   }
   cout<< count;
}
DFS(int v, int flag)
{   flag [v] = 1; Cout<<v;
    foreach adjacent node u of v
        if( flag [u] = = -1)
            DFS (u, flag)
}
```

Find Bridges(Cut Edge) in a Graph



Graph Theory

A *bridge* is an edge that if removed from a connected graph would create a disconnected graph.



Graph Theory

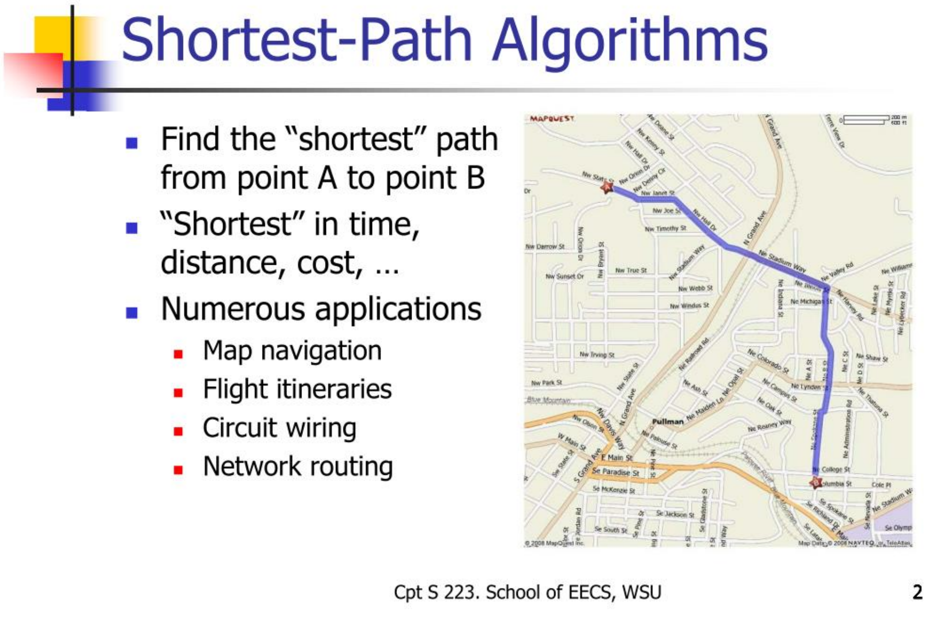
Cut-edge, Cut-vertex 1.2.12

❑ A ***cut-edge*** or ***cut-vertex*** of a graph is an edge or vertex whose deletion increases the number of components

Ch. 1. Fundamental Concept

42

SSSP(Shortest Path Algorithms)

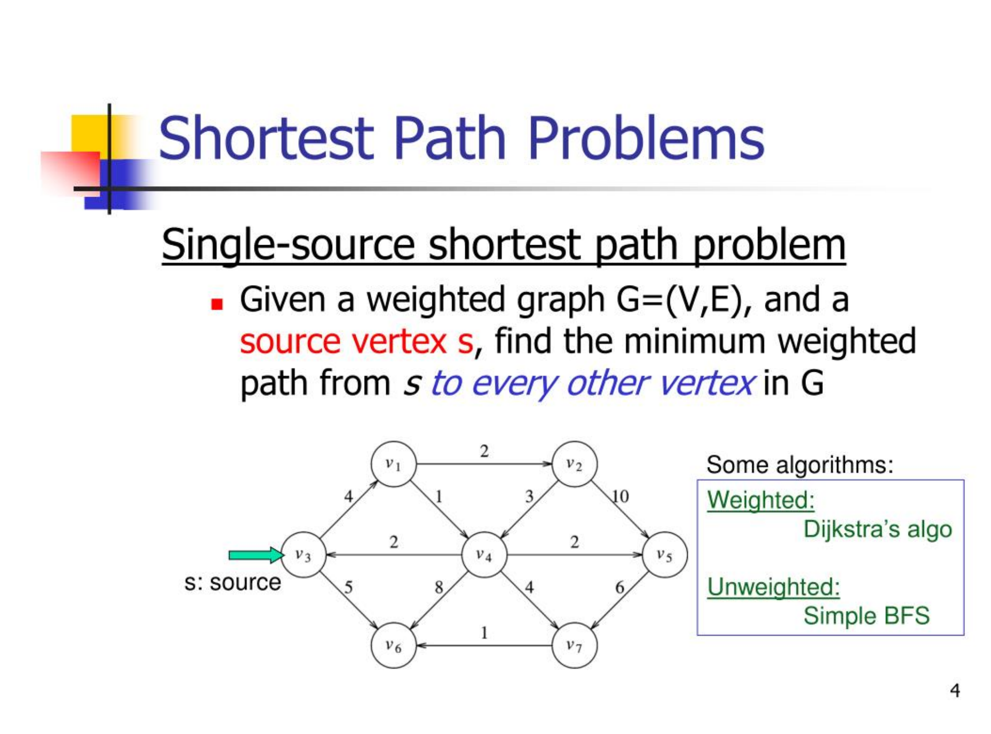


Shortest-Path Algorithms

- Find the "shortest" path from point A to point B
- "Shortest" in time, distance, cost, …
- Numerous applications
  - Map navigation
  - Flight itineraries
  - Circuit wiring
  - Network routing

Cpt S 223. School of EECS, WSU

2



Shortest Path Problems

Single-source shortest path problem

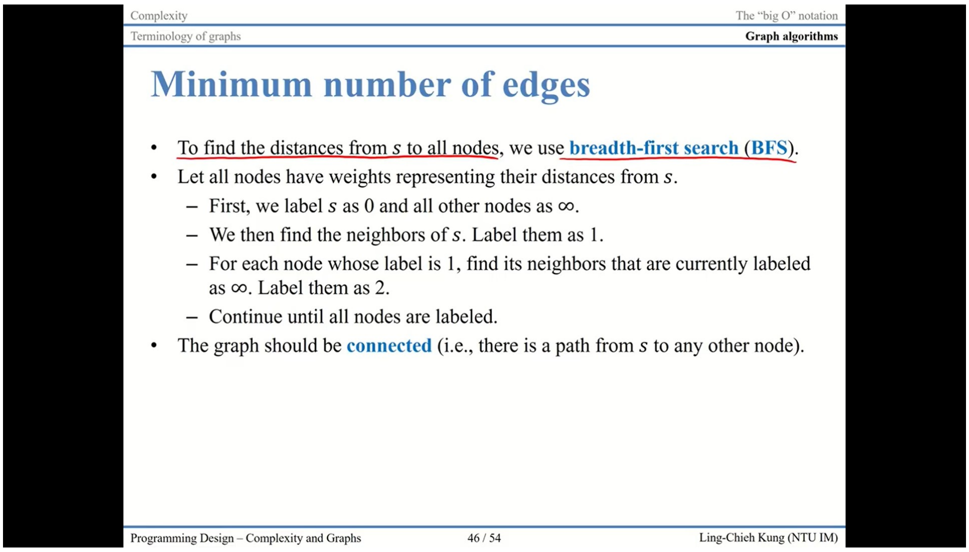
- Given a weighted graph G=(V,E), and a source vertex s, find the minimum weighted path from *s to every other vertex* in G

Some algorithms:

Weighted:
Dijkstra's algo

Unweighted:
Simple BFS

4

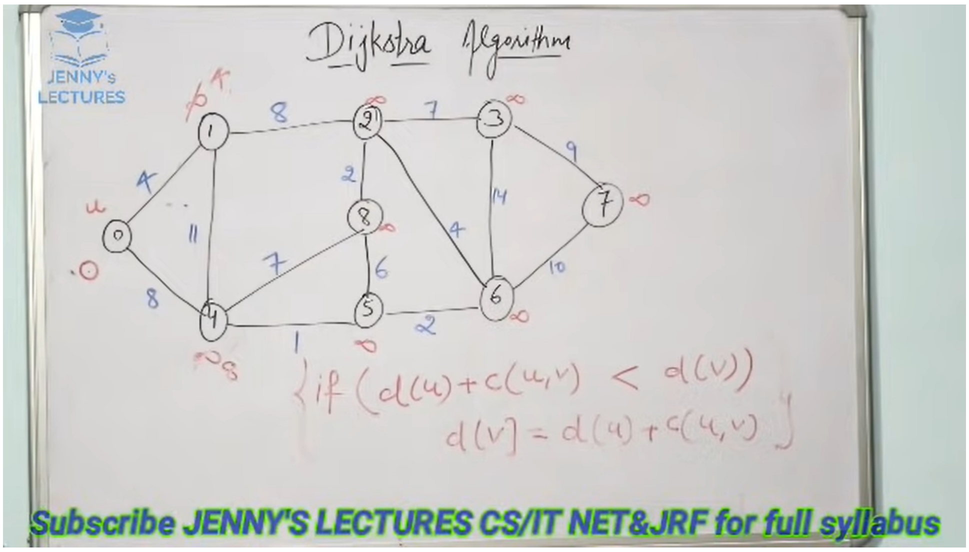


Complexity | The "big O" notation
Terminology of graphs | Graph algorithms

Minimum number of edges

- To find the distances from *s* to all nodes, we use **breadth-first search (BFS)**.
- Let all nodes have weights representing their distances from *s*.
  - First, we label *s* as 0 and all other nodes as ∞.
  - We then find the neighbors of *s*. Label them as 1.
  - For each node whose label is 1, find its neighbors that are currently labeled as ∞. Label them as 2.
  - Continue until all nodes are labeled.
- The graph should be **connected** (i.e., there is a path from *s* to any other node).

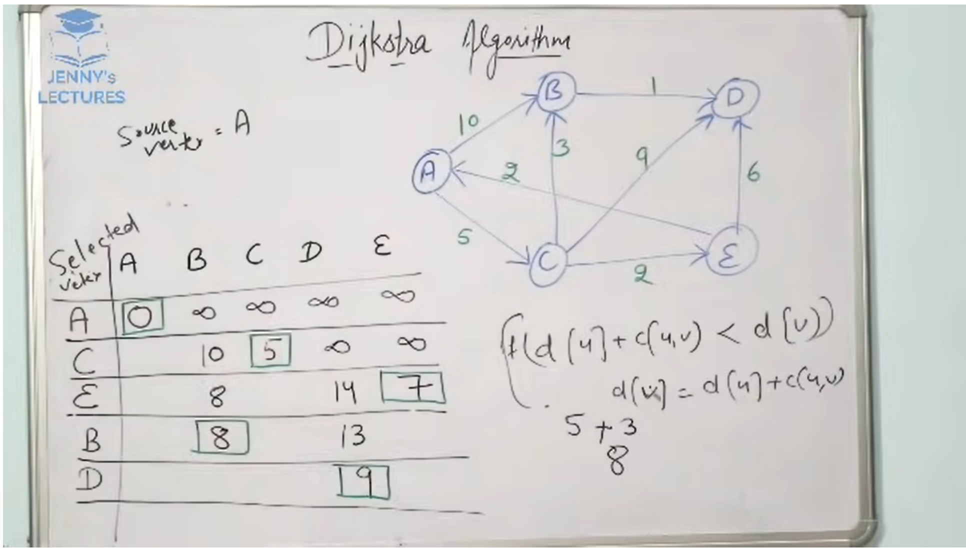
Programming Design – Complexity and Graphs 46 / 54 Ling-Chieh Kung (NTU IM)



Dijkstra Algorithm

{if (d(u)+c(u,v) < d(v))
d(v) = d(u)+c(u,v) }

Subscribe JENNY'S LECTURES CS/IT NET&JRF for full syllabus



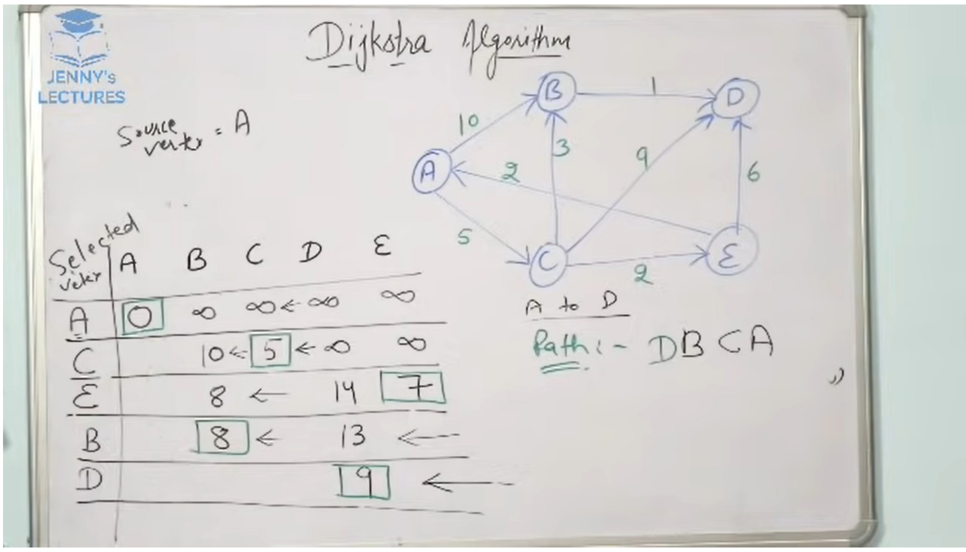
Dijkstra Algorithm

Source vertex = A

Selected vertex	A	B	C	D	E
A	$\boxed{0}$	∞	∞	∞	∞
C		10	$\boxed{5}$	∞	∞
E		8		14	$\boxed{7}$
B		$\boxed{8}$		13	
D				$\boxed{9}$	

$$\text{if}\,(d[u] + c(u,v) < d[v])$$
$$d[v] = d[u] + c(u,v)$$

$5 + 3$
8



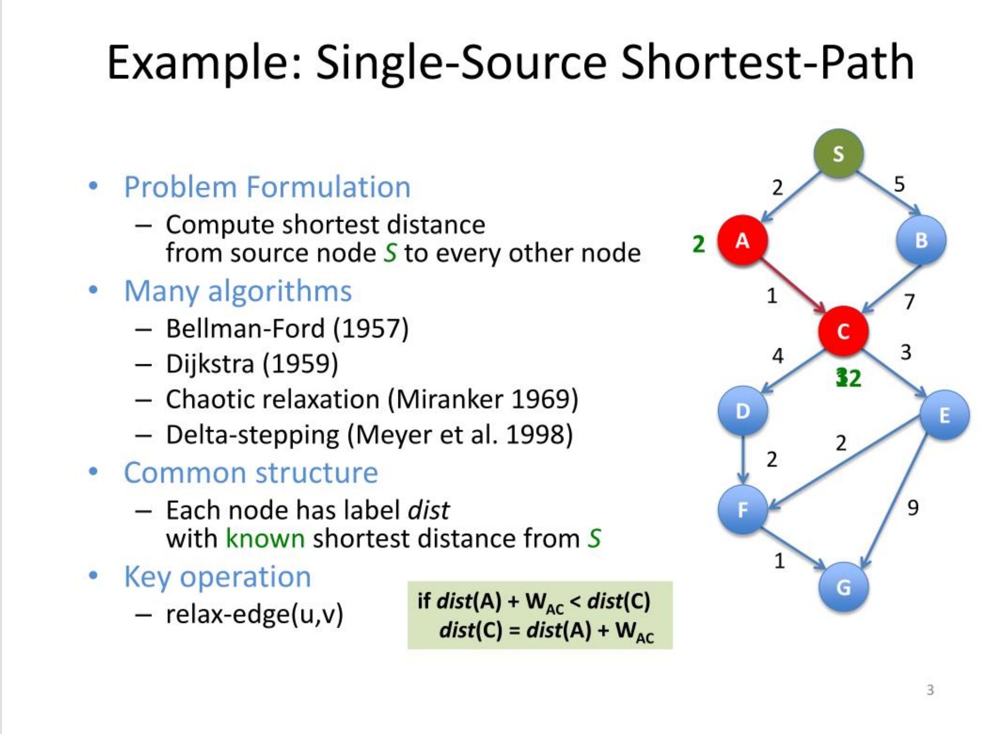
Dijkstra Algorithm

Source vertex = A

Selected vertex	A	B	C	D	E
A	$\boxed{0}$	∞	∞ ←	∞	∞
C		10 ⇐	$\boxed{5}$ ←	∞	∞
E		8 ←		14	$\boxed{7}$
B		$\boxed{8}$ ←		13	←—
D				$\boxed{9}$	←—

A to D

Path :- DBCA

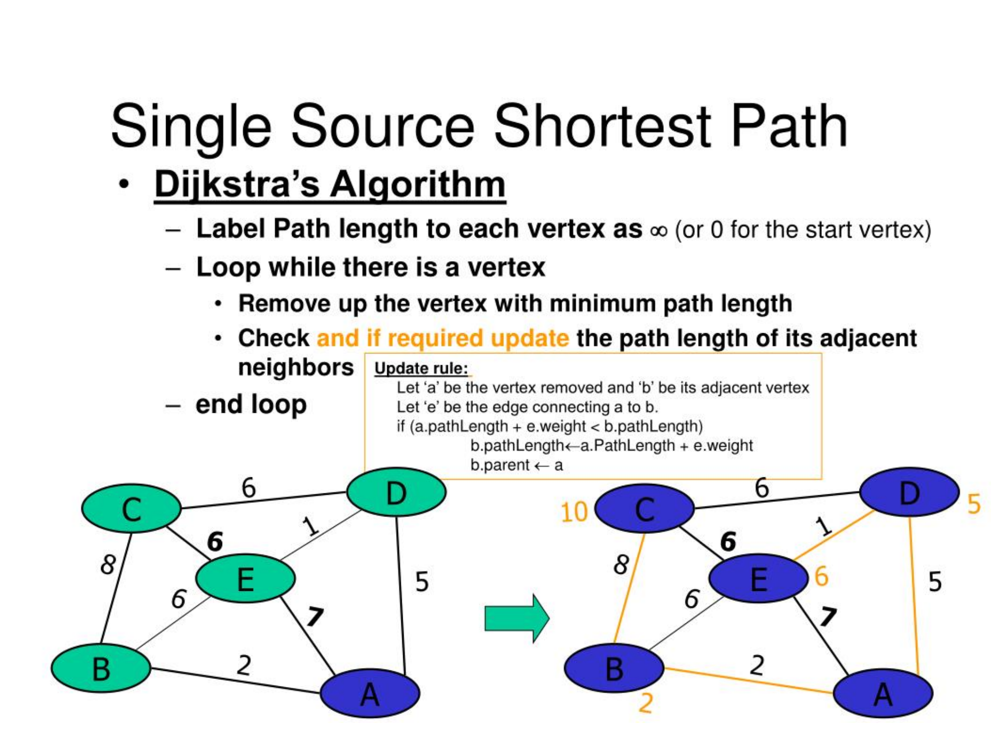


Example: Single-Source Shortest-Path

- Problem Formulation
  - Compute shortest distance from source node *S* to every other node
- Many algorithms
  - Bellman-Ford (1957)
  - Dijkstra (1959)
  - Chaotic relaxation (Miranker 1969)
  - Delta-stepping (Meyer et al. 1998)
- Common structure
  - Each node has label *dist* with known shortest distance from *S*
- Key operation
  - relax-edge(u,v)

if *dist*(A) + W_{AC} < *dist*(C)
dist(C) = *dist*(A) + W_{AC}

3

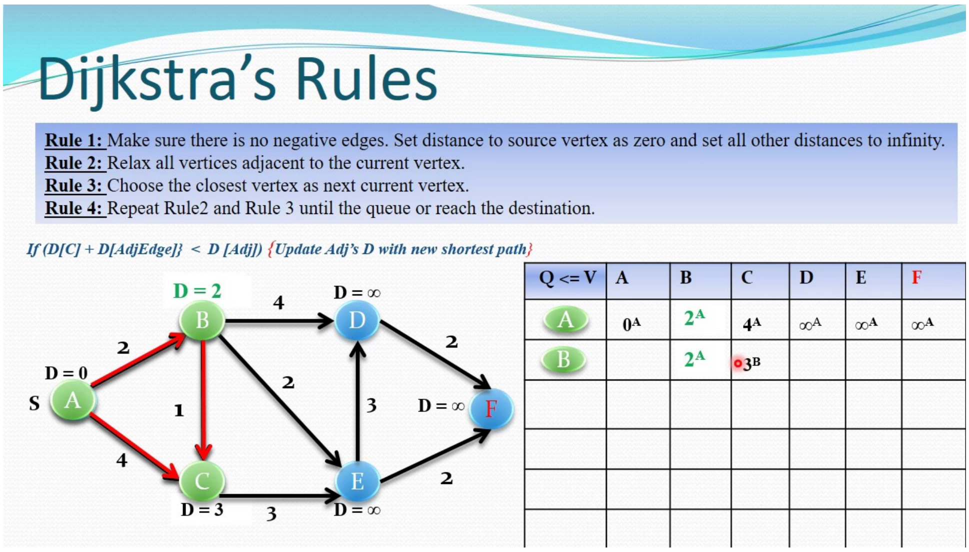


Single Source Shortest Path

- **Dijkstra's Algorithm**
  - **Label Path length to each vertex as ∞** (or 0 for the start vertex)
  - **Loop while there is a vertex**
    - **Remove up the vertex with minimum path length**
    - **Check and if required update the path length of its adjacent neighbors**
  - **end loop**

Update rule:

