

Basic Concepts

What is Program

- A Set of Instructions
- Data Structures + Algorithms
- Data Structure = A Container stores Data
- Algorithm = Logic + Control

What is it all about?

- Solving problems
 - Get me from home to work
 - Balance my checkbook
 - Simulate a jet engine
 - Graduate from SPU
- Using a computer to help solve problems
 - Designing programs (architecture, algorithms)
 - Writing programs
 - Verifying programs
 - Documenting programs

Data Structures and Algorithms

- **Algorithm**
 - Outline, the essence of a computational procedure, step-by-step instructions
- **Program** – an implementation of an algorithm in some programming language
- **Data structure**
 - **Organization** of data needed to solve the problem

Overall Picture

Data Structure and Algorithm Design Goals

Correctness



Efficiency



Implementation Goals

Robustness



Adaptability



Reusability



Functions of Data Structures

- Add
 - Index
 - Key
 - Position
 - Priority
- Get
- Change
- Delete

Common Data Structures

- Array
- Stack
- Queue
- Linked List
- Tree
- Heap
- Hash Table
- Priority Queue

How many Algorithms?

- Countless

Steps in Program Development

- Define the problem
- Outline the solution
- Develop the outline into an algorithm
- Test the algorithm for correctness
- Code the algorithm into a programming language
- Run the program on computer
- Document and maintain the program

Structured programming

Can be :

- 1.Top-down development
- 2.Modular design
- 3. Structure theorem
- 4. Algorithms
- 5. Pseudocode

Top-down development

- Consider the ***problem as a whole and break it down into component parts.***
- ***Each part becomes a problem to be solved*** in its own right.
- Using stepwise refinement, the problem is repeatedly ***broken down into smaller parts*** until it becomes manageable.

Modular design

- A module is
 - a step or *sub-task resulting from stepwise refinement.*
 - a problem solution which has a *well-defined set of inputs, processing and outputs.*
- Algorithms are written at module level.

The Structure Theorem

- Only three constructs should ever be used in programming:-
 - 1 Sequence
 - 2 Selection
 - 3 Iteration
- i.e. no unstructured GOTOs or STOPs.

Algorithm Definition

- Lists the steps involved in accomplishing a task which must
 - be lucid, precise and unambiguous
 - give the correct solution in all cases
 - eventually end
- Algorithms can be written **in pseudocode**, or can be expressed by means of a **flowchart**

Pseudocode

- Statements are written in simple English.
- Each instruction is written on a separate line.
- Keywords and indentation are used to signify particular control structures.
- Each set of instructions is written from top to bottom with only 1 entry and 1 exit.
- Groups of statements may be formed into modules and that group can be given a name.

Topics

- Introduction
- Definitions
- Classification of Data Structures
- Arrays and Linked Lists
- Abstract Data Types [ADT]
 - The List ADT
 - Array-based Implementation
 - Linked List Implementation
 - Cursor-based Implementation
- Doubly Linked Lists

Data Structure [Wikipedia]

- Data Structure is a particular way of storing and organizing data in a computer so that it can be used efficiently.
- Different kinds of data structures are suited to different kinds of applications.
- **Storing and retrieving** can be carried out on data stored in both **main memory and in secondary memory**.

Merriam-Webster's Definition

- Way in which data are stored for efficient search and retrieval.
- The simplest data structure is the one-dimensional (linear) array.
- Data items stored non-consecutively in memory may be linked by pointers.
- Many algorithms have been developed for storing data efficiently

Algorithms [Wikipedia]

- An algorithm is a step-by-step procedure for calculations.
- An algorithm is an effective method expressed as a **finite list of well-defined instructions** for calculating a function.
- The transition from one state to the next is not necessarily deterministic; some algorithms incorporate random input.

Merriam-Webster's Definition

- **Procedure that produces the answer to a question or the solution to a problem in a finite number of steps.**
- An **algorithm that produces a yes or no answer is called a decision procedure**; one that leads to a solution is a computation procedure.
- Example: A mathematical formula and the instructions in a computer program

Data Structure Classification

- **Primitive / Non-primitive**
 - Basic Data Structures available / Derived from Primitive Data Structures
- **Homogeneous / Heterogeneous**
 - Elements are of the same type / Different types
- **Static / Dynamic**
 - memory is allocated at the time of compilation / run-time
- **Linear / Non-linear**
 - Maintain a Linear relationship between element

ADT - General Concept

- Problem solving with a computer means **processing data**
- To process data, we need to define the data type and the operation to be performed on the data
- The definition of the data type and the definition of the operation to be applied to the data is part of the idea behind an Abstract Data Type (ADT)

ADT - General Concept

- The user of an ADT needs only to know that a set of operations are available for the data type, but does not need to know how they are applied
- Several simple ADTs, such as **integer, real, character, pointer** and so on, have been implemented and are available for use in most languages

Data Types

- A data type is characterized by:
 - A *set of values*
 - A *data representation*, which is common to all these values, and
 - A *set of operations*, which can be applied uniformly to all these values

Primitive Data Types

- Languages like ‘C’ provides the following ***primitive data types***:
 - boolean
 - char, byte, int
 - float, double
- ***Each primitive type has:***
 - A set of values
 - A data representation
 - A set of operations
- These are “set in stone”.

ADT Definition [Wikipedia]

- In computer science, an abstract data type (ADT) is a **mathematical model for a certain class** of data structures that **have similar behavior**.
- An abstract data type is defined indirectly, only by the operations that may be performed on it and by mathematical constraints on the effects (and possibly cost) of those operations.

ADT Definition [Wikipedia]

- An ADT may be implemented **by specific data types or data structures**, in many ways and in many programming languages; or described in a formal specification language.
- example, an abstract stack could be defined by three operations:
 - push, that inserts some data item onto the structure,
 - pop, that extracts an item from it, and
 - peek, that allows data on top of the structure to be examined without removal.

Definition from techforum4you

- Abstract data types or ADTs are a mathematical specification of a set of data and the set of operations that can be performed on the data.
- They are abstract in the sense that the focus is on the definitions and the various operations with their arguments.
- The actual implementation is not defined, and does not affect the use of the ADT.

ADT in Simple Words

- Definition:
 - Is a set of operation
 - Mathematical abstraction
 - No implementation detail
- Example:
 - Lists, sets, graphs, stacks are examples of ADT along with their operations

Why ADT?

- Modularity
 - divide program into small functions
 - easy to debug and maintain
 - easy to modify
 - group work
- Reuse
 - do some operations only once
- Easy to change the implementation
 - transparent to the program

Implementing an ADT

- To implement an ADT, you need to choose:
 - A data representation
 - must be able to represent all necessary values of the ADT
 - should be private
 - An algorithm for each of the necessary operation:
 - must be consistent with the chosen representation
 - all auxiliary (helper) operations that are not in the contract should be private
- Remember: Once other people are using it
 - It's easy to add functionality

The List ADT

- The List is an
 - Ordered sequence of data items called elements
 - $A_1, A_2, A_3, \dots, A_N$ is a list of size N
 - size of an empty list is 0
 - A_{i+1} succeeds A_i
 - A_{i-1} precedes A_i
 - Position of A_i is i
 - First element is A_1 called “head”
 - Last element is A_N called “tail”

Operations on Lists

- MakeEmpty
- PrintList
- Find
- FindKth
- Insert
- Delete
- Next
- Previous

List – An Example

- The elements of a list are 34, 12, 52, 16, 12
 - Find (52) -> 3
 - Insert (20, 4) -> 34, 12, 52, 20, 16, 12
 - Delete (52) -> 34, 12, 20, 16, 12
 - FindKth (3) -> 20

List - Implementation

- Lists can be implemented using:
 - Arrays
 - Linked List
 - Cursor [Linked List using Arrays]

Arrays

- Array is a static data structure that represents a collection of fixed number of homogeneous data items or
- A fixed-size indexed sequence of elements, all of the same type.
- The individual elements are typically stored in consecutive memory locations.
- The length of the array is determined when the array is created, and cannot be changed.

Arrays

- Any component of the array can be inspected or updated by using its index.
 - This is an efficient operation
 - $O(1)$ = constant time
- The array indices may be integers (C, Java) or other discrete data types (Pascal, Ada).
- The lower bound may be zero (C, Java), one (Fortran), or chosen by the programmer (Pascal, Ada)

Different Types of Arrays

- One-dimensional array: only one index is used
- Multi-dimensional array: array involving more than one index
- Static array: the compiler determines how memory will be allocated for the array
- Dynamic array: memory allocation takes place during execution

One Dimensional Static Array

- Syntax:
 - `ElementType arrayName [CAPACITY];`
 - `ElementType arrayName [CAPACITY] = { initializer_list };`
- Example in C++:
 - `int b [5];`
 - `int b [5] = {19, 68, 12, 45, 72};`

Array Output Function

```
void display(int array[],int num_values)
{
    for (int i = 0; i<num_values; i++)
        printf( "%d",array[i]);
}
```

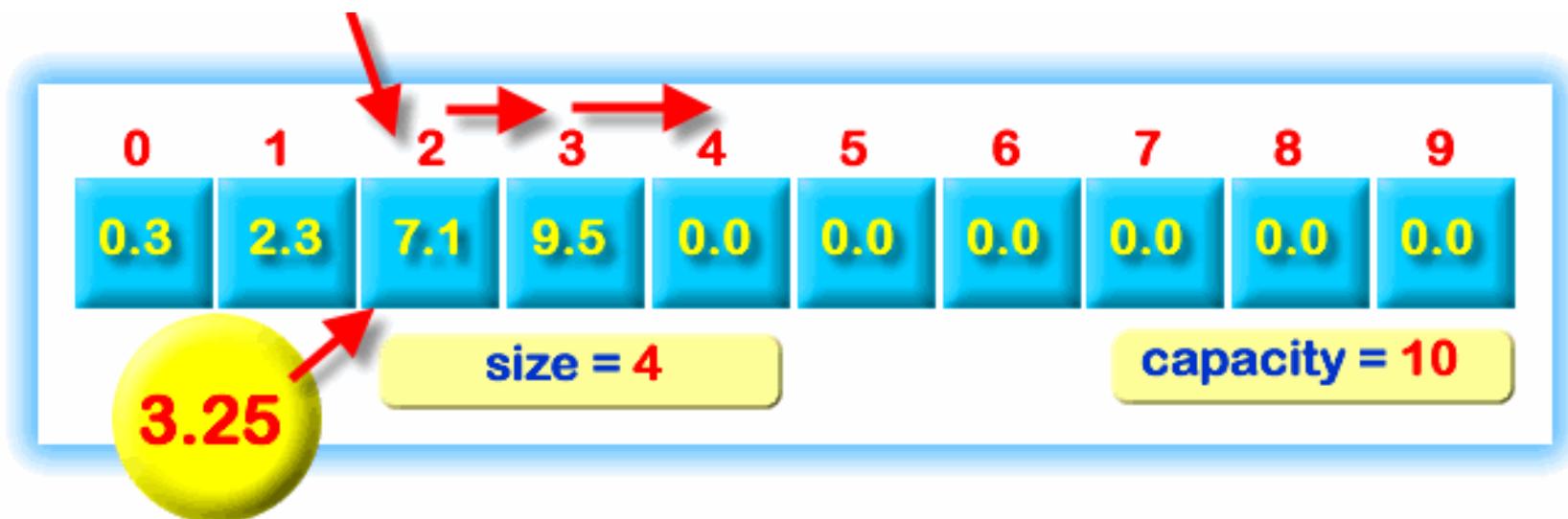
List Implemented Using Array



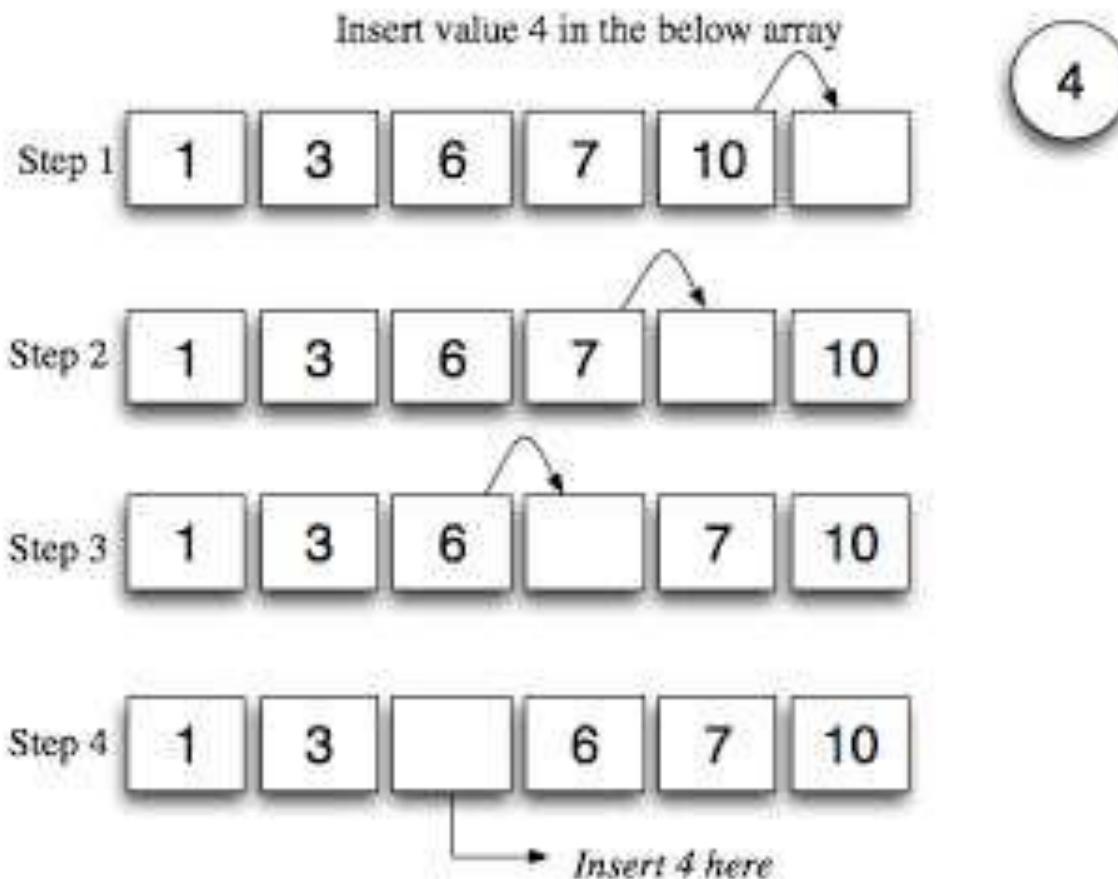
Operations On Lists

- We'll consider only few operations and not all operations on Lists
- Let us consider Insert
- There are two possibilities:
 - Ordered List
 - Unordered List

Insertion into an Ordered List



Insertion in Detail



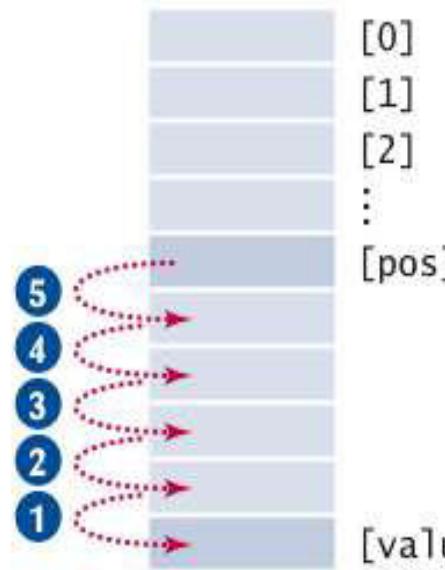
Insertion

[0]
[1]
[2]
⋮

Insert new element here

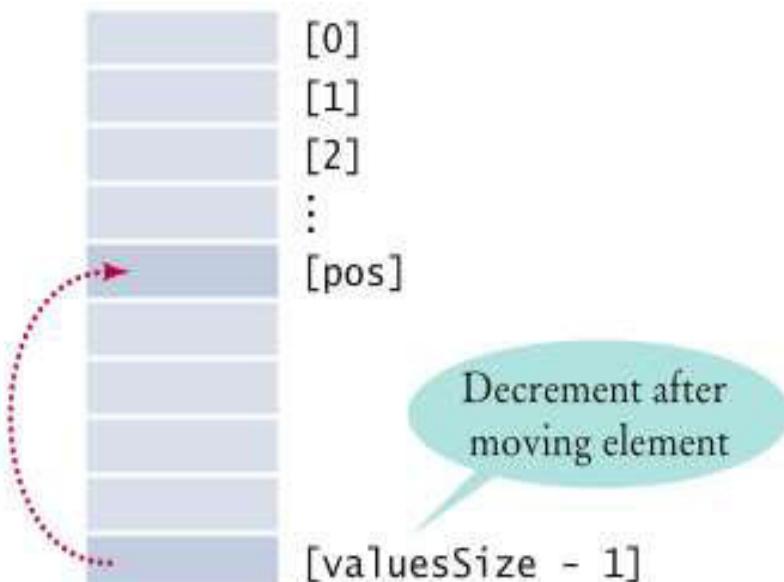
[valuesSize]

Inserting an Element in an Unordered Array

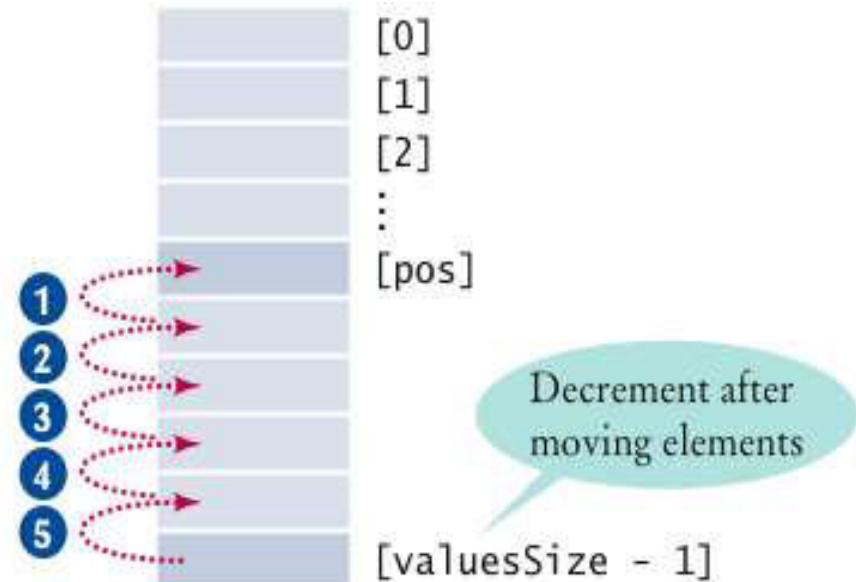


Inserting an Element in an Ordered Array

Deletion



Removing an Element in an Unordered Array



Removing an Element in an Ordered Array

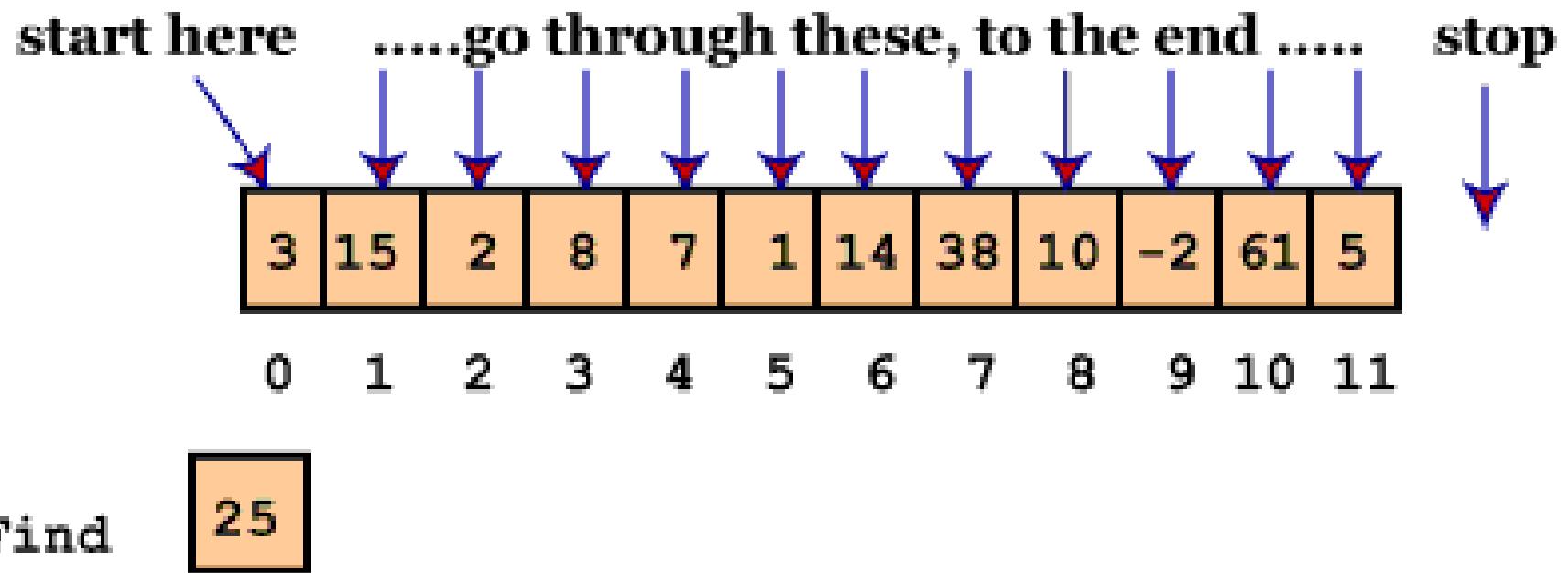
Find / Search

- Searching is the process of looking for a specific element in an array
- For example, discovering whether a certain score is included in a list of scores.
- Searching, like sorting, is a common task in computer programming.
- There are many algorithms and data structures devoted to searching.
- The most common one is the linear search.

Linear Search

- The linear search approach compares the given value with each element in the array.
- The method continues to do so until the given value matches an element in the list or the list is exhausted without a match being found.
- If a match is made, the linear search returns the index of the element in the array that matches the key.

Linear Search



Linear Search Function

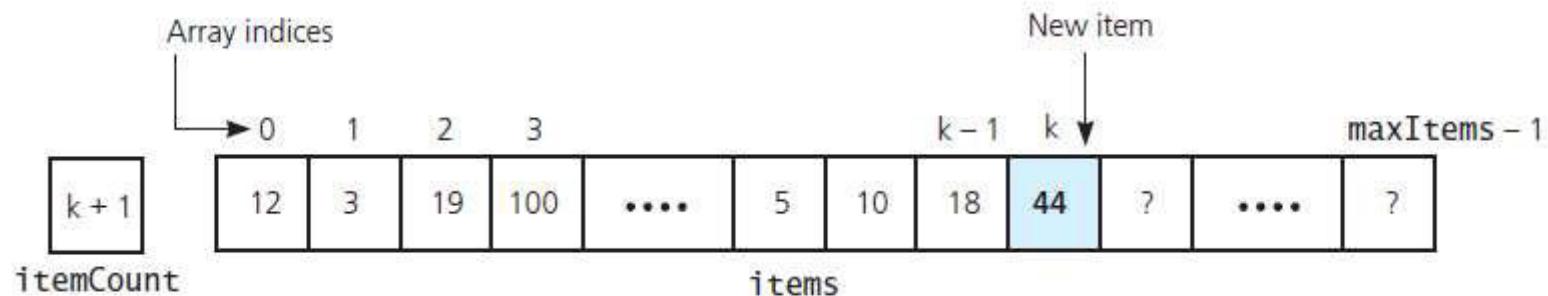
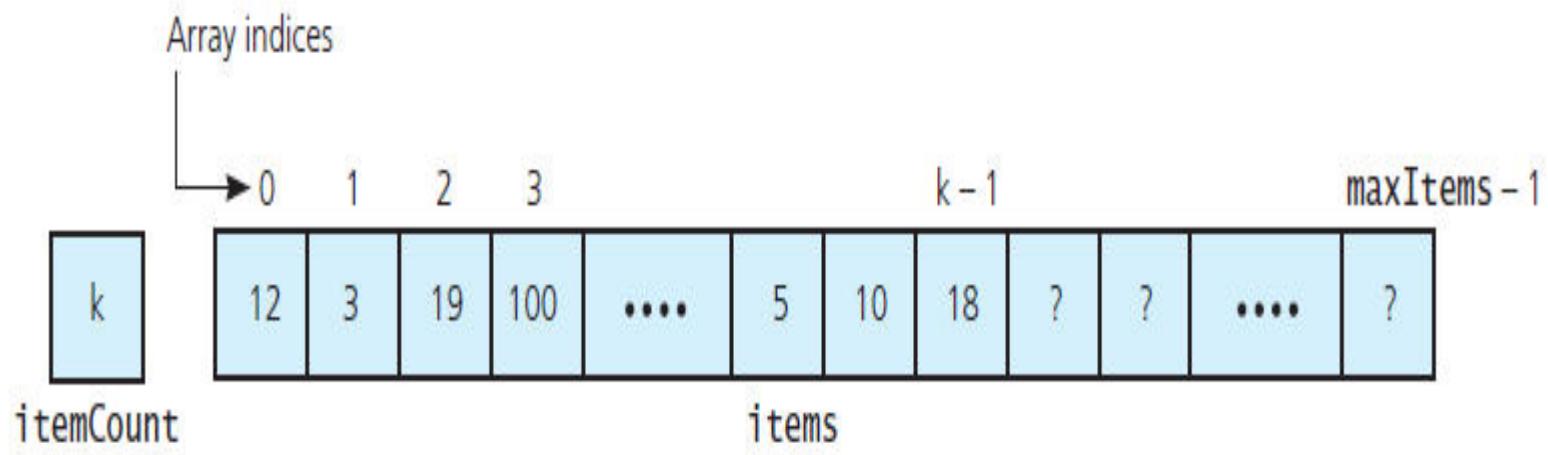
```
int LinearSearch (int a[], int n, int key)
{
    int i;
    for(i=0; i<n; i++)
    {
        if (a[i] == key)
            return i;
    }
    return -1;
}
```

Using the Function

- **LinearSearch (a, n, item, loc)**
- Here "a" is an array of the size n.
- This algorithm finds the location of the element "item" in the array "a".
- If search item is found, it sets loc to the index of the element; otherwise, it sets loc to -1
- **index=linearsearch(array, num, key)**

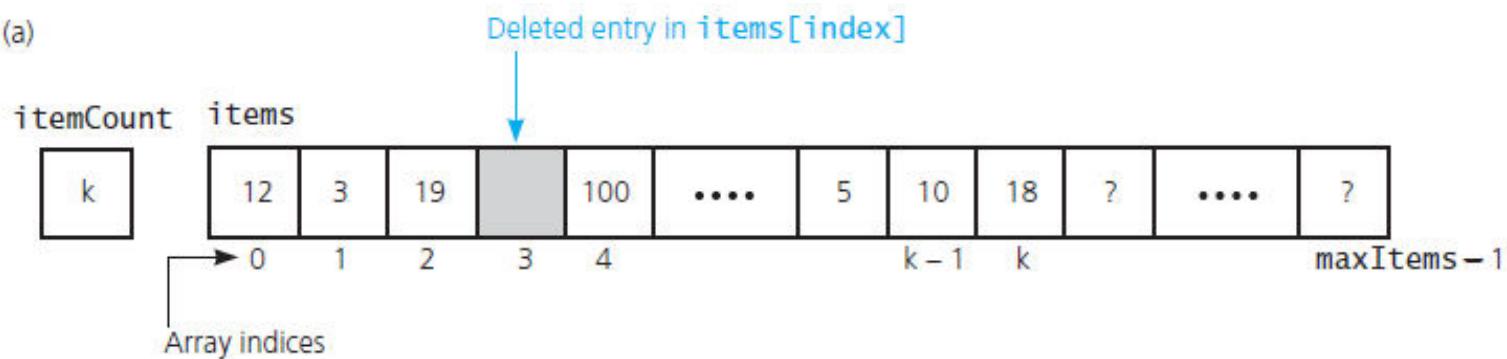
PrintList Operation

```
int myArray [5] = {19, 68, 12, 45, 72};  
/* To print all the elements of the array  
for (int i=0;i<5;i++)  
{  
printf("%d", myArray[i]);  
}
```

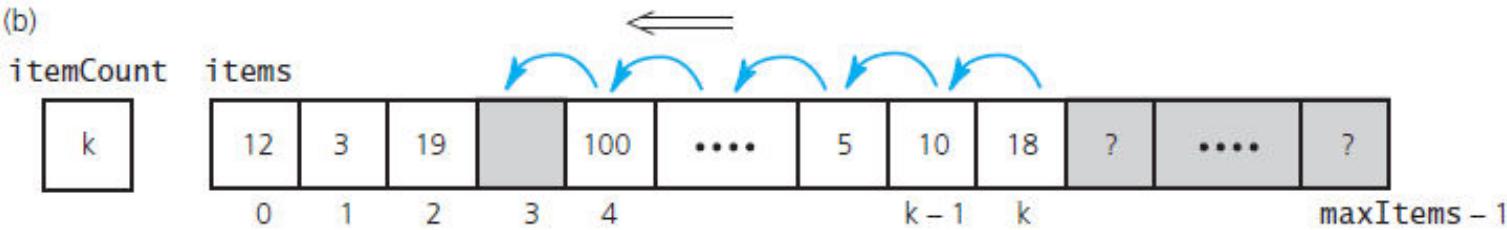


Implementing Deletion

(a)

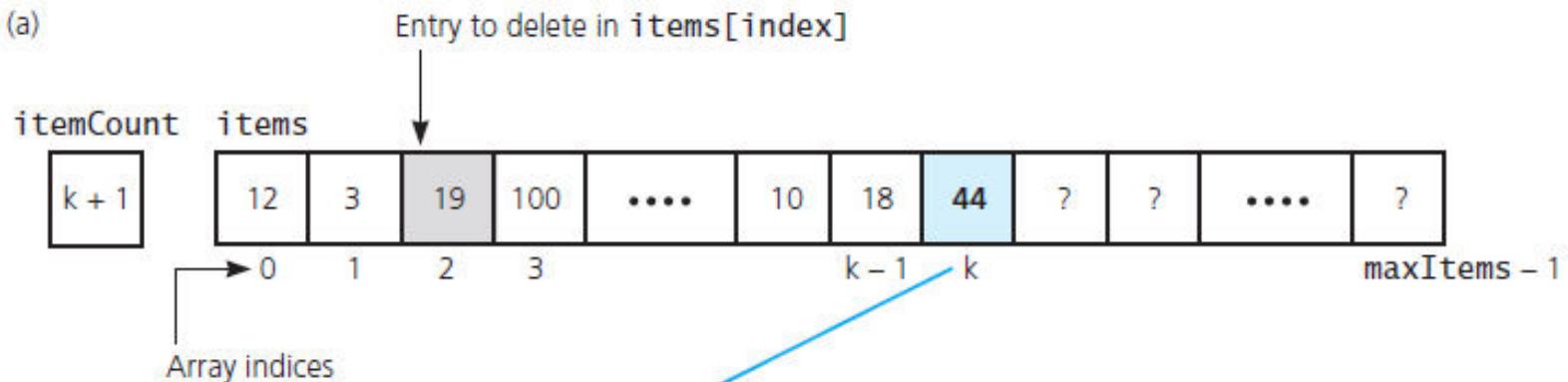


(b)

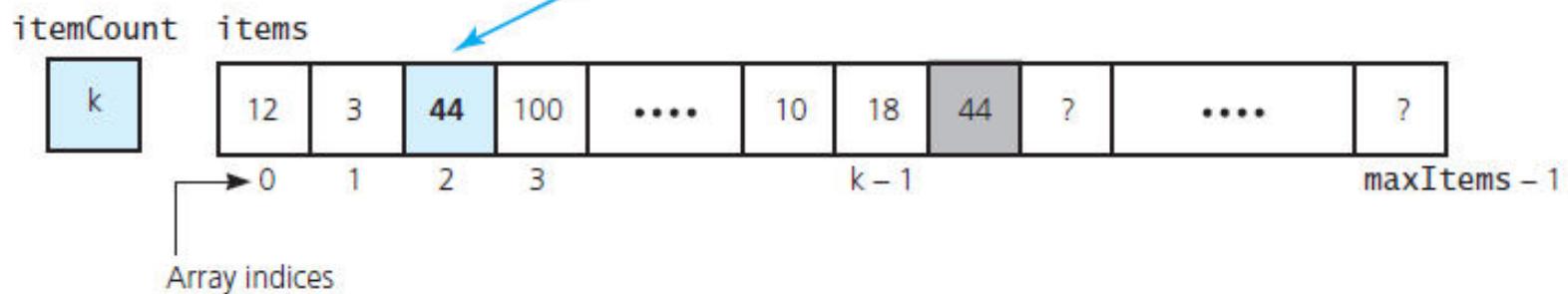


Deletion - Another Method

(a)



(b)



Operations Running Times

PrintList } O(N)
Find

Insert } O(N) (on average half
Delete needs to be moved)

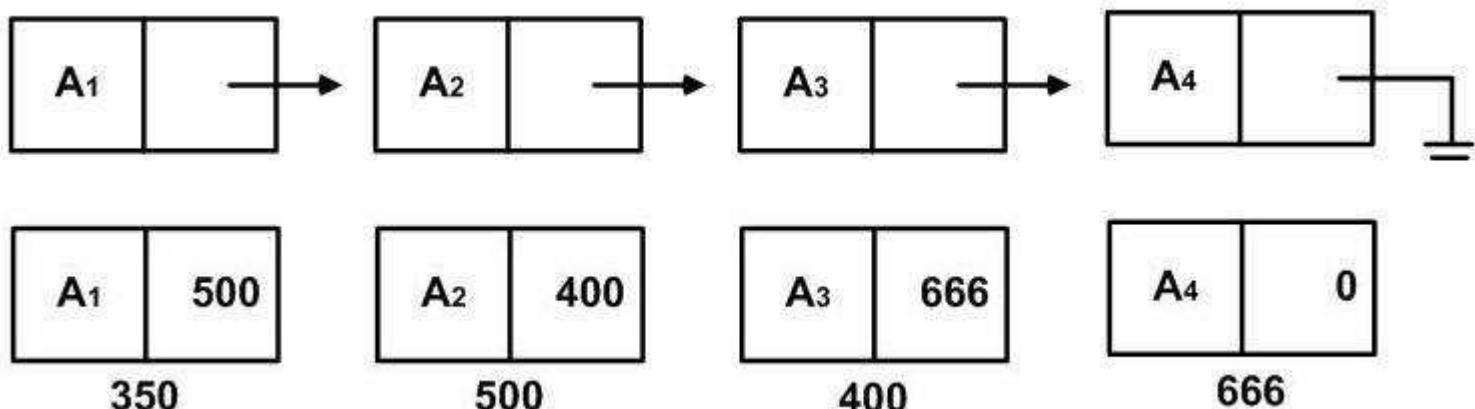
FindKth } O(1)
Next }
Previous

Disadvantages of Using Arrays

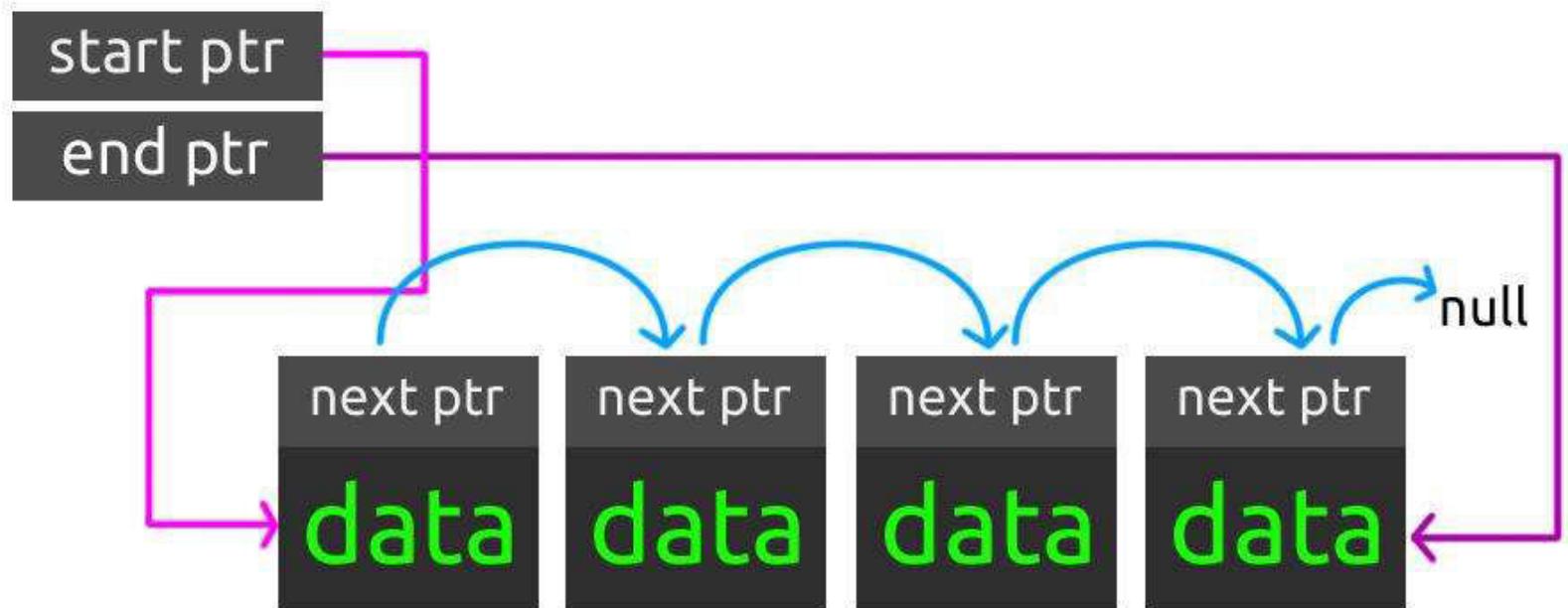
- Need to define a size for array
 - High overestimate (waste of space)
- insertion and deletion is very slow
 - need to move elements of the list
- redundant memory space
 - it is difficult to estimate the size of array

Linked List

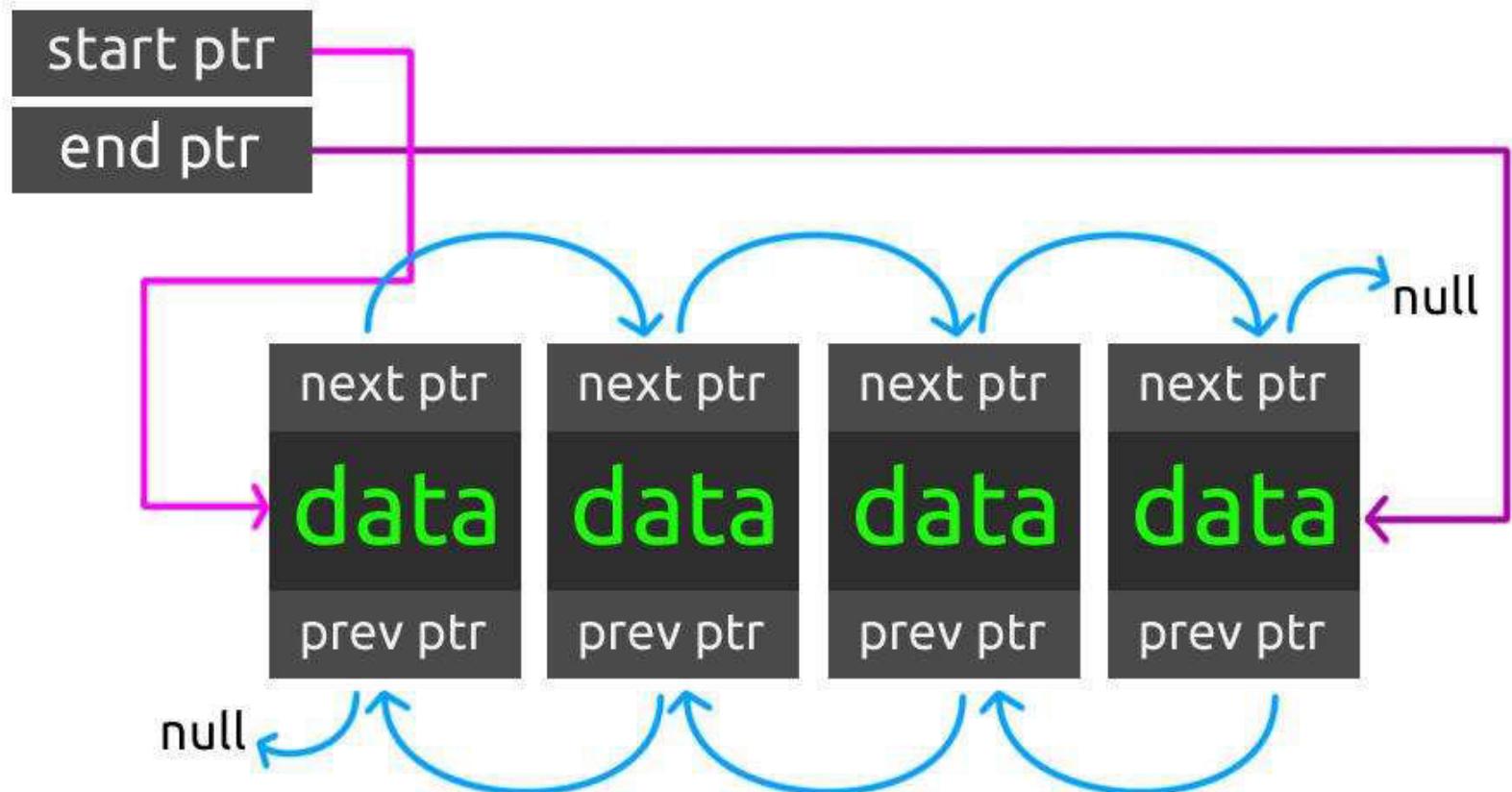
- Series of nodes
 - not adjacent in memory
 - contain the element and a pointer to a node containing its successor
- Avoids the linear cost of insertion and deletion!