```
Let 'a' be the vertex removed and 'b' be its adjacent vertex
Let 'e' be the edge connecting a to b.
if (a.pathLength + e.weight < b.pathLength)
        b.pathLength←a.PathLength + e.weight
        b.parent ← a
```



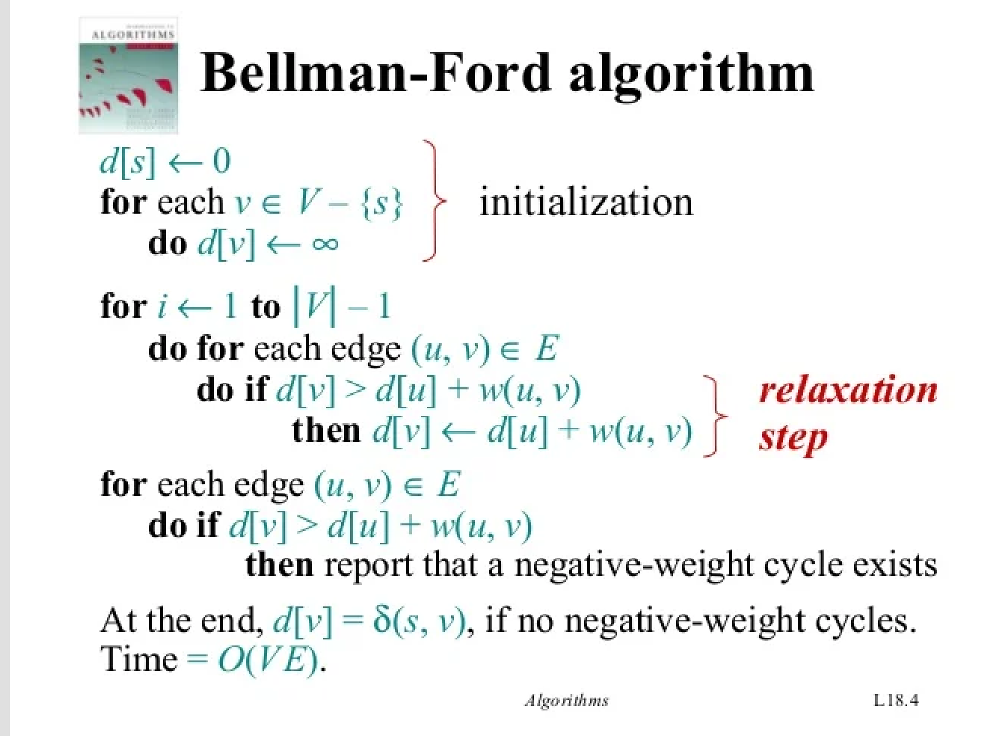
Dijkstra's Rules

Rule 1: Make sure there is no negative edges. Set distance to source vertex as zero and set all other distances to infinity.
Rule 2: Relax all vertices adjacent to the current vertex.
Rule 3: Choose the closest vertex as next current vertex.
Rule 4: Repeat Rule2 and Rule 3 until the queue or reach the destination.

If (D[C] + D[AdjEdge]} < D [Adj]) {Update Adj's D with new shortest path}

Q <= V	A	B	C	D	E	F
A	0^A	2^A	4^A	∞^A	∞^A	∞^A
B		2^A	3^B			

Bellman Ford Algorithm

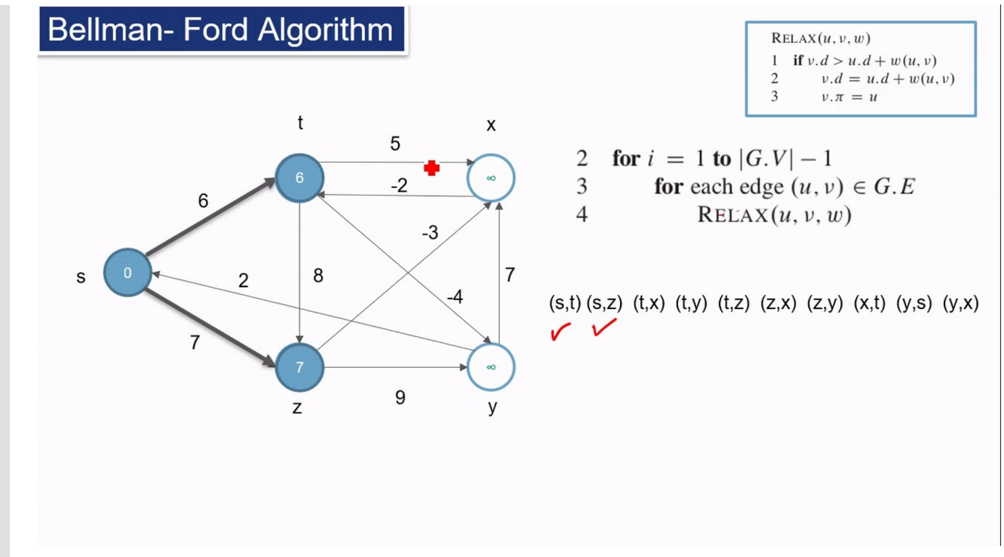


Bellman-Ford algorithm

```
$d[s] \leftarrow 0$
for each $v \in V - \{s\}$            } initialization
    do $d[v] \leftarrow \infty$

for $i \leftarrow 1$ to $|V| - 1$
    do for each edge $(u, v) \in E$
        do if $d[v] > d[u] + w(u, v)$              } relaxation
            then $d[v] \leftarrow d[u] + w(u, v)$  } step
for each edge $(u, v) \in E$
    do if $d[v] > d[u] + w(u, v)$
        then report that a negative-weight cycle exists
```

At the end, $d[v] = \delta(s, v)$, if no negative-weight cycles.
Time = $O(VE)$.

Algorithms L18.4



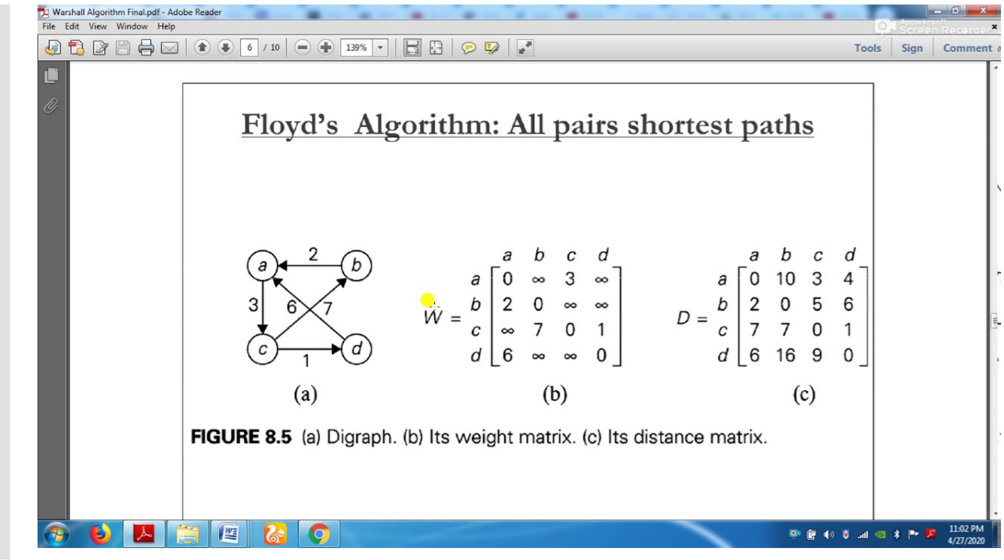
Bellman- Ford Algorithm

```
RELAX(u, v, w)
    if $v.d > u.d + w(u, v)$
        $v.d = u.d + w(u, v)$
        $v.\pi = u$
```

```
for $i = 1$ to $|G.V| - 1$
    for each edge $(u, v) \in G.E$
        RELAX$(u, v, w)$
```

(s,t) (s,z) (t,x) (t,y) (t,z) (z,x) (z,y) (x,t) (y,s) (y,x)
✓ ✓

Floyd Warshall Algorithm (All Pair Shortest Path algorithm)



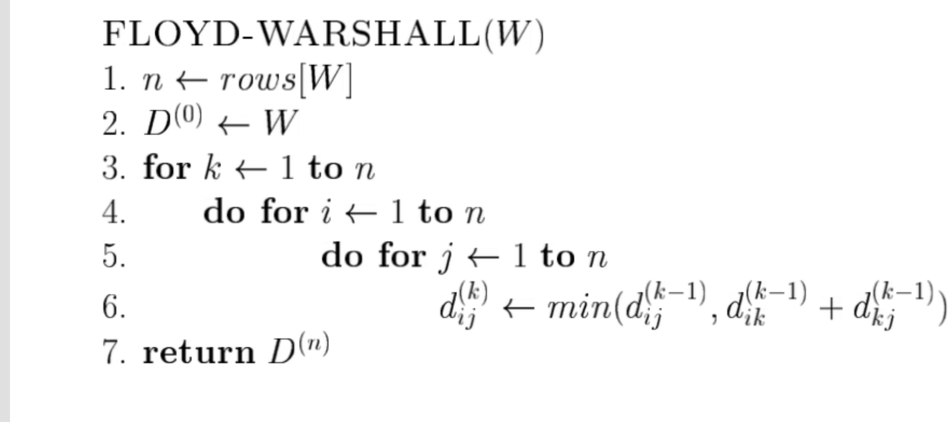
Warshall Algorithm Final.pdf - Adobe Reader
File Edit View Window Help
6 / 10
139%
Tools Sign Comment
11:02 PM
4/27/2020

Floyd's Algorithm: All pairs shortest paths

$$W = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & \infty & 3 & \infty \\ b & 2 & 0 & \infty & \infty \\ c & \infty & 7 & 0 & 1 \\ d & 6 & \infty & \infty & 0 \end{array} \qquad D = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 10 & 3 & 4 \\ b & 2 & 0 & 5 & 6 \\ c & 7 & 7 & 0 & 1 \\ d & 6 & 16 & 9 & 0 \end{array}$$

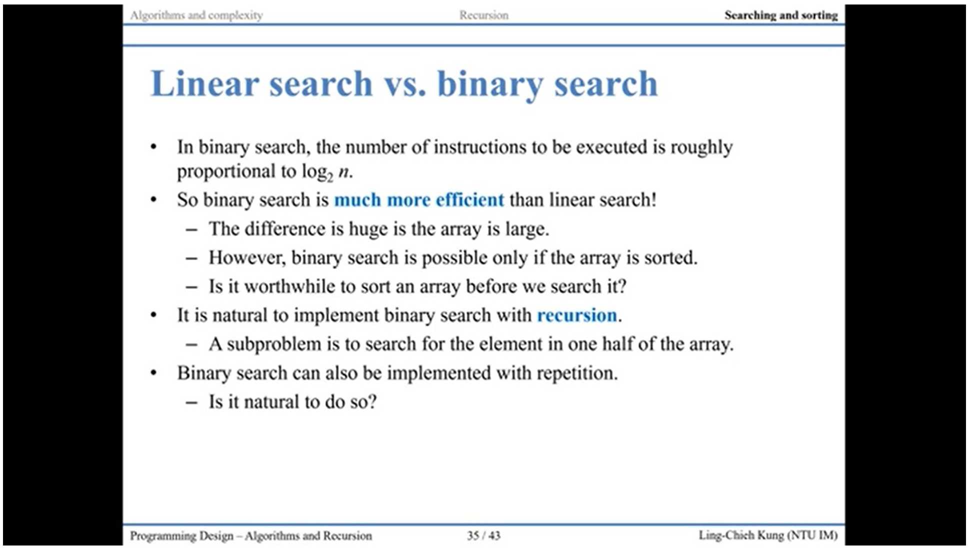
(a) (b) (c)

FIGURE 8.5 (a) Digraph. (b) Its weight matrix. (c) Its distance matrix.



```
FLOYD-WARSHALL($W$)
$n \leftarrow rows[W]$
$D^{(0)} \leftarrow W$
for $k \leftarrow 1$ to $n$
    do for $i \leftarrow 1$ to $n$
        do for $j \leftarrow 1$ to $n$
            $d_{ij}^{(k)} \leftarrow min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$
return $D^{(n)}$
```

Search Algorithm



Algorithms and complexity Recursion **Searching and sorting**

Linear search vs. binary search

- In binary search, the number of instructions to be executed is roughly proportional to $\log_2 n$.
- So binary search is **much more efficient** than linear search!
  - The difference is huge is the array is large.
  - However, binary search is possible only if the array is sorted.
  - Is it worthwhile to sort an array before we search it?
- It is natural to implement binary search with **recursion**.
  - A subproblem is to search for the element in one half of the array.
- Binary search can also be implemented with repetition.
  - Is it natural to do so?

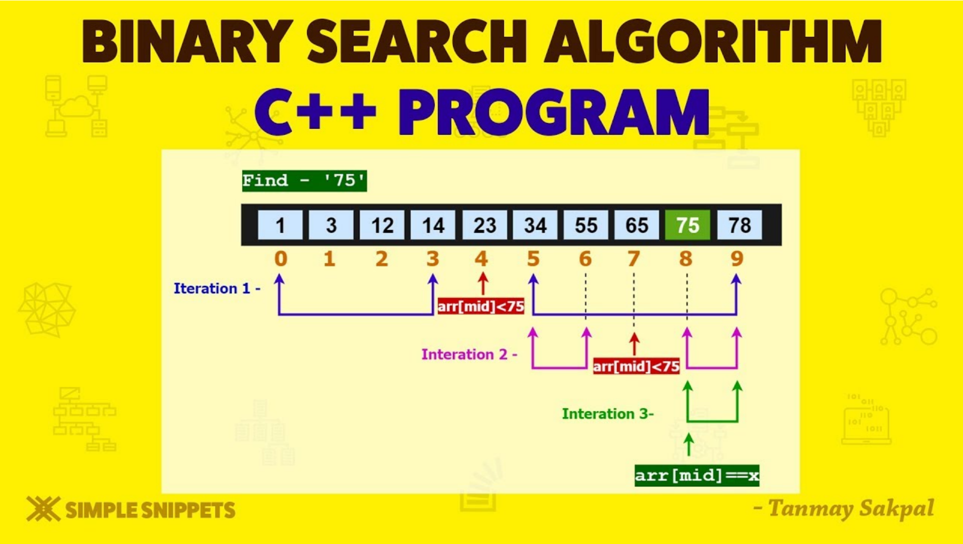
Programming Design – Algorithms and Recursion 35 / 43 Ling-Chieh Kung (NTU IM)

Linear Search

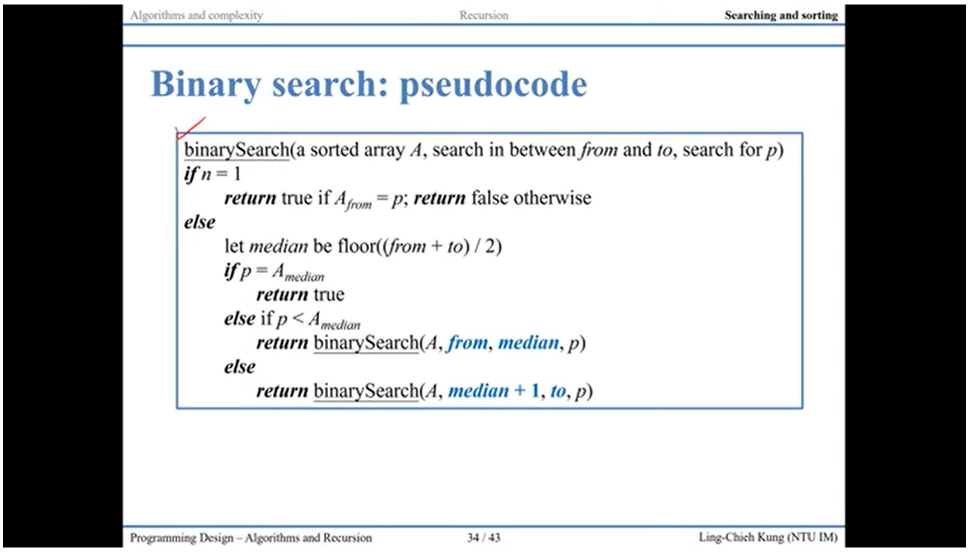
In [ ]:

```
#include <iostream>
using namespace std;


void linearSearch(int arr[], int n);

int main(){

    int arr[5] = {1, 22, 34, 5, 7};
    int searchNum;

    cout << "Please enter a number to search: ";
    cin >> searchNum;

    linearSearch(arr, searchNum);

    return 0;
}

void linearSearch(int arr[], int searchNum){
    int found = -1;
    for(int i = 0; i < 5; i++){
        if(arr[i] == searchNum){
            cout << "Found at index: " << i << endl;
            found = 1;
        }
    }

    if(found == -1){
        cout << "Not Found" << endl;
    }
}
```

Binary Search in sorted Array



In [ ]:
```
#include <iostream>
using namespace std;


int binarySearch(int arr[], int left, int right, int n);

int main(){

    int myArr[10];
    int searchNum;
    int index;

    cout << "Please enter 10 numbers in ASCENDING order" << endl;
    for(int i=0; i<10; i++){
        cin >> myArr[i];
    }
    cout << endl;

    cout << "Please enter a number to search: ";
    cin >> searchNum;
    cout << endl;

    index = binarySearch(myArr, 0, 9, searchNum);

    if(index == -1){
        cout << "No Match Found" << endl;
    }
    else {
        cout << "Found at index: " << index << endl;
    }

    return 0;
}


int binarySearch(int arr[], int left, int right, int x){

    while(left <= right){
        int mid = left + (right - right)/2;
        if(arr[mid] == x){
            return mid;
        }
        else if(arr[mid] < x){
            left = mid + 1;
        }
        else {
            right = mid - 1;
        }
    }

    return -1;
}
```



Algorithms and complexity Recursion Searching and sorting

Binary search: pseudocode

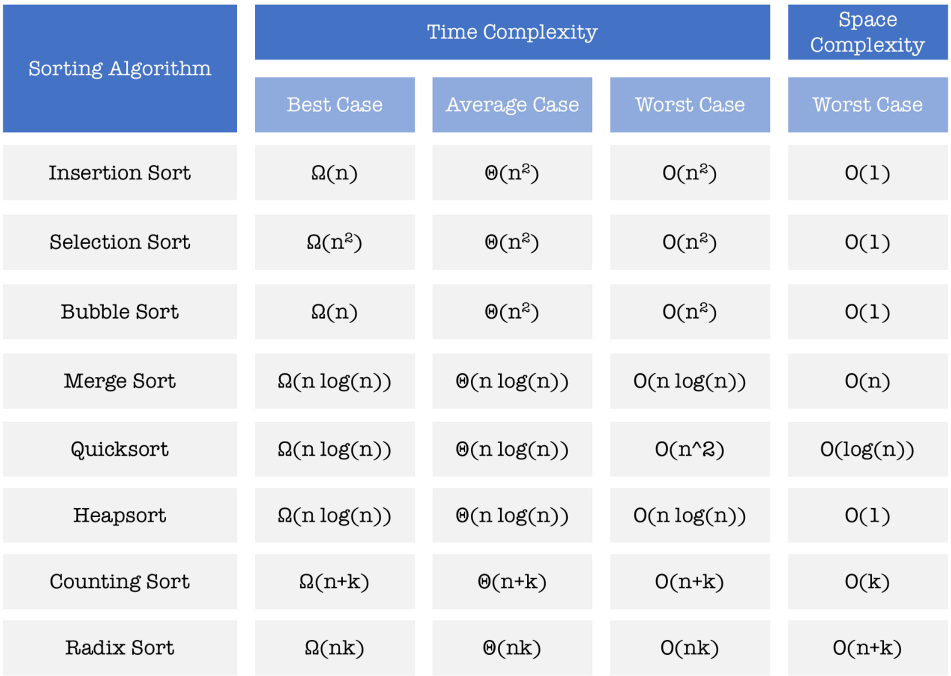
```
binarySearch(a sorted array $A$, search in between $from$ and $to$, search for $p$)
if $n = 1$
        return true if $A_{from} = p$; return false otherwise
else
        let $median$ be floor(($from + to$) / 2)
        if $p = A_{median}$
            return true
        else if $p < A_{median}$
            return binarySearch($A$, $from$, $median$, $p$)
        else
            return binarySearch($A$, $median + 1$, $to$, $p$)
```

Programming Design – Algorithms and Recursion 34 / 43 Ling-Chieh Kung (NTU IM)

In [ ]:

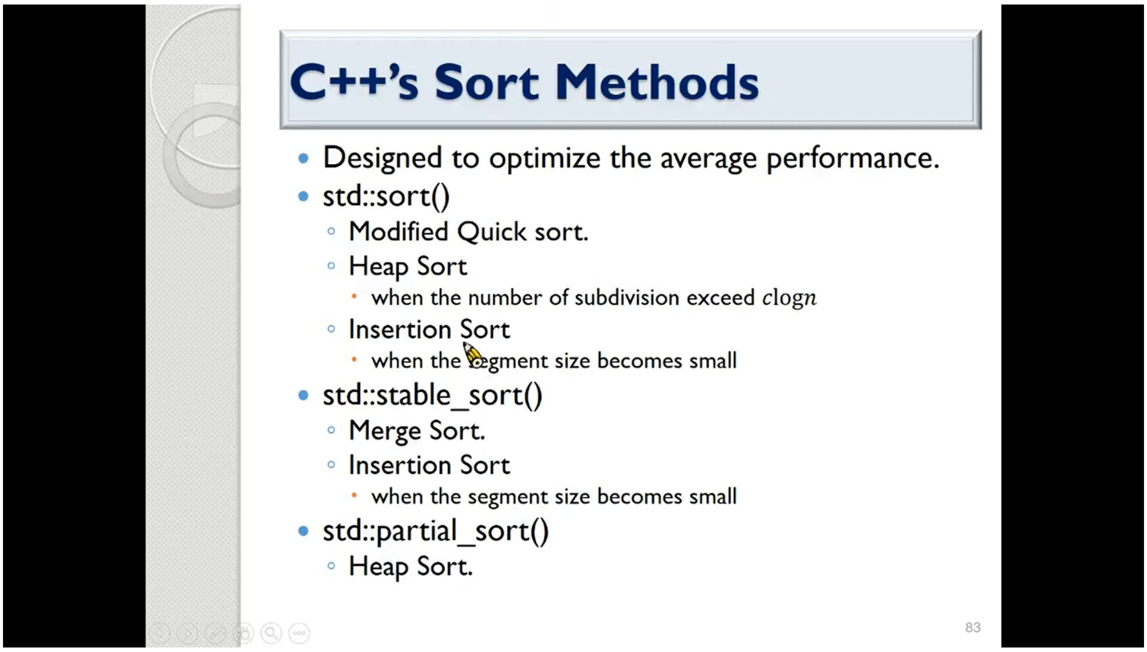
```
#include <iostream>
using namespace std;