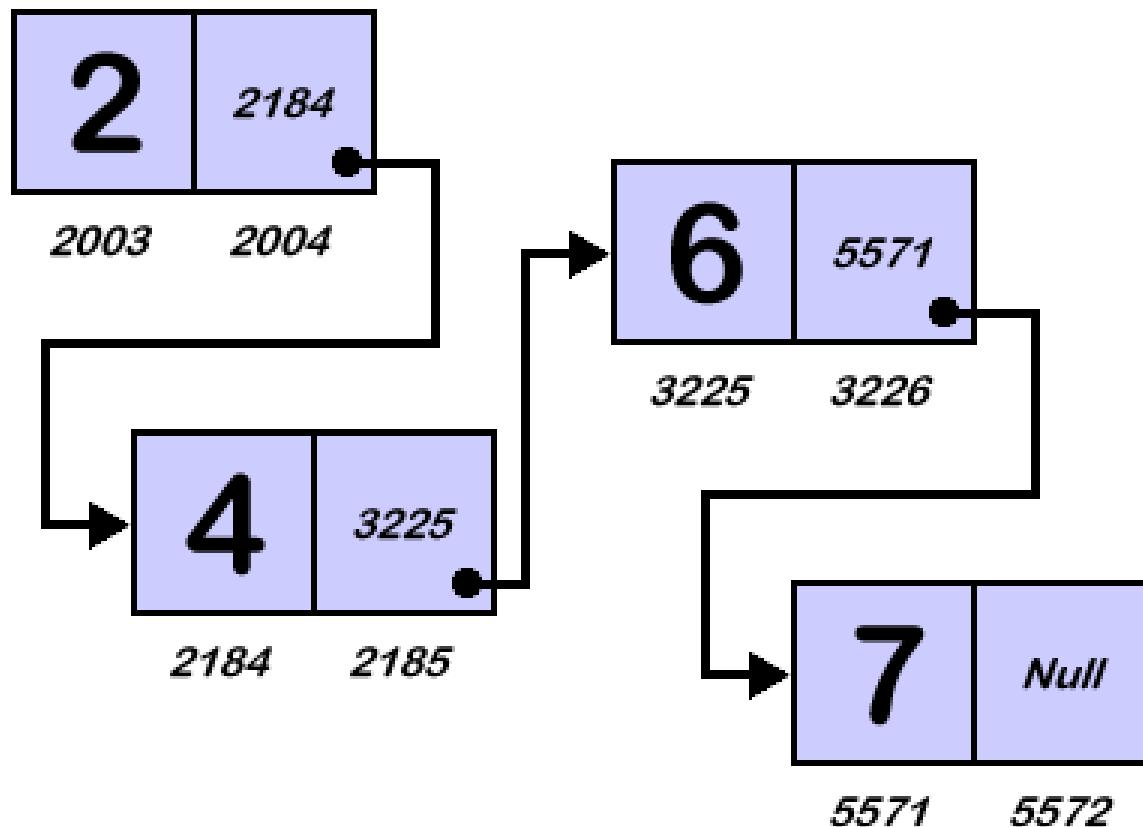
Singly Linked List



Doubly Linked List



Singly Linked List

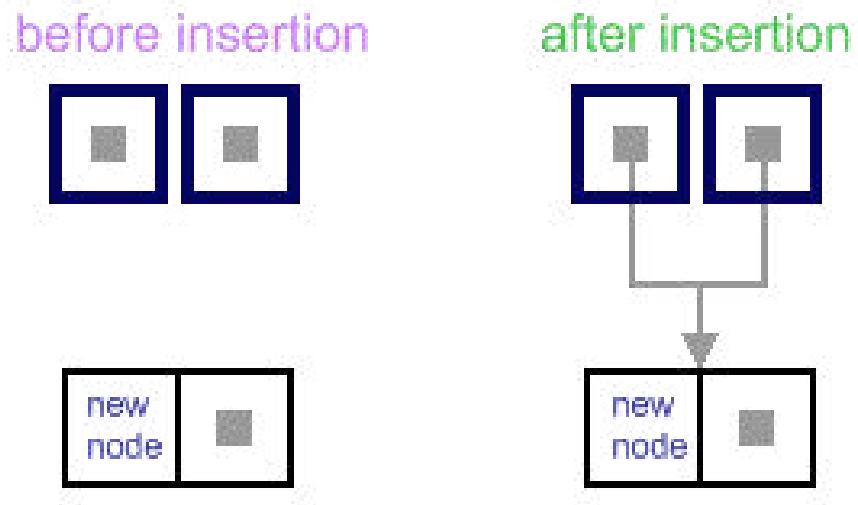


Singly-linked List - Addition

- Insertion into a singly-linked list has two special cases.
- It's insertion a new node before the head (to the very beginning of the list) and after the tail (to the very end of the list).
- In any other case, new node is inserted in the middle of the list and so, has a predecessor and successor in the list.

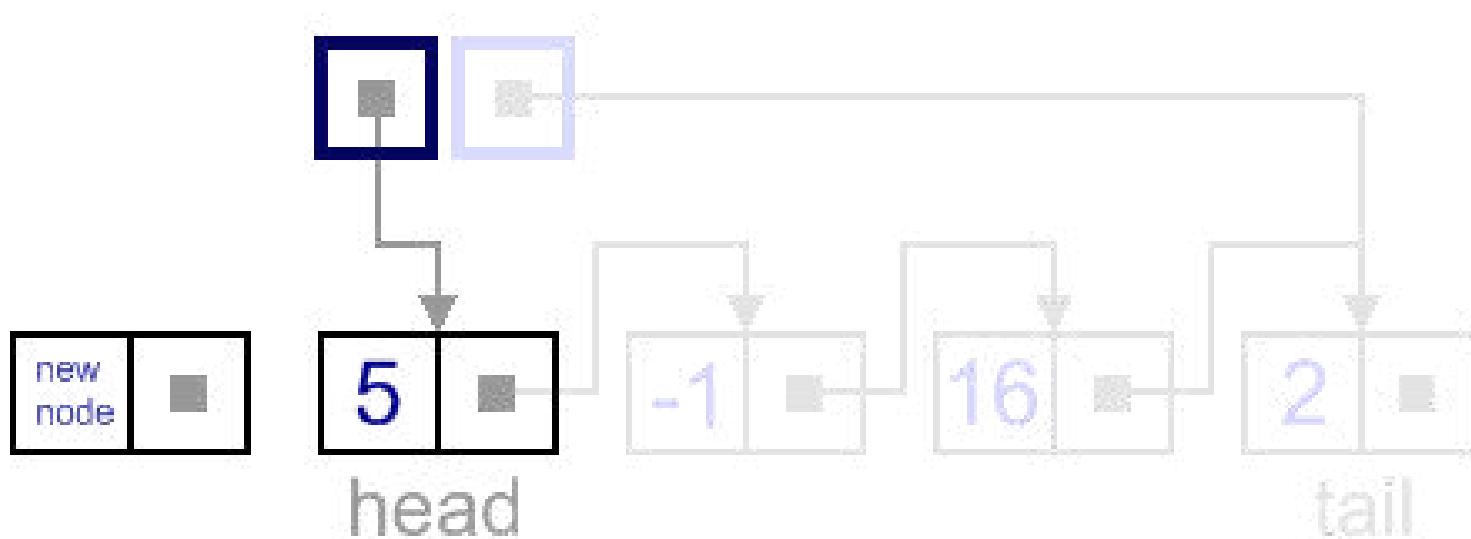
Empty list case

- When list is empty, which is indicated by (`head == NULL`) condition, the insertion is quite simple.
- Algorithm sets both head and tail to point to the new node.



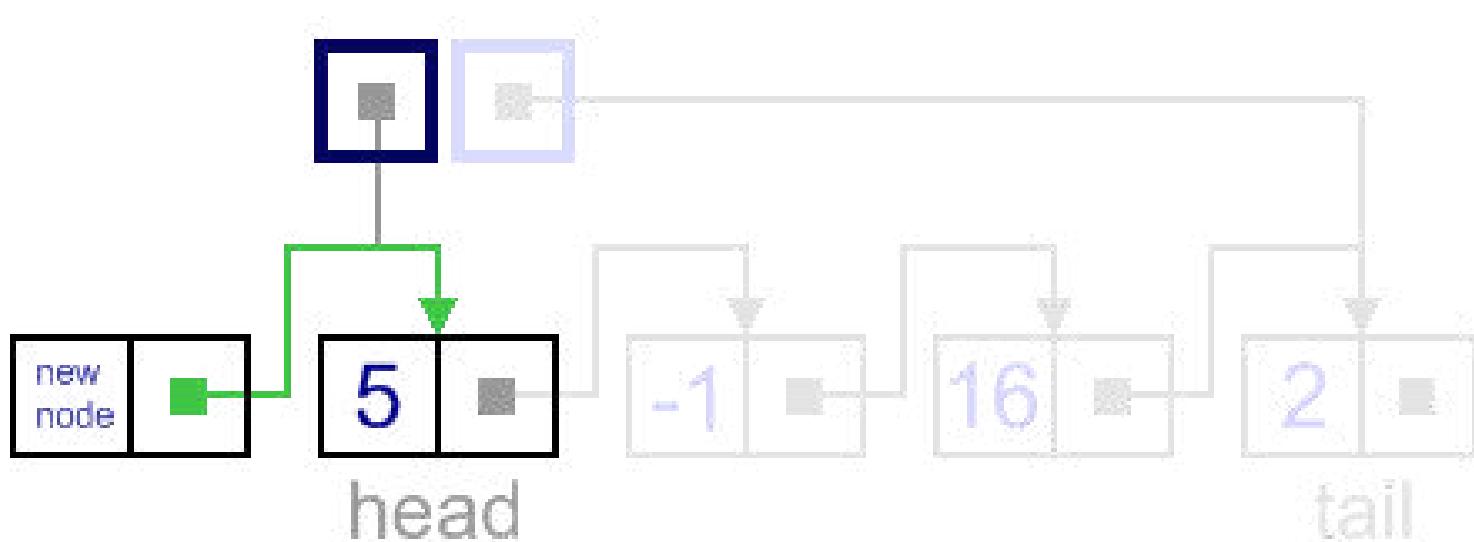
Add first

- In this case, new node is inserted right before the current head node.



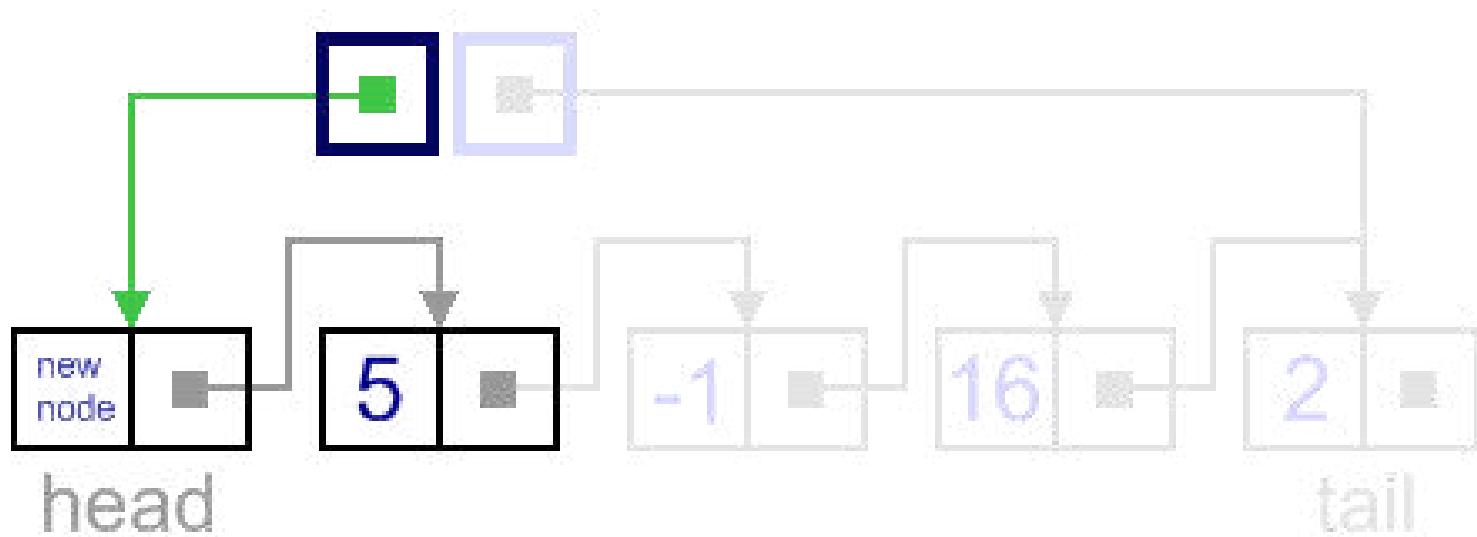
Add First - Step 1

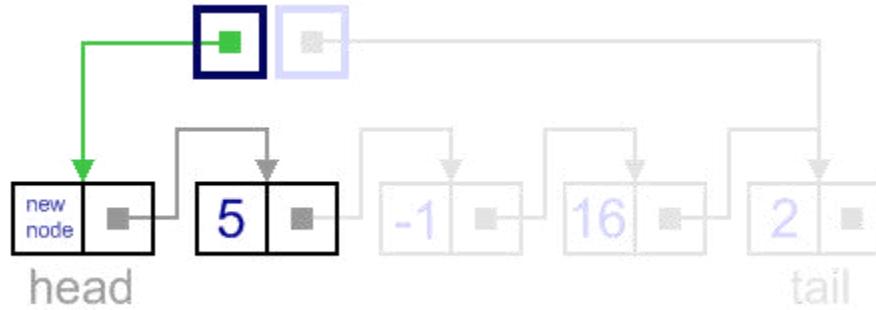
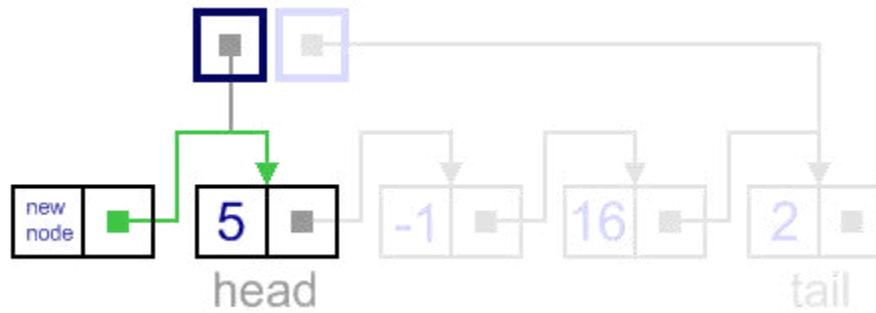
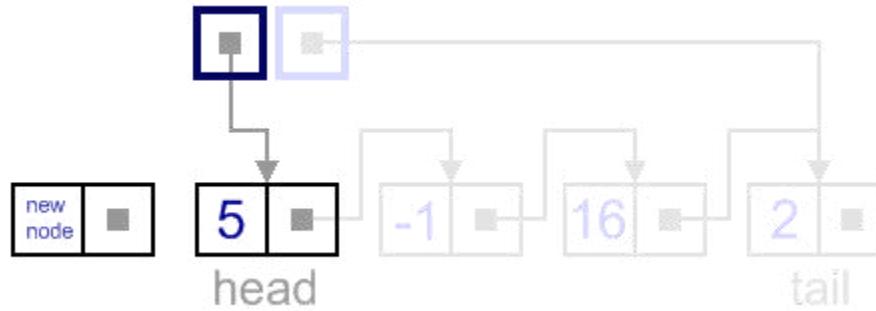
- It can be done in two steps:
 - Update the next link of the new node, to point to the current head node.



Add First - Step 2

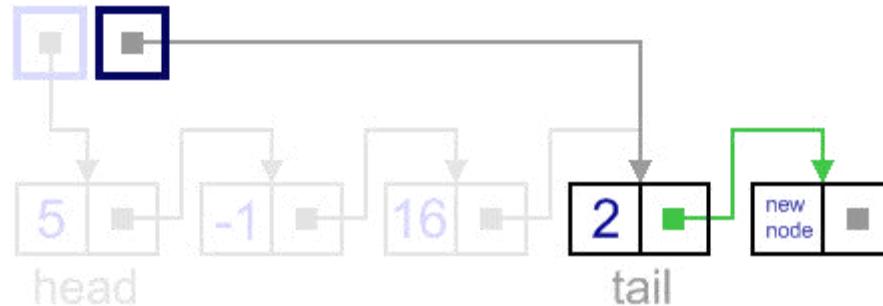
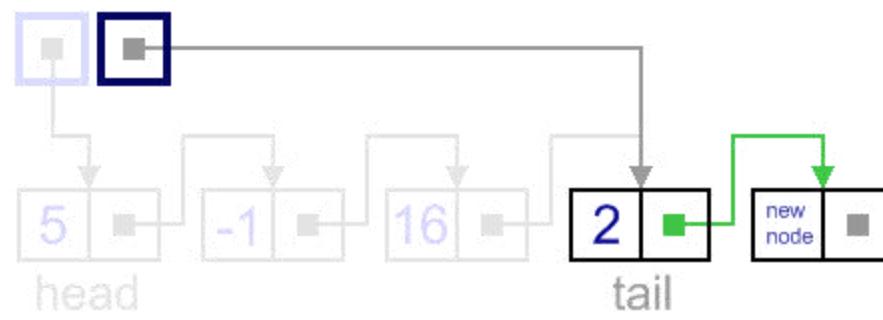
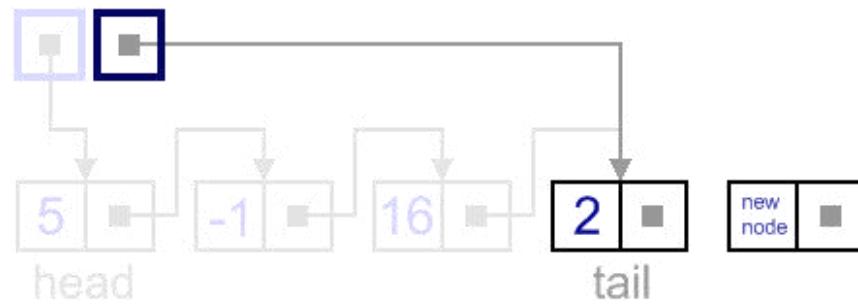
- Update head link to point to the new node.





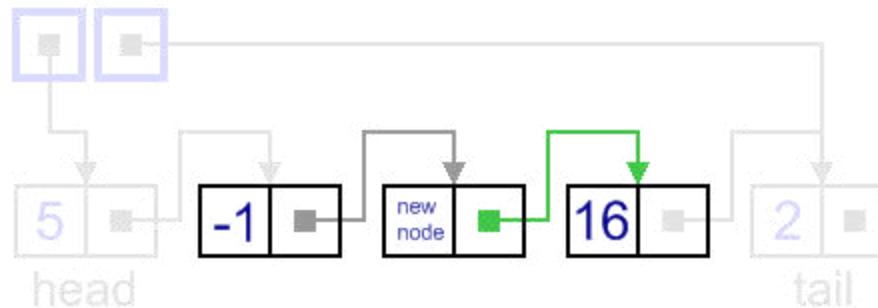
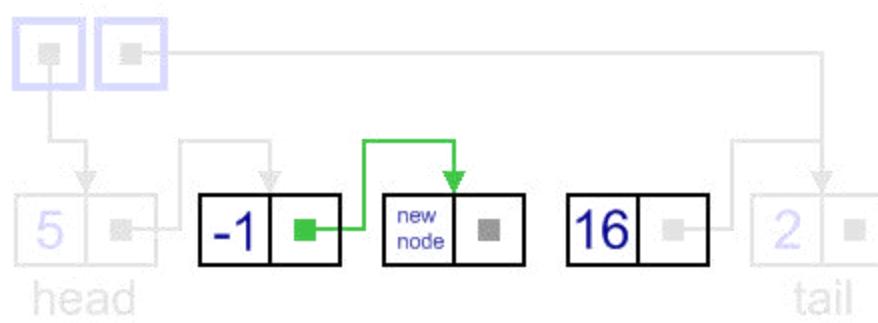
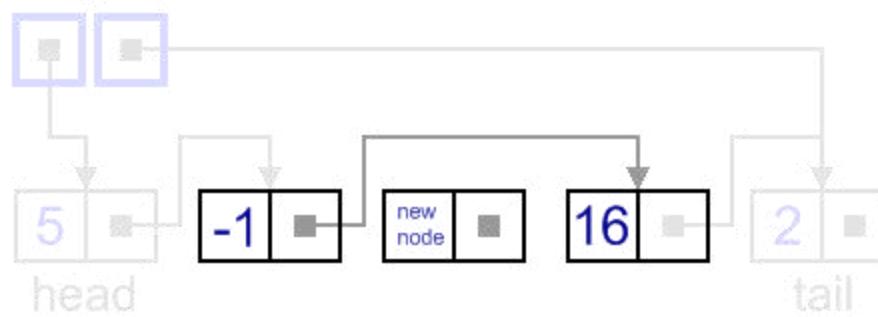
Add last

- In this case, new node is inserted right after the current tail node.
- It can be done in two steps:
 - Update the next link of the current tail node, to point to the new node.
 - Update tail link to point to the new node.



Insert - General Case

- In general case, new node is always inserted between two nodes, which are already in the list. Head and tail links are not updated in this case.
- We need to know two nodes "Previous" and "Next", between which we want to insert the new node.
- This also can be done in two steps:
 - Update link of the "previous" node, to point to the new node.
 - Update link of the new node, to point to the "next" node.

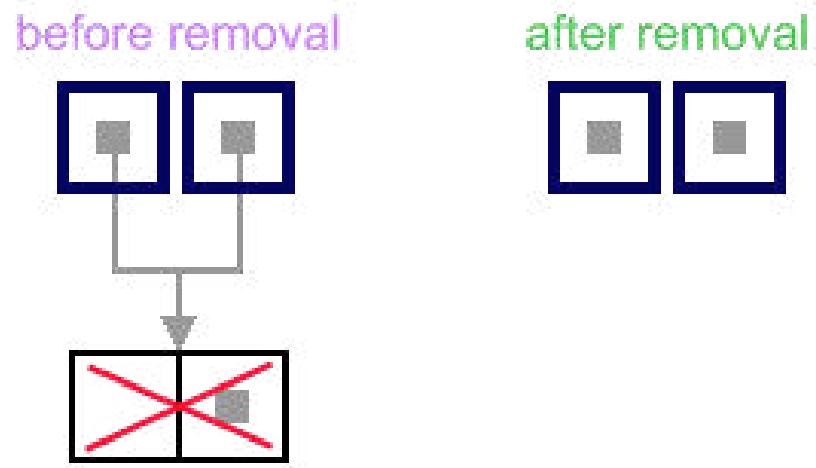


Singly-linked List - Deletion

- There are four cases, which can occur while removing the node.
- We have the same four situations, but the order of algorithm actions is opposite.
- Notice, that removal algorithm includes the disposal of the deleted node - unnecessary in languages with automatic garbage collection (Java).

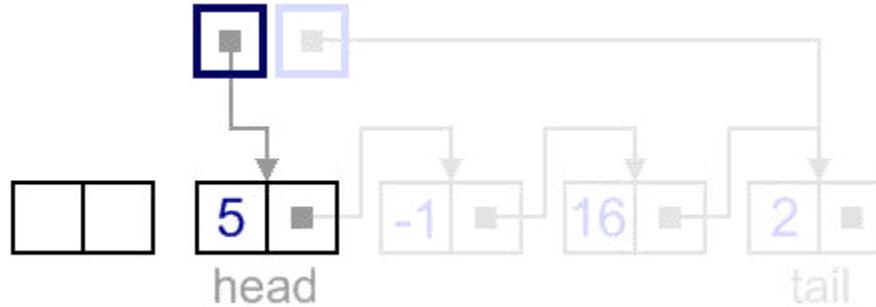
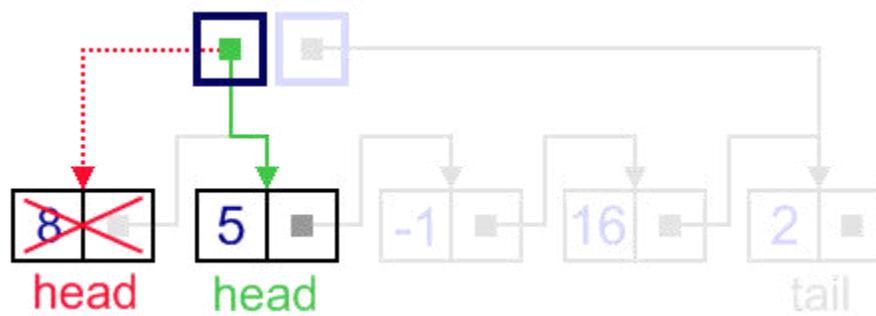
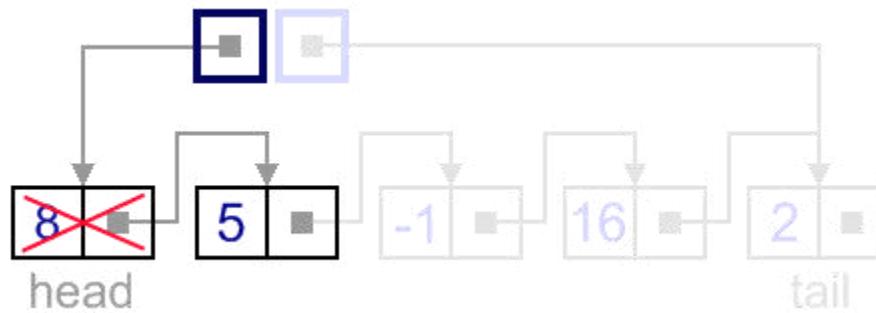
List has only one node

- When list has only one node, that the head points to the same node as the tail, the removal is quite simple.
- Algorithm disposes the node, pointed by head (or tail) and sets both head and tail to NULL.



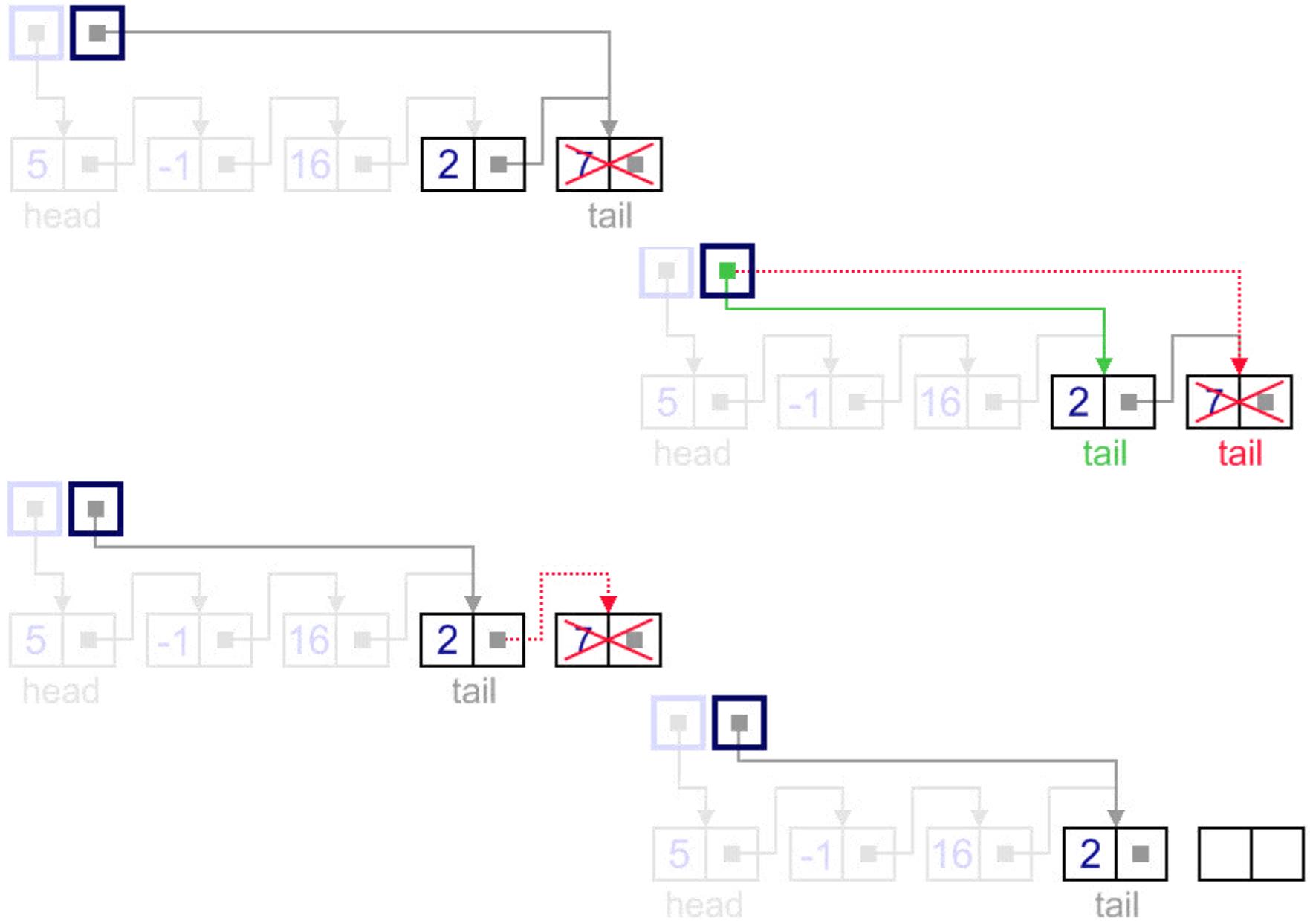
Remove First

- In this case, first node (current head node) is removed from the list.
- It can be done in two steps:
 - Update head link to point to the node, next to the head.
 - Dispose removed node.



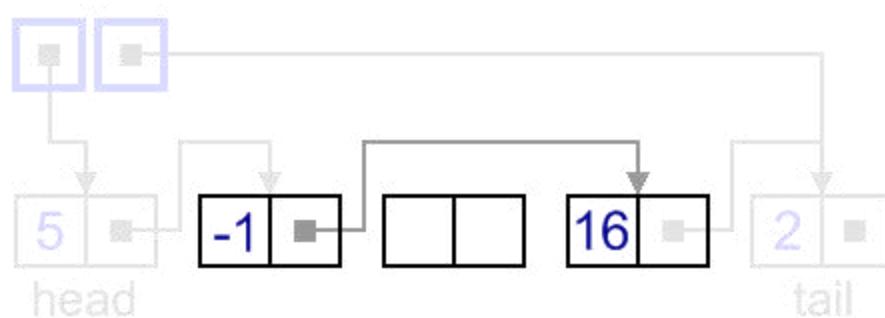
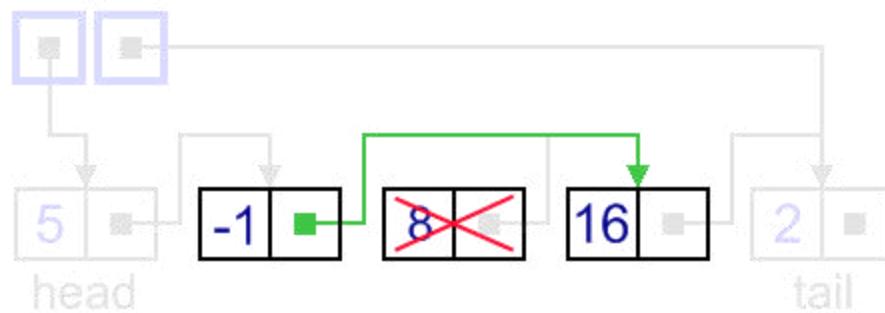
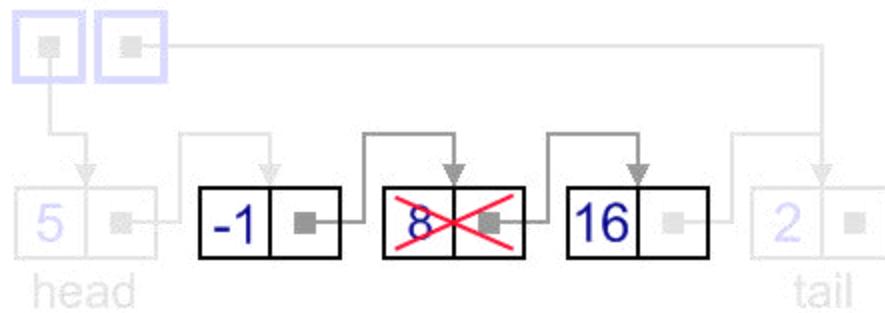
Remove Last

- In this case, last node (current tail node) is removed from the list. This operation is a bit more tricky, than removing the first node, because algorithm should find a node, which is previous to the tail first.
- It can be done in three steps:
 - Update tail link to point to the node, before the tail.
In order to find it, list should be traversed first, beginning from the head.
 - Set next link of the new tail to NULL.
 - Dispose removed node.



Remove - General Case

- In general case, node to be removed is always located between two list nodes. Head and tail links are not updated in this case.
- We need to know two nodes "Previous" and "Next", of the node which we want to delete.
- Such a removal can be done in two steps:
 - Update next link of the previous node, to point to the next node, relative to the removed node.
 - Dispose removed node.



Advantages of Using Linked Lists

- Need to know where the first node is
 - the rest of the nodes can be accessed
- No need to move the elements in the list for insertion and deletion operations
- No memory waste

Cursor Implementation

Problems with linked list implementation:

- Same language do not support pointers!
 - Then how can you use linked lists ?
- **new** and **free** operations are slow
 - Actually not constant time
- SOLUTION: Implement linked list on an array - called CURSOR

Cursor Implementation - Diagram

Slot	Element	Next
0		1
1		2
2		3
3		4
4		5
5		6
6		7
7		8
8		9
9		10
10		0

Cursor Implementation

Slot	Element	Next
0		1
1		2
2		3
3		4
4		5
5		6
6		7
7		8
8		9
9		10
10		0

Slot	Element	Next
0	-	6
1	B	9
2	F	0
3	Header	7
4	-	0
5	Header	10
6	-	4
7	C	8
8	D	2
9	E	0
10	A	1

If L = 5, then L represents list (A, B, E)

If M = 3, then M represents list (C, D, F)

Arrays - Pros and Cons

- Pros
 - Directly supported by C
 - Provides random access
- Cons
 - Size determined at compile time
 - Inserting and deleting elements is time consuming

Linked Lists - Pros and Cons

- Pros
 - Size determined during runtime
 - Inserting and deleting elements is quick
- Cons
 - No random access
 - User must provide programming support

Application of Lists

- Lists can be used
- To store the records sequentially
- For creation of stacks and queues
- For polynomial handling
- To maintain the sequence of operations for do / undo in software
- To keep track of the history of web sites visited

Why Doubly Linked List ?

- given only the pointer location, we cannot access its predecessor in the list.
- Another task that is difficult to perform on a linear linked list is traversing the list in reverse.
- Doubly linked list A linked list in which each node is linked to both its successor and its predecessor
- In such a case, where we need to access the node that precedes a given node, a doubly linked list is useful.

Doubly Linked List

- In a doubly linked list, the nodes are linked in both directions. Each node of a doubly linked list contains three parts:
 - Info: the data stored in the node
 - Next: the pointer to the following node
 - Back: the pointer to the preceding node



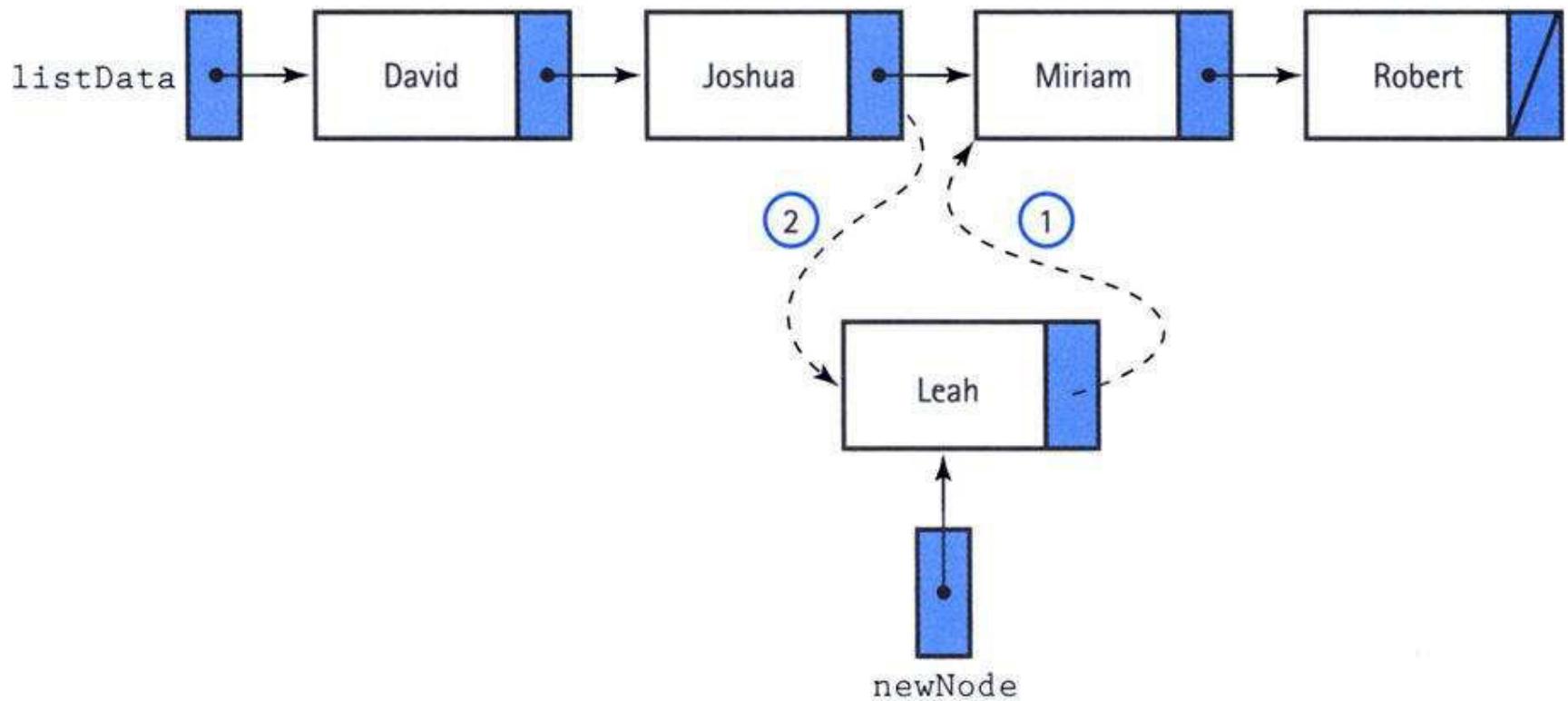
Operations on Doubly Linked Lists

- The algorithms for the insertion and deletion operations on a doubly linked list are somewhat more complicated than the corresponding operations on a singly linked list.
- The reason is clear: There are more pointers to keep track of in a doubly linked list.

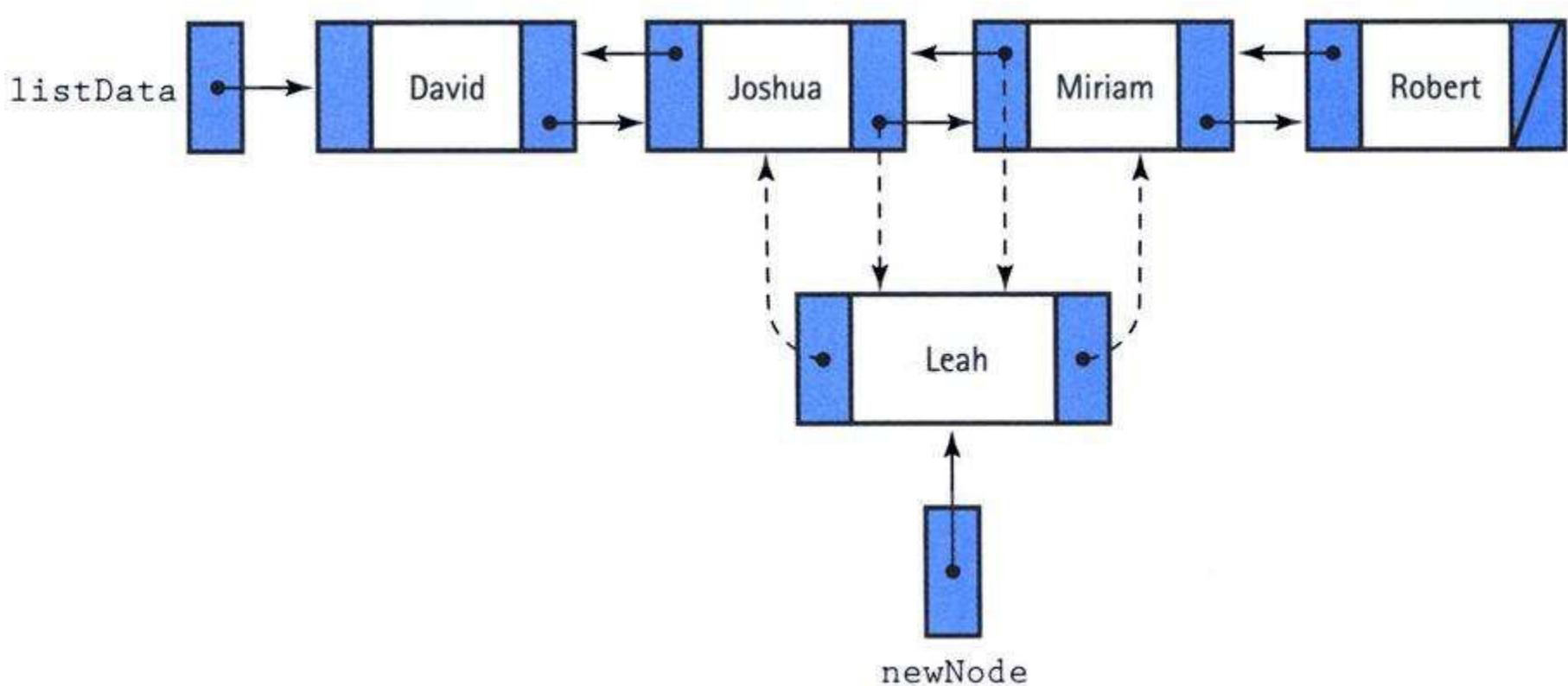
Inserting Item

- As an example, consider the Inserting an item.
- To link the new node, after a given node, in a singly linked list, we need to change two pointers:
 - `newNode->next` and
 - `location->next`.
- The same operation on a doubly linked list requires four pointer changes.

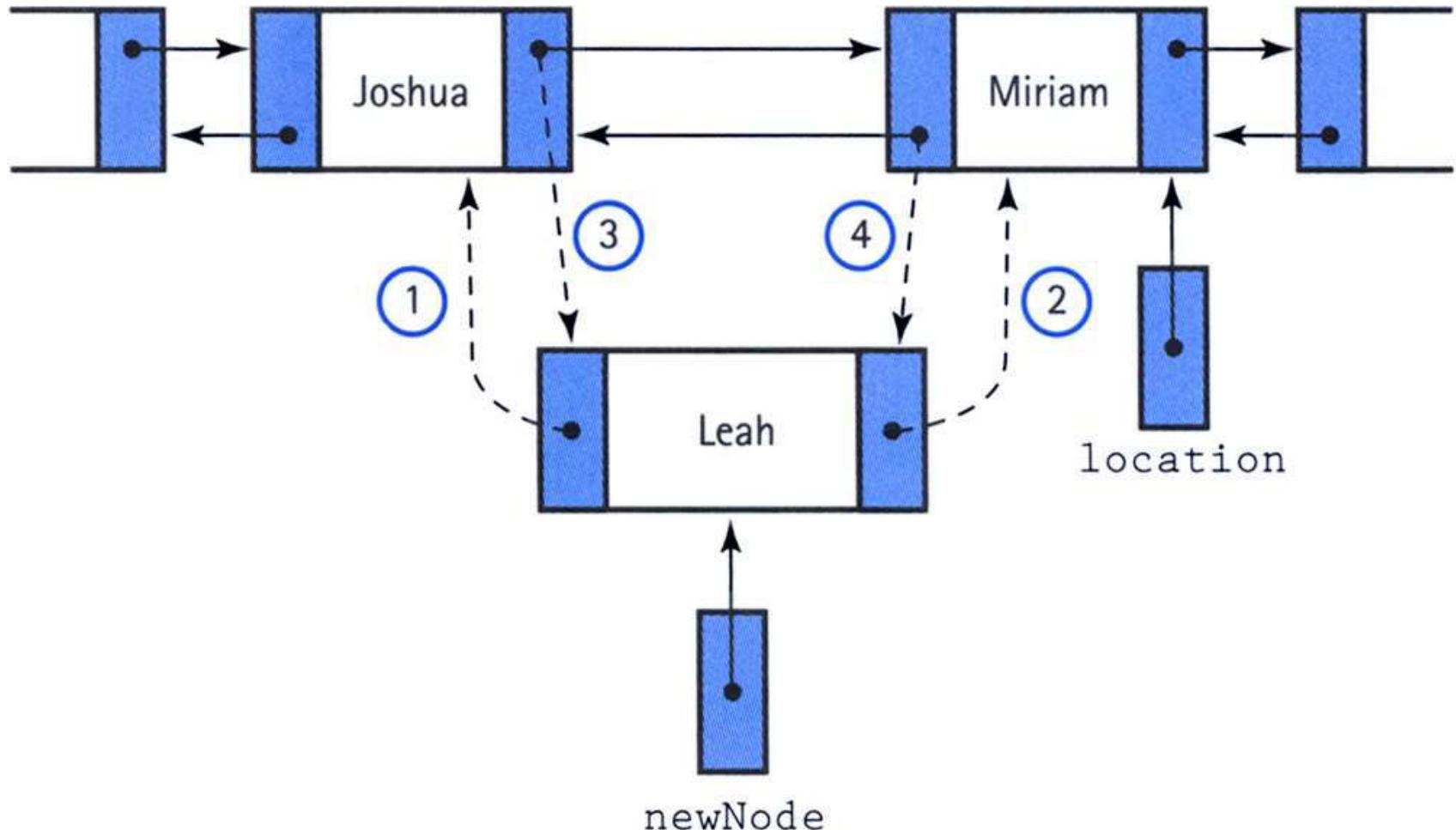
Singly Linked List Insertion



Doubly Linked List Insertion



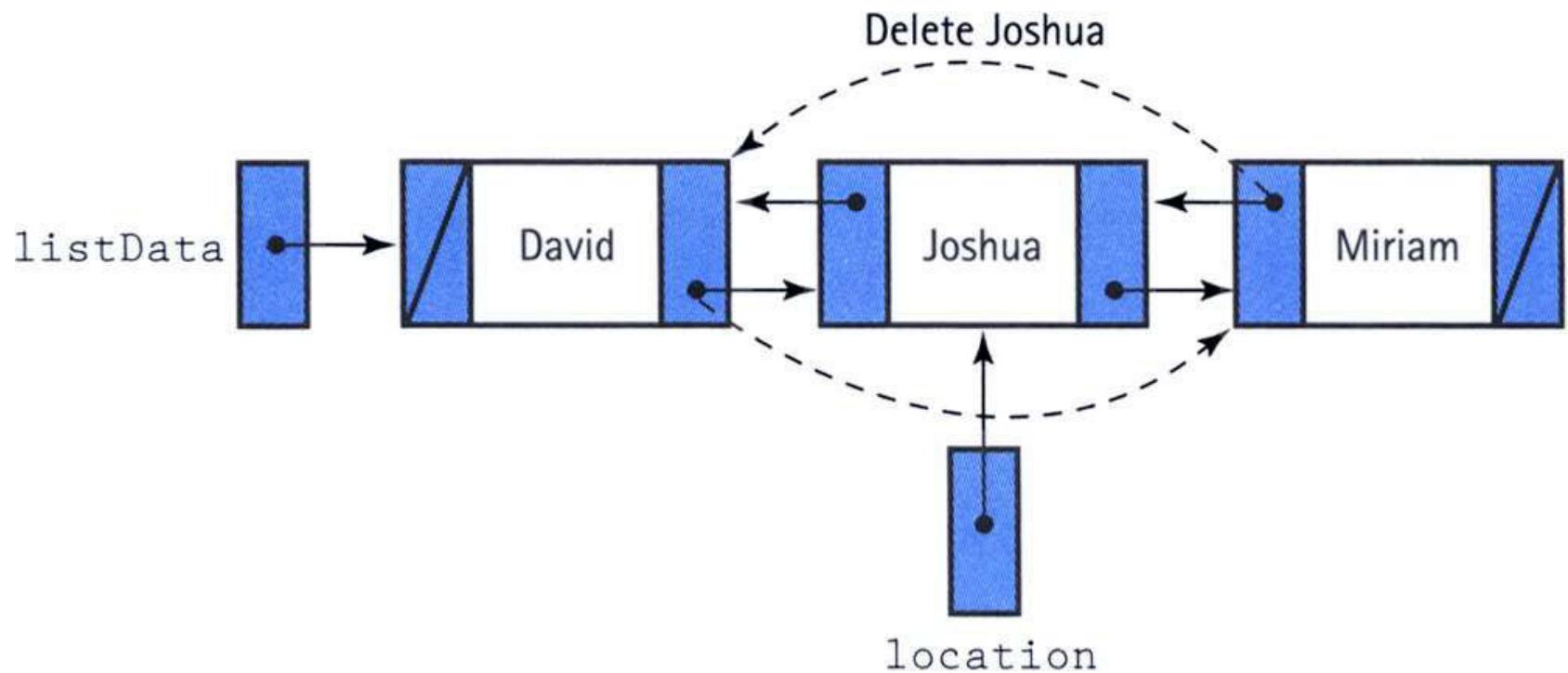
The Order is Important



Doubly Linked List - Deletion

- One useful feature of a doubly linked list is its elimination of the need for a pointer to a node's predecessor to delete the node.
- Through the back member, we can alter the next member of the preceding node to make it jump over the unwanted node.
- Then we make the back pointer of the succeeding node point to the preceding node.

Doubly Linked List - Deletion



Special Cases of Deletion

- We do, however, have to be careful about the end cases:
 - If location->back is NULL, we are deleting the first node
 - if location->next is NULL, we are deleting the last node.
 - If both location->back and location->next are NULL, we are deleting the only node.

DSA - Introduction

Topics

- Introduction
- Definitions
- Classification of Data Structures
- Arrays and Linked Lists
- Abstract Data Types [ADT]
 - The List ADT
 - Array-based Implementation
 - Linked List Implementation
 - Cursor-based Implementation

Data Structure [Wikipedia]

- Data Structure is a particular **way of storing and organizing data** in a computer so that it can be used efficiently.
- Different kinds of data structures are suited to different kinds of applications.
- **Storing and retrieving** can be carried out on data stored in both **main memory and in secondary memory**.

Merriam-Webster's Definition

- Way in which data are stored for efficient search and retrieval.
- The simplest data structure is the one-dimensional (linear) array.
- Data items stored non-consecutively in memory may be linked by pointers.
- Many algorithms have been developed for storing data efficiently

Algorithms [Wikipedia]

- An algorithm is a step-by-step procedure for calculations.
- An algorithm is an effective method expressed as a finite list of well-defined instructions for calculating a function.
- The transition from **one state to the next is not necessarily deterministic**; some algorithms incorporate random input.

Merriam-Webster's Definition

- **Procedure** that produces the answer to a question or the solution to a problem in a finite number of steps.
- An algorithm that produces a yes or no answer is called a **decision procedure**; one that leads to a solution is a **computation procedure**.
- Example: A mathematical formula and the instructions in a computer program

Data Structure Classification

- **Primitive / Non-primitive**
 - Basic Data Structures available(defined by programming languages like int , float etc...) / Derived from Primitive Data Structures(Array, structure, union, link list, stacks, queue etc...)
- **Homogeneous / Heterogeneous**
 - Elements are of the same type / Different types
- **Static / Dynamic**
 - memory is allocated at the time of compilation / run-time
- **Linear / Non-linear**
 - Maintain a Linear relationship between element

ADT - General Concept

- Problem solving with a computer means processing data
- To process data, we need to define the data type and the operation to be performed on the data
- The definition of the data type and the definition of the operation to be applied to the data is part of the idea behind an Abstract Data Type (ADT)

ADT(Abstract Data type) - General Concept

- The user of an ADT needs only to know that a set of operations are available for the data type, but does not need to know how they are applied
- Several simple ADTs, such as integer, real, character, pointer and so on, have been implemented and are available for use in most languages

Data Types

- A data type is characterized by:
 - A set of *values*
 - A *data representation*, which is common to all these values, and
 - A set of *operations*, which can be applied uniformly to all these values

Primitive Data Types

- *Languages like ‘C’ provides the following primitive data types:*
 - boolean
 - char, byte, int
 - float, double
- Each primitive type has:
 - A set of values
 - A data representation
 - A set of operations
- These are “set in stone”.

ADT Definition [Wikipedia]

- In computer science, an abstract data type (ADT) is a mathematical model for a certain class of data structures that have similar behavior.
- An abstract data type is defined indirectly, only by the operations that may be performed on it and by mathematical constraints on the effects (and possibly cost) of those operations.

ADT Definition [Wikipedia]

- An ADT may be implemented by specific data types or data structures, in many ways and in many programming languages; or described in a formal specification language.
- **example, an abstract stack could be defined by three operations:**
 - push, that inserts some data item onto the structure,
 - pop, that extracts an item from it, and
 - peek, that allows data on top of the structure to be examined without removal.

Definition from techforum4you

- Abstract data types or ADTs are a mathematical specification of a set of data and the set of operations that can be performed on the data.
- They are abstract in the sense that ***the focus is on the definitions and the various operations with their arguments.***
- The actual implementation is not defined, and does not affect the use of the ADT.

Why ADT?

- Modularity
 - divide program into small functions
 - easy to debug and maintain
 - easy to modify
 - group work
- Reuse
 - do some operations only once
- Easy to change the implementation
 - transparent to the program

The List ADT

- The List is an
 - Ordered sequence of data items called elements
 - $A_1, A_2, A_3, \dots, A_N$ is a list of size N
 - size of an empty list is 0
 - A_{i+1} succeeds A_i
 - A_{i-1} precedes A_i
 - Position of A_i is i
 - First element is A_1 called “head”
 - Last element is A_N called “tail”

Operations on Lists

- MakeEmpty
- PrintList
- Find
- Find k^{th}
- Insert
- Delete
- Next
- Previous

List – An Example

- The elements of a list are 34, 12, 52, 16, 12
 - Find (52) -> 3
 - Insert (20, 4) -> 34, 12, 52, 20, 16, 12
 - Delete (52) -> 34, 12, 20, 16, 12
 - FindKth (3) -> 20

List - Implementation

- Lists can be implemented using:
 - Arrays
 - Linked List
 - Cursor [Linked List using Arrays]

Arrays

- Array is a static data structure that represents a collection of fixed number of homogeneous data items or
- A fixed-size indexed sequence of elements, all of the same type.
- The individual elements are typically stored in consecutive memory locations.
- The length of the array is determined when the array is created, and cannot be changed.

Arrays

- Any component of the array can be inspected or updated by using its index.
 - This is an efficient operation
 - $O(1)$ = constant time
- The array indices may be integers (C, Java) or other discrete data types (Pascal, Ada).
- The lower bound may be zero (C, Java), one (Fortran), or chosen by the programmer (Pascal, Ada)

Different Types of Arrays

- **One-dimensional array:** only one index is used
- **Multi-dimensional array:** array involving more than one index
- **Static array:** the compiler determines how memory will be allocated for the array
- **Dynamic array:** memory allocation takes place during execution

One Dimensional Static Array

- Syntax:
 - `ElementType arrayName [CAPACITY];`
 - `ElementType arrayName [CAPACITY] = { initializer_list };`
- Example in C++:
 - `int b [5];`
 - `int b [5] = {19, 68, 12, 45, 72};`

Array Output Function

```
void display(int array[],int num_values)
{
    for (int i = 0; i<num_values; i++)
        printf( "%d", &array[i] );
}
```

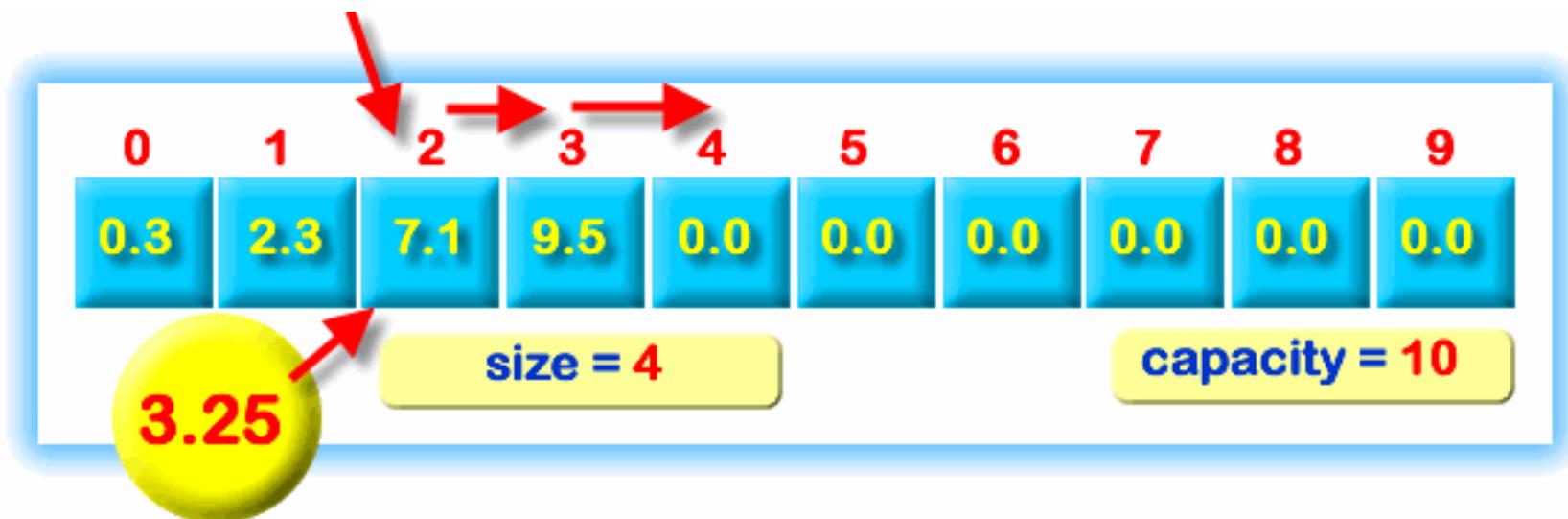
List Implemented Using Array



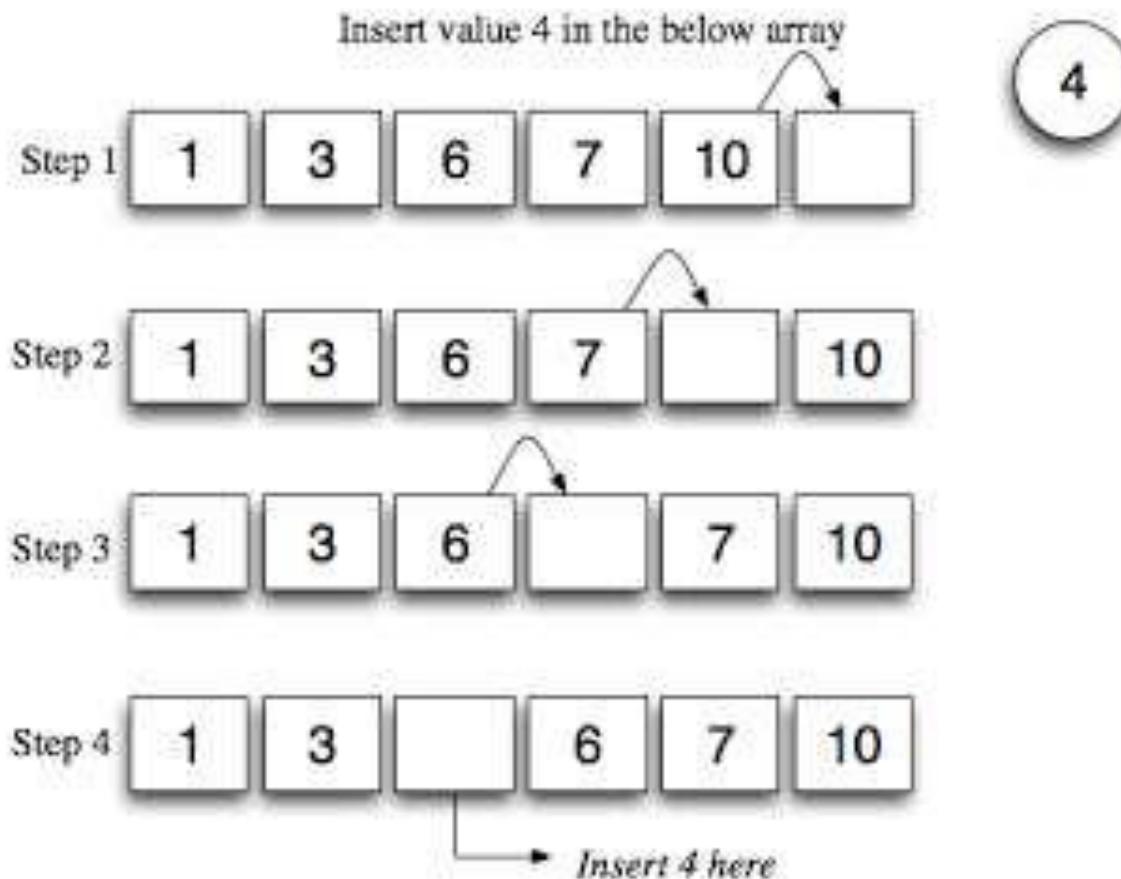
Operations On Lists

- We'll consider only few operations and not all operations on Lists
- Let us consider Insert
- There are two possibilities:
 - Ordered List
 - Unordered List

Insertion into an Ordered List



Insertion in Detail



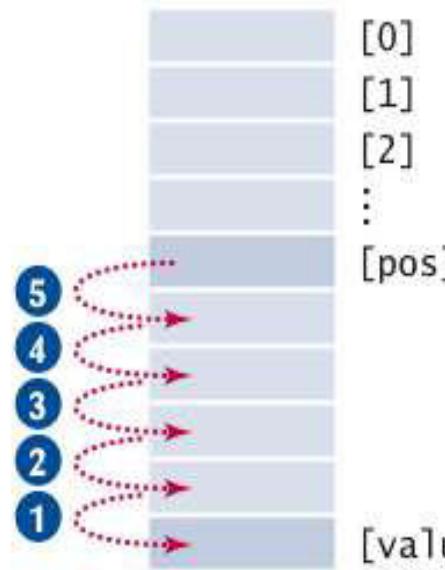
Insertion

[0]
[1]
[2]
⋮

Insert new element here

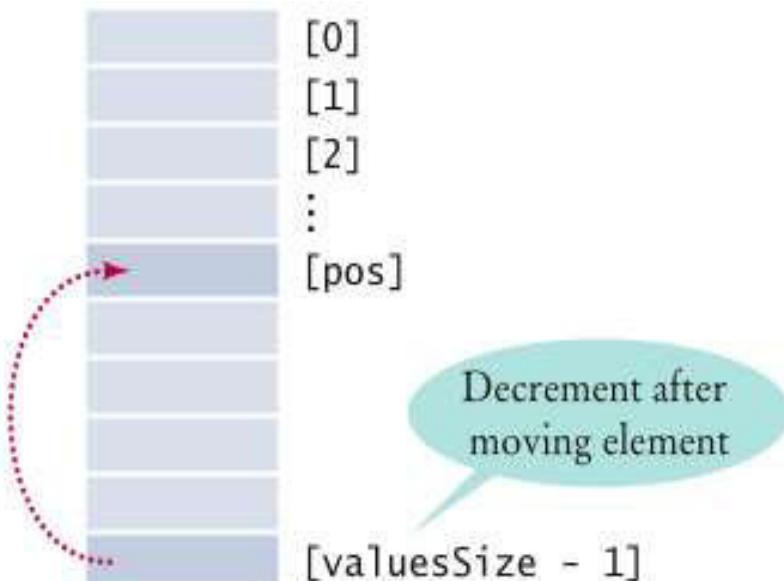
[valuesSize]

Inserting an Element in an Unordered Array

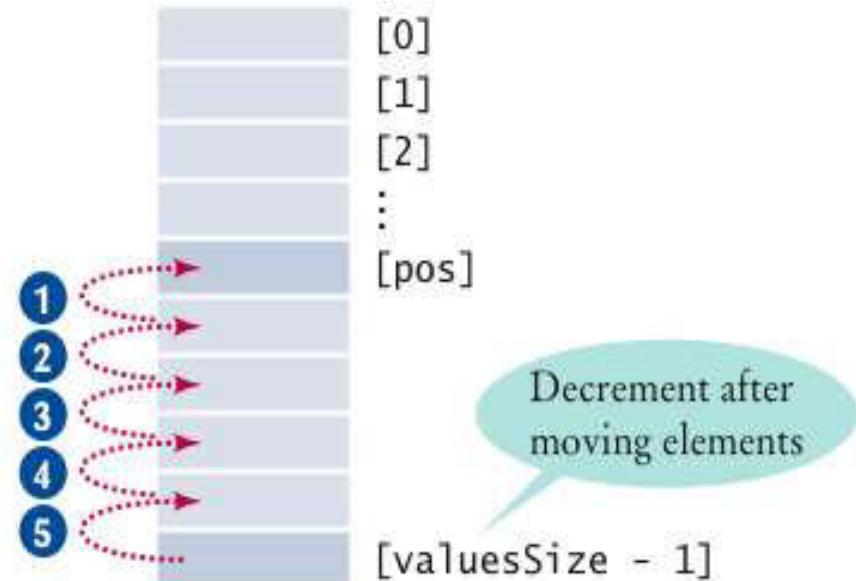


Inserting an Element in an Ordered Array

Deletion



Removing an Element in an Unordered Array



Removing an Element in an Ordered Array

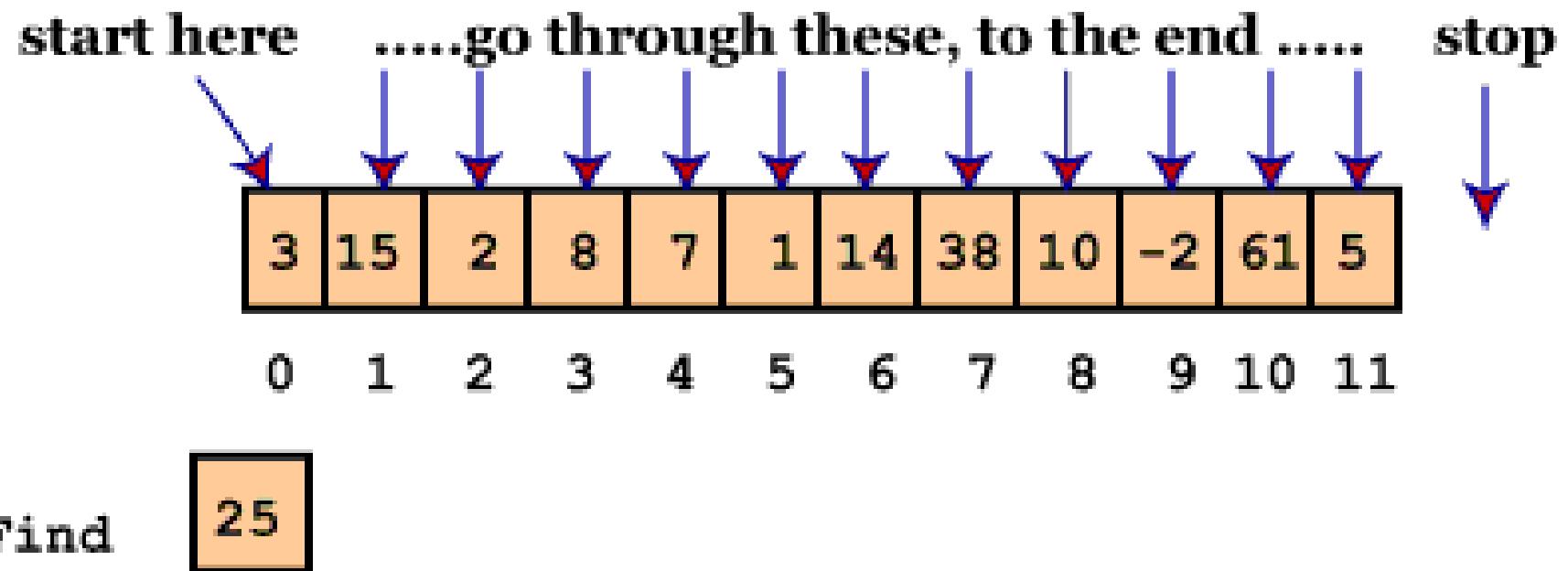
Find / Search

- Searching is the process of looking for a specific element in an array
- For example, discovering whether a certain score is included in a list of scores.
- Searching, like sorting, is a common task in computer programming.
- There are many algorithms and data structures devoted to searching.
- The most common one is the linear search.

Linear Search

- The linear search approach compares the given value with each element in the array.
- The method continues to do so until the given value matches an element in the list or the list is exhausted without a match being found.
- If a match is made, the linear search returns the index of the element in the array that matches the key.