int binarySearch(int arr[], int left, int right, int n);

int main(){

    int myArr[10];
    int searchNum;
    int index;

    cout << "Please enter 10 numbers in ASCENDING order" << endl;
    for(int i=0; i<10; i++){
        cin >> myArr[i];
    }
    cout << endl;
    cout << "Please enter a number to search: ";
    cin >> searchNum;
    cout << endl;

    index = binarySearch(myArr, 0, 9, searchNum);

    if(index == -1){
        cout << "No Match Found" << endl;
    }
    else {
        cout << "Found at index: " << index << endl;
    }

    return 0;
}


int binarySearch(int arr[], int left, int right, int x){

    while(left <= right){
        int mid = left + (right - right)/2;
        if(arr[mid] == x){
            return mid;
        }
        else if(arr[mid] < x){
            return binarySearch(arr, mid + 1, right, x);
        }
        else {
            return binarySearch(arr, left, mid - 1, x);
        }
    }

    return -1;
}
```

Sort Algorithm



Sorting Algorithm	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Insertion Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Bubble Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Merge Sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$
Quicksort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	O(n^2)	$O(\log(n))$
Heapsort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(1)$
Counting Sort	$\Omega(n+k)$	$\Theta(n+k)$	$O(n+k)$	$O(k)$
Radix Sort	$\Omega(nk)$	$\Theta(nk)$	$O(nk)$	$O(n+k)$

|>



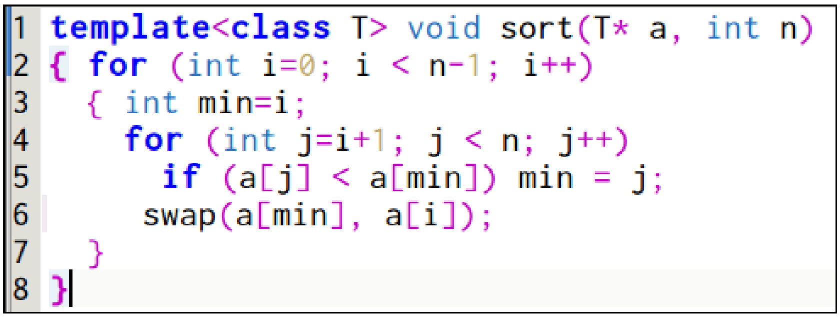
C++'s Sort Methods

- Designed to optimize the average performance.
- std::sort()
  - Modified Quick sort.
  - Heap Sort
    - when the number of subdivision exceed *clogn*
  - Insertion Sort
    - when the segment size becomes small
- std::stable_sort()
  - Merge Sort.
  - Insertion Sort
    - when the segment size becomes small
- std::partial_sort()
  - Heap Sort.

83

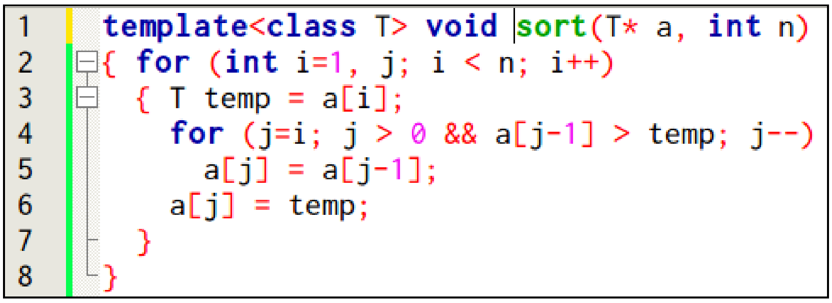
Sorting in Array

Here are the two C++ implementations to use for your tracing exercise. First is a **Selection Sort**



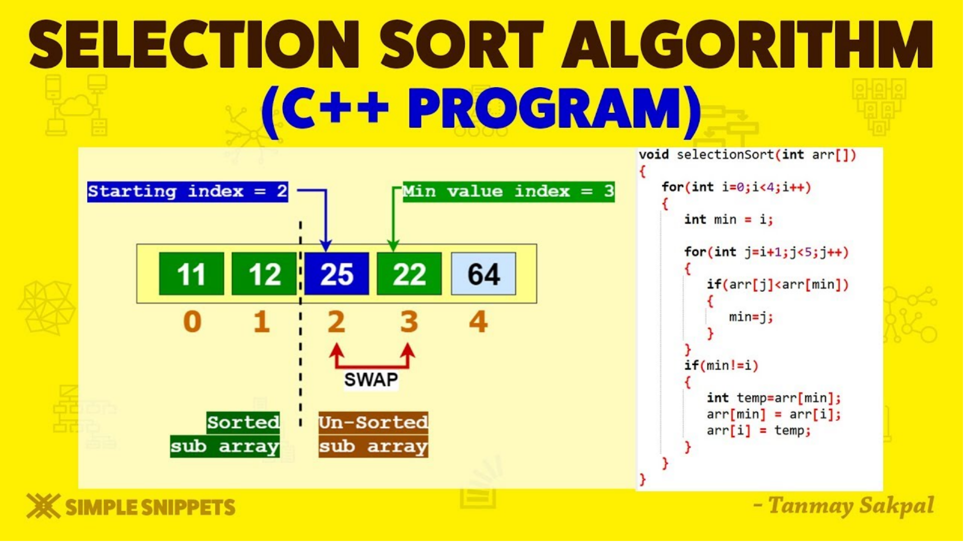
```
template<class T> void sort(T* a, int n)
{ for (int i=0; i < n-1; i++)
  { int min=i;
    for (int j=i+1; j < n; j++)
      if (a[j] < a[min]) min = j;
     swap(a[min], a[i]);
  }
}
```

The second is an **Insertion Sort**:



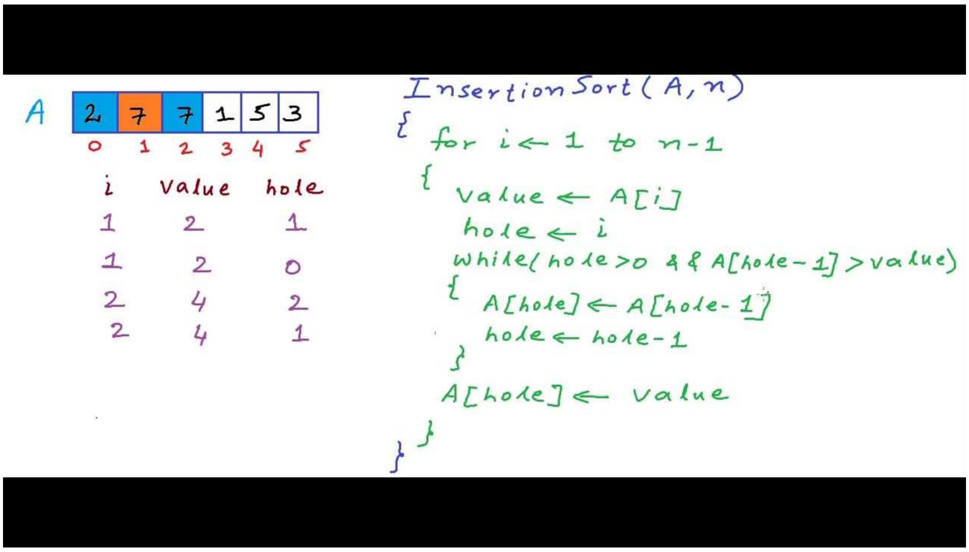
```
template<class T> void sort(T* a, int n)
{ for (int i=1, j; i < n; i++)
  { T temp = a[i];
    for (j=i; j > 0 && a[j-1] > temp; j--)
      a[j] = a[j-1];
    a[j] = temp;
  }
}
```

Selection Sort



```
void selectionSort(int arr[])
{
   for(int i=0;i<4;i++)
   {
      int min = i;

      for(int j=i+1;j<5;j++)
      {
         if(arr[j]<arr[min])
         {
            min=j;
         }
      }
      if(min!=i)
      {
         int temp=arr[min];
         arr[min] = arr[i];
         arr[i] = temp;
      }
   }
}
```

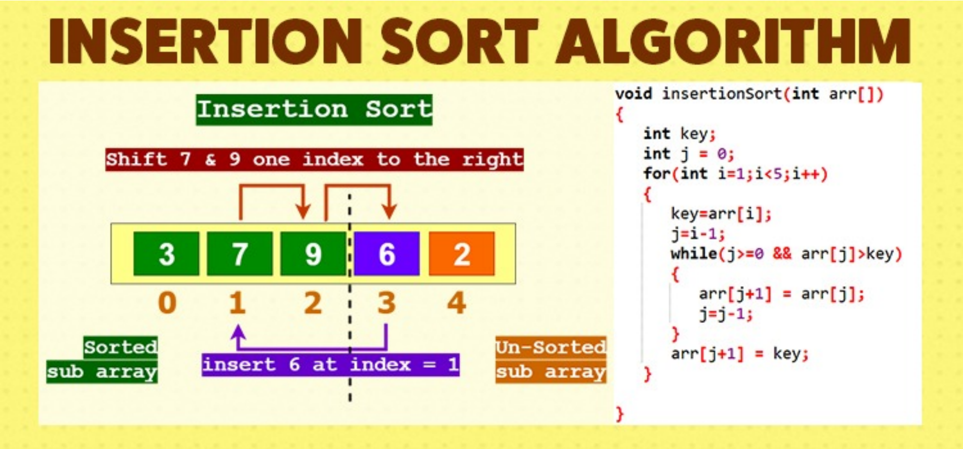
Insertion Sort



A	2	7	7	1	5	3
	0	1	2	3	4	5

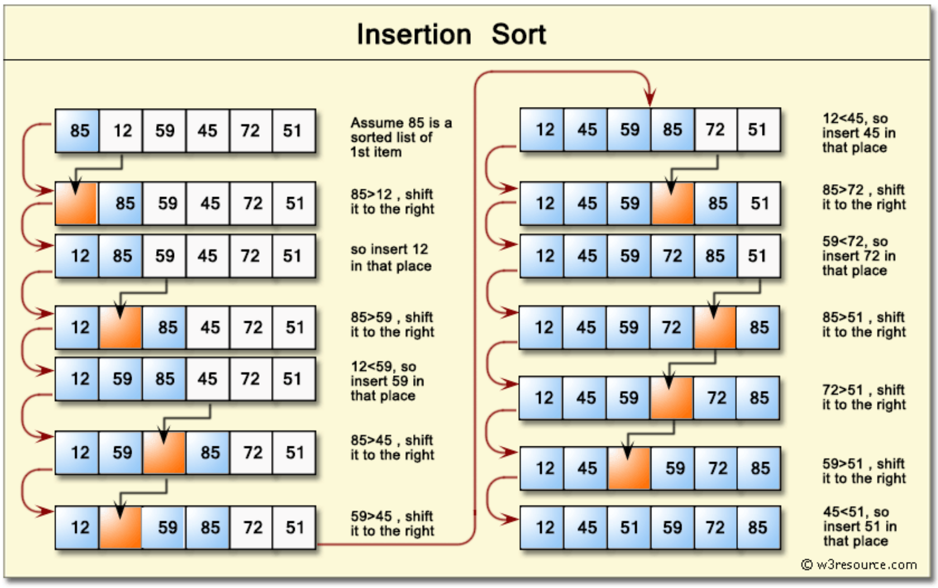
i	value	hole
1	2	1
1	2	0
2	4	2
2	4	1

```
InsertionSort(A, n)
{
  for i ← 1 to n-1
  {
    value ← A[i]
    hole ← i
    while(hole>0 && A[hole-1] > value)
    {
      A[hole] ← A[hole-1]
      hole ← hole-1
    }
    A[hole] ← value
  }
}
```



INSERTION SORT ALGORITHM

```
void insertionSort(int arr[])
{
    int key;
    int j = 0;
    for(int i=1;i<5;i++)
    {
        key=arr[i];
        j=i-1;
        while(j>=0 && arr[j]>key)
        {
            arr[j+1] = arr[j];
            j=j-1;
        }
        arr[j+1] = key;
    }
}
```

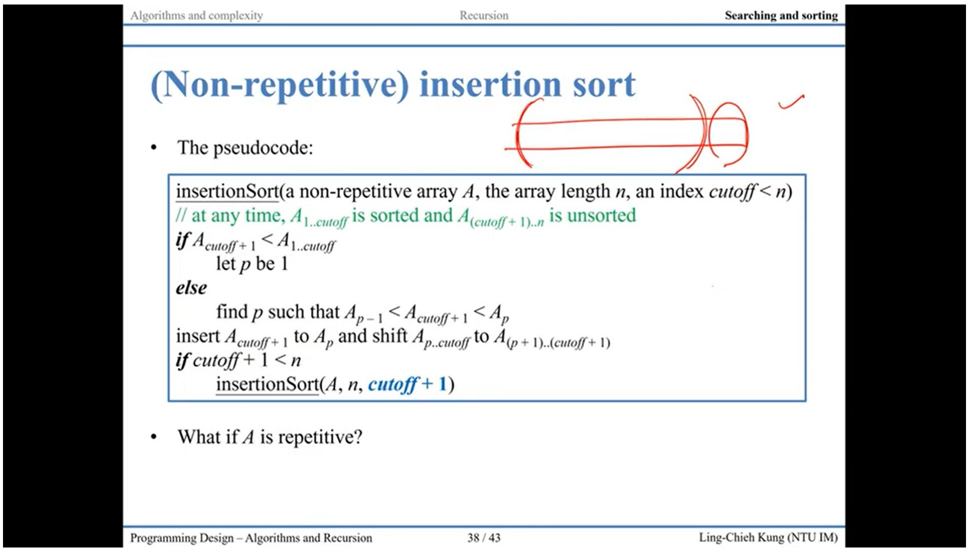


Insertion Sort

85	12	59	45	72	51	Assume 85 is a sorted list of 1st item
	85	59	45	72	51	85>12 , shift it to the right
12	85	59	45	72	51	so insert 12 in that place
12		85	45	72	51	85>59 , shift it to the right
12	59	85	45	72	51	12<59, so insert 59 in that place
12	59		85	72	51	85>45 , shift it to the right
12		59	85	72	51	59>45 , shift it to the right

12	45	59	85	72	51	12<45, so insert 45 in that place
12	45	59		85	51	85>72 , shift it to the right
12	45	59	72	85	51	59<72, so insert 72 in that place
12	45	59	72		85	85>51 , shift it to the right
12	45	59		72	85	72>51 , shift it to the right
12	45		59	72	85	59>51 , shift it to the right
12	45	51	59	72	85	45<51, so insert 51 in that place

© w3resource.com



Algorithms and complexity Recursion **Searching and sorting**

(Non-repetitive) insertion sort

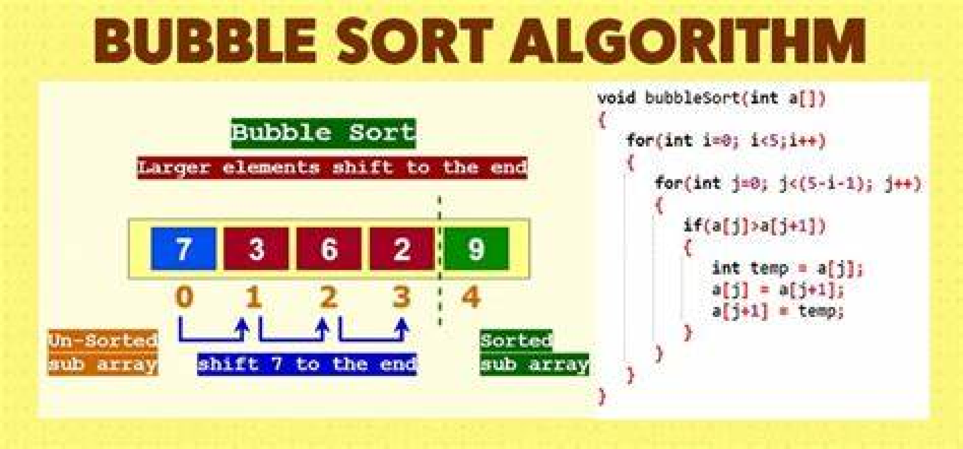
- The pseudocode:

```
insertionSort(a non-repetitive array $A$, the array length $n$, an index $cutoff < n$)
// at any time, $A_{1..cutoff}$ is sorted and $A_{(cutoff + 1)..n}$ is unsorted
if $A_{cutoff + 1} < A_{1..cutoff}$
    let $p$ be 1
else
    find $p$ such that $A_{p-1} < A_{cutoff + 1} < A_p$
insert $A_{cutoff + 1}$ to $A_p$ and shift $A_{p..cutoff}$ to $A_{(p + 1)..(cutoff + 1)}$
if $cutoff + 1 < n$
    insertionSort($A$, $n$, $cutoff + 1$)
```

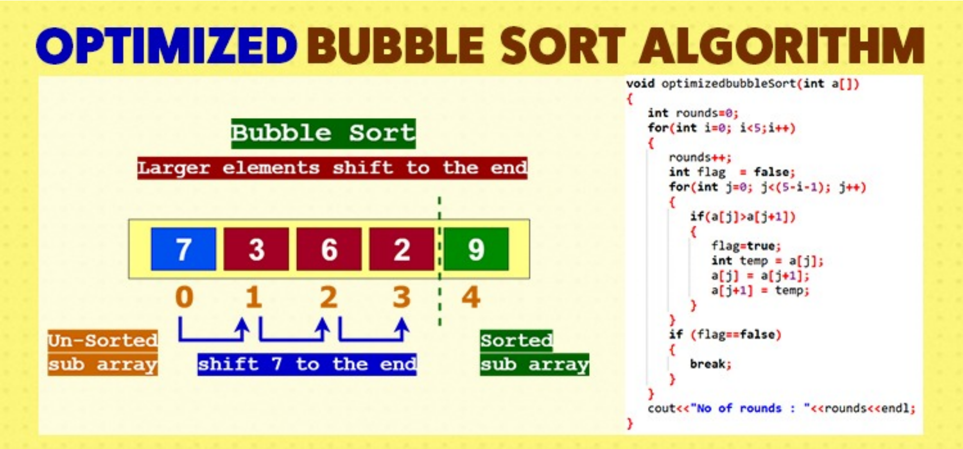
- What if A is repetitive?

Programming Design – Algorithms and Recursion 38 / 43 Ling-Chieh Kung (NTU IM)

Bubble Sort



```
void bubbleSort(int a[])
{
   for(int i=0; i<5;i++)
   {
      for(int j=0; j<(5-i-1); j++)
      {
         if(a[j]>a[j+1])
         {
            int temp = a[j];
            a[j] = a[j+1];
            a[j+1] = temp;
         }
      }
   }
}
```



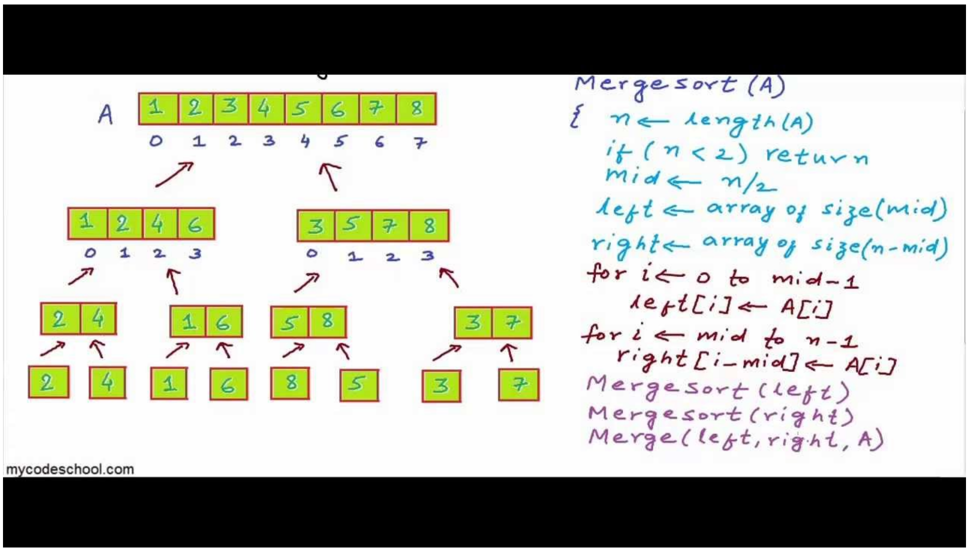
```
void optimizedbubbleSort(int a[])
{
   int rounds=0;
   for(int i=0; i<5;i++)
   {
      rounds++;
      int flag  = false;
      for(int j=0; j<(5-i-1); j++)
      {
         if(a[j]>a[j+1])
         {
            flag=true;
            int temp = a[j];
            a[j] = a[j+1];
            a[j+1] = temp;
         }
      }
      if (flag==false)
      {
         break;
      }
   }
   cout<<"No of rounds : "<<rounds<<endl;
}
```

In [ ]:

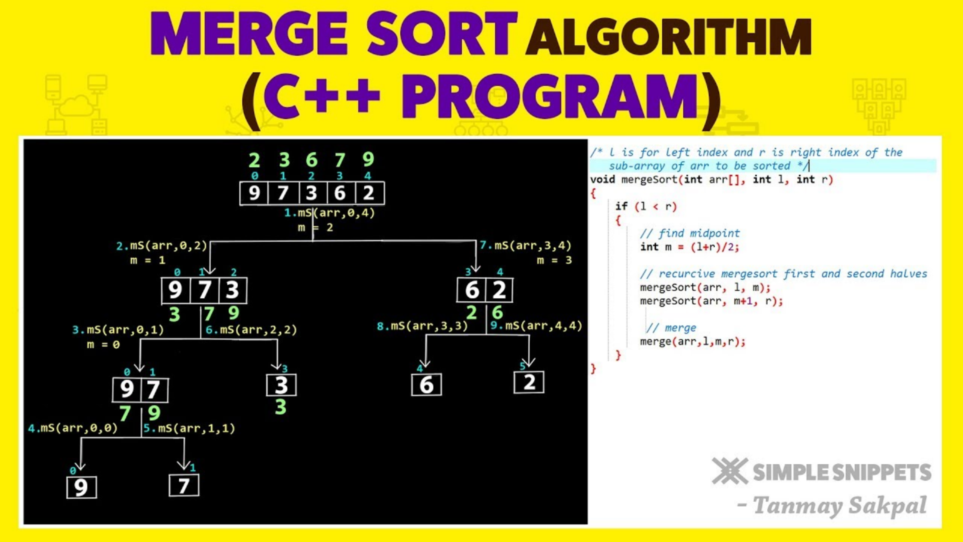
```
#include <iostream>
using namespace std;


void display(int* arr, int n);

void selectionSort(int* arr, int n);

void insertionSort(int* arr, int n);

void bubbleSort(int* arr, int n);

int main(){

    int option;
    int arraySize;

    do {
        cout << "Welcome to Learning C++ Data Structure and Algorithm" << endl;
        cout << "Topic: Sorting Algorithm" << endl << endl;

        cout << "1. Selection Sort" << endl;
        cout << "2. Insertion Sort" << endl;
        cout << "3. Bubble Sort" << endl;
        cout << "0. Exit" << endl;
        cout << endl;

        cout << "Select Your Option Number: ";
        cin >> option;
        cout << endl;

        if(option==0){
            break;
        }

        cout << "Enter your array size: ";
        cin >> arraySize;
        int arr[arraySize];
        cout << "\nEnter " << arraySize << " numbers in random order: " << endl;

        for(int i = 0; i < arraySize; i++){
            cin >> arr[i];
        }
        cout << endl;


        switch(option){
            //case 0://Quit
            //    break;
            case 1://Selection Sort
                selectionSort(arr, arraySize);
                display(arr, arraySize);
                break;
            case 2://Insertion Sort
                insertionSort(arr, arraySize);
                display(arr, arraySize);
                break;
            case 3://Bubble Sort
                bubbleSort(arr, arraySize);
                display(arr, arraySize);
                break;
            default:
                cout << "Enter the proper option number" << endl;
                cout << endl;
                break;
        }
    } while (option != 0);

    return 0;
}
```

```
void display(int* arr, int n){
    cout << "The Sorted Array: " << endl;
    for(int i=0; i<n; i++){
        cout << arr[i] << " ";
    }
    cout << endl << endl << endl;
}


void selectionSort(int* arr, int n){
    for(int i=0; i < n-1; i++){
        int min = i;
        for(int j=i+1; j<n; j++){
            if(arr[min] > arr[j]){
                min = j;
            }
        }
        if (min != i){
            int temp = arr[min];
            arr[min] = arr[i];
            arr[i] = temp;
        }
    }
}


void insertionSort(int* arr, int n){
    for(int i=1; i<=n-1; i++){
        int key = arr[i];
        int hole = i;
        while(hole>=0 && arr[hole-1]>key){
            arr[hole] = arr[hole-1];
            hole = hole-1;
        }
        arr[hole] = key;
    }
}

/*
void bubbleSort(int* arr, int n){
    for(int i=0; i<n-1; i++){
        for(int j=0; j<n-i-1; j++){
            if(arr[j] > arr[j+1]){
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}
*/

//Optimized Bubble Sort Algorithm
void bubbleSort(int* arr, int n){
    for(int i=0; i<n-1; i++){
        bool swapFlag = false;
        for(int j=0; j<n-i-1; j++){
            if(arr[j] > arr[j+1]){
                swapFlag = true;
        int temp = arr[j];
        arr[j] = arr[j+1];
        arr[j+1] = temp;
      }
    }
    if(swapFlag==false){
      break;
    }
  }
}
```

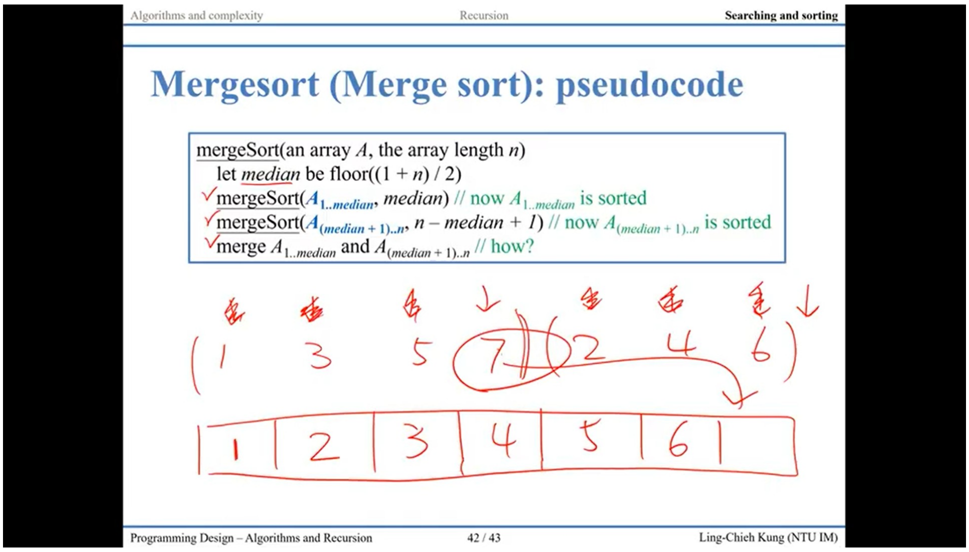
Merge Sort



```
Mergesort (A)
{ n ← length(A)
   if (n < 2) return
   mid ← n/2
   left ← array of size(mid)
   right ← array of size(n-mid)
  for i ← 0 to mid-1
      left[i] ← A[i]
  for i ← mid to n-1
     right[i-mid] ← A[i]
  Mergesort(left)
  Mergesort(right)
  Merge(left, right, A)
```



```
/* l is for left index and r is right index of the
   sub-array of arr to be sorted */
void mergeSort(int arr[], int l, int r)
{
    if (l < r)
    {
        // find midpoint
        int m = (l+r)/2;

        // recurcive mergesort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m+1, r);

        // merge
        merge(arr,l,m,r);
    }
}
```



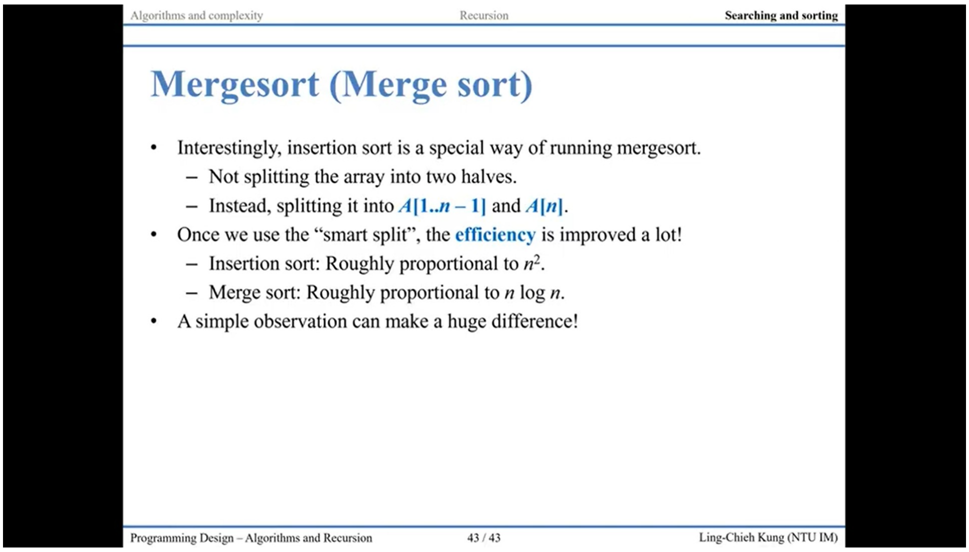
Algorithms and complexity Recursion **Searching and sorting**

Mergesort (Merge sort): pseudocode