Linear Search



Linear Search Function

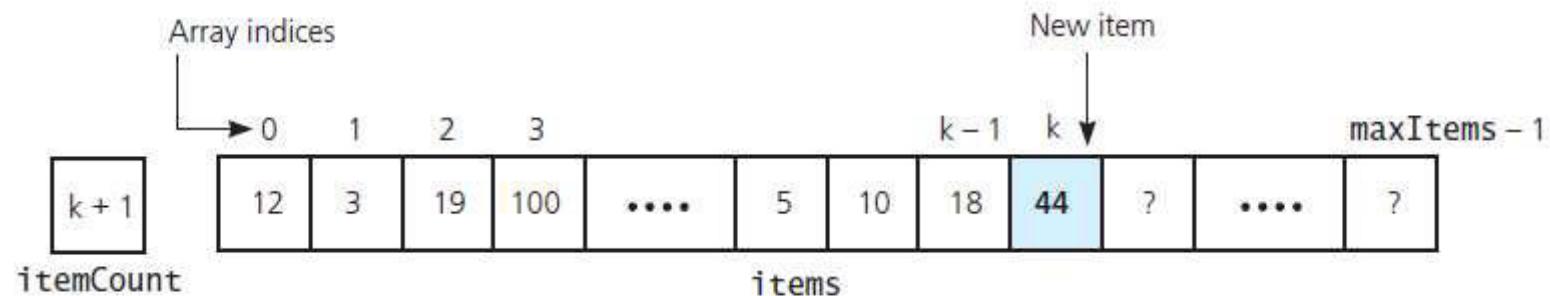
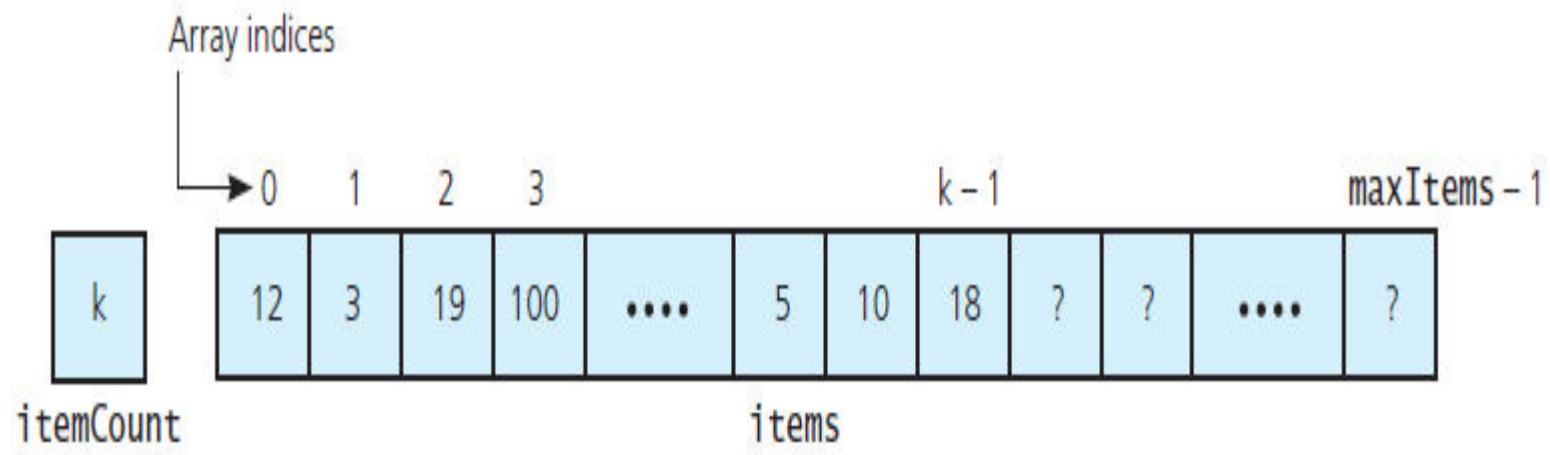
```
int LinearSearch (int a[], int n, int key)
{
    int i;
    for(i=0; i<n; i++)
    {
        if (a[i] == key)
            return i;
    }
    return -1;
}
```

Using the Function

- **LinearSearch (a, n, item, loc)**
- Here "a" is an array of the size n.
- This algorithm finds the location of the element "item" in the array "a".
- If search item is found, it sets loc to the index of the element; otherwise, it sets loc to -1
- **index=linearsearch(array, num, key)**

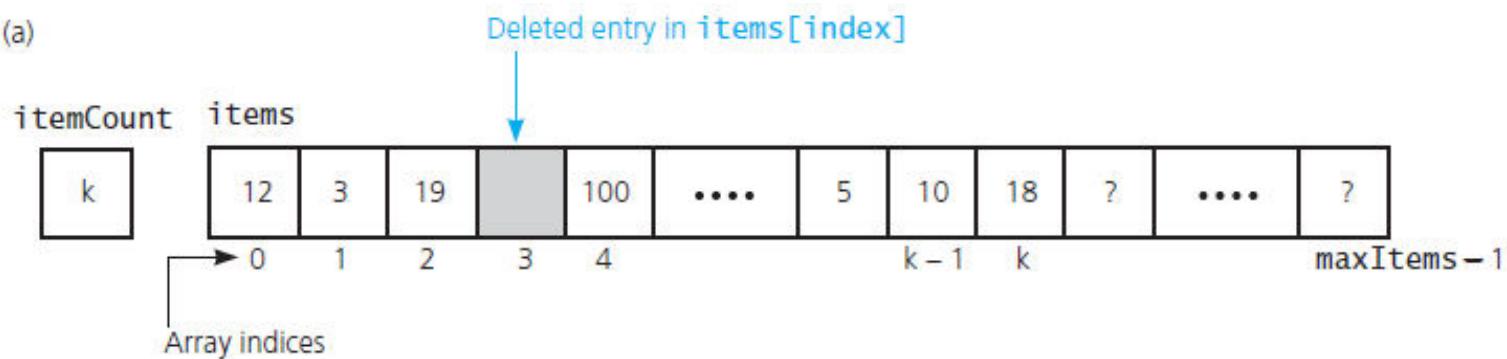
PrintList Operation

```
int myArray [5] = {19, 68, 12, 45, 72};  
/* To print all the elements of the array  
for (int i=0;i<5;i++)  
{  
printf("%d", myArray[i]);  
}
```

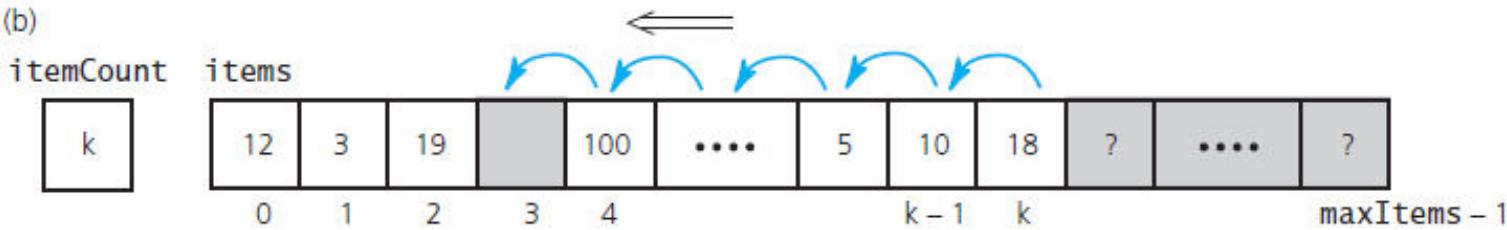


Implementing Deletion

(a)

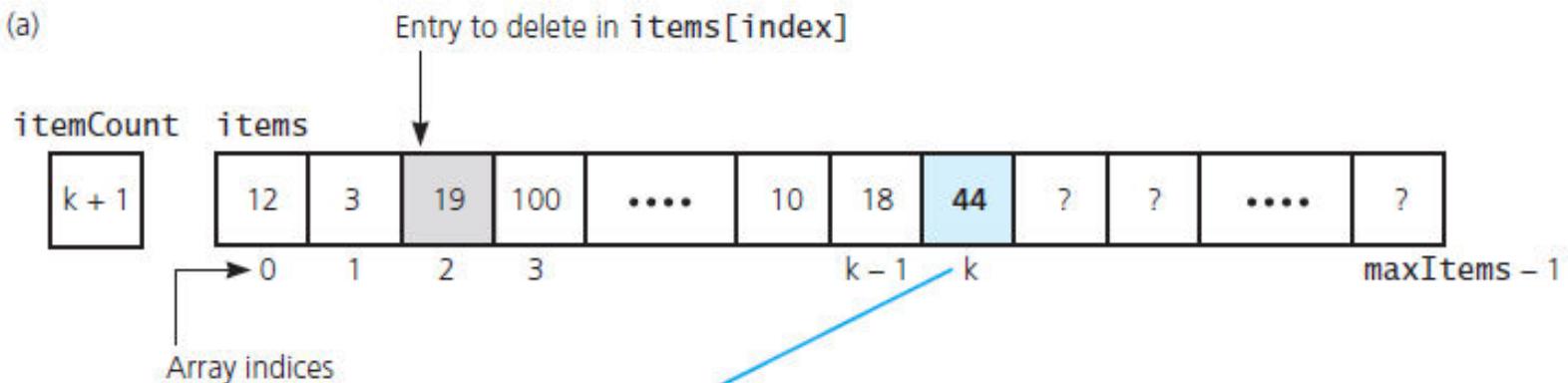


(b)

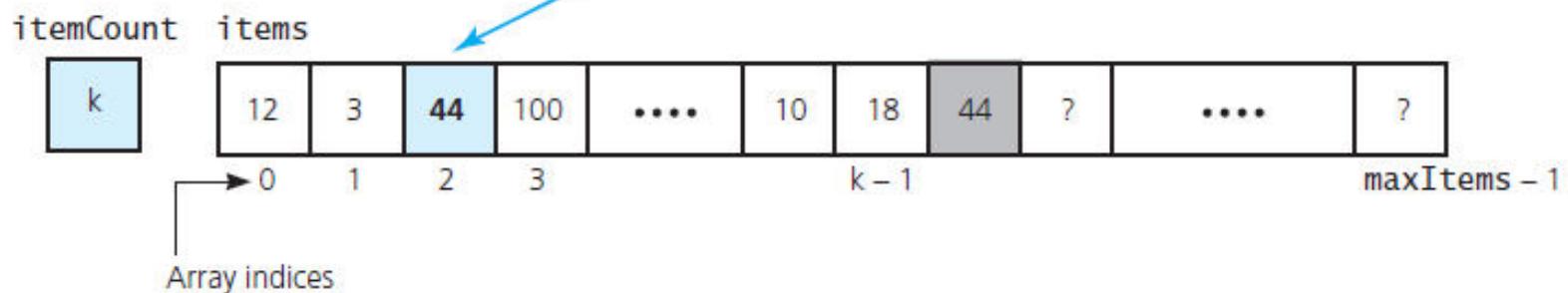


Deletion - Another Method

(a)



(b)



Operations Running Times

PrintList } O(N)
Find

Insert } O(N) (on average half
Delete) needs to be moved)

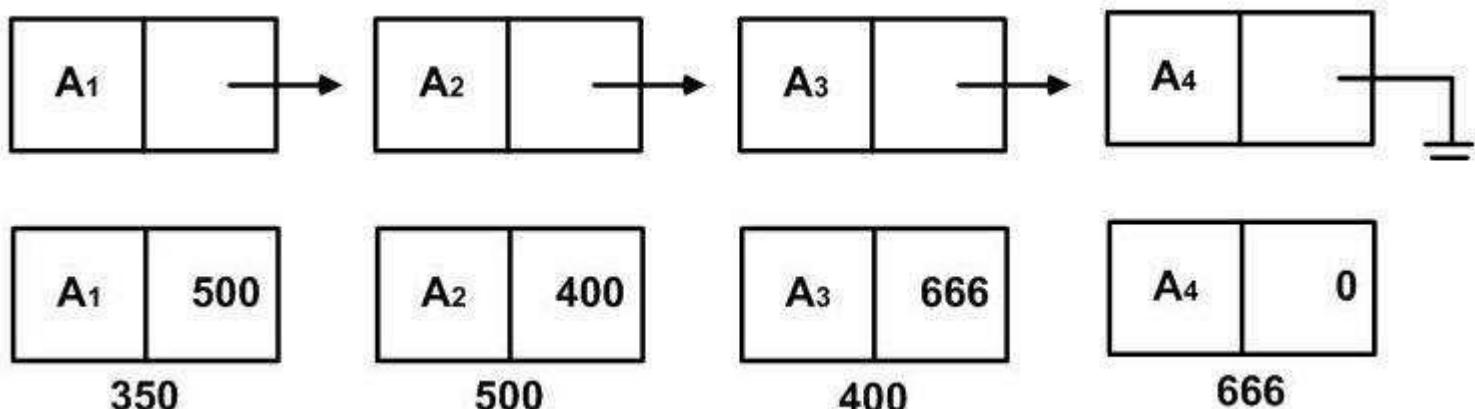
FindKth } O(1)
Next }
Previous

Disadvantages of Using Arrays

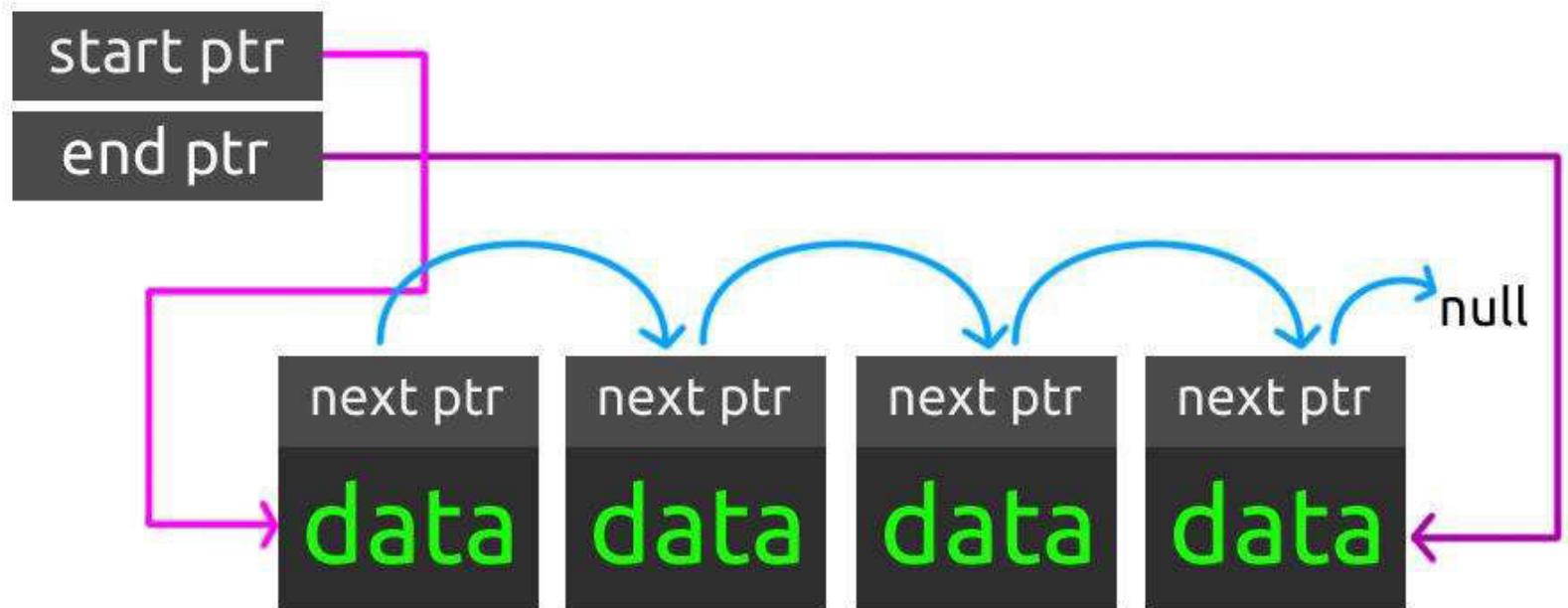
- Need to define a size for array
 - High overestimate (waste of space)
- insertion and deletion is very slow
 - need to move elements of the list
- redundant memory space
 - it is difficult to estimate the size of array

Linked List

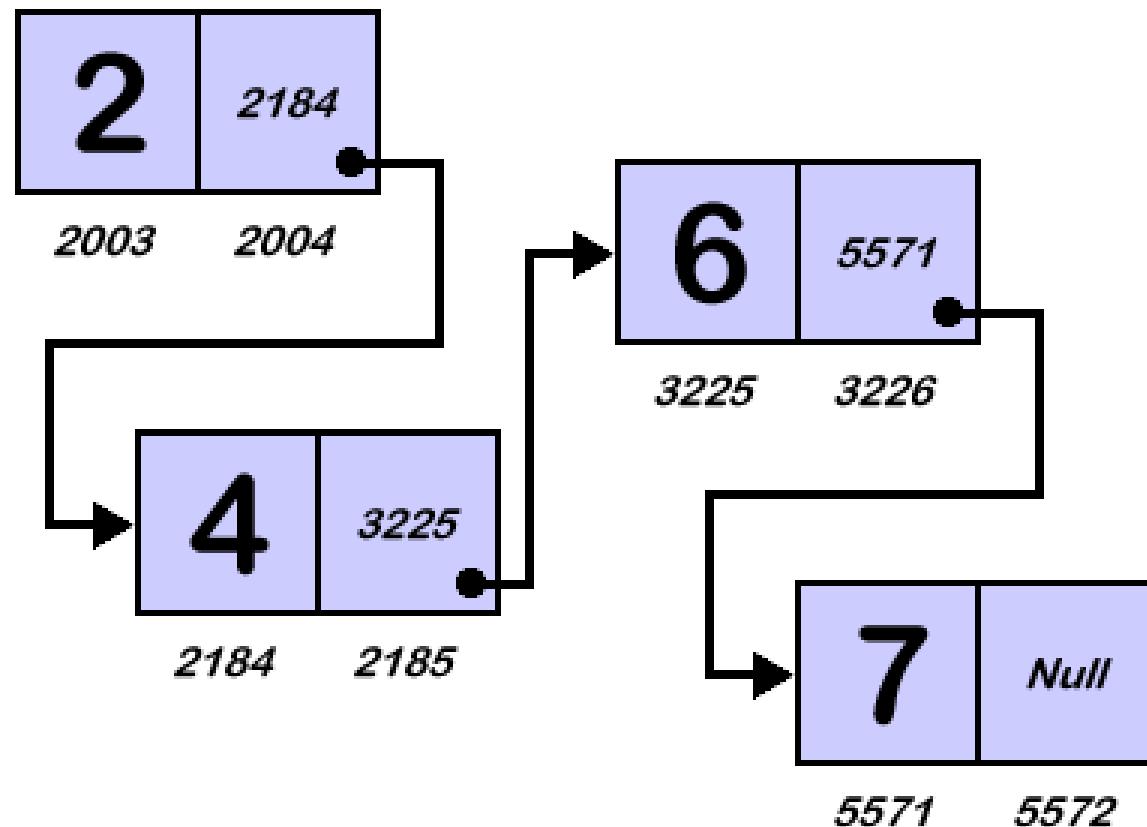
- Series of nodes
 - not adjacent in memory
 - contain the element and a pointer to a node containing its successor
- Avoids the linear cost of insertion and deletion!



Singly Linked List



Singly Linked List

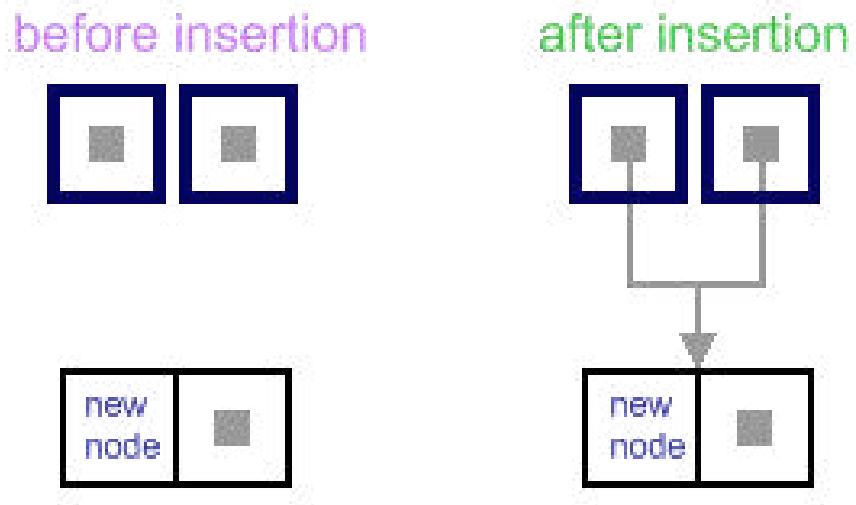


Singly-linked List - Addition

- Insertion into a singly-linked list has two special cases.
- It's insertion a new node before the head (to the very beginning of the list) and after the tail (to the very end of the list).
- In any other case, new node is inserted in the middle of the list and so, has a predecessor and successor in the list.

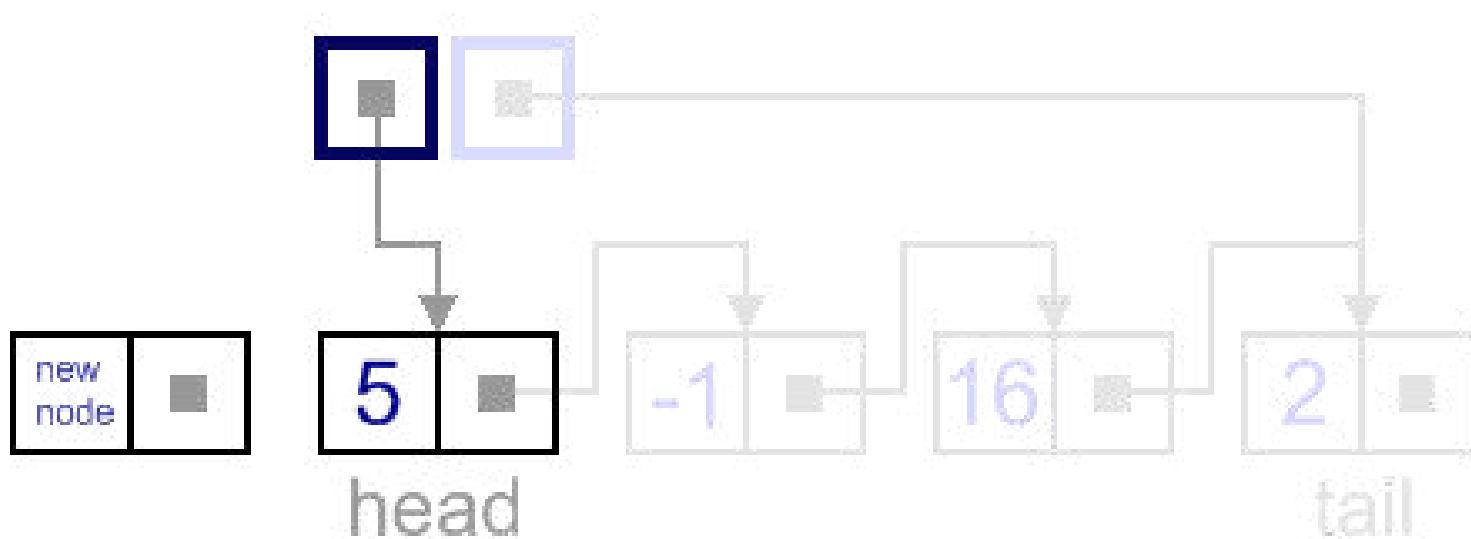
Empty list case

- When list is empty, which is indicated by (`head == NULL`) condition, the insertion is quite simple.
- Algorithm sets both head and tail to point to the new node.



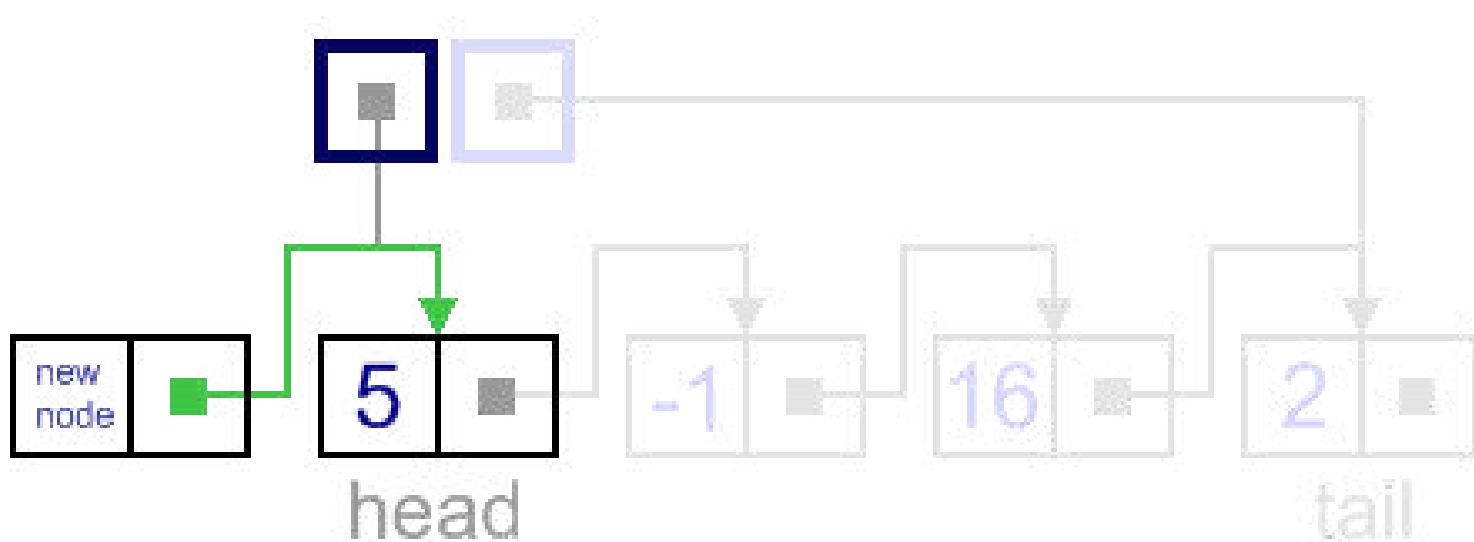
Add first

- In this case, new node is inserted right before the current head node.



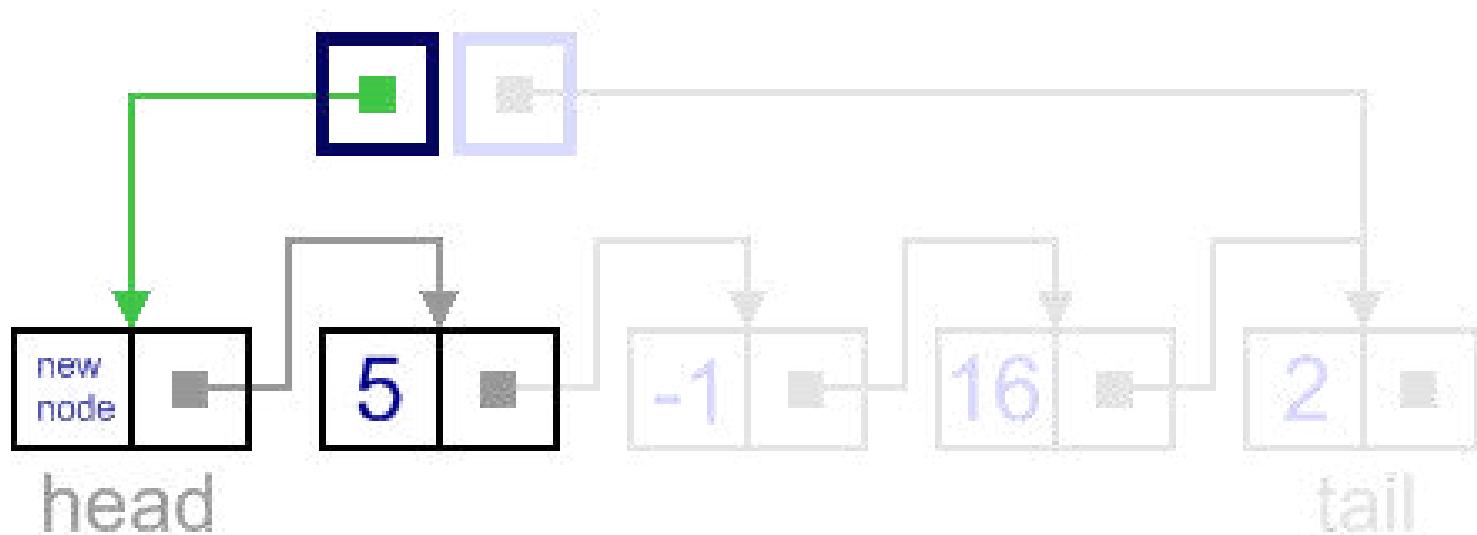
Add First - Step 1

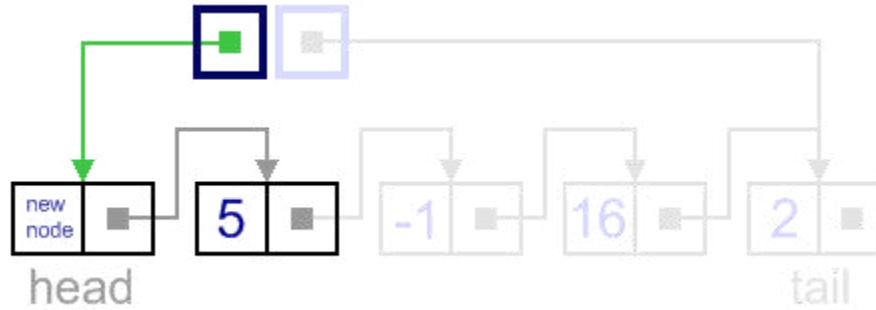
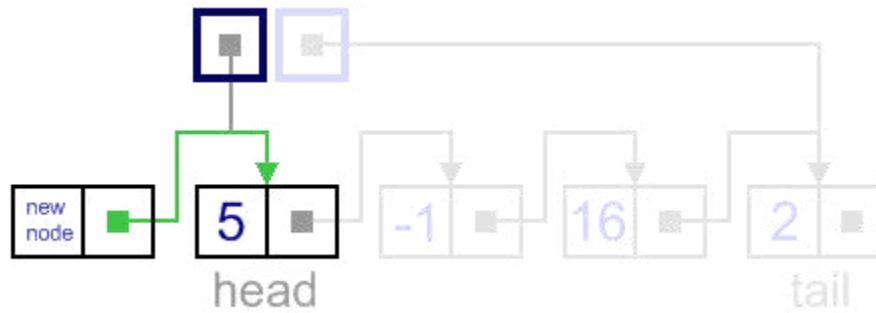
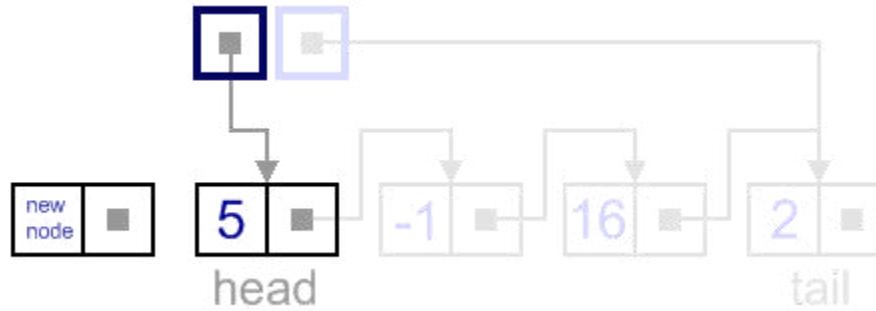
- It can be done in two steps:
 - Update the next link of the new node, to point to the current head node.



Add First - Step 2

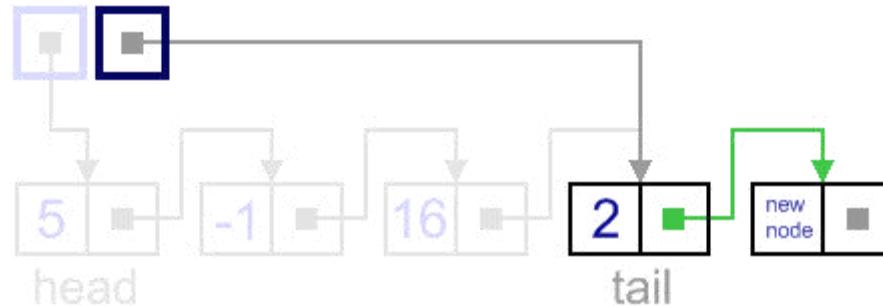
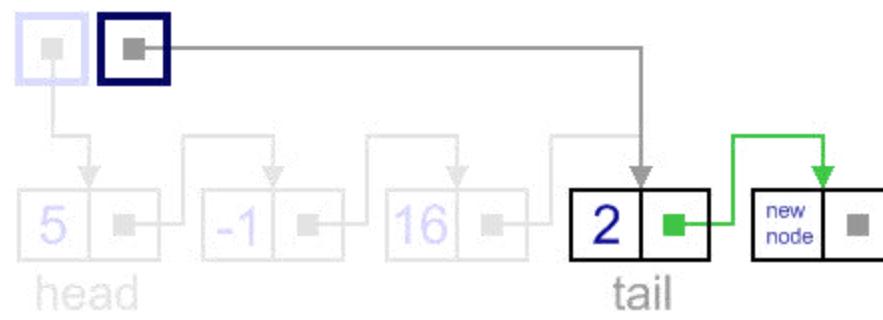
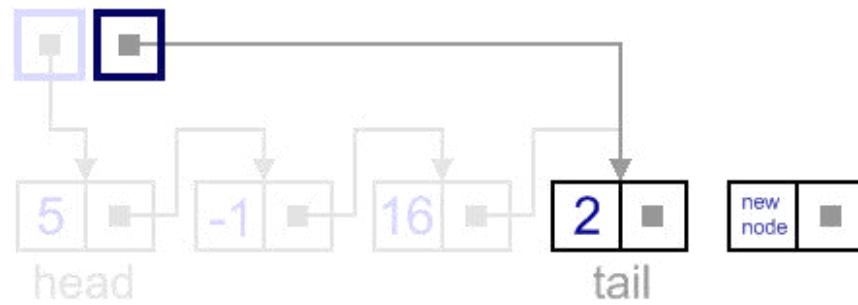
- Update head link to point to the new node.





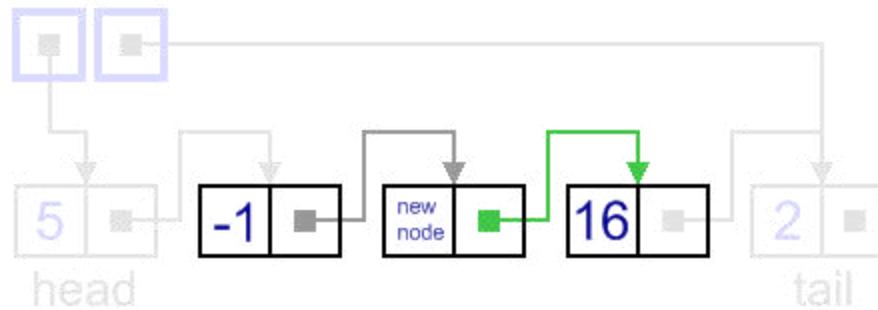
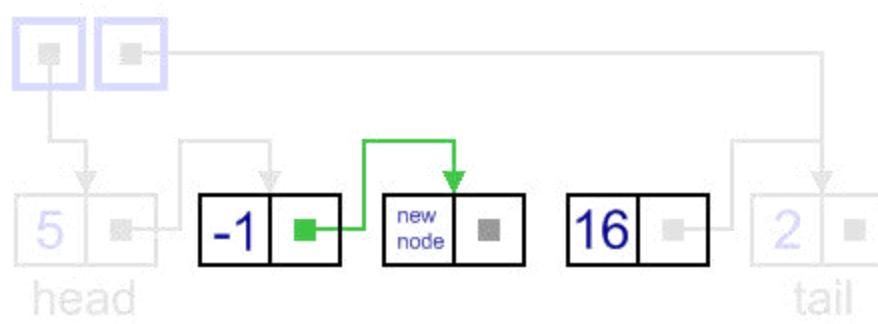
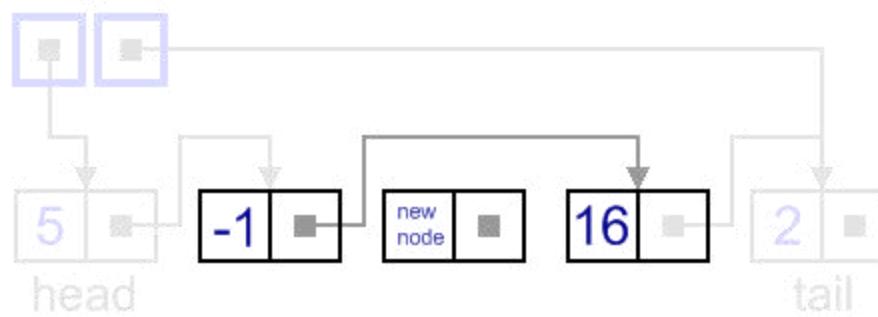
Add last

- In this case, new node is inserted right after the current tail node.
- It can be done in two steps:
 - Update the next link of the current tail node, to point to the new node.
 - Update tail link to point to the new node.



Insert - General Case

- In general case, new node is always inserted between two nodes, which are already in the list. Head and tail links are not updated in this case.
- We need to know two nodes "Previous" and "Next", between which we want to insert the new node.
- This also can be done in two steps:
 - Update link of the "previous" node, to point to the new node.
 - Update link of the new node, to point to the "next" node.

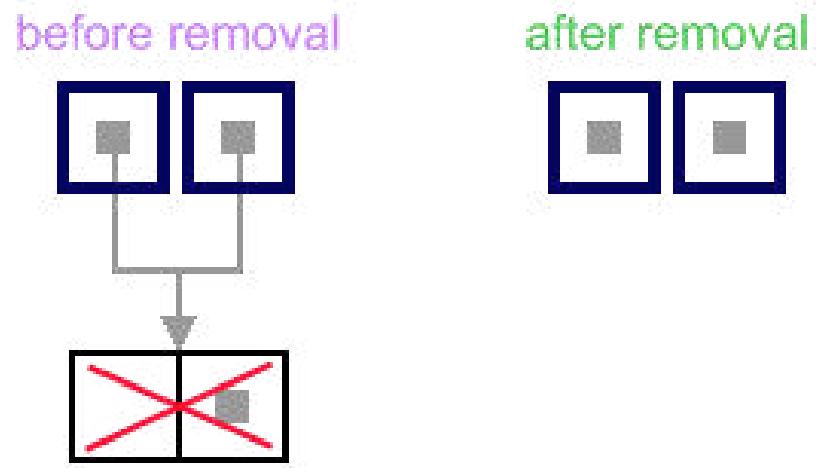


Singly-linked List - Deletion

- There are four cases, which can occur while removing the node.
- We have the same four situations, but the order of algorithm actions is opposite.
- Notice, that removal algorithm includes the disposal of the deleted node - unnecessary in languages with automatic garbage collection (Java).

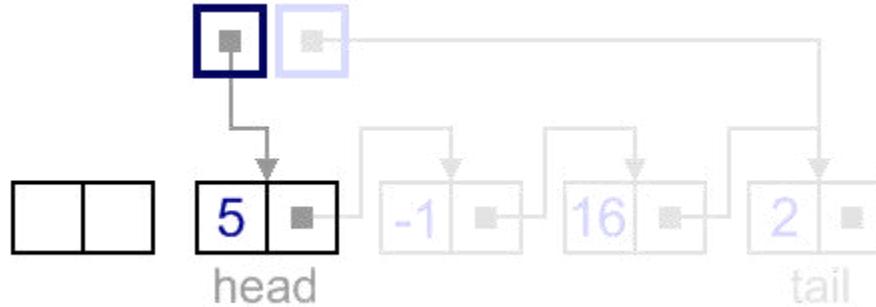
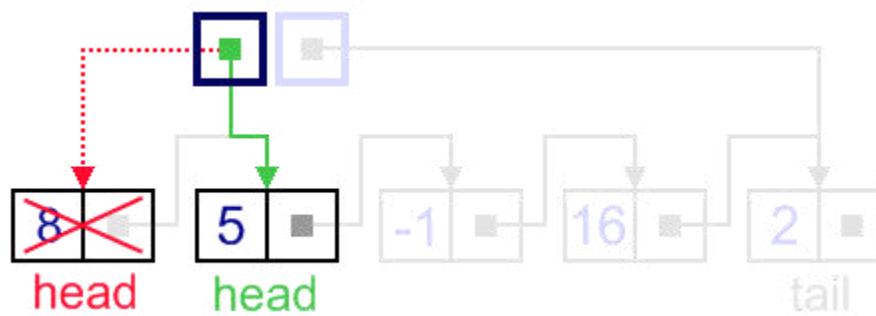
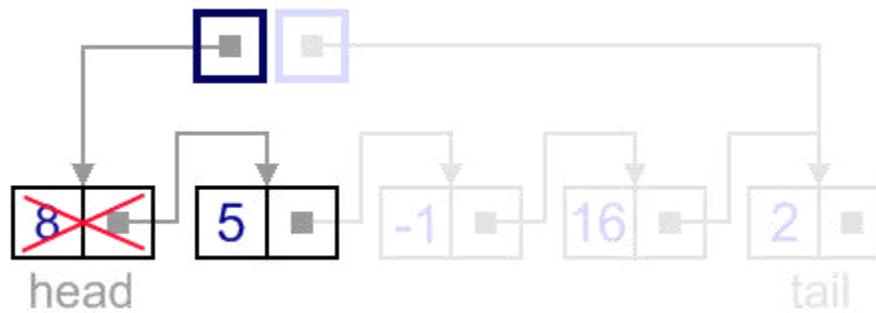
List has only one node

- When list has only one node, that the head points to the same node as the tail, the removal is quite simple.
- Algorithm disposes the node, pointed by head (or tail) and sets both head and tail to NULL.



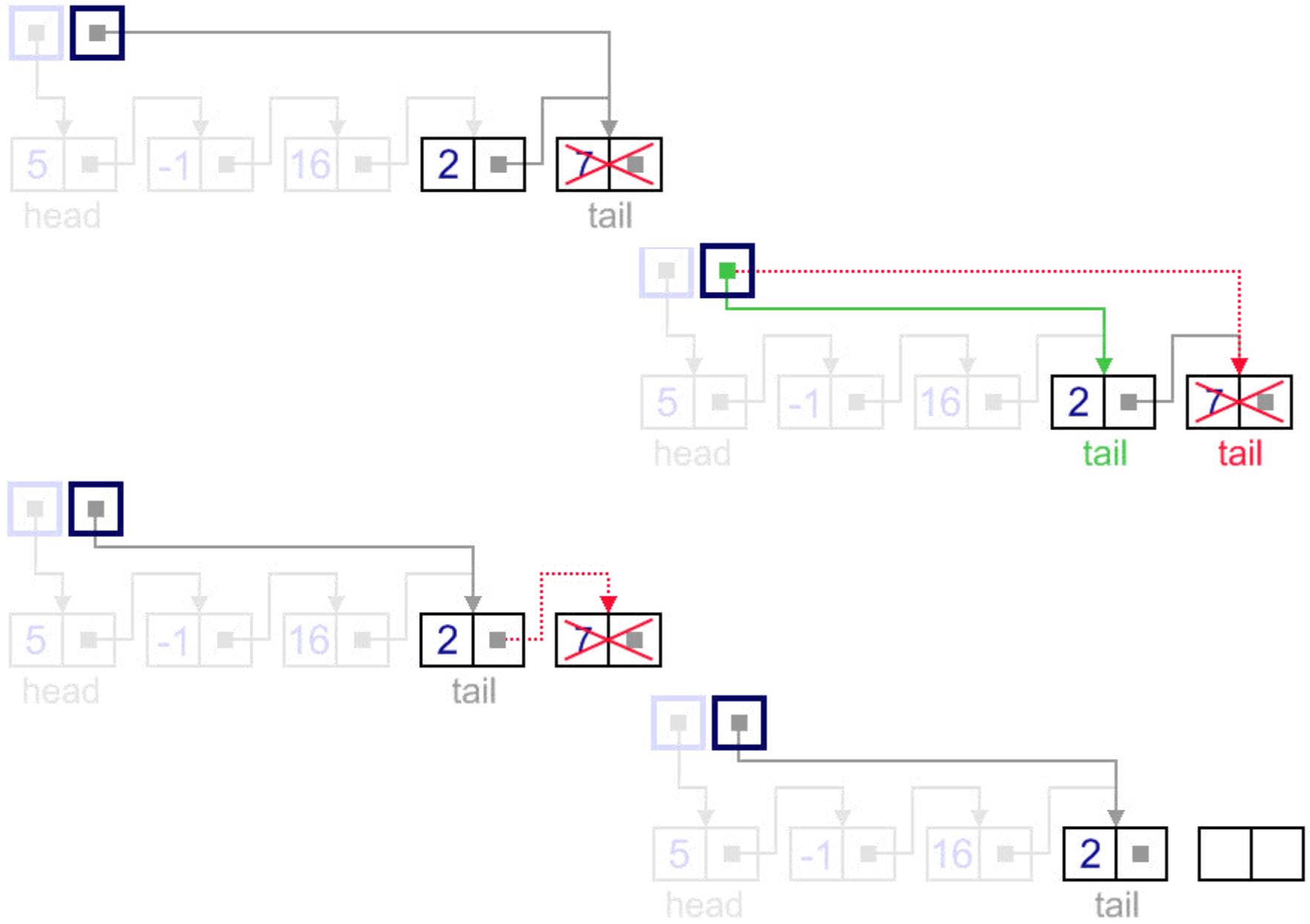
Remove First

- In this case, first node (current head node) is removed from the list.
- It can be done in two steps:
 - Update head link to point to the node, next to the head.
 - Dispose removed node.



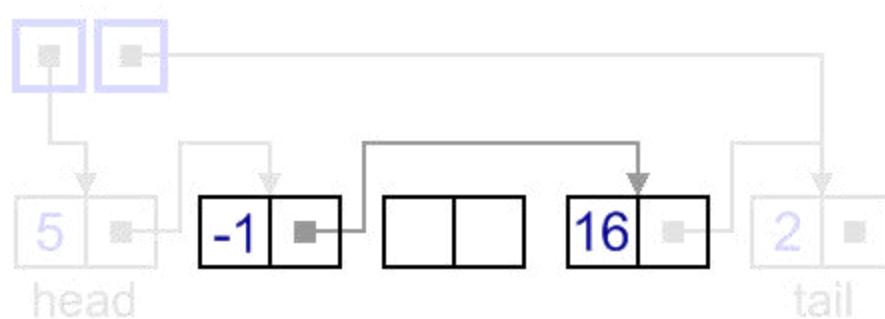
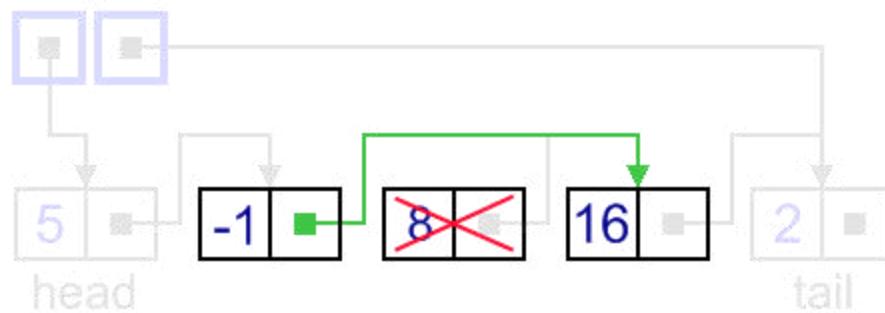
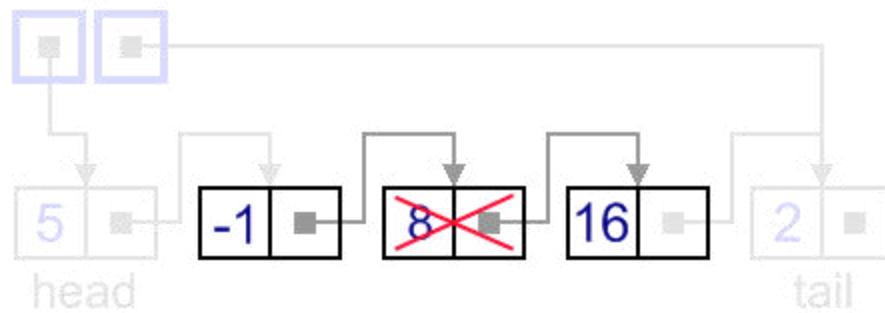
Remove Last

- In this case, last node (current tail node) is removed from the list. This operation is a bit more tricky, than removing the first node, because algorithm should find a node, which is previous to the tail first.
- It can be done in three steps:
 - Update tail link to point to the node, before the tail.
In order to find it, list should be traversed first, beginning from the head.
 - Set next link of the new tail to NULL.
 - Dispose removed node.



Remove - General Case

- In general case, node to be removed is always located between two list nodes. Head and tail links are not updated in this case.
- We need to know two nodes "Previous" and "Next", of the node which we want to delete.
- Such a removal can be done in two steps:
 - Update next link of the previous node, to point to the next node, relative to the removed node.
 - Dispose removed node.



Advantages of Using Linked Lists

- Need to know where the first node is
 - the rest of the nodes can be accessed
- No need to move the elements in the list for insertion and deletion operations
- No memory waste

Arrays - Pros and Cons

- Pros
 - Directly supported by C
 - Provides random access
- Cons
 - Size determined at compile time
 - Inserting and deleting elements is time consuming

Linked Lists - Pros and Cons

- Pros
 - Size determined during runtime
 - Inserting and deleting elements is quick
- Cons
 - No random access
 - User must provide programming support

Application of Lists

Lists can be used

- To store the records sequentially
- For creation of stacks and queues
- For polynomial handling
- To maintain the sequence of operations for do / undo in software
- To keep track of the history of web sites visited

Proof of correctness

Module -1

Algorithm (topics)

- Design of Algorithms (for simple problems)
- Analysis of algorithms
 - – Is it correct?
 - Loop invariants**
 - *a loop invariant* is a property of a *program loop that is true before (and after) each iteration*
 - – Is it “good”?
 - Efficiency**
 - – Is there a better algorithm?
 - Lower bounds**

State of Computation

- Most programming algorithms are based on the notion of transforming the algorithm to outputs

Loop Invariant

- The state of computation may be defined by examining the contents of key variables before and after the execution of each statement.

Loop Invariant : $x + y = x_0 + y_0$.

We want to prove that $x + y = x_0 + y_0$ is a *loop invariant* of the program. In other words, we want to prove that $x + y = x_0 + y_0$ is always true no matter how many time the loop executed. Consider the following predicate.

$L(n)$: If the program reaches point a after the loop has been executed n times,
then $x + y = x_0 + y_0$.

Inductive Base: $L(0)$

If the loop has not been executed, then $x = x_0$ and $y = y_0$. Therefore, $x + y = x_0 + y_0$.
 $L(0) = \text{true}$. The inductive base holds.

Inductive Hypothesis: Suppose $L(n) = \text{true}$.

Inductive Step: $L(n + 1)$

Suppose at the moment t_0 , the loop has been executed n times. Since $L(n)$ is true, $x + y = x_0 + y_0$. And, Suppose at this moment t_0 , $x = k$ and $y = l$, i.e, at the moment t_0 we have $x + y = x_0 + y_0 = k + l$.

If the loop will be executed one more time, then x will be decreased by 1 and y will be increased by 1, and then the program will reach point a at the moment t_1 . At this moment t_1 , $x + y = (k - 1) + (l + 1) = k + l = x_0 + y_0$. Therefore, $L(n + 1) = \text{true}$.

Assertions

- Assertions are facts about the state of the program variables
- It is wasteful to spend your time looking at variables that are not effected by a particular statement
- Default assertion
 - any variable not mentioned in the assertion for a statement do not affect the state of computation

Use of Assertions

- Pre-condition
 - assertion describing the state of computation before statement is executed
- Post condition
 - assertion describing the state of computation after a statement is executed
- Careful use of assertions as program comments can help control side effects

1.Simple Algorithm- Sum of two integers , Swapping of Two Numbers

- **Model {P} A {Q}**
 - P = pre-condition
 - A = Algorithm
 - Q = post condition
- **Sum of two numbers algorithm**
{pre: $x = x_0$ and $y = y_0$ }
$$z = x + y$$

{post: $z = x_0 + y_0$ }

Sequence Algorithm

- **Model**

if $\{P\} A_1 \{Q_1\}$ and $\{Q_1\} A_2 \{Q\}$

then $\{P\} A_1 ; A_2 \{Q\}$ is true

- **Swap algorithm**

{pre: $x = x_0$ and $y = y_0$ }

temp = x

$x = y$

$y = \text{temp}$

{post: $\text{temp} = x_0$ and $x = y_0$ and $y = x_0$ }

Intermediate Assertions

- Swap algorithm

{pre: $x = x_0$ and $y = y_0$ }

temp = x

{temp = x_0 and $x = x_0$ and $y = y_0$ }

x = y

{temp = x_0 and $x = y_0$ and $y = y_0$ }

y = temp

{post: temp = x_0 and $x = y_0$ and $y = x_0$ }

2. Conditional Statements- Absolute value

- **Absolute value**

{pre: $x = x_0$ }

if $x < 0$ then

$y = -x_0$

else

$y = x_0$

{post: $y = |x_0|$ }

Intermediate Assertions

if $x < 0$ **then**

{ $x = x_0$ and $x_0 < 0$ }

$y = -x_0$

{ $y = |x_0|$ }

else

{ $x = x_0$ and $x_0 \geq 0$ }

$y = x_0$

{ $y = |x_0|$ }

3. Proving the correctness of Algorithm- Sequential Search

- **Sequential Search**
- 1. $\text{index} = 1;$
- 2. While $\text{index} \leq n$ and $L[\text{index}] \neq x$ do
 - $\text{index} = \text{index} + 1;$
- 3. if $\text{index} > n$ then $\text{index} = 0;$

Defining the I/O of the algorithm

- Input : Given an array L containing n items ($n \geq 0$) and given x,
- Output:
 - the sequential search algorithm terminates
 - with index = first occurrence of x in L, if found
 - and, index = 0 otherwise.

Loop Invariant

- **Sequential Search**
 1. $\text{index} = 1;$
 2. While $\text{index} \leq n$ and $L[\text{index}] \neq x$ do
 $\text{index} = \text{index} + 1;$
 3. if $\text{index} > n$ then $\text{index} = 0;$
- **Hypothesis:** For $1 \leq k \leq n + 1$, $L(k)$: when the control reaches the test in line 2 for **kth** time
 - $\text{index} = k$ and,
 - for $1 \leq i \leq k-1$, $L[i] \neq x$.
- Prove the above hypothesis by induction.

Proving the Hypothesis by induction

- Base Case : $H[1]$ is true vacuously.

Let $H(k)$ be true

- We will prove that $H(k+1)$ is also true.

Control reaches the test in line 2 for $(k+1)$ th time only

if $L(k) \neq x \dots \dots \dots \text{(i)}$

Also since $H(k)$ is true

$1 \leq i \leq k-1, L[i] \neq x \dots \dots \dots \text{(ii)}$

from (i) and (ii) we get,

$1 \leq i \leq k, L[i] \neq x$

\therefore it holds for index = $k+1$

Thus by induction our hypothesis is true.

Correctness contd..

- Suppose the test condition is executed exactly k times. i.e. body of the loop is executed $k-1$ times.
 - Case 1: $k = n+1$, by loop invariant hypothesis, $\text{index} = n+1$ and for $1 \leq i \leq n$, $L[i] \neq x$.
Since $\text{index} = n+1$, line 3 sets index to 0 and **by second condition above x is not in the array**. So correct.
 - Case 2: $k \leq n$, $\Rightarrow \text{index} = k$ and loop terminated because $L[k] = x$. Thus index is the position of the first occurrence of x in the array.

Hence the algorithm is correct in either case.

4. How do you find the **max** of n numbers (stored in array A?)

- INPUT: A[1..n] - an array of integers
 - OUTPUT: an element m of A such that $A[j] \leq m$, $1 \leq j \leq \text{length}(A)$
-
- Find-max (A)
 1. $\text{max} \leftarrow A[1]$
 2. for $j \leftarrow 2$ to n
 3. do if ($\text{max} < A[j]$)
 4. $\text{max} \leftarrow A[j]$
 5. return max

Reasoning (formally) about algorithms

- 1. I/O specs: Needed for correctness proofs, performance analysis.
 - INPUT: $A[1..n]$ - an array of integers
 - OUTPUT: an element m of A such that $A[j] \leq m$, $1 \leq j \leq \text{length}(A)$
- 2. CORRECTNESS: The algorithm satisfies the output specs for EVERY valid input
- 3. ANALYSIS: Compute the running time, the space requirements, number of cache misses, disk accesses, network accesses,....

Correctness proofs of algorithms

Find-max (A)

```
1. max ← A[1]
2. for j ← 2 to length(A)
3.   do if (max < A[j])
4.     max ← A[j]
5. return max
```

- Prove that for any valid Input, the output of Find-max satisfies the output condition.
- **Proof by contradiction:**
 - Suppose the algorithm is incorrect.
 - Then for some input A,
 - **Case 1: max is not an element of A.** max is initialized to and assigned to elements of A – (a) is impossible.
 - **Case 2: ($\exists j \mid \text{max} < A[j]$).**
After the jth iteration of the for-loop (lines 2 – 4), $\text{max} \geq A[j]$. From lines 3,4, max only increases.
 - Therefore, upon termination, $\text{max} \geq A[j]$, which contradicts (b).

Correctness proofs using loop invariants

- while (condition), do (S)
- Loop invariant
 - An assertion that remains true each time S is executed.
 - p is a loop invariant if $(p \wedge \text{condition}) \{S\} p$ is true.
 - p is true before S is executed. q and l condition are true after termination.

$$(p \wedge \text{condition}) \{S\} p$$

$$\therefore p \{\text{while condition do } S\} (\text{lcondition} \wedge p)$$

Loop invariant proofs

Find-max (A)

1. $\text{max} \leftarrow A[1]$
2. $\text{for } j \leftarrow 2 \text{ to } \text{length}(A)$
3. $\text{do if } (\text{max} < A[j])$
4. $\text{max} \leftarrow A[j]$
5. return max

- Prove that for any valid Input, the output of Find-max satisfies the output condition.
- Proof by **loop invariants**:
 - Loop invariant: $I(j)$: At the beginning of iteration j of the loop, max contains the maximum of $A[1,..,j-1]$.
 - Proof:
 - True for $j=2$.
 - Assume that the loop invariant holds for the j iteration,
So at the beginning of iteration k , $\text{max} = \text{maximum of } A[1,..,j-1]$.

Loop invariant proofs

Find-max (A)

```
1. max ← A[1]
2. for j ← 2 to length(A)
3.   do if (max < A[j])
4.     max ← A[j]
5. return max
```

- For the $(j+1)$ th iteration
 - **Case 1:** $A[j]$ is the maximum of $A[1, \dots, j]$. In lines 3, 4, max is set to $A[j]$.
 - **Case 2:** $A[j]$ is not the maximum of $A[1, \dots, j]$. So the maximum of $A[1, \dots, j]$ is in $A[1, \dots, j-1]$. By our assumption max already has this value and by lines 3-4 max is unchanged in this iteration.

Loop invariant proofs

- STRATEGY: We proved that the invariant holds at the beginning of iteration j for each j used by Find-max.
 - Upon termination, $j = \text{length}(A)+1$. (WHY?)
 - The invariant holds, and so max contains the maximum of $A[1..n]$

Loop invariant proofs

- ▶ Advantages:
 - Rather than reason about the whole algorithm,
reason about SINGLE iterations of SINGLE loops.
- ▶ Structured proof technique
- ▶ Usually prove loop invariant via Mathematical Induction.

Algorithm Correctness Example

- Suppose you have a software component that accepts as input an array **T** of size **N**
- As output the component produces an **T'** which **contains the elements of T arranged in ascending order using Bubble Sort.**
- **Binary Search**
- How would we convert the code to its logical counterpart and prove its correctness?

Implementation of Stack and Queue using Array

```
#include<stdio.h>
#include<process.h>
#include<stdlib.h>
#define MAX 5 //Maximum number of elements that can be stored
    int top=-1, stack[MAX];
    void push();
    void pop();
    void display();
void main()
{
    int ch;
    while(1) //infinite loop, will end when choice will be 4
    {
        printf("\n*** Stack Menu ***");
        printf("\n\n1.Push\n2.Pop\n3.Display\n4.Exit");
        printf("\n\nEnter your choice(1-4):");
        scanf("%d",&ch);
```

```
switch(ch)
{
    case 1: push();
        break;
    case 2: pop();
        break;
    case 3: display();
        break;
    case 4: exit(0);

    default: printf("\nWrong Choice!!");

}
}
```

```
void push()
{
    int val;

    if (top==MAX-1)
    {
        printf("\nStack is full!!");
    }
    else
    {
        printf("\nEnter element to push:");
        scanf("%d",&val);
        top=top+1;
        stack[top]=val;
    }
}
```

```
void pop()
{
    if(top== -1)
    {
        printf("\nStack is empty!!");
    }
    else
    {
        printf("\nDeleted element is %d",stack[top]);
        top=top-1;
    }
}
```

```
void display()
{
    int i;

    if(top==-1)
    {
        printf("\nStack is empty!!");
    }
    else
    {
        printf("\nStack is... \n");
        for(i=top;i>=0;--i)
            printf("%d\n",stack[i]);
    }
}
```

- Output:

```
C:\Users\Admin\Desktop\DSA_WINTER2018_2019\Stack1.exe
*** Stack Menu ***
1.Push
2.Pop
3.Display
4.Exit

Enter your choice(1-4):1

Enter element to push:100

*** Stack Menu ***
1.Push
2.Pop
3.Display
4.Exit

Enter your choice(1-4):1

Enter element to push:200

*** Stack Menu ***
```

Queue Implementation using array

```
#include <stdio.h>
#define MAX 10
int queue_array[MAX];
    int rear = - 1;
    int front = - 1;
main()
{
    int choice;
    while (1)
    {
        printf("1.Insert element to queue \n");
        printf("2.Delete element from queue \n");
        printf("3.Display all elements of queue \n");
        printf("4.Quit \n");
        printf("Enter your choice : ");
        scanf("%d", &choice);
```

```
switch (choice)
{
    case 1:
        insert();
        break;
    case 2:
        delete();
        break;
    case 3:
        display();
        break;
    case 4:
        exit(1);
default:
    printf("Wrong choice \n");
}
/*End of switch*/
}
/*End of while*/
}
/*End of main()*/
```

```
insert()
{
    int add_item;
    if (rear == MAX-1)
        { printf("Queue Overflow \n");
        return; }

    if (front == - 1)          /*If queue is initially empty */
        { front = 0; }

    printf("Insert the element in queue : ");
    scanf("%d", &add_item);
    rear = rear + 1;
    queue_array[rear] = add_item;
}
} /*End of insert()*/
```

```
delete()
{
    if (front == - 1 || front > rear)
    {
        printf("Queue Underflow \n");
        return ;
    }
    else
    {
        printf("Element deleted from queue is :
                %d\n",queue_array[front]);
        front = front + 1;
    }
} /*End of delete() */
```

```
display()
{
    int i;
    if (front == - 1)
        printf("Queue is empty \n");
    else
    {
        printf("Queue is : \n");
        for (i = front; i <= rear; i++)
            printf("%d ", queue_array[i]);
        printf("\n");
    }
} /*End of display() */
```

C:\Users\Admin\Desktop\DSA_WINTER2018_2019\Queue.exe

File

```
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
```

6 Enter your choice : 1

6 Inset the element in queue : 100

```
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
```

7 Enter your choice : 1

7 Inset the element in queue : 200

```
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
```

7 Enter your choice : 3

7 Queue is :

7 100 200

```
i]);
```

```
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
```

7 Enter your choice : 2

- 2.Delete element from queue
- 3.Display all elements of queue
- 4.Quit

Enter your choice : 3

Queue is :

100 200

- 1.Insert element to queue
- 2.Delete element from queue
- 3.Display all elements of queue
- 4.Quit

Enter your choice : 2

Element deleted from queue is : 100

- 1.Insert element to queue
- 2.Delete element from queue
- 3.Display all elements of queue
- 4.Quit

Enter your choice : 3

Queue is :

200

- 1.Insert element to queue
- 2.Delete element from queue
- 3.Display all elements of queue
- 4.Quit

Enter your choice : ■

Application of Queue-Circular Queue

Drawback of Linear Queue

- Once the queue is full, even though few elements from the front are deleted and **some occupied space is relieved**, it is not possible to add anymore new elements, as the rear has already reached the Queue's rear most position.

Circular Queue

- This queue is not linear but circular.
- Its structure can be like the following figure:
- In circular queue, once the Queue is full the

"First" element of the Queue becomes the

"Rear" most element, if and only if the

"Front" has moved forward. otherwise it will

again be a "Queue overflow" state.

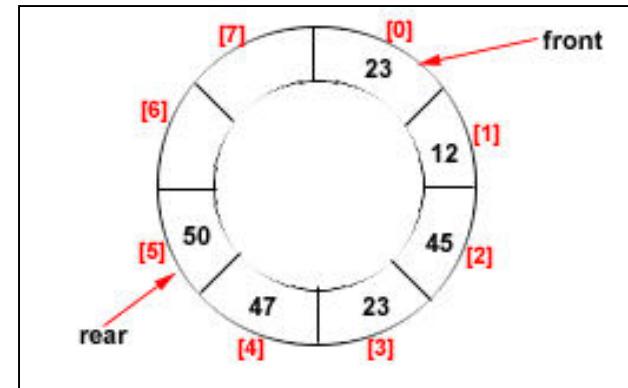


Figure: Circular Queue having
Rear = 5 and Front = 0

Algorithms for Insert and Delete Operations in Circular Queue

For Insert Operation

Insert-Circular-Q(CQueue, Rear, Front, N, Item)

Here, CQueue is a circular queue where to store data. Rear represents the location in which the data element is to be inserted and Front represents the location from which the data element is to be removed. Here N is the maximum size of CQueue and finally, Item is the new item to be added. Initially Rear = 0 and Front = 0.

1. If Front = 0 and Rear = 0 then Set Front := 1 and go to step 4.
2. If Front = 1 and Rear = N or Front = Rear + 1
then Print: “Circular Queue Overflow” and Return.
3. If Rear = N then Set Rear := 1 and go to step 5.
4. Set Rear := Rear + 1
5. Set CQueue [Rear] := Item.
6. Return

For Delete Operation

Delete-Circular-Q(CQueue, Front, Rear, Item)

Here, CQueue is the place where data are stored. Rear represents the location in which the data element is to be inserted and Front represents the location from which the data element is to be removed. Front element is assigned to Item. Initially, Front = 1.

1. If Front = 0 then

Print: “Circular Queue Underflow” and Return. /*..Delete without Insertion

2. Set Item := CQueue [Front]

3. If Front = N then Set Front = 1 and Return.

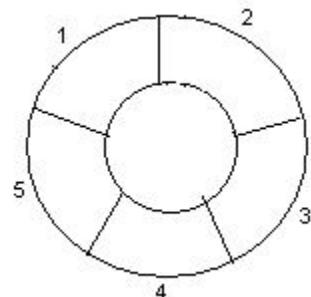
4. If Front = Rear then Set Front = 0 and Rear = 0 and Return.

5. Set Front := Front + 1

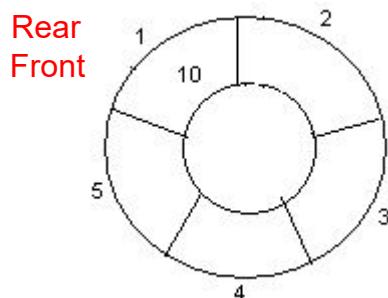
6. Return.

Example: Consider the following circular queue with $N = 5$.

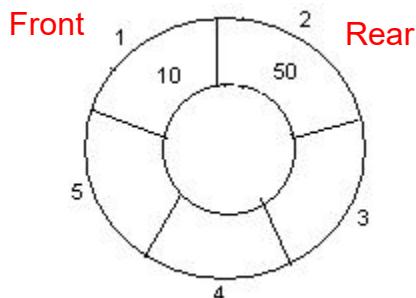
1. Initially, Rear = 0, Front = 0.



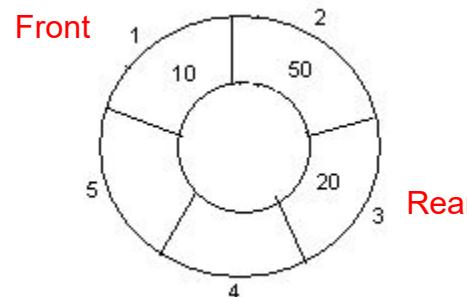
2. Insert 10, Rear = 1, Front = 1.



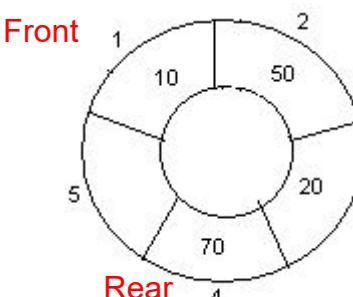
3. Insert 50, Rear = 2, Front = 1.



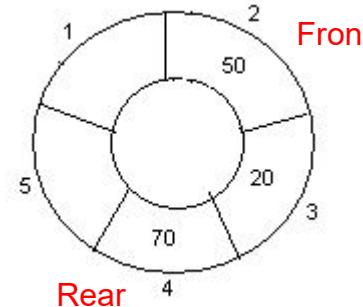
4. Insert 20, Rear = 3, Front = 0.



5. Insert 70, Rear = 4, Front = 1.



6. Delete front, Rear = 4, Front = 2.



Drawback of Linear Queue

- Once the queue is full, even though few elements from the front are deleted and **some occupied space is relieved**, it is not possible to add anymore new elements, as the rear has already reached the Queue's rear most position.

Circular Queue

- This queue is not linear but circular.
- Its structure can be like the following figure:
- In circular queue, once the Queue is full the

"First" element of the Queue becomes the

"Rear" most element, if and only if the

"Front" has moved forward. otherwise it will

again be a "Queue overflow" state.

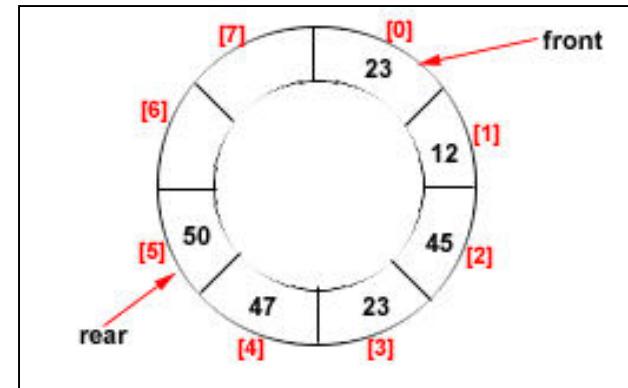


Figure: Circular Queue having
Rear = 5 and Front = 0

Algorithms for Insert and Delete Operations in Circular Queue

For Insert Operation

Insert-Circular-Q(CQueue, Rear, Front, N, Item)

Here, CQueue is a circular queue where to store data. Rear represents the location in which the data element is to be inserted and Front represents the location from which the data element is to be removed. Here N is the maximum size of CQueue and finally, Item is the new item to be added. Initially Rear = 0 and Front = 0.

1. If Front = 0 and Rear = 0 then Set Front := 1 and go to step 4.
2. If Front = 1 and Rear = N or Front = Rear + 1
then Print: “Circular Queue Overflow” and Return.
3. If Rear = N then Set Rear := 1 and go to step 5.
4. Set Rear := Rear + 1
5. Set CQueue [Rear] := Item.
6. Return

For Delete Operation

Delete-Circular-Q(CQueue, Front, Rear, Item)

Here, CQueue is the place where data are stored. Rear represents the location in which the data element is to be inserted and Front represents the location from which the data element is to be removed. Front element is assigned to Item. Initially, Front = 1.

1. If Front = 0 then

Print: “Circular Queue Underflow” and Return. /*..Delete without Insertion

2. Set Item := CQueue [Front]

3. If Front = N then Set Front = 1 and Return.

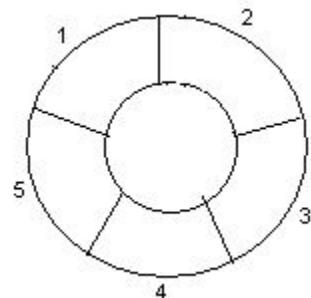
4. If Front = Rear then Set Front = 0 and Rear = 0 and Return.

5. Set Front := Front + 1

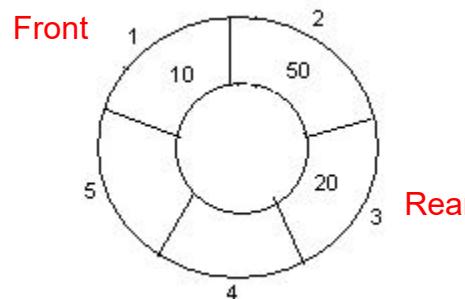
6. Return.

Example: Consider the following circular queue with N = 5.

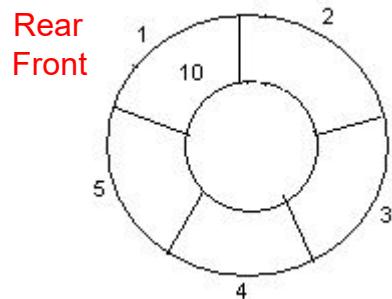
1. Initially, Rear = 0, Front = 0.



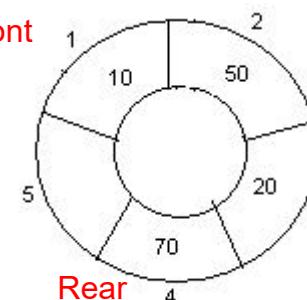
4. Insert 20, Rear = 3, Front = 1.



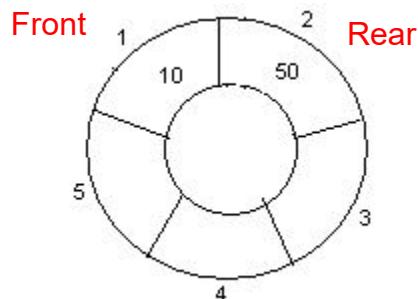
2. Insert 10, Rear = 1, Front = 1.



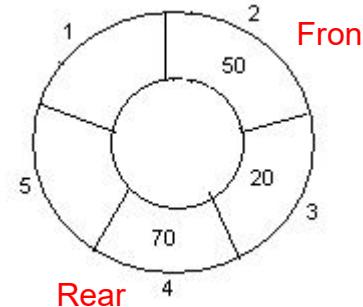
5. Insert 70, Rear = 4, Front = 1.