```
mergeSort(an array $A$, the array length $n$)
  let $median$ be floor((1 + $n$) / 2)
✓mergeSort($A_{1..median}$, $median$) // now $A_{1..median}$ is sorted
✓mergeSort($A_{(median+1)..n}$, $n - median + 1$) // now $A_{(median+1)..n}$ is sorted
✓merge $A_{1..median}$ and $A_{(median+1)..n}$ // how?
```

(1 3 5 7) (2 4 6)
1 2 3 4 5 6

Programming Design – Algorithms and Recursion 42 / 43 Ling-Chieh Kung (NTU IM)



Algorithms and complexity Recursion **Searching and sorting**

Mergesort (Merge sort)

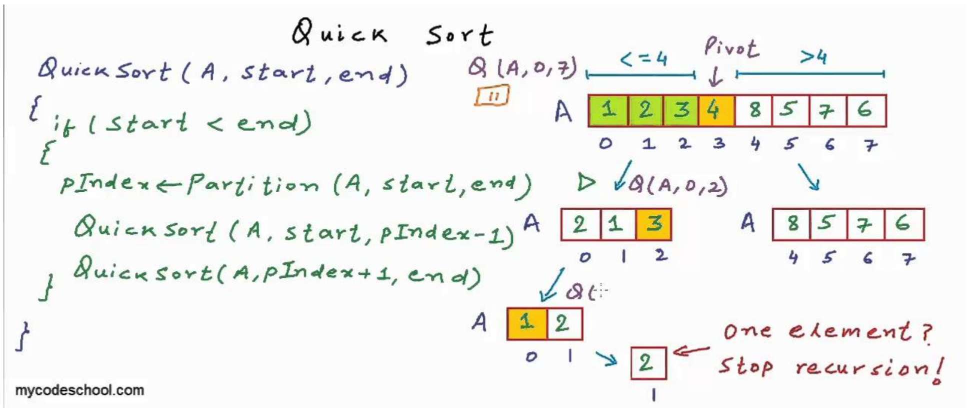
- Interestingly, insertion sort is a special way of running mergesort.
  - Not splitting the array into two halves.
  - Instead, splitting it into $A[1..n-1]$ and $A[n]$.
- Once we use the “smart split”, the **efficiency** is improved a lot!
  - Insertion sort: Roughly proportional to n^2.
  - Merge sort: Roughly proportional to $n \log n$.
- A simple observation can make a huge difference!

Programming Design – Algorithms and Recursion 43 / 43 Ling-Chieh Kung (NTU IM)

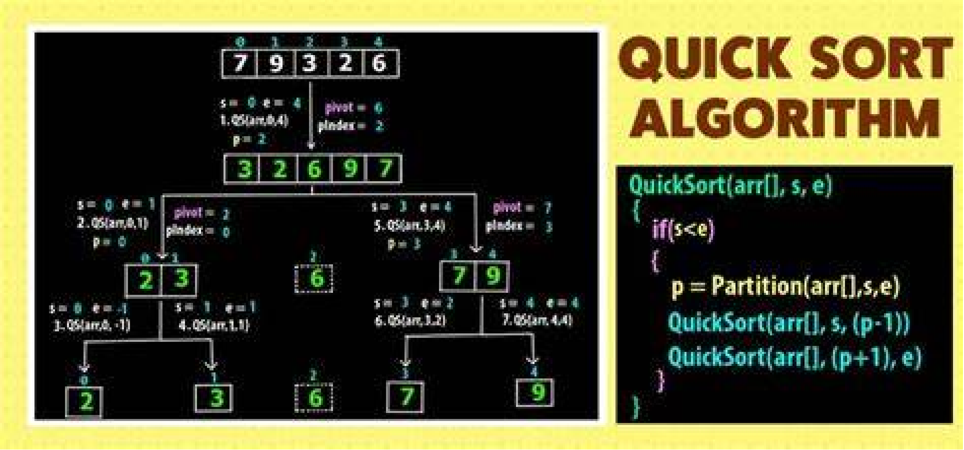
In [ ]:

```
#include <iostream>
using namespace std;


void display(int* arr, int n);

void mergeSort(int* arr, int l, int r, int n);

int main(){

    int arraySize;

    cout << "Welcome to Learning C++ Data Structure and Algorithm" << endl;
    cout << "Topic: MergeSorting Algorithm" << endl << endl;

    cout << "Enter your array size: ";
    cin >> arraySize;
    int arr[arraySize];
    cout << "\nEnter " << arraySize << " numbers in random order: " << endl;

    for(int i = 0; i < arraySize; i++){
        cin >> arr[i];
    }
    cout << endl;

    mergeSort(arr, 0, (arraySize-1), arraySize);
    display(arr, arraySize);

    return 0;
}



void display(int* arr, int n){
    cout << "The Sorted Array: " << endl;
    for(int i=0; i<n; i++){
        cout << arr[i] << " ";
    }
    cout << endl << endl << endl;
}


void merge(int* arr, int l, int m, int r, int n){
    int i = l;
    int j = m+1;
    int k = l;
    int temp[n];

    while(i<=m && j<=r){
        if(arr[i] <= arr[j]){
            temp[k] = arr[i];
            i++;
        }
        else {
            temp[k] = arr[j];
            j++;
        }
        k++;
    }
    if(i>m){
        while(j<=r){
            temp[k] = arr[j];
            j++;
            k++;
        }
    }
    else {
        while(i<=m){
            temp[k] = arr[i];
            i++;
            k++;
        }
    }
    for(k=l; k<=r; k++){
        arr[k] = temp[k];
    }
}

void mergeSort(int* arr, int l, int r, int n){
    if(l<r){
        int m = l + (r - l) / 2;
        mergeSort(arr, l, m, n);
        mergeSort(arr, m+1, r, n);
        merge(arr, l, m, r, n);
    }
}
```

Quick Sort



```
QuickSort(A, start, end)
{
  if (start < end)
  {
    pIndex←Partition(A, start, end)
    QuickSort(A, start, pIndex-1)
    QuickSort(A, pIndex+1, end)
  }
}
```

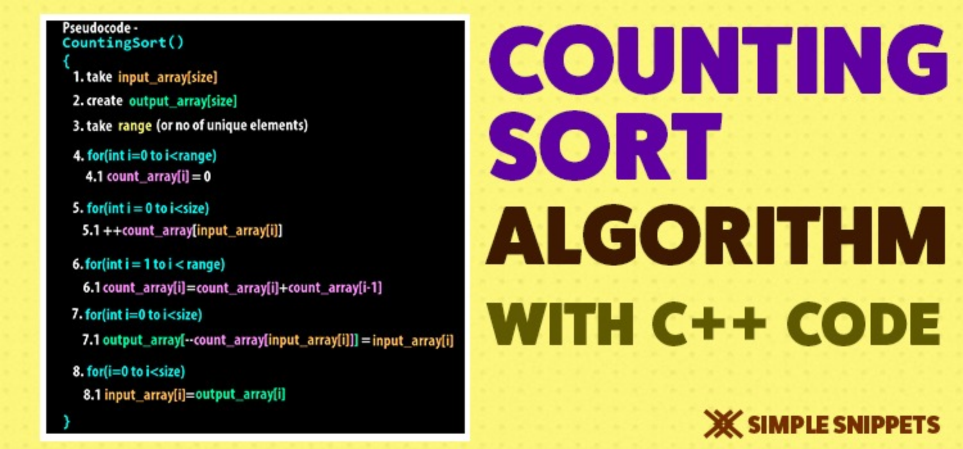


```
QuickSort(arr[], s, e)
{
    if(s<e)
    {
        p = Partition(arr[],s,e)
        QuickSort(arr[], s, (p-1))
        QuickSort(arr[], (p+1), e)
    }
}
```

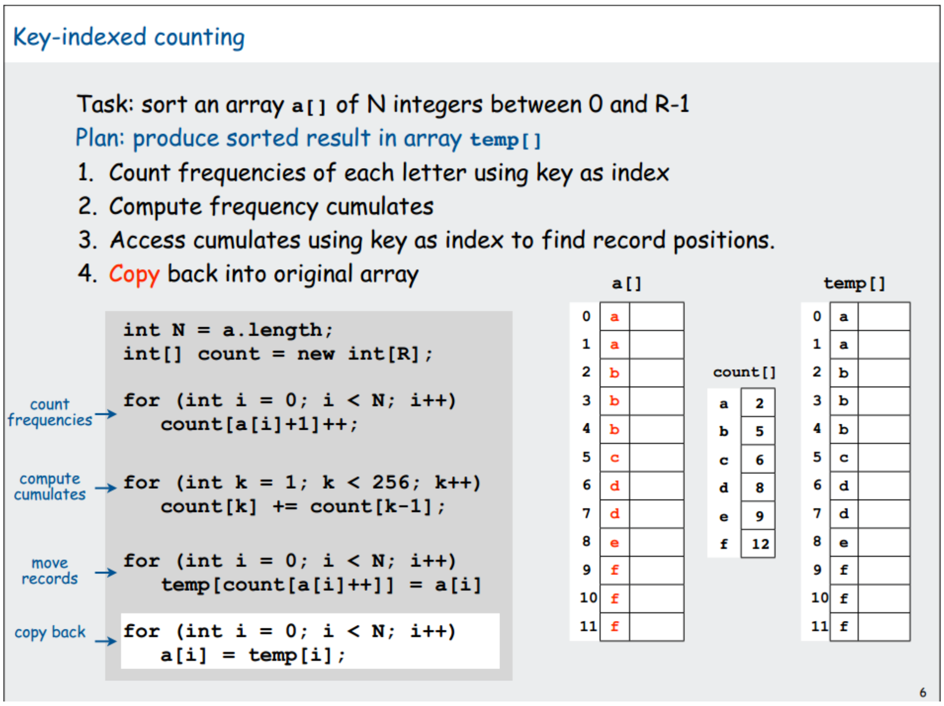
In [ ]:

```
#include <iostream>
using namespace std;


void display(int* arr, int n);

void quickSort(int* arr, int l, int r);

int main(){

    int arraySize;

    cout << "Welcome to Learning C++ Data Structure and Algorithm" << endl;
    cout << "Topic: QuickSorting Algorithm" << endl << endl;

    cout << "Enter your array size: ";
    cin >> arraySize;
    int arr[arraySize];
    cout << "\nEnter " << arraySize << " numbers in random order: " << endl;

    for(int i = 0; i < arraySize; i++){
        cin >> arr[i];
    }
    cout << endl;

    quickSort(arr, 0, (arraySize-1));

    display(arr, arraySize);
    return 0;
}



void display(int* arr, int n){
    cout << "The Sorted Array: " << endl;
    for(int i=0; i<n; i++){
        cout << arr[i] << " ";
    }
    cout << endl << endl << endl;
}


int partition(int* arr, int l, int r){
    int pivot = arr[r];
    int pIndex = l;
    for(int i=l; i<r; i++){
        if(arr[i]<=pivot){
            int temp = arr[i];
            arr[i] = arr[pIndex];
            arr[pIndex] = temp;
            pIndex++;
        }
    }
    int temp = arr[r];
    arr[r] = arr[pIndex];
    arr[pIndex] = temp;

    return pIndex;
}

void quickSort(int* arr, int l, int r){
    if(l<r){
        int pIndex = partition(arr, l, r);
        quickSort(arr, l, pIndex-1);
        quickSort(arr, pIndex+1, r);
    }
}
```

Counting Sort



```
Pseudocode -
CountingSort()
{
  1. take input_array[size]
  2. create output_array[size]
  3. take range (or no of unique elements)
  4. for(int i=0 to i<range)
     4.1 count_array[i] = 0
  5. for(int i = 0 to i<size)
     5.1 ++count_array[input_array[i]]
  6. for(int i = 1 to i < range)
     6.1 count_array[i]=count_array[i]+count_array[i-1]
  7. for(int i=0 to i<size)
     7.1 output_array[--count_array[input_array[i]]] = input_array[i]
  8. for(i=0 to i<size)
     8.1 input_array[i]=output_array[i]
}
```



Key-indexed counting

Task: sort an array `a[]` of N integers between 0 and R-1

Plan: produce sorted result in array `temp[]`

1. Count frequencies of each letter using key as index
2. Compute frequency cumulates
3. Access cumulates using key as index to find record positions.
4. Copy back into original array

```
int N = a.length;
int[] count = new int[R];

for (int i = 0; i < N; i++)
   count[a[i]+1]++;

for (int k = 1; k < 256; k++)
   count[k] += count[k-1];

for (int i = 0; i < N; i++)
   temp[count[a[i]++]] = a[i]

for (int i = 0; i < N; i++)
   a[i] = temp[i];
```

a[]

0	a
1	a
2	b
3	b
4	b
5	c
6	d
7	d
8	e
9	f
10	f
11	f

count[]

a	2
b	5
c	6
d	8
e	9
f	12

temp[]

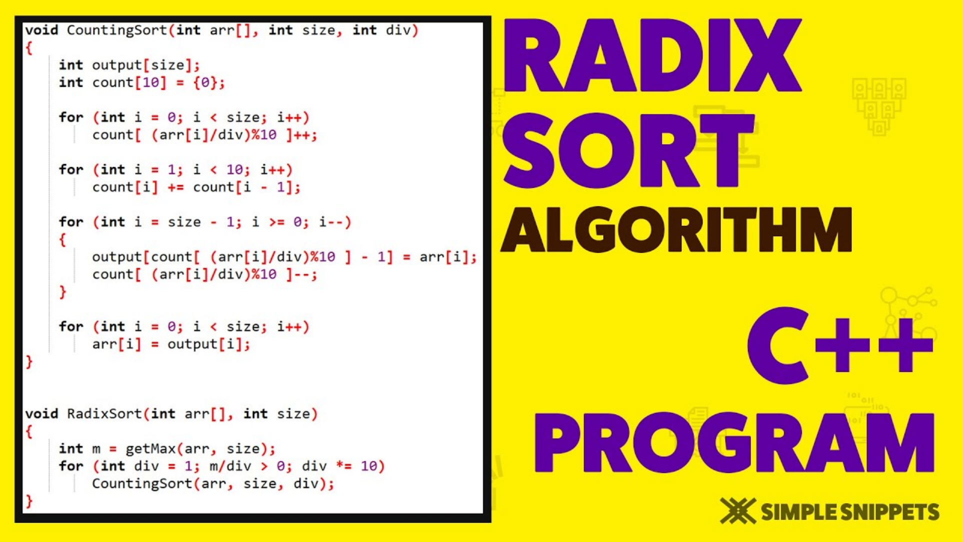
0	a
1	a
2	b
3	b
4	b
5	c
6	d
7	d
8	e
9	f
10	f
11	f

6

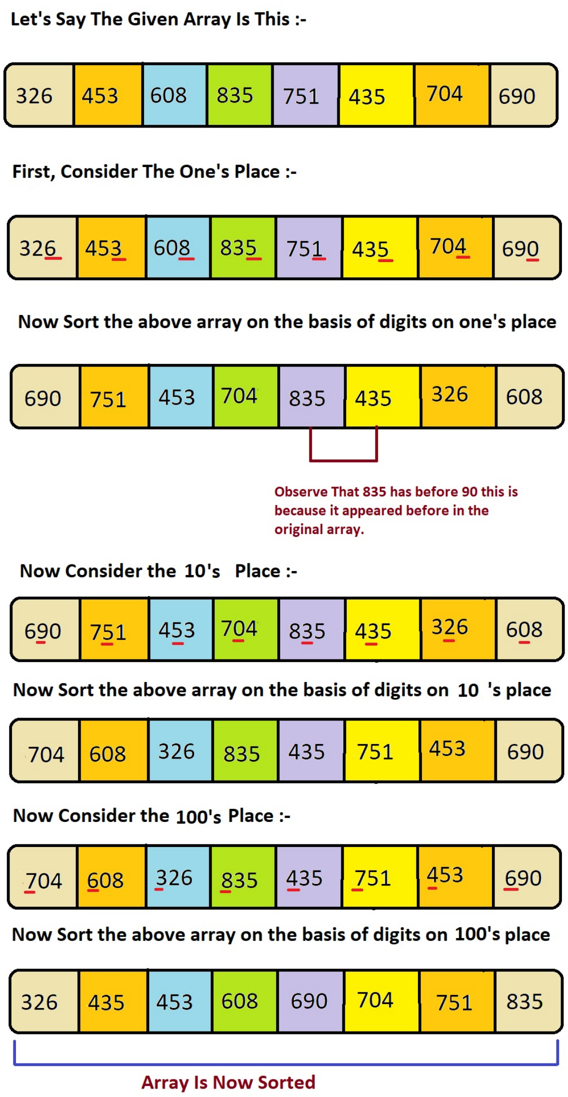
In [ ]:

```
#include <iostream>
#include <string>
using namespace std;


void display(int* arr, int n);

void countingSort(int* arr, int n);

int main(){

    cout << "Welcome to Learning C++ Data Structure and Algorithm" << endl;
    cout << "Topic: CountingSorting Algorithm" << endl << endl;

    int arraySize = 0;

    cout << "Enter your array size: ";
    cin >> arraySize;
    int arr[arraySize];
    cout << "\nEnter " << arraySize << " numbers in random order in range 0-9: " <<

    for(int i = 0; i < arraySize; i++){
        cin >> arr[i];
    }
    cout << endl;

    countingSort(arr, arraySize);

    display(arr, arraySize);

    return 0;
}



void display(int* arr, int n){
    cout << "The Sorted Array: " << endl;
    for(int i=0; i<n; i++){
        cout << arr[i] << " ";
    }
    cout << endl << endl << endl;
}

void countingSort(int* arr, int n){
    int countArray[10] = {0};
    int outputArray[n];

    // to take a count of all elements in the input array
    for(int i=0; i<n; i++){
        ++countArray[arr[i]];
    }

    // update countArray to get the position in the outputArray
    for(int i=1; i<10; i++){
        countArray[i] = countArray[i] + countArray[i-1];
    }

    // placing inputArray elements into outputAarray
    for(int i=n-1; i>=0; i--){
        outputArray[--countArray[arr[i]]] = arr[i];
    }

    for(int i=0; i<n; i++){
        arr[i] = outputArray[i];
    }
}
```

Radix Sort



```
void CountingSort(int arr[], int size, int div)
{
    int output[size];
    int count[10] = {0};

    for (int i = 0; i < size; i++)
        count[ (arr[i]/div)%10 ]++;

    for (int i = 1; i < 10; i++)
        count[i] += count[i - 1];

    for (int i = size - 1; i >= 0; i--)
    {
        output[count[ (arr[i]/div)%10 ] - 1] = arr[i];
        count[ (arr[i]/div)%10 ]--;
    }

    for (int i = 0; i < size; i++)
        arr[i] = output[i];
}


void RadixSort(int arr[], int size)
{
    int m = getMax(arr, size);
    for (int div = 1; m/div > 0; div *= 10)
        CountingSort(arr, size, div);
}
```



Let's Say The Given Array Is This :-

326	453	608	835	751	435	704	690

First, Consider The One's Place :-

326	453	608	835	751	435	704	690

Now Sort the above array on the basis of digits on one's place

690	751	453	704	835	435	326	608

Observe That 835 has before 90 this is because it appeared before in the original array.

Now Consider the 10's Place :-

690	751	453	704	835	435	326	608

Now Sort the above array on the basis of digits on 10 's place

704	608	326	835	435	751	453	690

Now Consider the 100's Place :-

704	608	326	835	435	751	453	690

Now Sort the above array on the basis of digits on 100's place

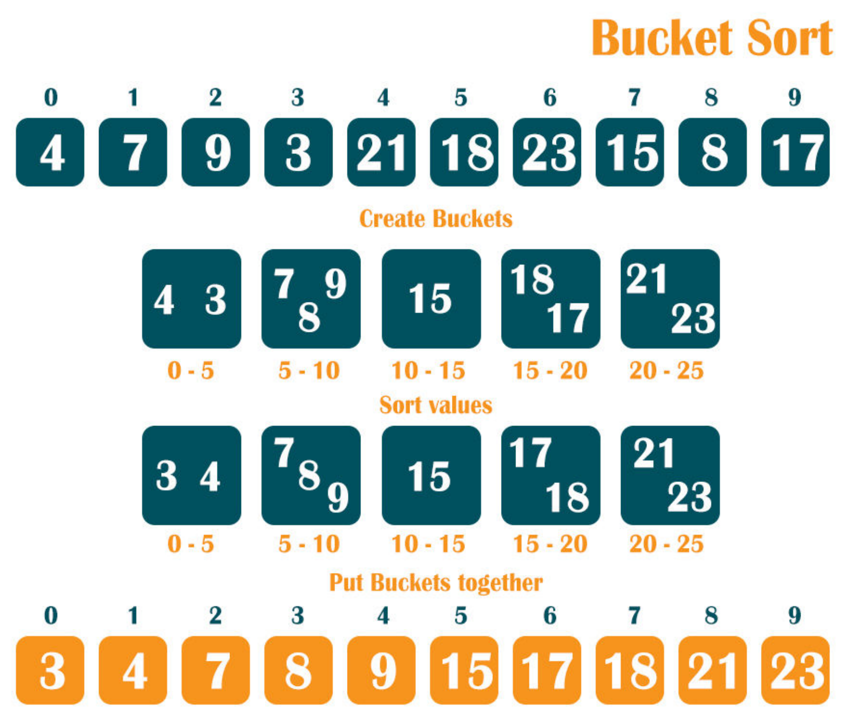
326	435	453	608	690	704	751	835

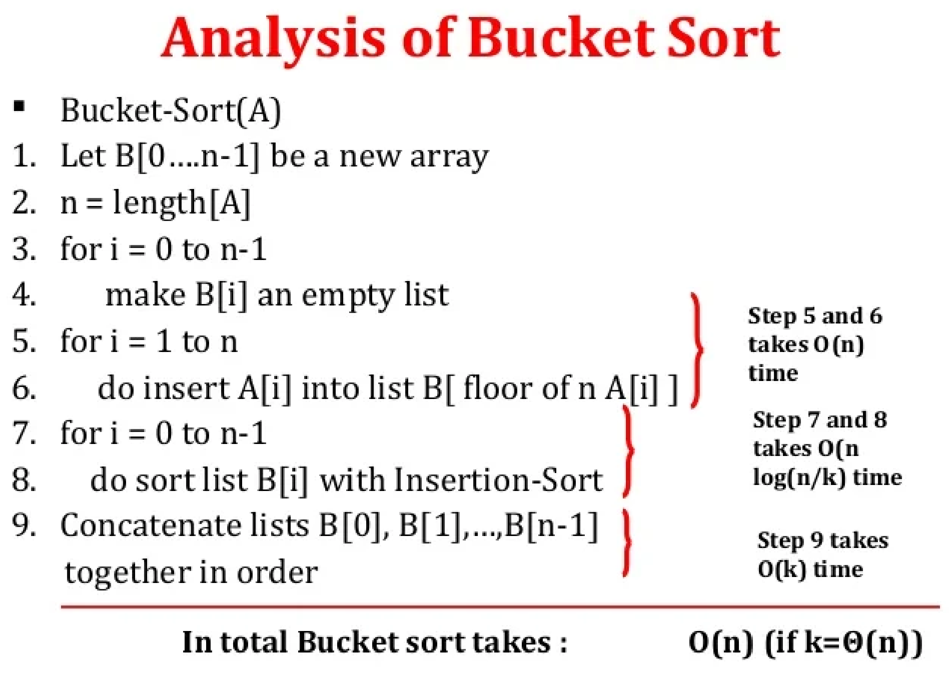
Array Is Now Sorted

In [ ]:

```cpp
#include <iostream>
#include <string>
using namespace std;


void display(int* arr, int n);

void radixSort(int* arr, int n);

int main(){

    cout << "Welcome to Learning C++ Data Structure and Algorithm" << endl;
    cout << "Topic: RadixSorting Algorithm" << endl << endl;

    int arraySize = 0;
    int countSize = 10;

    cout << "Enter your array size: ";
    cin >> arraySize;
    int arr[arraySize];
    cout << "\nEnter " << arraySize << " numbers in random order: " << endl;

    for(int i = 0; i < arraySize; i++){
        cin >> arr[i];
    }
    cout << endl;

    radixSort(arr, arraySize);

    display(arr, arraySize);

    return 0;
}



void display(int* arr, int n){
    cout << "The Sorted Array: " << endl;
    for(int i=0; i<n; i++){
        cout << arr[i] << " ";
    }
    cout << endl << endl << endl;
}


void countSort(int* arr, int n, int pos){
    int countArray[10] = {0}; // initialize with "0"
    int outputArray[n];

    // to take a count of all elements in the input array
    for(int i=0; i<n; i++){
        ++countArray[(arr[i]/pos) % 10];
    }

    // update countArray to get the position in the outputArray
    for(int i=1; i<10; i++){
        countArray[i] = countArray[i] + countArray[i-1];
    }

    // placing inputArray elements into outputAarray
    for(int i=n-1; i>=0; i--){
        outputArray[--countArray[(arr[i]/pos) % 10]] = arr[i];
    }

    for(int i=0; i<n; i++){
        arr[i] = outputArray[i];
    }
}

int getMax(int* arr, int n){
    int maxNumber = arr[0];
    for(int i=1; i<n; i++){
        if(maxNumber<arr[i]){
            maxNumber = arr[i];
        }
    }
    return maxNumber;
}


void radixSort(int* arr, int n){
    int maxNumber = getMax(arr, n);

    for(int pos=1; maxNumber/pos>0; pos *= 10){
        countSort(arr, n, pos);
    }
}
```

Bucket Sort





Analysis of Bucket Sort

- Bucket-Sort(A)

```
1. Let B[0....n-1] be a new array
2. n = length[A]
3. for i = 0 to n-1
4.     make B[i] an empty list
5. for i = 1 to n
6.     do insert A[i] into list B[ floor of n A[i] ]
7. for i = 0 to n-1
8.     do sort list B[i] with Insertion-Sort
9. Concatenate lists B[0], B[1],...,B[n-1]
   together in order
```

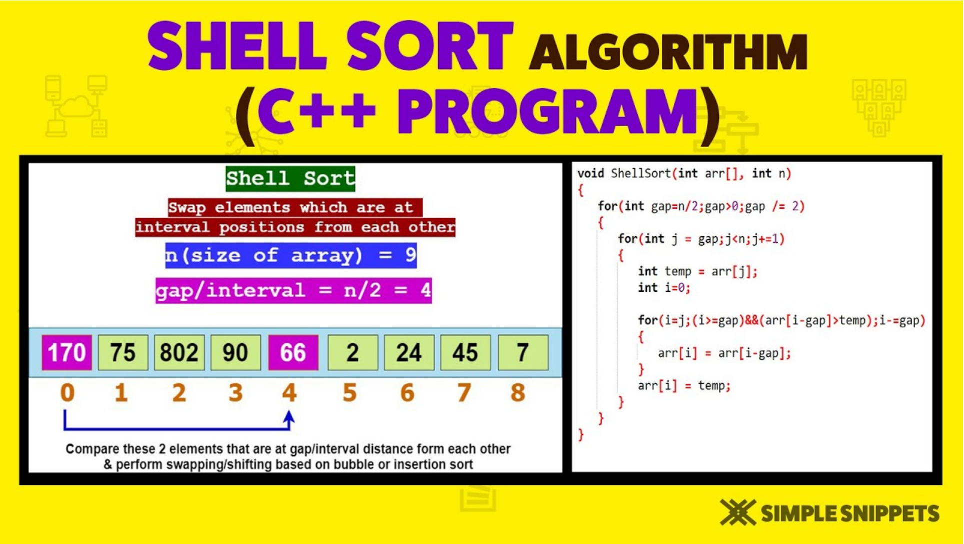
Step 5 and 6 takes O(n) time

Step 7 and 8 takes O(n log(n/k) time

Step 9 takes O(k) time

In total Bucket sort takes : **O(n) (if k=Θ(n))**

Shell Sort



SHELL SORT ALGORITHM (C++ PROGRAM)

Shell Sort

Swap elements which are at interval positions from each other

n(size of array) = 9

gap/interval = n/2 = 4

Compare these 2 elements that are at gap/interval distance form each other & perform swapping/shifting based on bubble or insertion sort

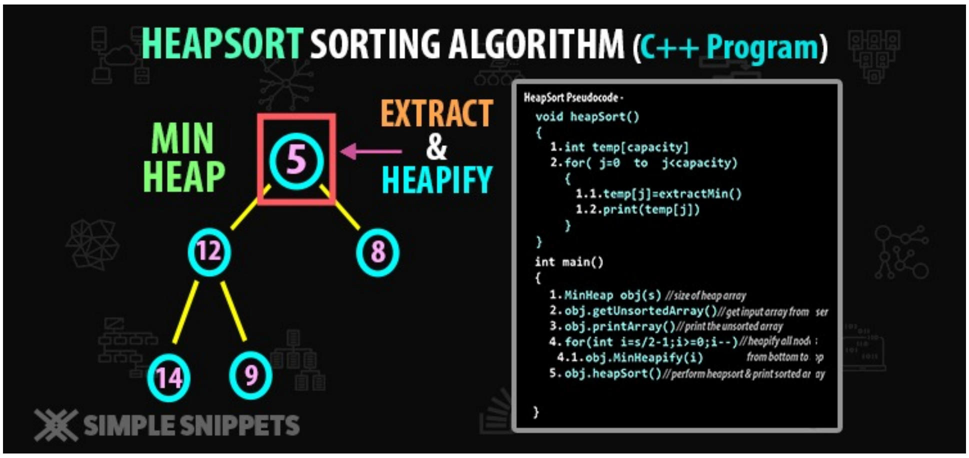
```
void ShellSort(int arr[], int n)
{
    for(int gap=n/2;gap>0;gap /= 2)
    {
        for(int j = gap;j<n;j+=1)
        {
            int temp = arr[j];
            int i=0;

            for(i=j;(i>=gap)&&(arr[i-gap]>temp);i-=gap)
            {
                arr[i] = arr[i-gap];
            }
            arr[i] = temp;
        }
    }
}
```

SIMPLE SNIPPETS

In [ ]:
```
#include <iostream>
#include <string>
using namespace std;


void display(int* arr, int n);

void shellSort(int* arr, int n);

int main(){

    cout << "Welcome to Learning C++ Data Structure and Algorithm" << endl;
    cout << "Topic: shellSorting Algorithm" << endl << endl;

    int arraySize = 0;
    int countSize = 10;

    cout << "Enter your array size: ";
    cin >> arraySize;
    int arr[arraySize];
    cout << "\nEnter " << arraySize << " numbers in random order: " << endl;

    for(int i = 0; i < arraySize; i++){
        cin >> arr[i];
    }
    cout << endl;

    shellSort(arr, arraySize);

    display(arr, arraySize);

    return 0;
}



void display(int* arr, int n){
    cout << "The Sorted Array: " << endl;
    for(int i=0; i<n; i++){
        cout << arr[i] << " ";
    }
    cout << endl << endl << endl;
}
```