3. Insert 50, Rear = 2, Front = 1.

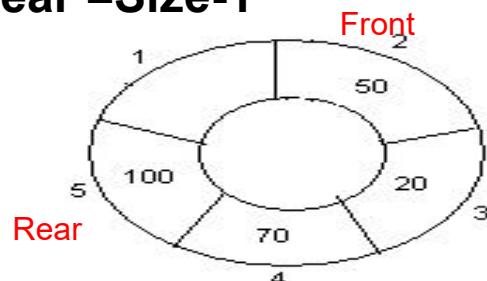


6. Delete front, Rear = 4, Front = 2.

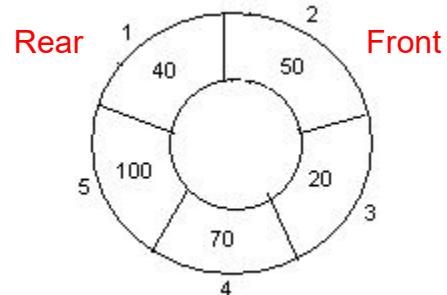


7. Insert 100, Rear = 5, Front = 2.

Rear =Size-1

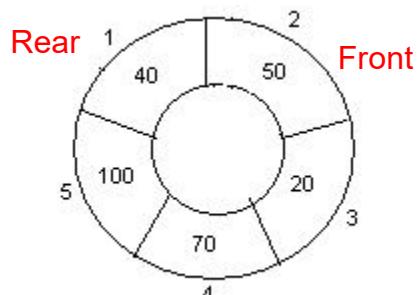


8. Insert 40, Rear = 1, Front = 2.

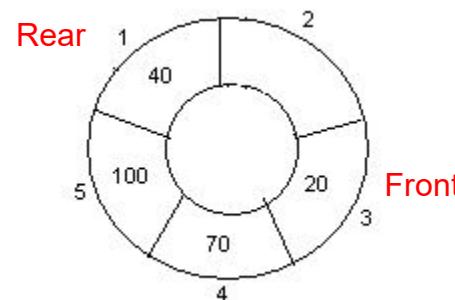


9. Insert 140, Rear = 1, Front = 2.

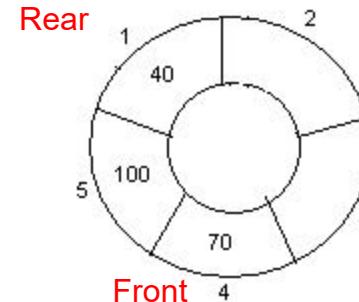
As Front = Rear + 1, so Queue overflow.



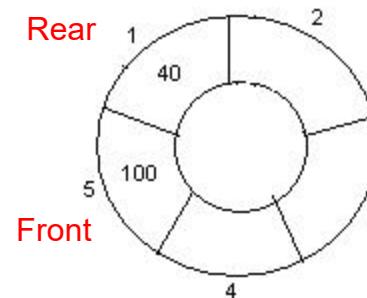
10.Delete front, Rear = 1, Front = 3.



11.Delete front, Rear = 1, Front = 4.



12. Delete front, Rear = 1, Front = 5.



```
//Implementation of Circular Queue Using Array
```

```
#include<stdio.h>
```

```
#define SIZE 3
```

```
int queue[SIZE];
```

```
int front = -1, rear = -1;
```

```
int isFull()
```

```
{
```

```
    if ((front == 0 && rear == SIZE-1) || (front == rear+1))
```

```
        return 1;
```

```
    else
```

```
        return 0;
```

```
}
```

```
int isEmpty()
```

```
{
```

```
    if ( front == -1 )
```

```
        return 1;
```

```
    else
```

```
        return 0;
```

```
}
```

```
void insert(int data)
{
if (front == -1)          //when the queue is initially empty
    front = rear = 0;
else {
    if ( isFull() ) {
        printf("\n Queue is full, cannot insert(overflow)\n");
        return;
    }
    else {
        if (rear == SIZE - 1)      // Reached end of queue When partially full- checking
            rear = 0;
        else
            rear = rear + 1;
    }
}
queue[rear] = data;
}
```

```
void dDelete()
{
    int data;
    if ( isEmpty() )
        printf("\n Queue is empty (underflow)\n");
    else
    {
        data = queue[front];
        printf("\n Element deleted from queue is : %d\n", data);
        if (front == rear)      // End of queue
        {
            front = -1;
            rear = -1;
        }
        else {
            if (front == SIZE - 1)  // rear is any where in the queue
                front = 0;
            else
                front = front + 1; // Normal deletion
        }
    }
}
```

```
void display()
{
    int frontPos = front, rearPos = rear;
    if ( isEmpty() ) {
        printf("\n Queue is empty ... \n");
        return;
    }
    printf("\n The elements in the queue are: \n ");
    if (frontPos <= rearPos)  {
        while (frontPos <= rearPos) {
            printf("%d  ", queue[frontPos]);
            frontPos++;
        }
    } else {
        while (frontPos <= SIZE - 1) {
            printf("%d  ", queue[frontPos]);
            frontPos++;
        }
    }
    frontPos = 0;
    while (frontPos <= rearPos) {
        printf("%d  ", queue[frontPos]);
        frontPos++;    }  } }
```

```
int main()
{
    char ch ='y';
    int choice, data;
    do
    {
        printf("\nEnter your choice for the operation: \n");
        printf("\n1.Insert");
        printf("\n2.Delete");
        printf("\n3.Display\n\n");

        scanf("%d",&choice);
```

```
switch(choice)
{
    case 1:
        printf("\nEnter the data to be inserted:\n");
        scanf("%d",&data);
        insert(data);
        display();
        break;

    case 2:
        Delete();
        display();
        break;

    case 3:
        display();
        break;

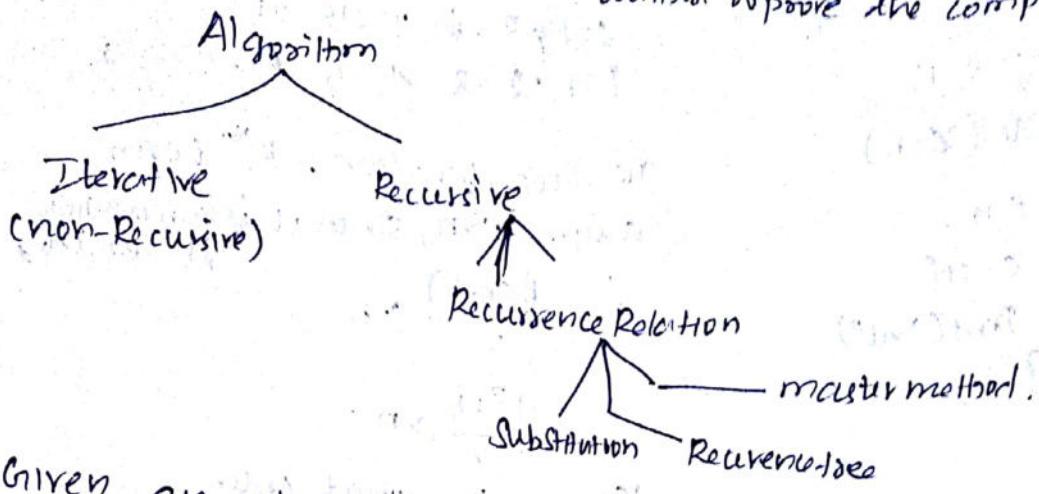
    default: printf("Wrong choice\n");
        break;
}
```

```
printf("\nDo you want to continue(y/n):\n");
fflush(stdin);
ch=getchar();
}
while(ch=='y' | | ch=='Y');
return 0;
}
```

Algorithms Analysis

Iterative analysing

Algorithm to prove the complexity.



A (C)

```
{  
    for i = 1 to n  
        max(a, b)  
    }  
}
```

looping

Recursion

```
A(n)  
{  
    if ( )  
        A(n/b)  
    }  
}
```

↔
conversion
is possible.

Recap:

O - Worst case

Ω - Best case

Θ - Average case

Example to search an element.

Iterative method:

A (C)

```
{  
    int i, j  
    for (i = 1 to n)  
        print ('val')  
    }  
}
```

adding of one more loop
← for (j = 1 to n)

Takes $O(n)$

$O(n^2)$ time

Moser Method

Logarithm

$\lg n = \log_2 n$ - binary algorithm

$\ln n = \log_e n$ - natural algorithm

$\lg^k n = (\lg n)^k$ - exponentiation

$\lg \lg n = \lg(\lg n)$ - composition

If $a > 0$, $b > 0$, $c > 0$ and n

$$a = b^{\log_b a}$$

$$\log_c(cab) = \log_c a + \log_c b$$

$$\log_b a^n = n \log_b a$$

$$\log_b a = \frac{\log_c a}{\log_c b} \quad \text{change of base.}$$

$$\log_b(1/a) = -\log_b a$$

$$\log_b a = \frac{y}{\log_a b}$$

$$a^{\log_b c} = c^{\log_b a}$$

Note: changing the base of the given problem to base 2.

Suppose that $x > 0$ is some number and

$$\log_4 x = y.$$

That means that $4^y = x$

$$\text{Now } 4 = 2^2, \text{ so } x = (2^2)^y = 2^{2y}$$

That means that $\log_2 x = 2y$

In other words we've just for $x > 0$

$$\boxed{\log_2 x = 2 \log_4 x}$$

\Rightarrow Special cases

Example:

$$\log_4 8 = n^{\log_2 8}$$

Apply the formula as

$$\frac{1}{2} \log_2 x = \log_4 x$$

$$\Rightarrow n^{1/2 \log_2 2}$$

$$\Rightarrow \boxed{n^{\log_4 8} = n^{3/2}}$$

Master Method - Recurrence relation.

$$T(n) = aT(n/b) + f(n)$$

$$a \geq 1 \quad b > 1 \quad f(n) = \Theta(n^k \log^p n)$$

Two information

1. $\log_b a$
2. k

Three cases:

1. case 1: If $\log_b a > k$ then $\Theta(n^{\log_b a})$

case 2: If $\log_b a = k$

If $p > -1 \quad \Theta(n^k \log^{p+1} n)$

If $p = -1 \quad \Theta(n^k \log \log n)$

If $p < -1 \quad \Theta(n^k)$

case 3: If $\log_b a < k$

If $p \geq 0 \quad \Theta(n^k \log^p n)$

If $p < 0 \quad \Theta(n^k)$

Example:

For case 1

$$T(n) = 2T(n/2) + 1$$

$$a=2, b=2, f(n)=\Theta(1)$$

$$= \Theta(n^0 \log^0 n)$$

$$\therefore \text{here } k=0 \text{ then } \log_b a = \log_2 2 = 1 > k=0$$

\therefore case 1 satisfies.

$$\Theta(n^0)$$

$$2. T(n) = 4T(n/2) + n$$

$$a=4, b=2, f(n)=n.$$

$$= \Theta(n^1 \log^0 n)$$

$$\log_2 4 = 2, k=1, P=0$$

$$\log_2 4 = 2 > k=1, \text{satisfies case.1}$$

$$\therefore \Theta(n^2)$$

$$3. T(n) = 8T(n/2) + n^1, a=8, b=2, f(n)=\Theta(n^1 \log^0 n)$$

$$\log_2 8 = 3 > k=1, P=0$$

$$\Theta(n^3) \text{ by case 1.}$$

$$4. T(n) = 9T(n/3) + 1 \quad \therefore a=9, b=3, f(n)=\Theta(n^0 \log^0 n)$$

$$= 9T(n/3) + n^0, P=0$$

$$\log_3 9 = 2 > k=0 \text{ case 1}$$

$$\Theta(n^2)$$

$$5. T(n) = 9T(n/3) + n^2 \quad \therefore a=9, b=3, f(n)=\Theta(n^2 \log^0 n)$$

$$\log_3 9 = 2 = k=2, P=0$$

$$\text{case 1 } \Theta(n^2)$$

For case 2.

$$1. T(n) = 2T(n/2) + n^1 \quad \therefore a=2, b=2, f(n)=\Theta(n \log^0 n)$$

$$\log_2 2 = 1 \quad k=1, \text{ here } \underline{P=0} \quad (\because P>-1)$$

$$\therefore \Theta(n^k \log^{P+1} n) \Rightarrow \Theta(n \log n)$$

$$2. T(n) = 4T(n/2) + n^2 \quad \because a=4, b=2, f(n)=\Theta(n^2 \log n)$$

$$\log_2 4 = 2, k=2, P=0 \quad [C : P > -1]$$

$$\Theta(n^2 \log n)$$

$$3. T(n) = 4T(n/2) + n^2 \log^2 n \quad \because a=4, b=2, f(n)=\Theta(n^2 \log^2 n)$$

$$\log_2 4 = 2, k=2, P=2 \quad [C : P > -1]$$

$$\therefore \Theta(n^k \log^{P+1} n) = \Theta(n^2 \log^3 n)$$

$$4. T(n) = 8T(n/2) + n^3 \log^5 n \quad \because a=8, b=2, f(n)=\Theta(n^3 \log^5 n)$$

$$\log_2 8 = 3, k=3, P=5 \quad [C : P > -1]$$

$$\therefore \Theta(n^k \log^{P+1} n) = \Theta(n^3 \log^6 n)$$

$$5. T(n) = 2T(n/2) + n/\log n \Rightarrow 2T(n/2) + n \log^{-1} n.$$

$$\log_2 2 = 1, k=1, P=-1 \quad [C : P < -1]$$

$$\therefore \Theta(n^k \log \log n) = \Theta(n \log \log n)$$

$$6. T(n) = 2T(n/2) + n/\log^2 n = 2T(n/2) + n \log^{-2} n.$$

$$\log_2 2 = 1, k=1, P=-2 \quad [C : P < -1]$$

$$\therefore \Theta(n^k) = \Theta(n)$$

For case: 3

$$1. T(n) = T(n/2) + n^2$$

$$\log_2 1 = 0 < k=2, P=0$$

$$\Theta(n^k \log^P n) = \Theta(n^2)$$

$$2. T(n) = 2T\left(\frac{n}{2}\right) + n^2 \log^2 n$$

$$\log_2^2 = 1 \quad \leftarrow k=2, P=2$$

$$\therefore \Theta(n^k \log^P n) = \Theta(n^2 \log^2 n)$$

$$3. T(n) = 4T\left(\frac{n}{2}\right) + n^3 / \log n = 4T\left(\frac{n}{2}\right) + n^3 \log^{-1} n.$$

$$\log_2 \neq = 2 \quad k=3, P=-1$$

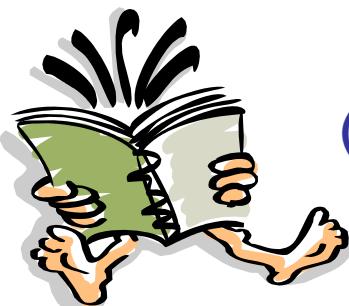
$$\therefore \Theta(n^k) = \Theta(n^3)$$

Analysis of Algorithms

CS 477/677

Asymptotic Analysis

Instructor: George Bebis



(Chapter 3, Appendix A)

Analysis of Algorithms

- An *algorithm* is a finite set of precise instructions for performing a computation or for solving a problem.
- What is the goal of analysis of algorithms?
 - To compare algorithms mainly in terms of running time but also in terms of other factors (e.g., memory requirements, programmer's effort etc.)
- What do we mean by running time analysis?
 - Determine how running time increases as the **size of the problem increases**.

Input Size

- Input size (number of elements in the input)
 - size of an array
 - polynomial degree
 - # of elements in a matrix
 - # of bits in the binary representation of the input
 - vertices and edges in a graph

Types of Analysis

- Worst case
 - Provides an upper bound on running time
 - An absolute **guarantee** that the algorithm would not run longer, no matter what the inputs are
- Best case
 - Provides a lower bound on running time
 - Input is the one for which the algorithm runs the fastest

$$\textit{Lower Bound} \leq \textit{Running Time} \leq \textit{Upper Bound}$$

- Average case
 - Provides a **prediction** about the running time
 - Assumes that the input is random

How do we compare algorithms?

- We need to define a number of objective measures.

(1) Compare execution times?

Not good: times are specific to a particular computer !!

(2) Count the number of statements executed?

Not good: number of statements vary with the programming language as well as the style of the individual programmer.

Ideal Solution

- Express running time as a function of the input size n (i.e., $f(n)$).
- Compare different functions corresponding to running times.
- Such an analysis is independent of machine time, programming style, etc.

Example

- Associate a "cost" with each statement.
- Find the "total cost" by finding the total number of times each statement is executed.

Algorithm 1

	Cost
arr[0] = 0;	c ₁
arr[1] = 0;	c ₁
arr[2] = 0;	c ₁
...	...
arr[N-1] = 0;	c ₁

Algorithm 2

	Cost
for(i=0; i<N; i++)	c ₂
arr[i] = 0;	c ₁

$$c_1 + c_1 + \dots + c_1 = c_1 \times N$$

$$(N+1) \times c_2 + N \times c_1 = \\ (c_2 + c_1) \times N + c_2$$

Another Example

<i>Algorithm 3</i>	<i>Cost</i>
sum = 0;	c_1
for(i=0; i<N; i++)	c_2
for(j=0; j<N; j++)	c_2
sum += arr[i][j];	c_3

$$c_1 + c_2 \times (N+1) + c_2 \times N \times (N+1) + c_3 \times N^2$$

Asymptotic Analysis

- To compare two algorithms with running times $f(n)$ and $g(n)$, we need a **rough measure** that characterizes **how fast each function grows**.
- Hint: use *rate of growth*
- Compare functions in the limit, that is, **asymptotically!**
(i.e., for large values of n)

Rate of Growth

- Consider the example of buying *elephants* and *goldfish*:

Cost: `cost_of_elephants + cost_of_goldfish`

Cost \sim `cost_of_elephants` (approximation)

- The low order terms in a function are relatively insignificant for **large n**

$$n^4 + 100n^2 + 10n + 50 \quad \sim \quad n^4$$

i.e., we say that $n^4 + 100n^2 + 10n + 50$ and n^4 have the same **rate of growth**

Asymptotic Notation

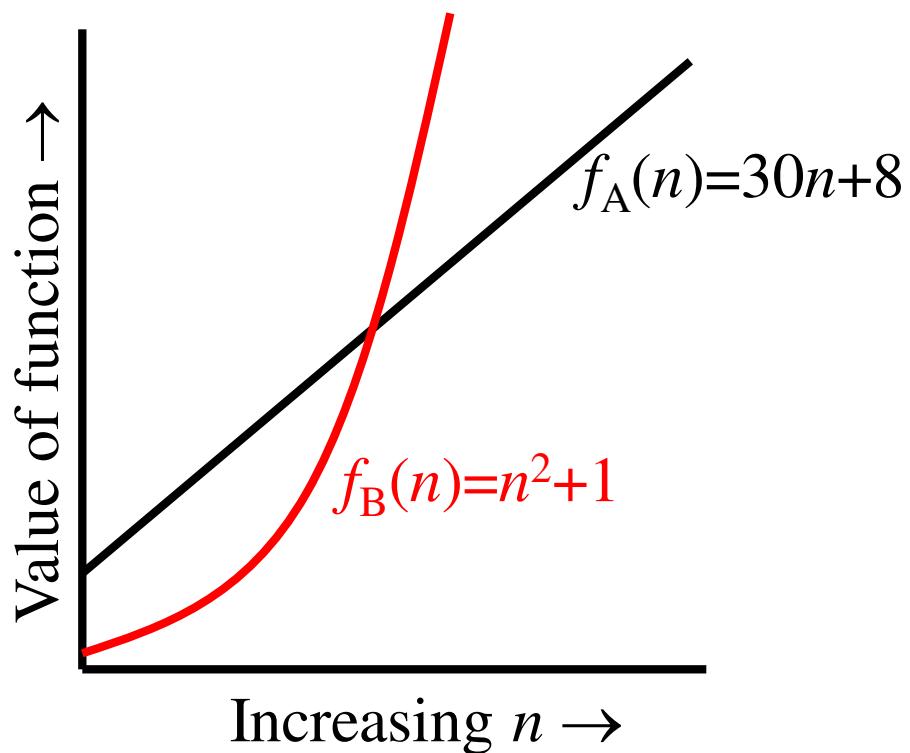
- O notation: asymptotic “less than”:
 - $f(n)=O(g(n))$ implies: $f(n) \leq g(n)$
- Ω notation: asymptotic “greater than”:
 - $f(n)=\Omega(g(n))$ implies: $f(n) \geq g(n)$
- Θ notation: asymptotic “equality”:
 - $f(n)=\Theta(g(n))$ implies: $f(n) = g(n)$

Big-O Notation

- We say $f_A(n)=30n+8$ is *order n*, or $O(n)$. It is, at most, roughly *proportional* to n .
- $f_B(n)=n^2+1$ is *order n^2* , or $O(n^2)$. It is, at most, roughly proportional to n^2 .
- In general, any $O(n^2)$ function is faster-growing than any $O(n)$ function.

Visualizing Orders of Growth

- On a graph, as you go to the right, a faster growing function eventually becomes larger...



More Examples ...

- $n^4 + 100n^2 + 10n + 50$ is $O(n^4)$
- $10n^3 + 2n^2$ is $O(n^3)$
- $n^3 - n^2$ is $O(n^3)$
- constants
 - 10 is $O(1)$
 - 1273 is $O(1)$

Back to Our Example

Algorithm 1

	Cost
arr[0] = 0;	c_1
arr[1] = 0;	c_1
arr[2] = 0;	c_1
...	
arr[N-1] = 0;	c_1

$$\text{-----}$$
$$c_1 + c_1 + \dots + c_1 = c_1 \times N$$

Algorithm 2

	Cost
for(i=0; i<N; i++)	c_2
arr[i] = 0;	c_1

$$\text{-----}$$
$$(N+1) \times c_2 + N \times c_1 =$$
$$(c_2 + c_1) \times N + c_2$$

- Both algorithms are of the same order: $O(N)$

Example (cont'd)

Algorithm 3

```
sum = 0;  
for(i=0; i<N; i++)  
    for(j=0; j<N; j++)  
        sum += arr[i][j];
```

Cost

c_1

c_2

c_2

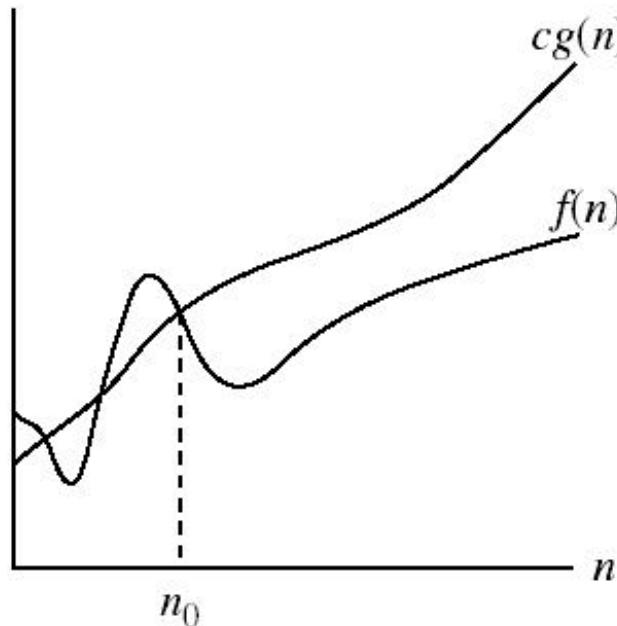
c_3

$$c_1 + c_2 \times (N+1) + c_2 \times N \times (N+1) + c_3 \times N^2 = O(N^2)$$

Asymptotic notations

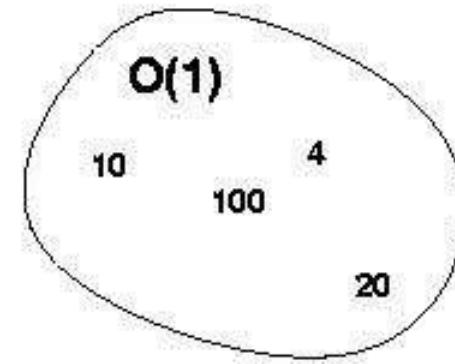
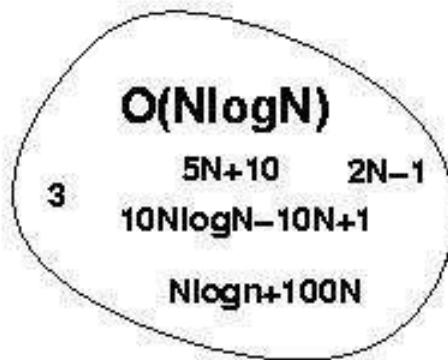
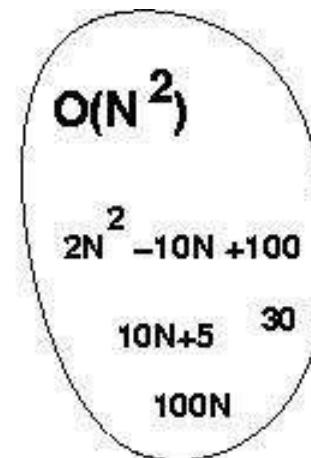
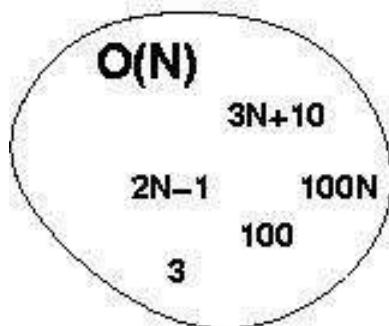
- *O-notation*

$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that}$
 $0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}.$



$g(n)$ is an *asymptotic upper bound* for $f(n)$.

Big-O Visualization



$O(g(n))$ is the set of functions with smaller or same order of growth as $g(n)$

Examples

- $2n^2 = O(n^3)$: $2n^2 \leq cn^3 \Rightarrow 2 \leq cn \Rightarrow c = 1$ and $n_0 = 2$
- $n^2 = O(n^2)$: $n^2 \leq cn^2 \Rightarrow c \geq 1 \Rightarrow c = 1$ and $n_0 = 1$
- $1000n^2 + 1000n = O(n^2)$:

$$1000n^2 + 1000n \leq 1000n^2 + n^2 = 1001n^2 \Rightarrow c = 1001 \text{ and } n_0 = 1000$$

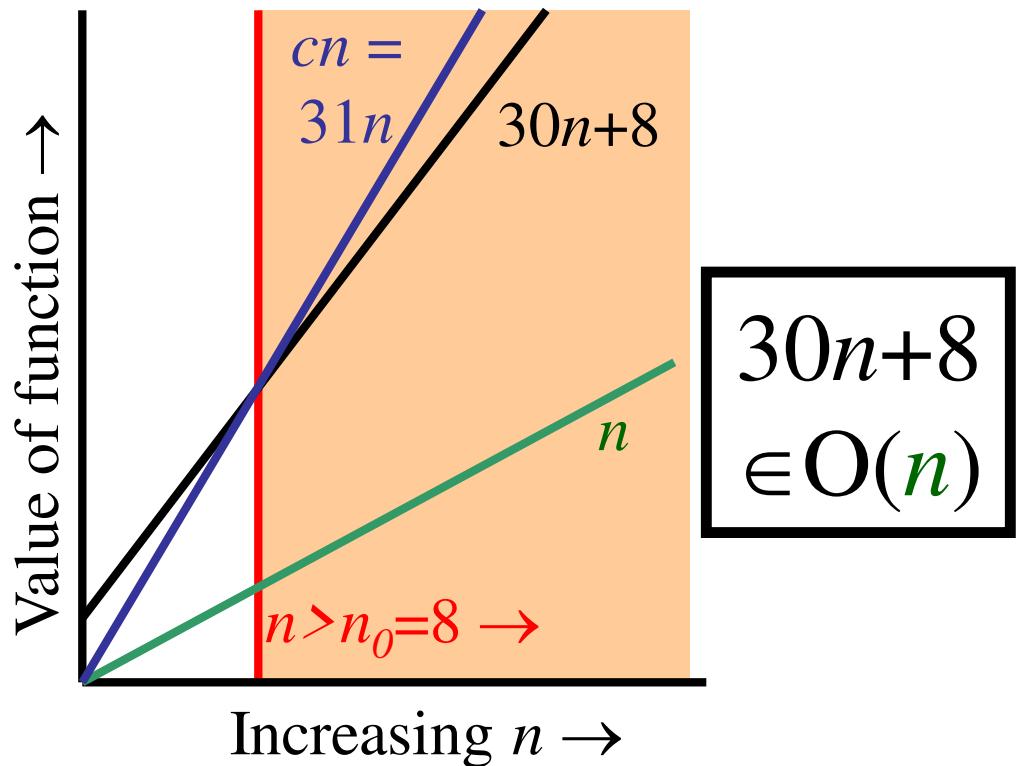
- $n = O(n^2)$: $n \leq cn^2 \Rightarrow cn \geq 1 \Rightarrow c = 1$ and $n_0 = 1$

More Examples

- Show that $30n+8$ is $O(n)$.
 - Show $\exists c, n_0: 30n+8 \leq cn, \forall n > n_0$.
 - Let $c=31, n_0=8$. Assume $n > n_0 = 8$. Then
 $cn = 31n = 30n + n > 30n+8$, so $30n+8 < cn$.

Big-O example, graphically

- Note $30n+8$ isn't less than n anywhere ($n>0$).
- It isn't even less than $31n$ everywhere.
- But it *is* less than $31n$ everywhere to the right of $n=8$.



No Uniqueness

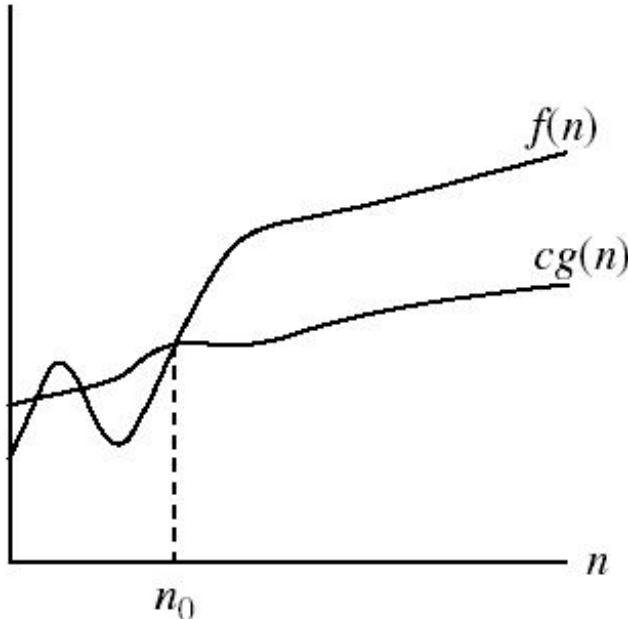
- There is no unique set of values for n_0 and c in proving the asymptotic bounds
- Prove that $100n + 5 = O(n^2)$
 - $100n + 5 \leq 100n + n = 101n \leq 101n^2$
for all $n \geq 5$
 - $n_0 = 5$ and $c = 101$ is a solution
 - $100n + 5 \leq 100n + 5n = 105n \leq 105n^2$
for all $n \geq 1$
 - $n_0 = 1$ and $c = 105$ is also a solution

Must find **SOME** constants c and n_0 that satisfy the asymptotic notation relation

Asymptotic notations (cont.)

- Ω - notation

$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}$.



$\Omega(g(n))$ is the set of functions
with larger or same order of
growth as $g(n)$

$g(n)$ is an *asymptotic lower bound* for $f(n)$.

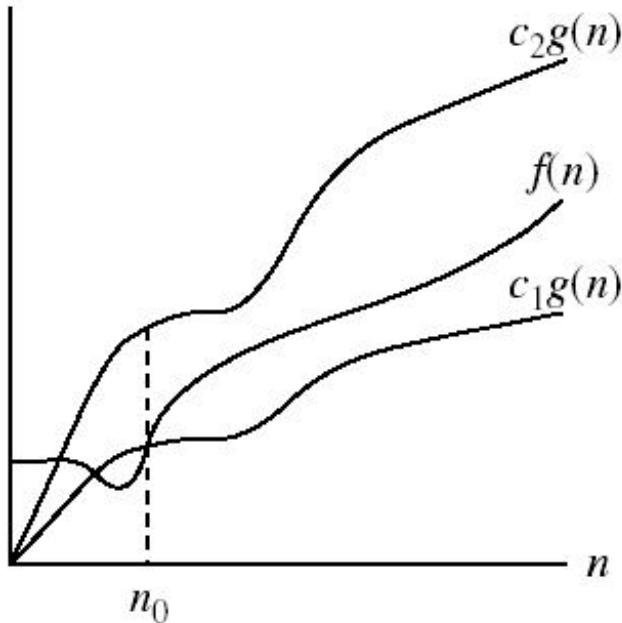
Examples

- $5n^2 = \Omega(n)$
 $\exists c, n_0$ such that: $0 \leq cn \leq 5n^2 \Rightarrow cn \leq 5n^2 \Rightarrow c = 1$ and $n_0 = 1$
- $100n + 5 \neq \Omega(n^2)$
 $\exists c, n_0$ such that: $0 \leq cn^2 \leq 100n + 5$
 $100n + 5 \leq 100n + 5n (\forall n \geq 1) = 105n$
 $cn^2 \leq 105n \Rightarrow n(cn - 105) \leq 0$
Since n is positive $\Rightarrow cn - 105 \leq 0 \Rightarrow n \leq 105/c$
 \Rightarrow contradiction: n cannot be smaller than a constant
- $n = \Omega(2n), n^3 = \Omega(n^2), n = \Omega(\log n)$

Asymptotic notations (cont.)

- Θ -notation

$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that}$
 $0 \leq c_1g(n) \leq f(n) \leq c_2g(n) \text{ for all } n \geq n_0\}$.



$\Theta(g(n))$ is the set of functions
with the same order of growth
as $g(n)$

$g(n)$ is an *asymptotically tight bound* for $f(n)$.

Examples

$$- n^2/2 - n/2 = \Theta(n^2)$$

$$\bullet \frac{1}{2} n^2 - \frac{1}{2} n \leq \frac{1}{2} n^2 \quad \forall n \geq 0 \quad \Rightarrow \quad c_2 = \frac{1}{2}$$

$$\bullet \frac{1}{2} n^2 - \frac{1}{2} n \geq \frac{1}{2} n^2 - \frac{1}{2} n * \frac{1}{2} n \quad (\forall n \geq 2) = \frac{1}{4} n^2$$

$$\Rightarrow c_1 = \frac{1}{4}$$

$$- n \neq \Theta(n^2): c_1 n^2 \leq n \leq c_2 n^2$$

\Rightarrow only holds for: $n \leq 1/c_1$

Examples

- $6n^3 \neq \Theta(n^2)$: $c_1 n^2 \leq 6n^3 \leq c_2 n^2$

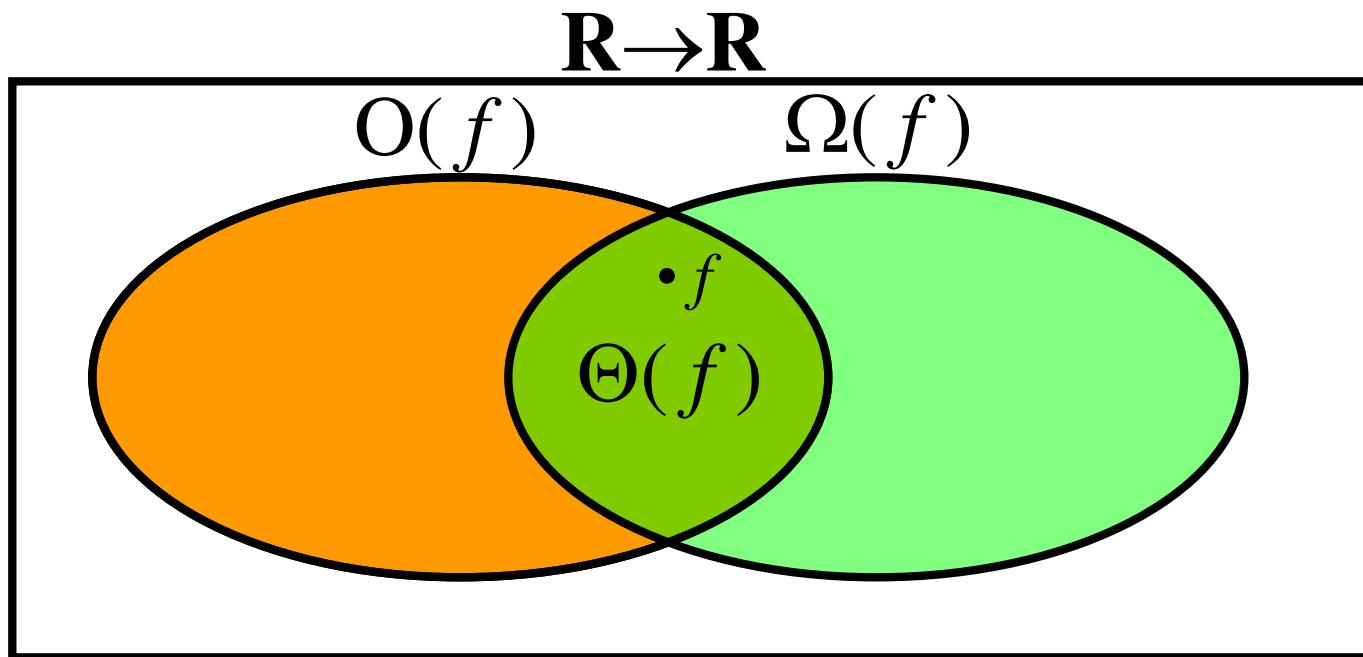
\Rightarrow only holds for: $n \leq c_2 / 6$

- $n \neq \Theta(\log n)$: $c_1 \log n \leq n \leq c_2 \log n$

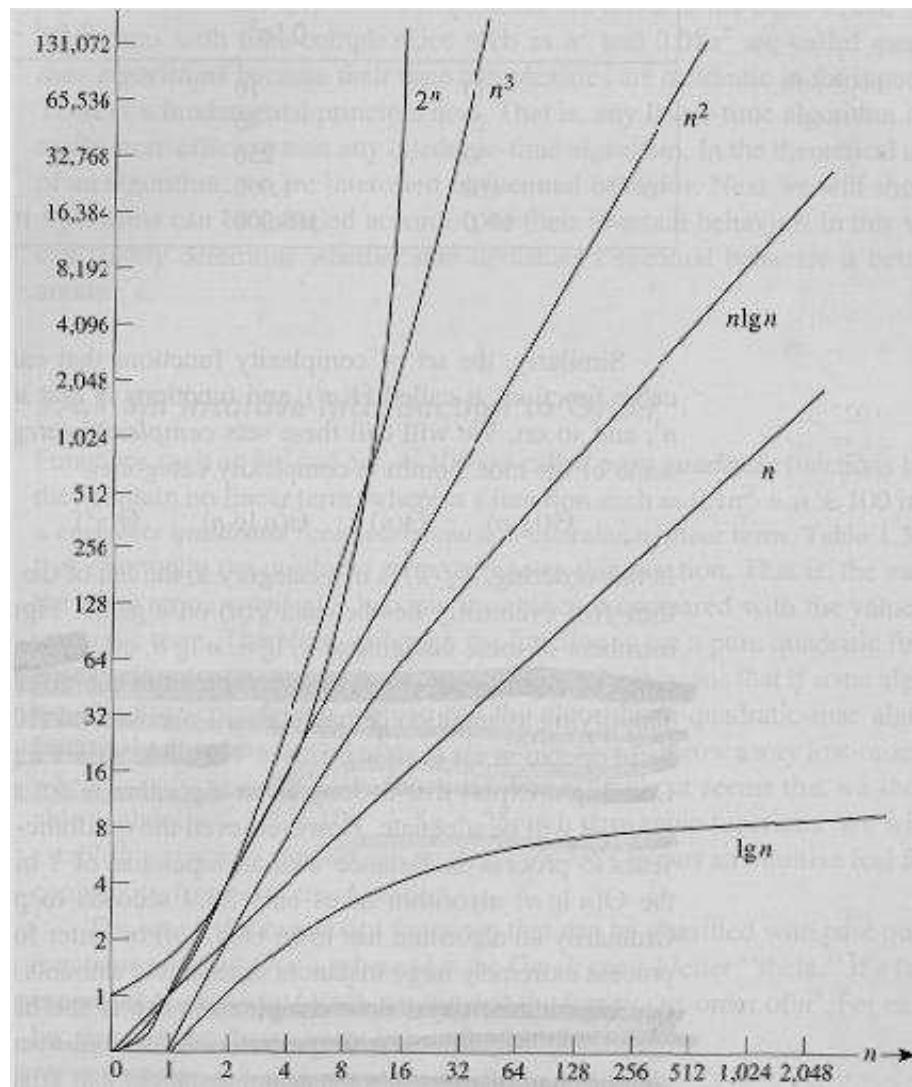
$\Rightarrow c_2 \geq n/\log n, \forall n \geq n_0$ - impossible

Relations Between Different Sets

- Subset relations between order-of-growth sets.



Common orders of magnitude



Common orders of magnitude

Table 1.4 Execution times for algorithms with the given time complexities

n	$f(n) = \lg n$	$f(n) = n$	$f(n) = n \lg n$	$f(n) = n^2$	$f(n) = n^3$	$f(n) = 2^n$
10	0.003 μs^*	0.01 μs	0.033 μs	0.1 μs	1 μs	1 μs
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	8 μs	1 ms [†]
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	27 μs	1 s
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	64 μs	18.3 min
50	0.005 μs	0.05 μs	0.282 μs	2.5 μs	25 μs	13 days
10^2	0.007 μs	0.10 μs	0.664 μs	10 μs	1 ms	4×10^{15} years
10^3	0.010 μs	1.00 μs	9.966 μs	1 ms	1 s	
10^4	0.013 μs	0 μs	130 μs	100 ms	16.7 min	
10^5	0.017 μs	0.10 ms	1.67 ms	10 s	11.6 days	
10^6	0.029 μs	1 ms	19.93 ms	16.7 min	31.7 years	
10^7	0.023 μs	0.01 s	0.23 s	1.16 days	31,709 years	
10^8	0.027 μs	0.10 s	2.66 s	115.7 days	3.17×10^7 years	
10^9	0.030 μs	1 s	29.90 s	31.7 years		

*1 $\mu s = 10^{-6}$ second.

[†]1 ms = 10^{-3} second.

Logarithms and properties

- In algorithm analysis we often use the notation “ $\log n$ ” without specifying the base

Binary logarithm $\lg n = \log_2 n$

Natural logarithm $\ln n = \log_e n$

$$\lg^k n = (\lg n)^k$$

$$\lg \lg n = \lg(\lg n)$$

$$\log x^y = y \log x$$

$$\log xy = \log x + \log y$$

$$\log \frac{x}{y} = \log x - \log y$$

$$a^{\log_b x} = x^{\log_b a}$$

$$\log_b x = \frac{\log_a x}{\log_a b}$$

More Examples

- For each of the following pairs of functions, either $f(n)$ is $O(g(n))$, $f(n)$ is $\Omega(g(n))$, or $f(n) = \Theta(g(n))$. Determine which relationship is correct.

- $f(n) = \log n^2$; $g(n) = \log n + 5$	$f(n) = \Theta(g(n))$
- $f(n) = n$; $g(n) = \log n^2$	$f(n) = \Omega(g(n))$
- $f(n) = \log \log n$; $g(n) = \log n$	$f(n) = O(g(n))$
- $f(n) = n$; $g(n) = \log^2 n$	$f(n) = \Omega(g(n))$
- $f(n) = n \log n + n$; $g(n) = \log n$	$f(n) = \Omega(g(n))$
- $f(n) = 10$; $g(n) = \log 10$	$f(n) = \Theta(g(n))$
- $f(n) = 2^n$; $g(n) = 10n^2$	$f(n) = \Omega(g(n))$
- $f(n) = 2^n$; $g(n) = 3^n$	$f(n) = O(g(n))$

Properties

- *Theorem:*

$$f(n) = \Theta(g(n)) \Leftrightarrow f = O(g(n)) \text{ and } f = \Omega(g(n))$$

- **Transitivity:**

- $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$
- Same for O and Ω

- **Reflexivity:**

- $f(n) = \Theta(f(n))$
- Same for O and Ω

- **Symmetry:**

- $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$

- **Transpose symmetry:**

- $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$

Asymptotic Notations in Equations

- On the right-hand side

- $\Theta(n^2)$ stands for some anonymous function in $\Theta(n^2)$

$2n^2 + 3n + 1 = 2n^2 + \Theta(n)$ means:

There exists a function $f(n) \in \Theta(n)$ such that

$$2n^2 + 3n + 1 = 2n^2 + f(n)$$

- On the left-hand side

$$2n^2 + \Theta(n) = \Theta(n^2)$$

No matter how the anonymous function is chosen on the left-hand side, there is a way to choose the anonymous function on the right-hand side to make the equation valid.

Common Summations

- Arithmetic series:

$$\sum_{k=1}^n k = 1 + 2 + \dots + n = \frac{n(n+1)}{2}$$

- Geometric series:

– Special case: $|x| < 1$:

$$\sum_{k=0}^n x^k = 1 + x + x^2 + \dots + x^n = \frac{x^{n+1} - 1}{x - 1} (x \neq 1)$$

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$$

- Harmonic series:

$$\sum_{k=1}^n \frac{1}{k} = 1 + \frac{1}{2} + \dots + \frac{1}{n} \approx \ln n$$

- Other important formulas:

$$\sum_{k=1}^n \lg k \approx n \lg n$$

$$\sum_{k=1}^n k^p = 1^p + 2^p + \dots + n^p \approx \frac{1}{p+1} n^{p+1}$$

Mathematical Induction

- A powerful, rigorous technique for proving that a statement $S(n)$ is true for every natural number n , no matter how large.
- Proof:
 - **Basis step:** prove that the statement is true for $n = 1$
 - **Inductive step:** assume that $S(n)$ is true and prove that $S(n+1)$ is true for all $n \geq 1$
- Find case n “within” case $n+1$

Example

- Prove that: $2n + 1 \leq 2^n$ for all $n \geq 3$
- **Basis step:**
 - $n = 3: 2 * 3 + 1 \leq 2^3 \Leftrightarrow 7 \leq 8$ TRUE
- **Inductive step:**
 - Assume inequality is true for n , and prove it for $(n+1)$:
 $2n + 1 \leq 2^n$ must prove: $2(n + 1) + 1 \leq 2^{n+1}$
 $2(n + 1) + 1 = (2n + 1) + 2 \leq 2^n + 2 \leq$
 $\leq 2^n + 2^n = 2^{n+1}$, since $2 \leq 2^n$ for $n \geq 1$

Linked List

Linked List and Application of Linked
List

Linked List

- Difference between array and linked list
- Advantages of linked list
- Disadvantages of linked list
- Definition of linked list
- Structure of Linked list
- Creation of node – malloc()

Linked List

- Dynamic data structure- Shrink or grow
- Elements of linked list are called NODE
- Self Referential data type - point to itself in the declaration
- *Insertion/Deletion of nodes at any point is allowed*
- Random Access of elements is not allowed.

Advantages of Linked Lists

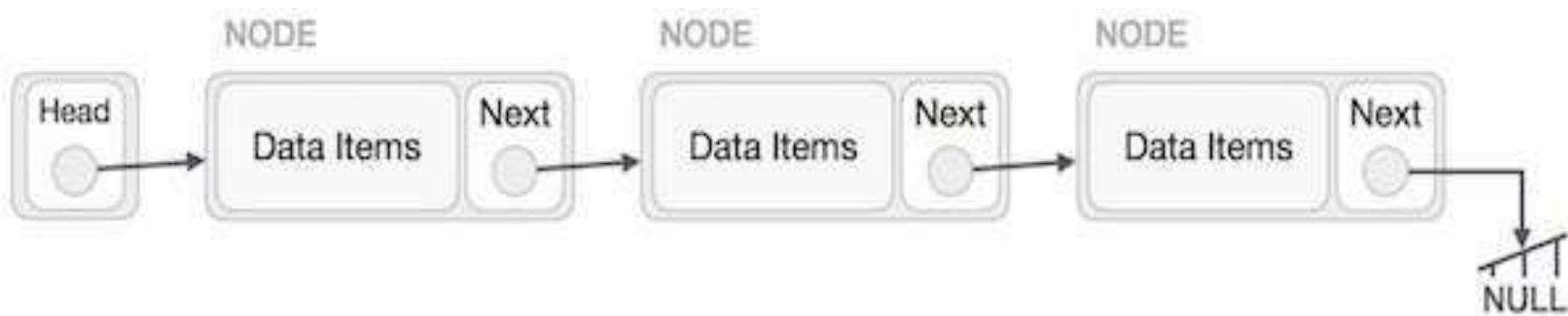
- They are dynamic in nature which allocates the memory when required.
- Linked List can grow and shrink during run time.
- Insertion and deletion operations can be easily implemented.
- Stacks and queues can be easily executed.

Disadvantages of Linked List

- The memory is wasted as pointers require extra memory for storage.
- No element can be accessed randomly; it has to access each node sequentially.
- Reverse Traversing is difficult in linked list.

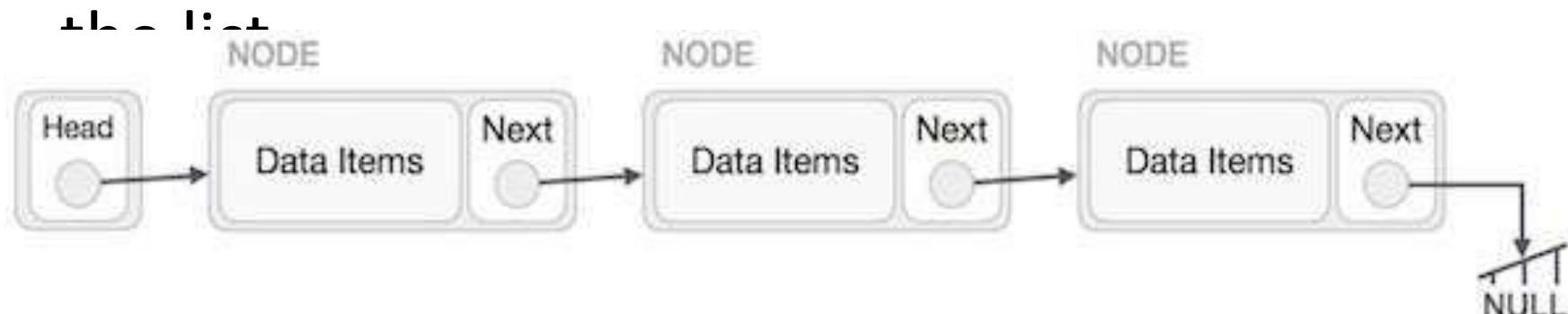
Linked List Representation

- Linked list can be visualized as a chain of nodes, where every node points to the next node.



As per the above illustration, following are the important points to be considered.

- Linked List contains a link element called first/Head.
- Each link carries a data field(s)/items and a link field called next.
- Each link is linked with its next link using its next link.
- Last link carries a link as null to mark the end of

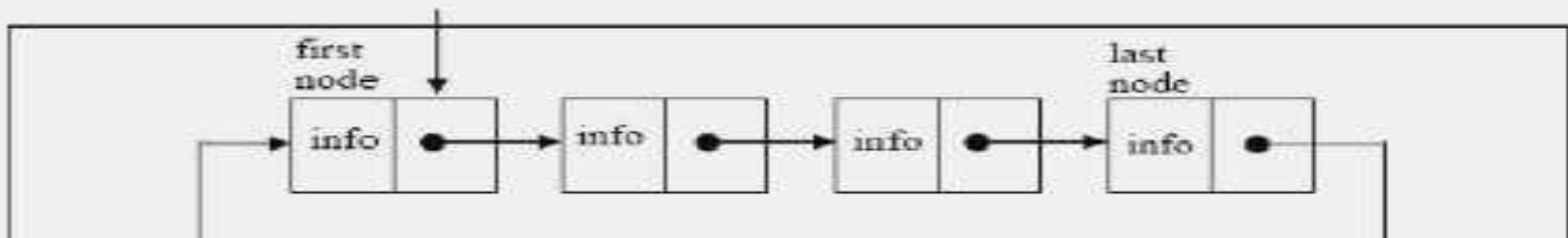


Various types of linked list

Following are the various types of linked list.

- **Simple Linked List** – Item navigation is forward only.
- **Doubly Linked List** – Items can be navigated forward and backward.
- **Circular Linked List** – Last item contains link of the first element as next and the first element has a link to the last element as previous.

Circular Linked Lists



a circular linked list without header node

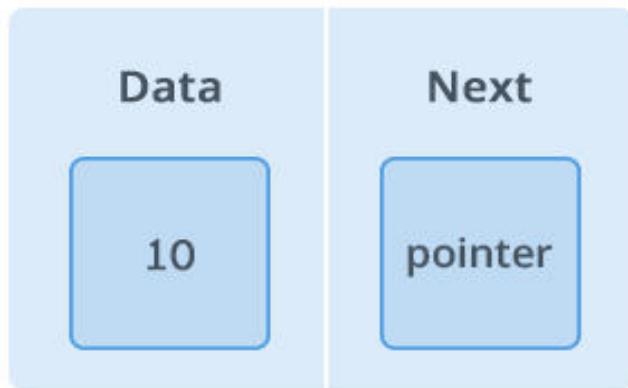
Applications of linked list:

1. Linked lists are used to represent and manipulate polynomial.
2. Hash tables use linked lists for collision resolution.
3. Any "File Requester" dialog uses a linked list.
4. Linked lists use to implement stack, queue, trees and graphs.
5. To implement the symbol table in compiler construction.

Basic Operations

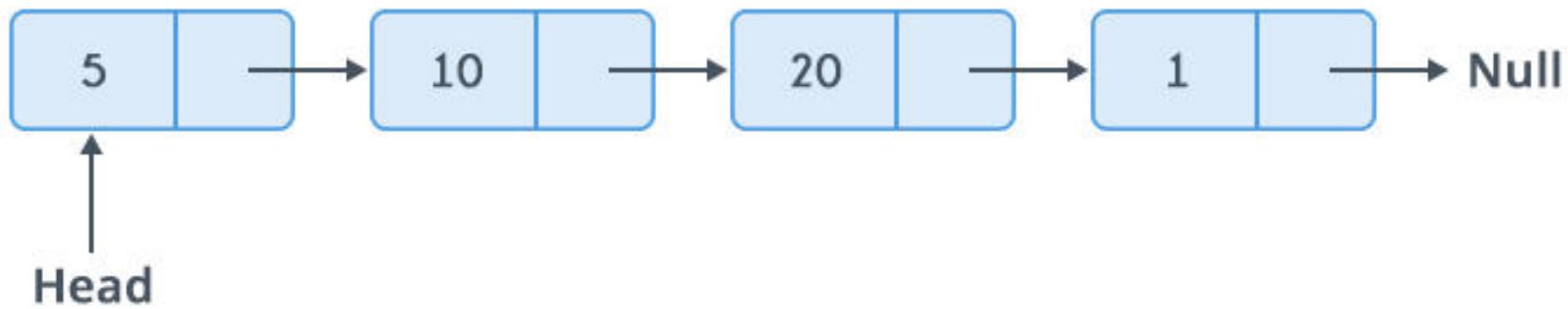
- Following are the basic operations supported by a list.
- **Insertion** – Adds an element at the beginning of the list.
- **Deletion** – Deletes an element at the beginning of the list.
- **Display** – Displays the complete list.
- **Search** – Searches an element using the given key.
- **Delete** – Deletes an element using the given key.

Node:

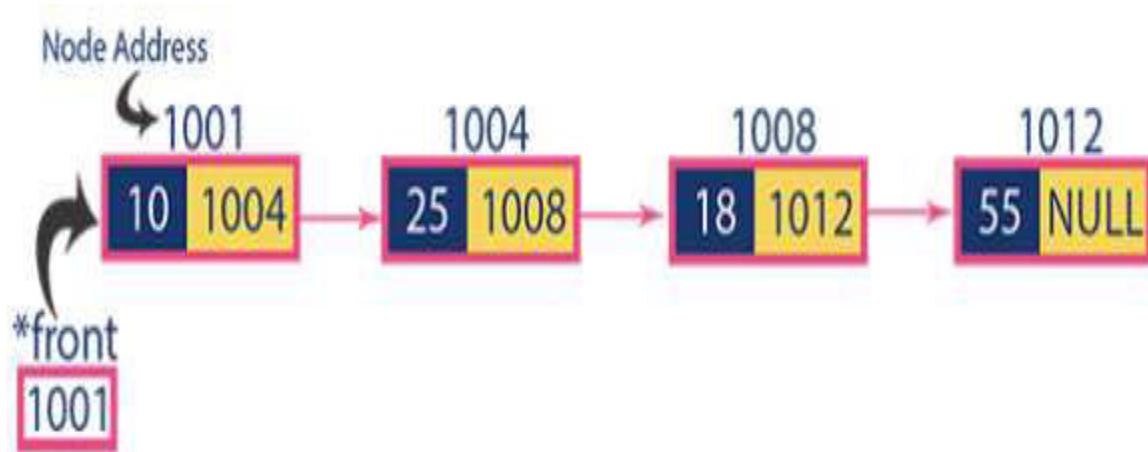


A **node** is a collection of two sub-elements or parts. A **data** part that stores the element and a **next** part that stores the link to the next node.

Linked List:



Example



malloc()

- The name malloc stands for "memory allocation".
- The function malloc() reserves a **block of memory of specified size** and **return a pointer of type void which can be casted into pointer of any form.**
- **Syntax of malloc()**

ptr = (cast-type*) malloc(byte-size)

- Here, ptr is pointer of cast-type. The malloc() function returns a pointer to an area of memory with size of byte size. If the space is insufficient, allocation fails and returns NULL pointer.
- **ptr = (int*) malloc(100 * sizeof(int));** This statement will allocate either 200 or 400 according to size of int 2 or 4 bytes respectively and **the pointer points to the address of first byte of memory.**

self referential data structure

- A self referential data structure is essentially a *structure definition which includes at least one member that is a pointer to the structure of its own kind.*

A chain of such structures can thus be expressed as follows.

```
struct name { member 1;  
             member 2; . . .  
             struct name *pointer; };
```

- The above illustrated structure prototype describes one node that comprises of two logical segments.
- One of them **stores data/information** and the **other one is a pointer indicating where the next component can be found**. Several such interconnected nodes create a chain of structures.

Linked list

```
struct node_type  
{ int data;  
  struct node_type *next; };  
  struct node_type *start = NULL;
```

Creating a Node of Linked list

```
//--Self-referential Structure-----//  
struct node  
{  
    int data;  
    struct node *next;  
};  
struct node *start;
```

```
void creat()
{ char ch;
struct node *new_node,*current;

do {
    new_node=(struct node *)malloc(sizeof(struct node));
    printf("\nEnter the data : "); //storing data to the node
    scanf("%d",&new_node->data);
    new_node->next=NULL;
    if(start==NULL) // create a new node and link
    { start=new_node;
        current=new_node; }
    else
    { current->next=new_node;
        current=new_node; }
    printf("Do you want to create another : ");
    ch=getche();
}while(ch!='n');
}
```

```
void display()
{
struct node *new_node;
printf("The Linked List : n");
new_node=start;
while(new_node!=NULL)
{
    printf("%d--->",new_node->data);
    new_node=new_node->next;
}
printf("NULL");
}
```

```
void main()
{
create();
display();
}
```

Inserting a node at the beginning

```
struct node *new_node,*current;
void insert_at_beg()
{
    new_node=(struct node *)malloc(sizeof(struct
node));
    if(new_node == NULL) {
        printf("\nFailed to Allocate memory");
        exit(0);
    }
    printf("Enter the data : ");
    scanf("%d",&new_node->data);
    new_node->next=NULL;
    if(start==NULL)
    { start=new_node; current=new_node; }
    else
    { new_node->next=start; start=new_node; }
```

Inserting a Node in the middle

```
void insert_mid()
{
    int pos,i;
    struct node *new_node,*current,*temp,*temp1;
    new_node=(struct node *)malloc(sizeof(struct
node));
    printf("\nEnter the data : ");
    scanf("%d",&new_node->data);
    new_node->next=NULL;
//Get the position to insert
    printf("\nEnter the position : ");
    scanf("%d",&pos);
```

//search for the position and insert the data

```
if(start==NULL)
{
    start=new_node;
    current=new_node;
}
else
{
    temp = start;
    for(i=1;i< pos-1;i++)
    {
        temp = temp->next;
    }
    temp1=temp->next;
    temp->next = new_node;
    new_node->next=temp1;
}}
```

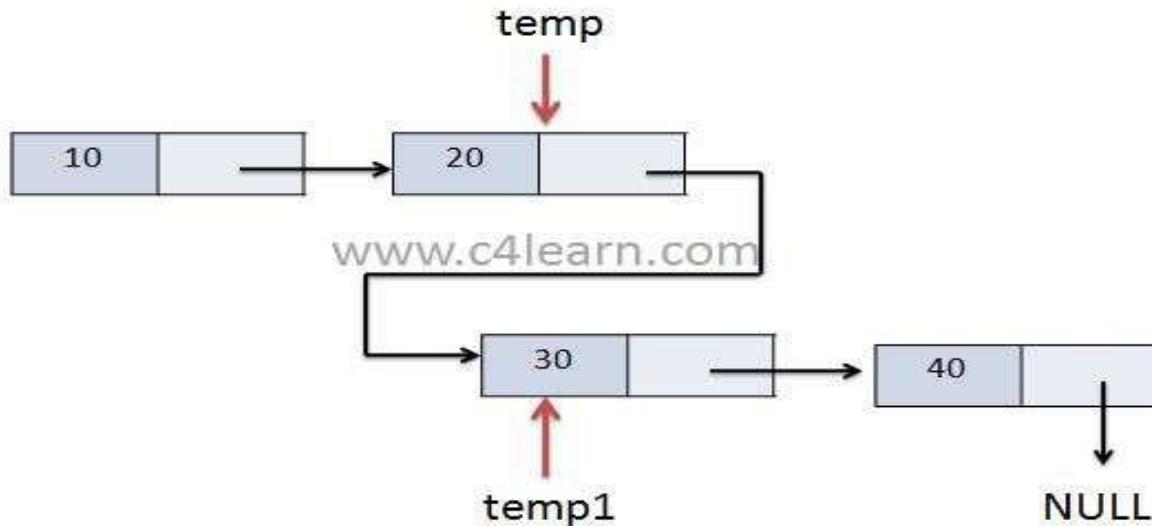
```
temp = start;
```

```
for(i=1;i< pos-1;i++)
```

```
{ temp = temp->next; }
```

```
temp1 = temp->next
```

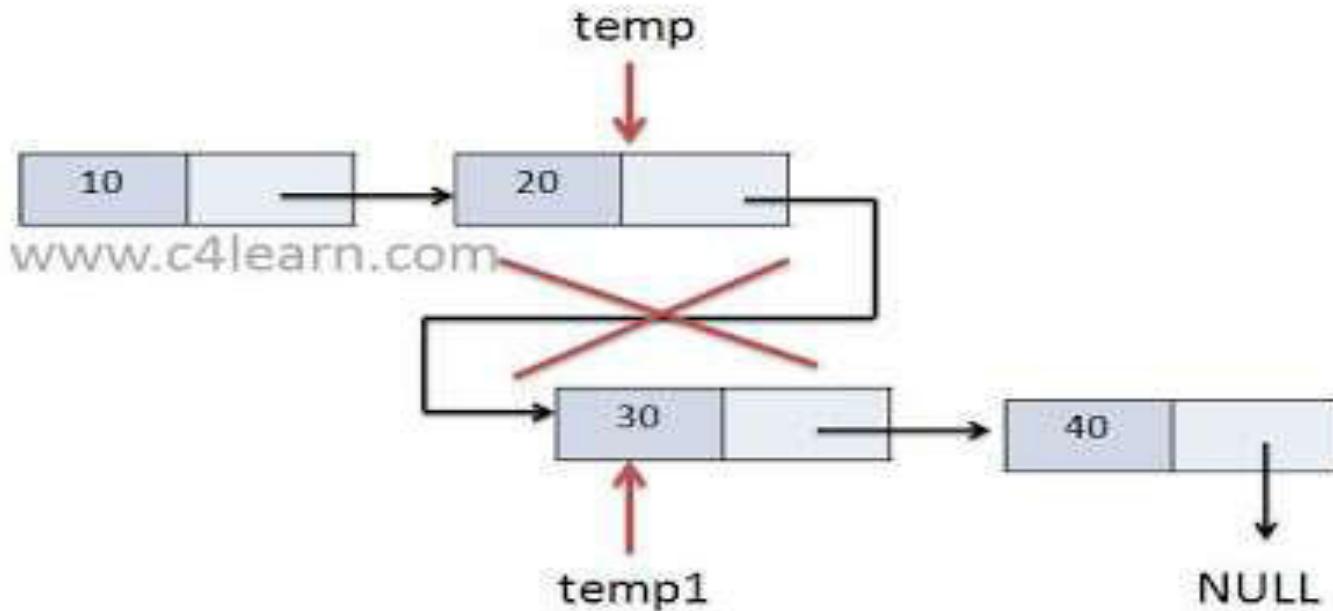
www.c4learn.com



`temp1=temp->next;`

Remove Link Between `temp` and `temp1`

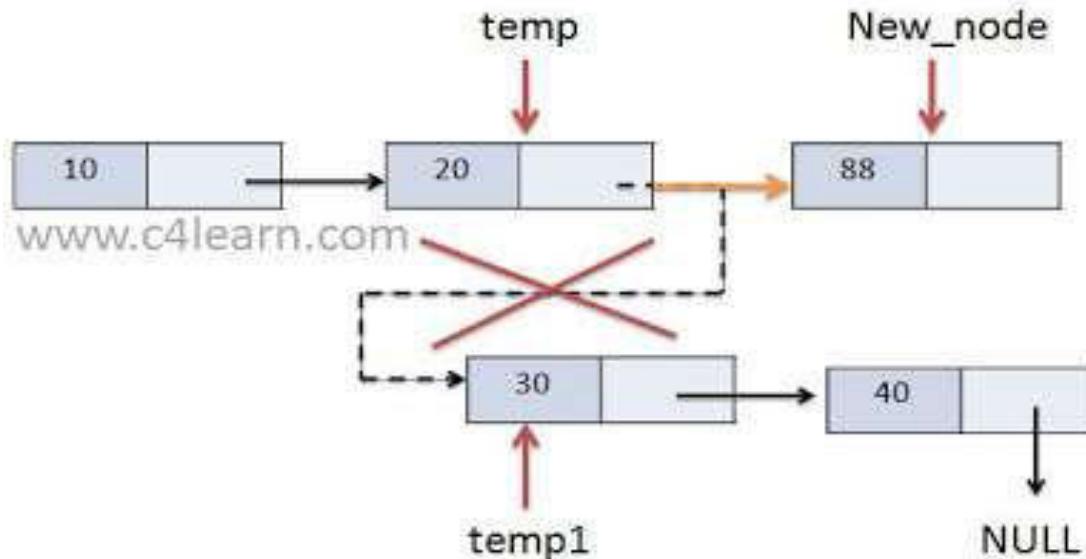
www.c4learn.com



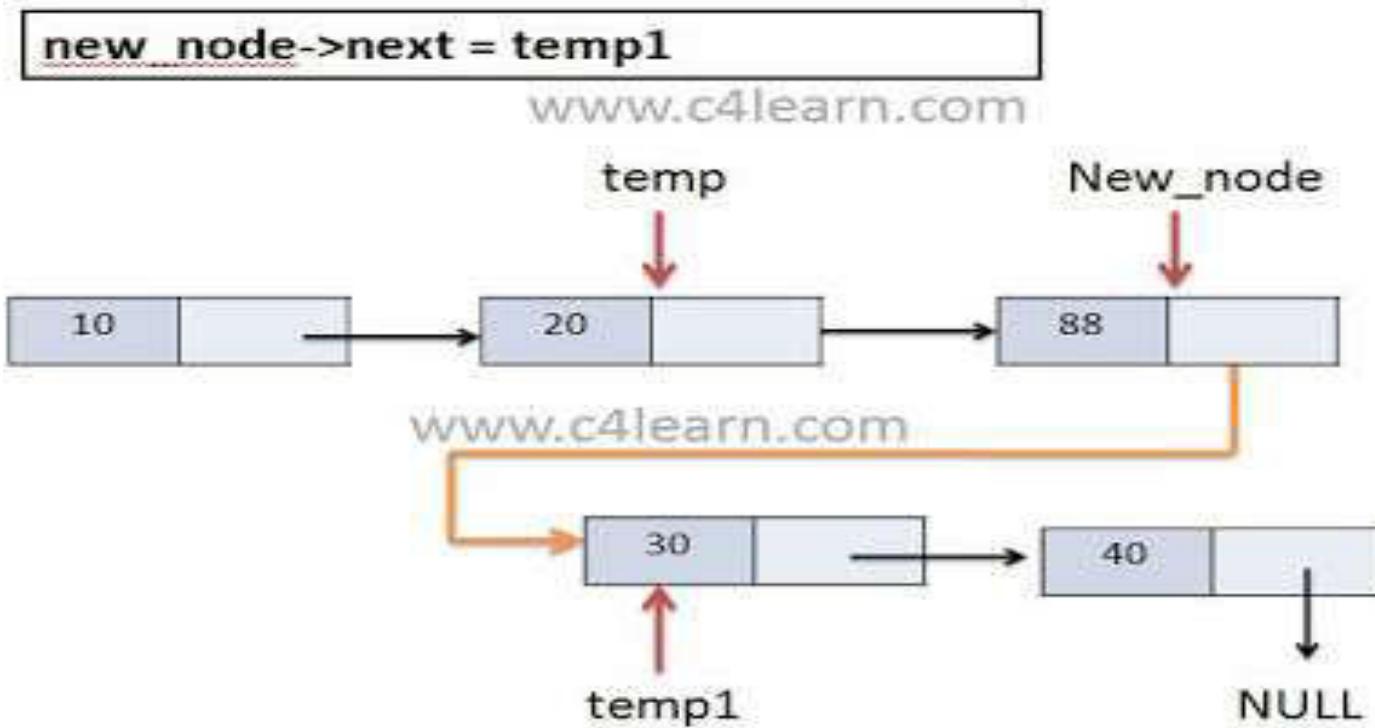
`temp->next = new_node;`

Temp->next = New_node

www.c4learn.com



`new_node->next = temp1`



Deletion of an element from the end of the linked list

DeleteFromend()

{ if (Start==Null)

 display"Deletion not possible"

Else // create a pointer prev and temp

 prev= NULL

 temp= Start

 While(temp->link != NULL)

 { prev= temp

 temp= temp->link

```
If(prev==NULL) // Just one Element in the list  
{ Start=NULL  
}  
  
else  
{ prev->next=NULL // more element in the list  
free(temp)  
}  
}
```

Deleting first node

```
void del_beg()
{
    struct node *temp;
    temp = start;
    start = start->next;
    free(temp);
    printf("The Element deleted Successfully ");
}
```

Searching for an element

```
//provide the value to be searched as num
{  int flag = 0;
struct node *temp;
    temp =start;
while(temp!=NULL)
{  if(temp->data == num)
    return(temp);    //Found Or return(1) – if found
    temp = temp->next;
}
if(flag == 0)
return(start);  // Not found }
```

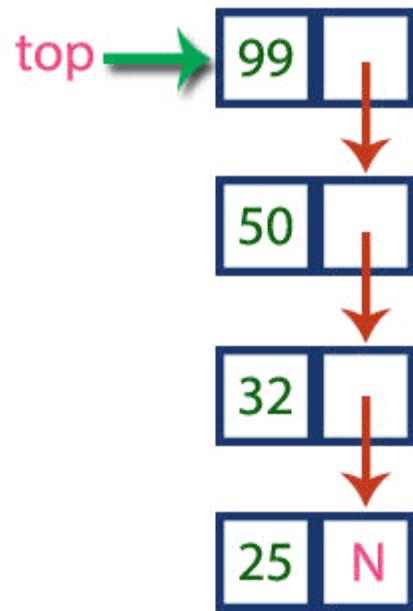
APPLICATION OF LINKED LIST

- STACK IMPLEMENTATION using Linked List
- QUEUE IMPLEMENTATION using Linked List

Stack implementation using Linked list

- Disadvantages of array implementation - it works only for fixed number of data values, size of the array unknown .
- A stack data structure can be implemented by using linked list data structure.
- The stack implemented using linked list can work **for unlimited number of values**. Stack implemented using linked **list works for variable size of data..**
In linked list implementation of a stack, every new element is inserted as '**top**' element. That means every newly inserted element is pointed by '**top**'.
- To remove an element from the stack, remove the node which is pointed by '**top**' by moving '**top**' to its next node in the list.
- The **next** field of the **first element** must be always **NULL**.

- **Example**



- In above example, the last inserted node is 99 and the first inserted node is 25. The order of elements inserted is 25, 32, 50 and 99.