```
void countSort(int* arr, int n, int pos){
    int countArray[10] = {0}; // initialize with "0"
    int outputArray[n];

    // to take a count of all elements in the input array
    for(int i=0; i<n; i++){
        ++countArray[(arr[i]/pos) % 10];
    }

    // update countArray to get the position in the outputArray
    for(int i=1; i<10; i++){
        countArray[i] = countArray[i] + countArray[i-1];
    }

    // placing inputArray elements into outputAarray
    for(int i=n-1; i>=0; i--){
        outputArray[--countArray[(arr[i]/pos) % 10]] = arr[i];
    }

    for(int i=0; i<n; i++){
        arr[i] = outputArray[i];
    }
}


void shellSort(int* arr, int n){
    for(int gap=n/2; gap>0; gap/=2){
        for(int j=gap; j<n; j++){
            for(int i=j-gap; i>=0; i-=gap){
                if(arr[i+gap] > arr[i]){
                    break;
                }
                else{
                    int temp = arr[i];
                    arr[i] = arr[i+gap];
                    arr[i+gap] = temp;
                }
            }
        }
    }
}
```

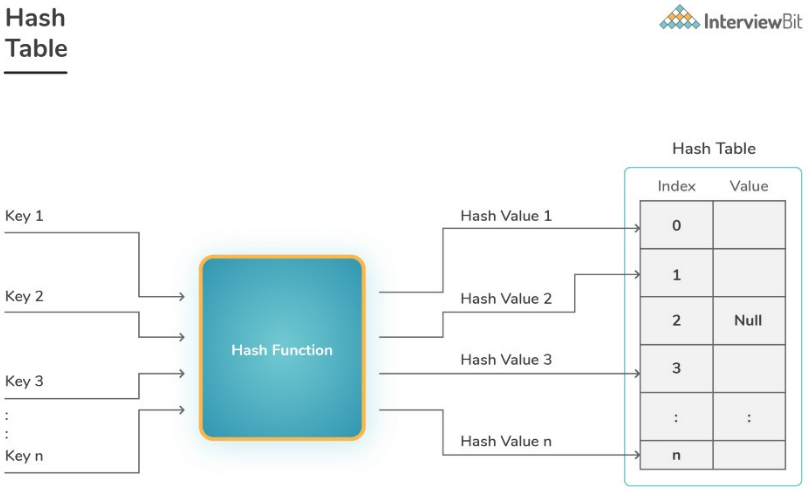
Heap Sort



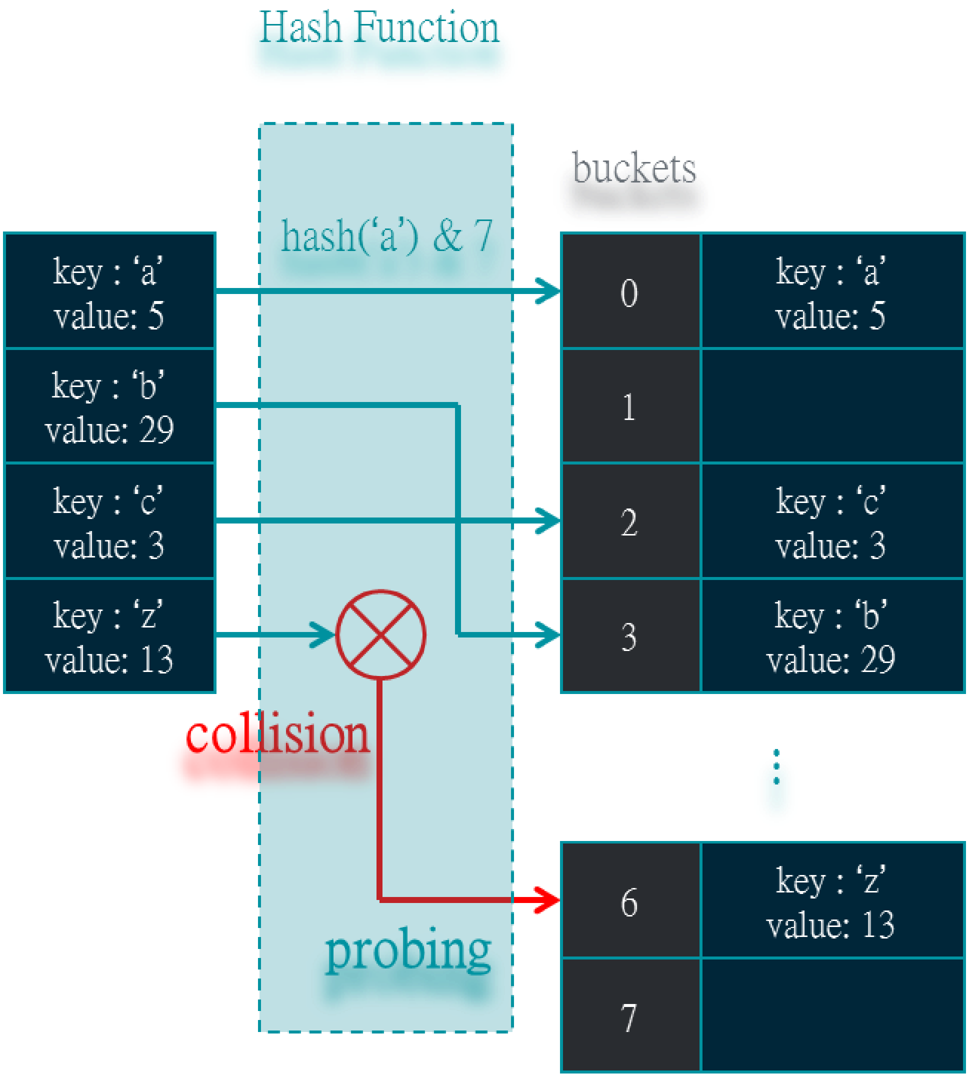
HeapSort Pseudocode -

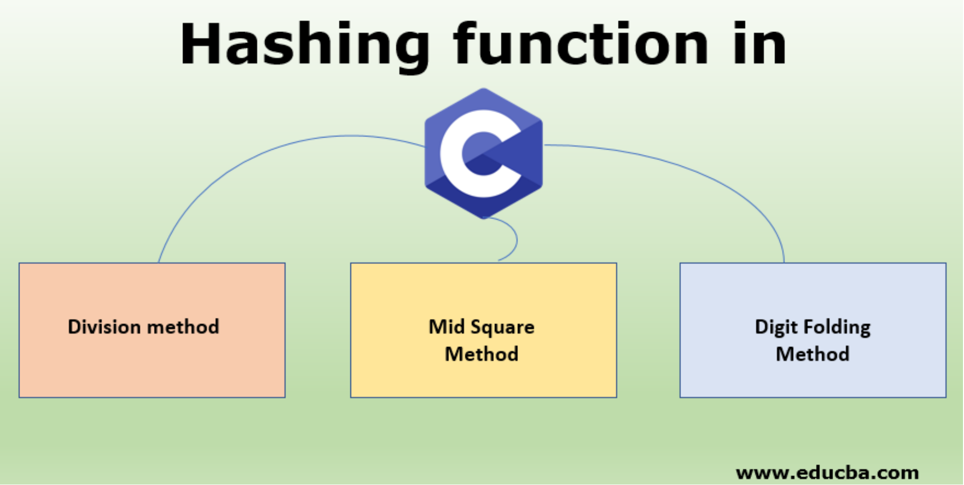
```
void heapSort()
{
  1.int temp[capacity]
  2.for( j=0  to  j<capacity)
    {
      1.1.temp[j]=extractMin()
      1.2.print(temp[j])
    }
}
int main()
{
  1.MinHeap obj(s) //size of heap array
  2.obj.getUnsortedArray()//get input array from  ser
  3.obj.printArray()//print the unsorted array
  4.for(int i=s/2-1;i>=0;i--)//heapify all nod :
   4.1.obj.MinHeapify(i)          from bottom to :p
  5.obj.heapSort()//perform heapsort & print sorted ar ay


}
```

Hashing

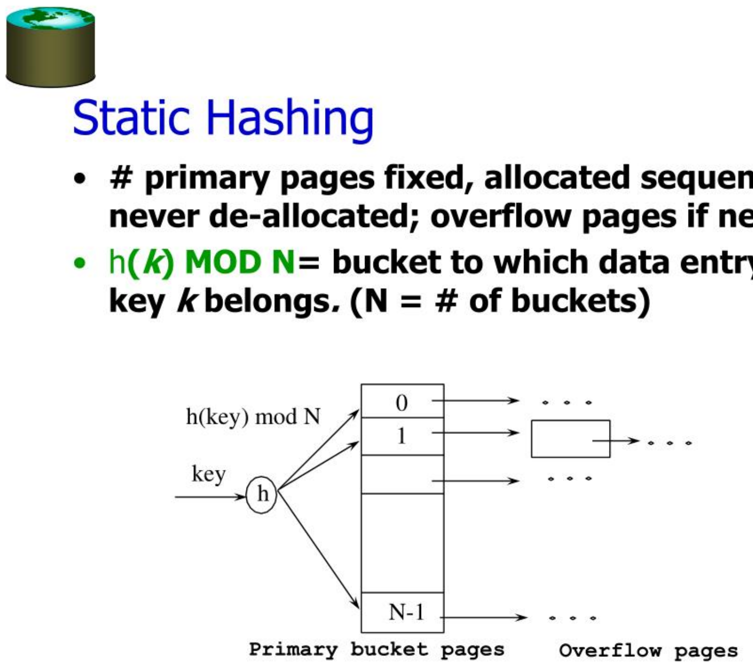


Index	Value
0	
1	
2	Null
3	
:	:
n	


Hash Function
buckets
hash('a') & 7
key : 'a'
value: 5
key : 'b'
value: 29
key : 'c'
value: 3
key : 'z'
value: 13
0
key : 'a'
value: 5
1
2
key : 'c'
value: 3
3
key : 'b'
value: 29
collision
6
key : 'z'
value: 13
probing
7

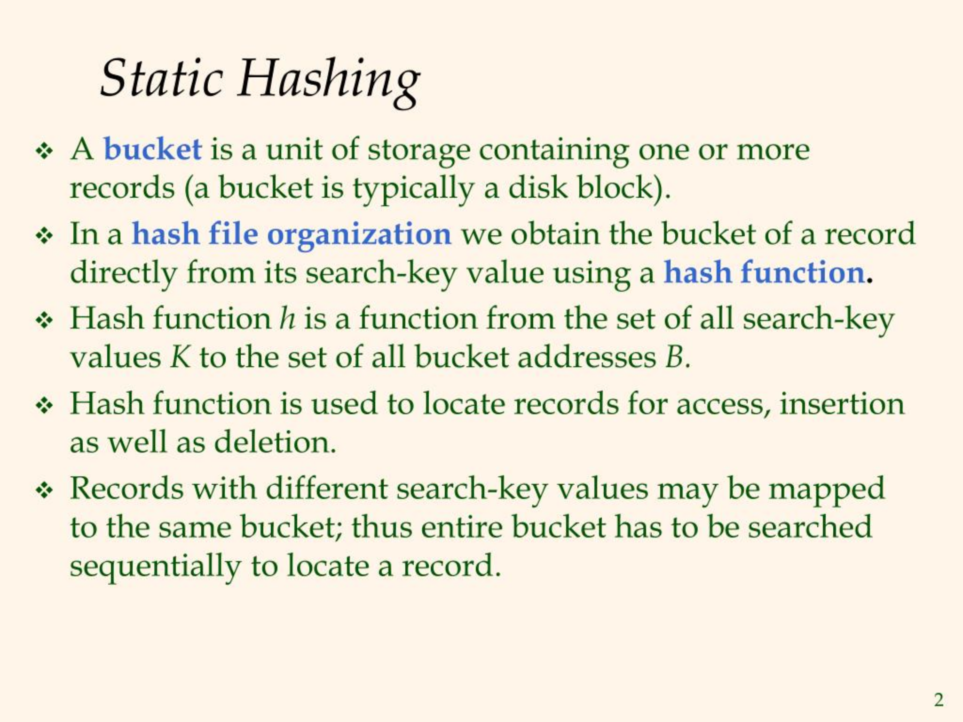

Hashing function in
Division method
Mid Square Method
Digit Folding Method
www.educba.com

Static Hash



Static Hashing

- **# primary pages fixed, allocated sequentially, never de-allocated; overflow pages if needed.**
- h(***k***) **MOD N= bucket to which data entry with key *k* belongs. (N = # of buckets)**



Static Hashing

- A **bucket** is a unit of storage containing one or more records (a bucket is typically a disk block).
- In a **hash file organization** we obtain the bucket of a record directly from its search-key value using a **hash function.**
- Hash function *h* is a function from the set of all search-key values *K* to the set of all bucket addresses *B*.
- Hash function is used to locate records for access, insertion as well as deletion.
- Records with different search-key values may be mapped to the same bucket; thus entire bucket has to be searched sequentially to locate a record.

2

-- Memo End --