Stack implementation of linked list

```
#include<stdio.h>
#include<conio.h>
// Structure definition
struct Node
{
    int data;
    struct Node *next;
}*top = NULL;
// all operations of STACK
void push(int);
void pop();
void display();
```

```
void main()
{
    int choice, value;

printf("\n:: Stack using Linked List ::\n");
while(1){
    printf("\n***** MENU *****\n");
    printf("1. Push\n2. Pop\n3. Display\n4. Exit\n");
    printf("Enter your choice: ");
    scanf("%d",&choice);
    switch(choice){
        case 1: printf("Enter the value to be insert: ");
                  scanf("%d", &value);
                  push(value);
                  break;
        case 2: pop(); break;
        case 3: display(); break;
        case 4:
        default: printf("\nWrong selection!!! Please try again!!!\n");
    }
}
```

PUSH()

```
void push(int value)
{
    struct Node *newNode;
    newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    if(top == NULL)
        newNode->next = NULL;
    else
        newNode->next = top;

    top = newNode;
    printf("\nInsertion is Success!!!\n");
}
```

POP()

```
void pop()
{
    if(top == NULL)
        printf("\nStack is Empty!!!\n");
    else{
        struct Node *temp = top;
        printf("\nDeleted element: %d", temp->data);
        top = temp->next;
        free(temp);
    }
}
```

Display ()

```
void display()
{
    if(top == NULL)
        printf("\nStack is Empty!!!\n");
    else{
        struct Node *temp = top;
        while(temp->next != NULL){
            printf("%d--->",temp->data);
            temp = temp -> next;
        }
        printf("%d--->NULL",temp->data);
    }
}
```

QUEUE using linked list

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
struct node
```

```
{   int info;
```

```
    struct node *ptr;
```

```
}*front,*rear,*temp,*front1;
```

int frontelement();

void enq(int data);

void deq();

void empty();

void display();

void create();

void queuesize();

int count = 0;

```
main()
```

```
{
```

```
    int no, ch, e;
```

```
    printf("\n 1 - Enque");
```

```
    printf("\n 2 - Deque");
```

```
    printf("\n 3 - Front element");
```

```
    printf("\n 4 - Empty");
```

```
    printf("\n 5 - Exit");
```

```
    printf("\n 6 - Display");
```

```
    printf("\n 7 - Queue size");
```

```
    create();
```

```
    while (1)
```

```
{
```

```
    printf("\n Enter choice : ");
```

```
    scanf("%d", &ch);
```

```
switch (ch)
{
    case 1:
        printf("Enter data : ");
        scanf("%d", &no);
        enq(no);
        break;
    case 2:
        deq();
        break;
    case 3:
        e = frontelement();
        if (e != 0)
            printf("Front element : %d", e);
        else
            printf("\n No front element in
Queue as queue is empty");
        break;
    case 5:
        exit(0);
    case 6:
        display();
        break;
    case 7:
        queuesize();
        break;
    default:
        printf("Wrong choice, Please enter
correct choice ");
        break;
}
```

```
/* Create an empty queue */
```

```
void create()
```

```
{
```

```
    front = rear = NULL;
```

```
}
```

```
/* Returns queue size */
```

```
void queuesize()
```

```
{
```

```
    printf("\n Queue size : %d", count);
```

```
}
```

```
/* Enqueing the queue */

void enq(int data)
{   if (rear == NULL) // First node created
{
    rear = (struct node *)malloc(1*sizeof(struct node));
    rear->ptr = NULL;
    rear->info = data;
    front = rear;
}
else //Already nodes available. We add a node here
{
    temp=(struct node *)malloc(1*sizeof(struct node));
    rear->ptr = temp;
    temp->info = data;
    temp->ptr = NULL;
    rear = temp;
}
count++; }
```

```
/* Dequeing the queue */

void deq()
{
    front1 = front;
    if (front1 == NULL)           / Empty Queue
    {
        printf("\n Error: Trying to display elements from empty queue");
        return;
    }
    else                         /More nodes available .Delete from front
    if (front1->ptr != NULL)
    {
        front1 = front1->ptr;
        printf("\n Dequeued value : %d", front->info);
        free(front);
        front = front1;
    }
    else                         / Only one node available to delete
    {
        printf("\n Dequeued value : %d", front->info);
        free(front);
        front = NULL;
        rear = NULL;
    }   count--;
}
```

```
/* Displaying the queue elements */
```

```
void display()
```

```
{
```

```
    front1 = front;
```

```
    if ((front1 == NULL) && (rear == NULL))
```

```
{
```

```
    printf("Queue is empty");
```

```
    return;
```

```
}
```

```
    while (front1 != rear)
```

```
{
```

```
    printf("%d ", front1->info);
```

```
    front1 = front1->ptr;
```

```
}
```

```
    if (front1 == rear)
```

```
        printf("%d", front1->info);
```

```
}
```

Adding TWO polynomials

A polynomial in a single indeterminate x can always be written

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_2 x^2 + a_1 x + a_0,$$

where a_0, \dots, a_n are constants and x is the indeterminate. TI

- To add two polynomial $P(x)+Q(x)$ we can use linked list
- To create a node : 3 information required

Co-efficient

Exponent

Link field

Definition of Structure

Struct Poly

```
{ int coeff;
```

```
int Exp;
```

```
Struct poly *ptr;};
```

- **Algorithm:**
 1. Read no of terms in the first polynomial
 2. Read coefficient in first polynomial
 3. Read no of terms in the Second polynomial
 4. Read coefficient in Second polynomial
 5. Set temp pointers p and q to traverse two polynomial
 6. Compare the exponents of two Polynomials starting from first node.

- While performing the addition these conditions to be taken care of.
 1. If BOTH exponents are equal then add coefficient and store in the RESULT linked list.
 $p=p+1$ and $q=q+1$ ie., Move pointers of p&q to point NEXT NODE
 2. If exponent of P< exponent of Q then ADD terms of Q in the RESULT and Move q to point NEXT NODE. [$q=q+1$]
 3. If Exponent of P>Exponent of Q then ADD terms of P in the RESULT and Move p to point NEXT NODE [$p=p+1$]

$$P(x) = 3x^3 + 2x + 7$$

$$Q(x) = 5x^2 + 3x + 5$$

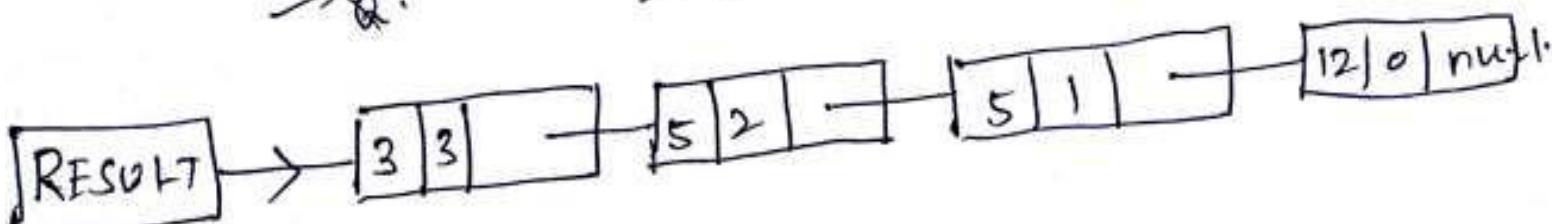
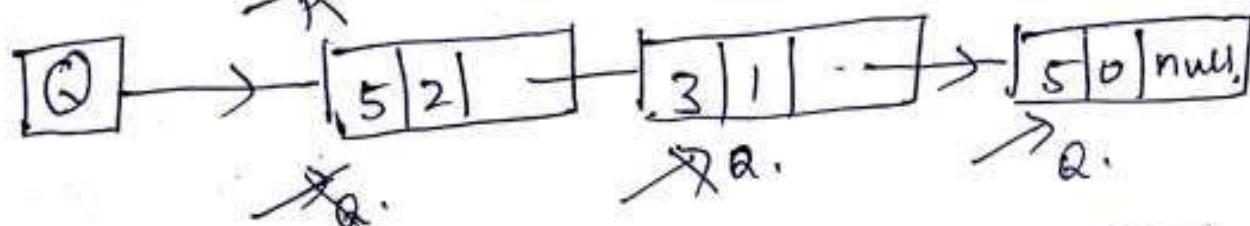
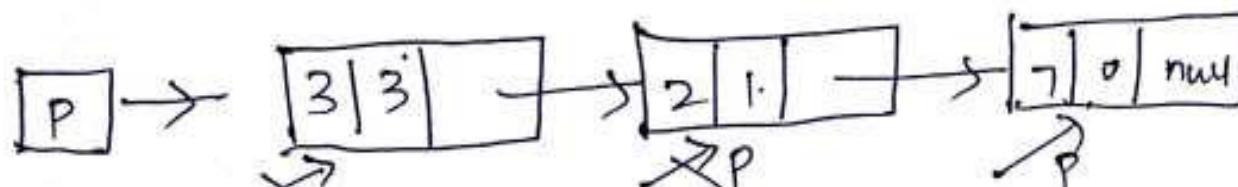
1. $P > Q \quad P = P + 1$

2. $Q > P \quad Q = Q + 1$

3. $P = Q$ Add coefficients

$$\begin{aligned} 2+3 &= 5 \\ P = P + 1 &\text{ & } Q = Q + 1 \end{aligned}$$

4. $P = Q \cdot 7 + 5 = 12$



Substitution method:

- we guess a bound and then use mathematical induction to prove our guess correct.

TWO STEPS

1. Guess the form of the solution
 2. Use mathematical induction to find constants and show that the solution works.
- This method can be applied only in cases when it is easy to guess the form of the answer.
 - This method used to establish upper or lower bounds on a recurrence.

Substitution Method.

$$T(n) = 2T\left(\frac{n}{2}\right) + An \rightarrow ①$$

Assume $T(1) = 4$

Build solution

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + An \\ &= 2^1 [2T\left(\frac{n}{2^2}\right) + A\frac{n}{2}] + An \\ &= 2^2 [2T\left(\frac{n}{2^3}\right) + A\frac{n}{2^2} + An] \\ &= 2^3 [2T\left(\frac{n}{2^4}\right) + A\frac{n}{2^3} + A\frac{n}{2^2} + An] \\ &= 2^4 [2T\left(\frac{n}{2^5}\right) + A\frac{n}{2^4} + A\frac{n}{2^3} + A\frac{n}{2^2} + An] \\ &= 2^i T\left(\frac{n}{2^i}\right) + i(An) \rightarrow ② \end{aligned}$$

$$T(n) = \begin{cases} 4 & n=1 \\ 2T\left(\frac{n}{2}\right) + An & \text{otherwise} \end{cases}$$

Expand

$$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{2^2}\right) + A\left(\frac{n}{2}\right)$$

$$T\left(\frac{n}{2^2}\right) = 2T\left(\frac{n}{2^3}\right) + A\left(\frac{n}{2^2}\right)$$

$$T\left(\frac{n}{2^3}\right) = 2T\left(\frac{n}{2^4}\right) + A\left(\frac{n}{2^3}\right)$$

Now let $\frac{n}{2^i}=1$, since we've taken $T(1)=4$, $n=1$

$$\therefore \frac{n}{2^i}=1 \Rightarrow n=2^i \Rightarrow \text{taking log we get}$$

$$\log_2 n = i$$

Sub i in eq ② we get

$$2^i T\left(\frac{n}{2^i}\right) + i(An) \text{ becomes}$$

~~$$2^{\log_2 n} T(1) + \log_2 n (An)$$~~

~~$$2^{\log_2 n} T(1) + \log_2 n (An) \rightarrow ③$$~~

But $2^{\log_2 n}$ can be written as $n^{\log_2 2} = n^1 = n$

Eqr. ③ becomes

$$n^1 A + An \log_2 n$$

$$\Rightarrow An + An \log_2 n \Rightarrow O(n \log n)$$

$$[a^{\log_b c} = c^{\log_b a}]$$

$$2. T(n) = T(n-1) + n$$

$$\Rightarrow T(1) = 1 \quad \text{--- ①}$$

$$T(n) = \begin{cases} 1 & \text{if } n=1 \\ T(n-1) + n & \end{cases}$$

$$T(n) = \underbrace{T(n-1)}_{\dots} + n$$

$$= T(n-2) + n-1 + n$$

$$= T(n-3) + n-2 + n-1 + n$$

! ! !

$$= T(n-k) + (n-(k-1)) + (n-(k-2)) + \dots + (n-2) + (n-1) + n$$

$$n-k=1 \quad \text{by ①}$$

$$= T(1) + \underbrace{n-k+1}_{1} + n-k+2 + \dots + n$$

~~We know that~~

$$T(1) = 1 \quad \text{and} \quad n-k=1 \Rightarrow n-k+1 = 1+1=2$$

$$\therefore T(1) + 2 + 3 + \dots + n$$

$$= 1 + 2 + 3 + \dots + n$$

$$= \frac{n(n+1)}{2} = O(n^2)$$

3.

$$T(n) = \begin{cases} 1 & \text{if } n=1 \\ T(n-1)*n & \end{cases} \Rightarrow T(1)=1$$

$$T(n) = \underbrace{T(n-1)}_{\dots} * n$$

$$= T(n-2) * (n-1) * n$$

$$= T(n-3) * (n-2) * (n-1) * n$$

⋮

$$T(n-k) * (n-(k-1)) * (n-(k-2)) \dots \\ (n-1) * n$$

We know that $n-k=1 \Rightarrow$

$$\text{if } n-k=1 \quad n-k+1=1+1=2$$

$$\& T(n-k)=1$$

$$= T(1) * 2 * 3 * 4 \dots (n-1) * n$$

$$= 1 * 2 * 3 * 4 \dots (n-1) * n$$

$$= n!$$

$$= O(n!)$$

$$4. T(n) = \begin{cases} 1 & \text{if } n=1 \\ T(n-1) + \gamma_n & \end{cases} \Rightarrow T(1)=1$$

$$T(n) = T(n-1) + \gamma_n$$

$$= T(n-2) + \gamma_{n-1} + \gamma_n$$

$$= T(n-3) + \gamma_{n-2} + \gamma_{n-1} + \gamma_n$$

$$\vdots$$

$$= T(n-k) + \gamma_{n-(k-1)} + \gamma_{n-(k-2)} + \dots + \gamma_n$$

$$\text{here } n-k=1, T(1)=1 \Rightarrow$$

$$= T(1) + \gamma_2 + \gamma_3 + \dots + \gamma_n$$

$$= \underbrace{1 + \gamma_2 + \gamma_3 + \dots + \gamma_n}_{\text{logarithmic series}} = O(\log_2 n)$$

$$= O(\log_2 n)$$

$$5. T(n) = \begin{cases} 1 & \text{if } n=1 \\ 2T(n/2) + n & \end{cases} \Rightarrow T(1)=1$$

$$T(n) = 2T(n/2) + n$$

~~2 times~~

$$= 2(2T(n/4) + n/2) + n$$

$$= 2^2 T(n/4) + n/2 + n$$

$$= 2^2 [2T(n/8) + n/4] + n/2 + n$$

$$= 2^3 T(n/16) + n/4 + n/2 + n$$

\vdots

$$= 2^K T(n/2^K) + n/2^K + n/4^K + \dots \text{--- k times}$$

$$\text{Let } n/2^K = 1 \Rightarrow n = 2^K$$

$$\log_2 n = K.$$

$$= 2^{\log_2 n} T(1) + n * K.$$

$$= n \times 1 + kn \Rightarrow kn + n \log_2 n \Rightarrow O(n \log_2 n)$$

$$T(n-1) = T(n-2) + \gamma_{n-1}$$

$$T(n-2) = T(n-3) + \gamma_{n-2}$$

$$\vdots$$

$$T(n-k) = T(n-(k+1)) + \gamma_{n-(k+1)}$$

$$\vdots$$

$$T(1) = T(1) + \gamma_2 + \gamma_3 + \dots + \gamma_n$$

$$= T(1) + \underbrace{\gamma_2 + \gamma_3 + \dots + \gamma_n}_{\text{logarithmic series}} = O(\log_2 n)$$

$$= T(1) + O(\log_2 n) = O(\log_2 n)$$

$$= 1 + O(\log_2 n) = O(\log_2 n)$$

$$= O(\log_2 n)$$

Proof of correctness

Module -1

Algorithm (topics)

- Design of Algorithms (for simple problems)
- Analysis of algorithms
 - – Is it correct?
 - Loop invariants**
 - *a loop invariant* is a property of a *program loop that is true before (and after) each iteration*
 - – Is it “good”?
 - Efficiency**
 - – Is there a better algorithm?
 - Lower bounds**

State of Computation

- Most programming algorithms are based on the notion of transforming the algorithm to outputs

Loop Invariant

- The state of computation may be defined by examining the contents of *key variables* before and after the execution of each statement.

Loop Invariant : $x + y = x_0 + y_0$.

We want to prove that $x + y = x_0 + y_0$ is a *loop invariant* of the program. In other words, we want to prove that $x + y = x_0 + y_0$ is always true no matter how many time the loop executed. Consider the following predicate.

$L(n)$: If the program reaches point a after the loop has been executed n times,
then $x + y = x_0 + y_0$.

Inductive Base: $L(0)$

If the loop has not been executed, then $x = x_0$ and $y = y_0$. Therefore, $x + y = x_0 + y_0$.
 $L(0) = \text{true}$. The inductive base holds.

Inductive Hypothesis: Suppose $L(n) = \text{true}$.

Inductive Step: $L(n + 1)$

Suppose at the moment t_0 , the loop has been executed n times. Since $L(n)$ is true, $x + y = x_0 + y_0$. And, Suppose at this moment t_0 , $x = k$ and $y = l$, i.e, at the moment t_0 we have $x + y = x_0 + y_0 = k + l$.

If the loop will be executed one more time, then x will be decreased by 1 and y will be increased by 1, and then the program will reach point a at the moment t_1 . At this moment t_1 , $x + y = (k - 1) + (l + 1) = k + l = x_0 + y_0$. Therefore, $L(n + 1) = \text{true}$.

Assertions

- Assertions are facts about the state of the program variables
- It is wasteful to spend your time looking at variables that are not effected by a particular statement
- Default assertion
 - any variable not mentioned in the assertion for a statement do not affect the state of computation

Use of Assertions

- Pre-condition
 - assertion describing the state of computation before statement is executed
- Post condition
 - assertion describing the state of computation after a statement is executed
- Careful use of assertions as program comments can help control side effects

1. Simple Algorithm- Sum of two integers , Swapping of Two Numbers

- **Model {P} A {Q}**
 - P = pre-condition
 - A = Algorithm
 - Q = post condition
- **Sum of two numbers algorithm**
{pre: $x = x_0$ and $y = y_0$ }
$$z = x + y$$

{post: $z = x_0 + y_0$ }

Sequence Algorithm

- **Model**

if $\{P\} A_1 \{Q_1\}$ and $\{Q_1\} A_2 \{Q\}$

then $\{P\} A_1 ; A_2 \{Q\}$ is true

- **Swap algorithm**

{pre: $x = x_0$ and $y = y_0$ }

temp = x

$x = y$

$y = \text{temp}$

{post: $\text{temp} = x_0$ and $x = y_0$ and $y = x_0$ }

Intermediate Assertions

- Swap algorithm

{pre: $x = x_0$ and $y = y_0$ }

temp = x

{temp = x_0 and $x = x_0$ and $y = y_0$ }

x = y

{temp = x_0 and $x = y_0$ and $y = y_0$ }

y = temp

{post: temp = x_0 and $x = y_0$ and $y = x_0$ }

2. Conditional Statements- Absolute value

- **Absolute value**

{pre: $x = x_0$ }

if $x < 0$ then

$y = -x_0$

else

$y = x_0$

{post: $y = |x_0|$ }

Intermediate Assertions

if $x < 0$ **then**

{ $x = x_0$ and $x_0 < 0$ }

$y = -x_0$

{ $y = |x_0|$ }

else

{ $x = x_0$ and $x_0 \geq 0$ }

$y = x_0$

{ $y = |x_0|$ }

3. Proving the correctness of Algorithm- Sequential Search

- **Sequential Search**
- 1. $\text{index} = 1;$
- 2. While $\text{index} \leq n$ and $L[\text{index}] \neq x$ do
 - $\text{index} = \text{index} + 1;$
- 3. if $\text{index} > n$ then $\text{index} = 0;$

Defining the I/O of the algorithm

- **Input :** Given an array L containing n items ($n \geq 0$) and given x ,
- **Output:**
 - the *sequential search* algorithm terminates
 - with *index = first occurrence of x in L* , if found and, *index = 0* otherwise.

Loop Invariant

- **Sequential Search**
 1. $\text{index} = 1;$
 2. While $\text{index} \leq n$ and $L[\text{index}] \neq x$ do
 $\text{index} = \text{index} + 1;$
 3. if $\text{index} > n$ then $\text{index} = 0;$
- **Hypothesis H:** For $1 \leq k \leq n + 1$, $L(k)$: when the control reaches the test in line 2 for **kth** time
 - $\text{index} = k$ and,
 - for $1 \leq i \leq k-1$, $L[i] \neq x$.
- Prove the above hypothesis by induction.

Proving the Hypothesis by induction

- **Base Case :** $H[1]$ is true vacuously.

Let $H(k)$ be true

- We will prove that $H(k+1)$ is also true.

Control reaches the test in line 2 for $(k+1)$ th time only

if $L(k) \neq x$ (i)

Also since $H(k)$ is true

$1 \leq i \leq k-1, L[i] \neq x$ (ii)

from (i) and (ii) we get,

$1 \leq i \leq k, L[i] \neq x$

\therefore it holds for index = $k+1$

Thus by induction our hypothesis is true.

Correctness contd..

- Suppose the test condition is executed exactly k times. i.e. body of the loop is executed $k-1$ times.
 - Case 1: $k = n+1$, by loop invariant hypothesis, $\text{index} = n+1$ and for $1 \leq i \leq n$, $L[i] \neq x$.
Since $\text{index} = n+1$, line 3 sets index to 0 and by second condition above x is not in the array. So correct.
 - Case 2: $k \leq n, \Rightarrow \text{index} = k$ and loop terminated because $L[k] = x$. Thus index is the position of the first occurrence of x in the array.

Hence the algorithm is correct in either case.

4. How do you find the **max** of n numbers (stored in array A?)

- INPUT: $A[1..n]$ - an array of integers
 - OUTPUT: an element m of A such that $A[j] \leq m$, $1 \leq j \leq \text{length}(A)$
-
- Find-max (A)
 1. $\text{max} \leftarrow A[1]$
 2. $\text{for } j \leftarrow 2 \text{ to } n$
 3. $\text{do if } (\text{max} < A[j])$
 4. $\text{max} \leftarrow A[j]$
 5. return max

Reasoning (formally) about algorithms

- 1. I/O specs: Needed for correctness proofs, performance analysis.
 - INPUT: $A[1..n]$ - an array of integers
 - OUTPUT: an element m of A such that $A[j] \leq m$, $1 \leq j \leq \text{length}(A)$
- 2. CORRECTNESS: The algorithm satisfies the output specs for EVERY valid input
- 3. ANALYSIS: Compute the running time, the space requirements, number of cache misses, disk accesses, network accesses,....

Correctness proofs of algorithms

Find-max (A)

```
1. max ← A[1]
2. for j ← 2 to length(A)
3.   do if (max < A[j])
4.     max ← A[j]
5. return max
```

- Prove that for any valid Input, the output of Find-max satisfies the output condition.
- **Proof by contradiction:**
 - Suppose the algorithm is incorrect.
 - Then for some input A,
 - **Case 1:** max is not an element of A. max is initialized to and assigned to elements of A – which is impossible.
 - **Case 2:** ($\exists j \mid \text{max} < A[j]$).
After the jth iteration of the for-loop (lines 2 – 4), $\text{max} \geq A[j]$. From lines 3,4, max only increases.
 - Therefore, upon termination, $\text{max} \geq A[j]$, which contradicts our assumption that $\text{max} < A[j]$

Loop invariant proofs

Find-max (A)

1. $\text{max} \leftarrow A[1]$
2. $\text{for } j \leftarrow 2 \text{ to } \text{length}(A)$
3. $\text{do if } (\text{max} < A[j])$
4. $\text{max} \leftarrow A[j]$
5. return max

- Prove that for any valid Input, the output of Find-max satisfies the output condition.
- Proof by **loop invariants**:
 - Loop invariant: $I(j)$: At the beginning of iteration j of the loop, max contains the maximum of $A[1,..,j-1]$.
 - Proof:
 - True for $j=2$.
 - Assume that the loop invariant holds for the j iteration,
So at the beginning of iteration k , $\text{max} = \text{maximum of } A[1,..,j-1]$.

Loop invariant proofs

Find-max (A)

```
1. max ← A[1]
2. for j ← 2 to length(A)
3.   do if (max < A[j])
4.     max ← A[j]
5. return max
```

- For the $(j+1)$ th iteration
 - **Case 1:** $A[j]$ is the maximum of $A[1, \dots, j]$. In lines 3, 4, max is set to $A[j]$.
 - **Case 2:** $A[j]$ is not the maximum of $A[1, \dots, j]$. So the maximum of $A[1, \dots, j]$ is in $A[1, \dots, j-1]$. By our assumption max already has this value and by lines 3-4 max is unchanged in this iteration.

Loop invariant proofs

- STRATEGY: We proved that the invariant holds at the beginning of iteration j for each j used by Find-max.
 - Upon termination, $j = \text{length}(A)+1$. (WHY?)
 - The invariant holds, and so max contains the maximum of $A[1..n]$

Loop invariant proofs

- ▶ Advantages:
 - Rather than reason about the whole algorithm,
reason about SINGLE iterations of SINGLE loops.
- ▶ Structured proof technique
- ▶ Usually prove loop invariant via Mathematical Induction.

Algorithm Correctness Example

- Suppose you have a software component that accepts as input an array **T** of size **N**
- As output the component produces an **T'** which **contains the elements of T arranged in ascending order using Bubble Sort.**
- **Binary Search**
- How would we convert the code to its logical counterpart and prove its correctness?

Assertions, pre & post conditions

- Syntax: assert(bool exp)
- Pre condition
 - Requires a bool condition satisfied
- Post condition
 - Ensures a bool condition satisfied

```
require (y>=0)
int f(int x, int y) {
    assert(x>=0)
    x += y;
    x *= x;
    return x;
} ensures (?)
```

How can we show?

A loop terminates

A loop produces the desired output

Loop Invariant

A bool statement that is true at the start of the loop

at the end of each iteration

Why Study loop invariants?

To prove properties of loops

&

to prove partial correctness of
loops

(equivalent to induction)

Examples

Simple Examples

$i = 0; x = 0;$

while ($i < n$) {

$i = i + 1;$

$x = x + i;$

}

\rightarrow ($i \leq n$) ~~bb~~ ($i > n$)
 \Rightarrow ($i = n$)

$P_0: i = i \checkmark ?$

$P_1: i = i \checkmark ?$

$P_2: i < n \times$

$i = 0 < n \checkmark$

$i = i + 1 < n \times$

$P_3: i \leq n$

$i = 0 \leq n \checkmark$

$i = i + 1 \leq n$

since $i < n$



3:54 / 10:50



Finding a loop invariant

Finding a loop invariant

```
int g (int x) {  
    int i = 0;  
    int j = 0;  
    int k = 0;  
    while (j < x) {  
        j = j+k+3*i+1;  
        k = k+6*i+3;  
        i = i+1;  
    }  
    return i;  
}
```

$$g(0)$$

$$g(1)$$

$$g(2)$$

$$\vdots$$

X	i	j	K
4	0	0	0
1	1		3
2	8		12

$$i^3 = j$$

$$K = 3 \cdot i^2$$

Proving the loop invariant

Proving the loop invariant

```
int g (int x) {
    int i = 0;
    int j = 0;
    int k = 0;
    while (j < x) {
        j = j+k+3*i+1;
        k = k+6*i+3;
        i = i+1;
    }
    return i;
}
```

The handwritten proof shows the derivation of the loop invariant $i^3 = j$ and $k = 3i^2$. It starts with the initial state $i^0 = 0$, $j^0 = 0$, and $k^0 = 0$. The first iteration leads to $i^1 = 1$, $j^1 = 1$, and $k^1 = 3$. The proof then shows that after each iteration, $j^l = (l+1)$ and $k^l = j^l + 3i^l + 1$. This is derived from the recurrence relations: $j^l = j^{l-1} + k^l + 3i^l + 1$ and $k^l = k^{l-1} + 6i^l + 3$. The final step shows that $k^l = 3i^l$, which is equivalent to $(k^l + 6i^l + 3) = 3(i^l + 1)^2$.

$$i^0 = 0$$
$$j^0 = 0$$
$$k^0 = 0$$
$$i^1 = 1$$
$$j^1 = 1$$
$$k^1 = 3$$
$$i^l = l+1$$
$$j^l = l+1$$
$$k^l = j^l + 3i^l + 1$$
$$k^l = 3i^l$$
$$(k^l + 6i^l + 3) = 3(i^l + 1)^2$$
$$3i^l + 6i^l + 3 = 3i^l + 6i^l + 3 \checkmark$$

Proving loop termination

- Since J increases and $J \geq x$ the loop terminates.

Post condition from loop invariant

Loop invariant & post condition

$$i^3 = j$$

$$K = 3i^2$$

$$\begin{array}{c} j < x \\ \parallel \\ i^3 < x \end{array}$$

MS 101: Algorithms

Instructor

Neelima Gupta

ngupta@cs.du.ac.in

Proving Correctness of Algorithms

Proving the correctness of Algorithm

Sequential Search

1. $\text{index} = 1;$
2. While $\text{index} \leq n$ and $L[\text{index}] \neq x$ do
 $\quad \text{index} = \text{index} + 1;$
3. if $\text{index} > n$ then $\text{index} = 0;$

Defining the I/O of the algorithm

- Input : Given an array L containing n items ($n \geq 0$) and given x,
- Output:
 - the sequential search algorithm terminates
 - with $\text{index} = \text{first occurrence of } x \text{ in } L$, if found
 - and, $\text{index} = 0$ otherwise.

Loop Invariant

- Hypothesis: For $1 \leq k \leq n + 1$, $L(k)$: when the control reaches the test in line 2 for k th time
 - index = k and,
 - for $1 \leq i \leq k-1$, $L[i] \neq x$.
- Prove the above hypothesis by induction

Proving the Hypothesis by induction

- Base Case : $H[1]$ is true vacuously.
- Let $H(k)$ be true
- We will prove that $H(k+1)$ is also true.

control reaches the test in **line 2** for $(k+1)$ th time only if
 $L(k) \neq x$ (i)

Also since $H(k)$ is true

$1 \leq i \leq k-1, L[i] \neq x$ (ii)

from (i) and (ii) we get,

$1 \leq i \leq k, L[i] \neq x$

\therefore it holds for index = $k+1$

Thus by induction our hypothesis is true.

Correctness contd..

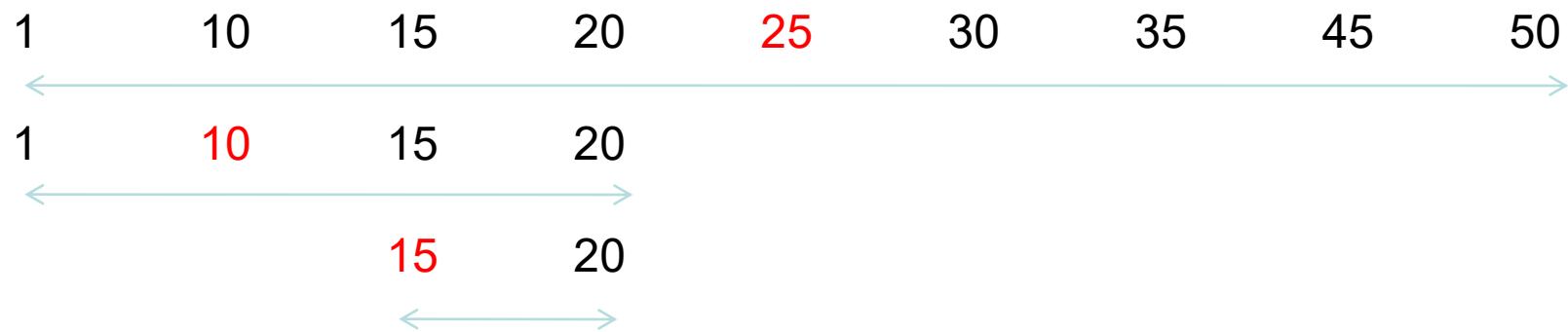
- Suppose the test condition is executed exactly k times. i.e. body of the loop is executed $k-1$ times.
 - Case 1: $k = n+1$, by loop invariant hypothesis, index = $n+1$ and for $1 \leq i \leq n$, $L[i] \neq x$.
Since index = $n+1$, line 3 sets index to 0 and by second condition above x is not in the array. So correct.
 - Case 2: $k \leq n$, => index = k and loop terminated because $L[k] = x$. Thus index is the position of the first occurrence of x in the array.
Hence the algorithm is correct in either case.

Binary Search

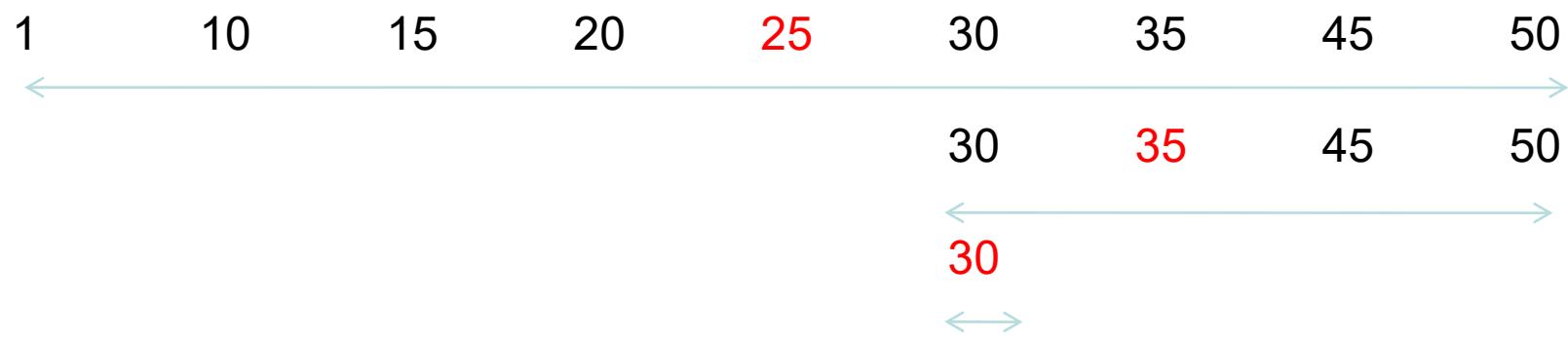
- Input : Given an array L containing n items ($n \geq 0$) ordered such that $L(1) \leq L(2) \leq L(3) \leq \dots \leq L(n)$ and given x,
- Output:
 - The binary search algorithm terminates
 - with index = an occurrence of x in L, if found
 - and, index = 0 otherwise.

Binary search can only be applied if the array to be searched is already sorted.

Search for the no. 15



Search for the no. 30



```
1. index_first = 1;
2. index_last=n;
3. While index_first<=index_last
4. do
5.     index_mid= floor((index_first+index_last)/2);
6.     if L[index_mid]=x
7.         Exit loop
8.     if L[index_mid]>x
9.         index_last=index_mid-1;
10.    if L[index_mid]<x
11.        index_first=index_mid+1;
12.
13. If index_first>index_last then
14.     index_mid=0;
```

Proof of correctness

The Loop Invariant

Let r be the maximum number of times the loop starting from line 3 will run.

- **Hypothesis:** For $1 \leq k \leq r + 1$, $H(k)$: when the control reaches the test in line 3 for k^{th} time
 - $L[i] \neq x$ for every $i < \text{first}$ & for every $i > \text{last}$
- Prove the above hypothesis by induction

Correctness of the algorithm assuming the hypothesis (the loop invariant)

Suppose the test condition is executed t times.

- **Case 1:** $\text{first} \leq \text{last}$. We exit from the loop because $L[m_t] = x$. Thus, mid is a position of occurrence of x as $mid = m_t$.
- **Case 2:** If $\text{first} > \text{last}$, we exit from the loop starting at line 3. Line 13 sets the value of mid to 0 since $\text{first} > \text{last}$. By loop invariant hypothesis, for $i < \text{first}$ and $i > \text{last}$, $L[i] \neq x$ i.e. x is not found in the array. The algorithm correctly returns 0 in mid .

Hence assuming that the statement $H(k)$ is correct the algorithm works correctly.

Proof of Induction Hypothesis

Let f_i and l_i be the values of first and last when the test condition at line 3 is executed for the i^{th} time with f_1 and l_1 being 1 and n respectively.

- Assume that the statement is true for $H(k)$
- We will prove that it is true for $H(k+1)$

In the k^{th} iteration

either first **was set to** $(f_k + l_k)/2 + 1$

i.e. $f_{k+1} = (f_k + l_k)/2 + 1$ [Index[mid]+1]

or last was set to $(f_k + l_k)/2 - 1$

i.e $l_{k+1} = (f_k + l_k)/2 - 1$

Proof of Induction Hypothesis contd..

- **Case 1:** $f_{k+1} > l_{k+1}$ (only if $f_k = l_k$ and $L[(f_k + l_k)/2] \neq x$)

By induction hypothesis

$$L[i] \neq x \quad \forall i < f_k$$

$$L[i] \neq x \quad \forall i > l_k (= f_k)$$

And, also $L[f_k] = L[(f_k + l_k)/2] \neq x$

So x is not present, hence trivially

$$L[i] \neq x \quad \forall i < f_{k+1}$$

$$L[i] \neq x \quad \forall i > l_{k+1}$$

Thus $H(k+1)$ is true.

Proof of Induction Hypothesis contd..

Case 2: $f_{k+1} \leq l_{k+1}$

a:) when $L[(f_k + l_k)/2] < x$

then $f_{k+1} = (f_k + l_k)/2 + 1$

$\forall i < f_k + 1, L[i] \leq L[(f_k + l_k)/2] < x$ (Why?)

Proof of Induction Hypothesis contd..

Case 2: $f_{k+1} \leq l_{k+1}$

a:) when $L[(f_k + l_k)/2] < x$

then $f_{k+1} = (f_k + l_k)/2 + 1$

$\forall i < f_k + 1, L[i] \leq L[(f_k + l_k)/2] < x$ (input is sorted)

i.e. $L[i] \neq x \quad \forall i \leq f_{k+1} - 1$ or $\forall i < f_{k+1}$

Also, by induction hypothesis

$L[i] \neq x \quad \forall i > l_k (= l_{k+1})$

Thus $H(k+1)$ is true.

Proof of Induction Hypothesis contd..

b:) When $L[(f_k + l_k)/2] > x$

then $l_{k+1} = (f_k + l_k)/2 - 1$

$\forall i > l_{k+1}$, $L[i] \geq L[(f_k + l_k)/2] > x$ (Why?)

i.e. $L[i] \neq x \quad \forall i > l_{k+1}$

Proof of Induction Hypothesis contd..

b:) When $L[(f_k + l_k)/2] > x$

then $l_{k+1} = (f_k + l_k)/2 - 1$

$\forall i > l_{k+1}$, $L[i] \geq L[(f_k + l_k)/2] > x$ (Why?)

i.e. $L[i] \neq x \quad \forall i > l_{k+1}$

Also, by induction hypothesis

$L[i] \neq x \quad \forall i < f_{k+1} (= f_k)$

Hence $H(k+1)$ is true.

This proves that our hypothesis is correct

Assignment 5

- Show that Insertion sort works correctly by using Induction.

Hashing

WHY?

- Time taken for a search depends on the **size of the collection**

- Binary Search $O(\log n)$
- Linear Search $O(n)$

Hashing:

- Storage & retrieval of data in an **average time**, does not depend on the collection size { **$O(1)$** }

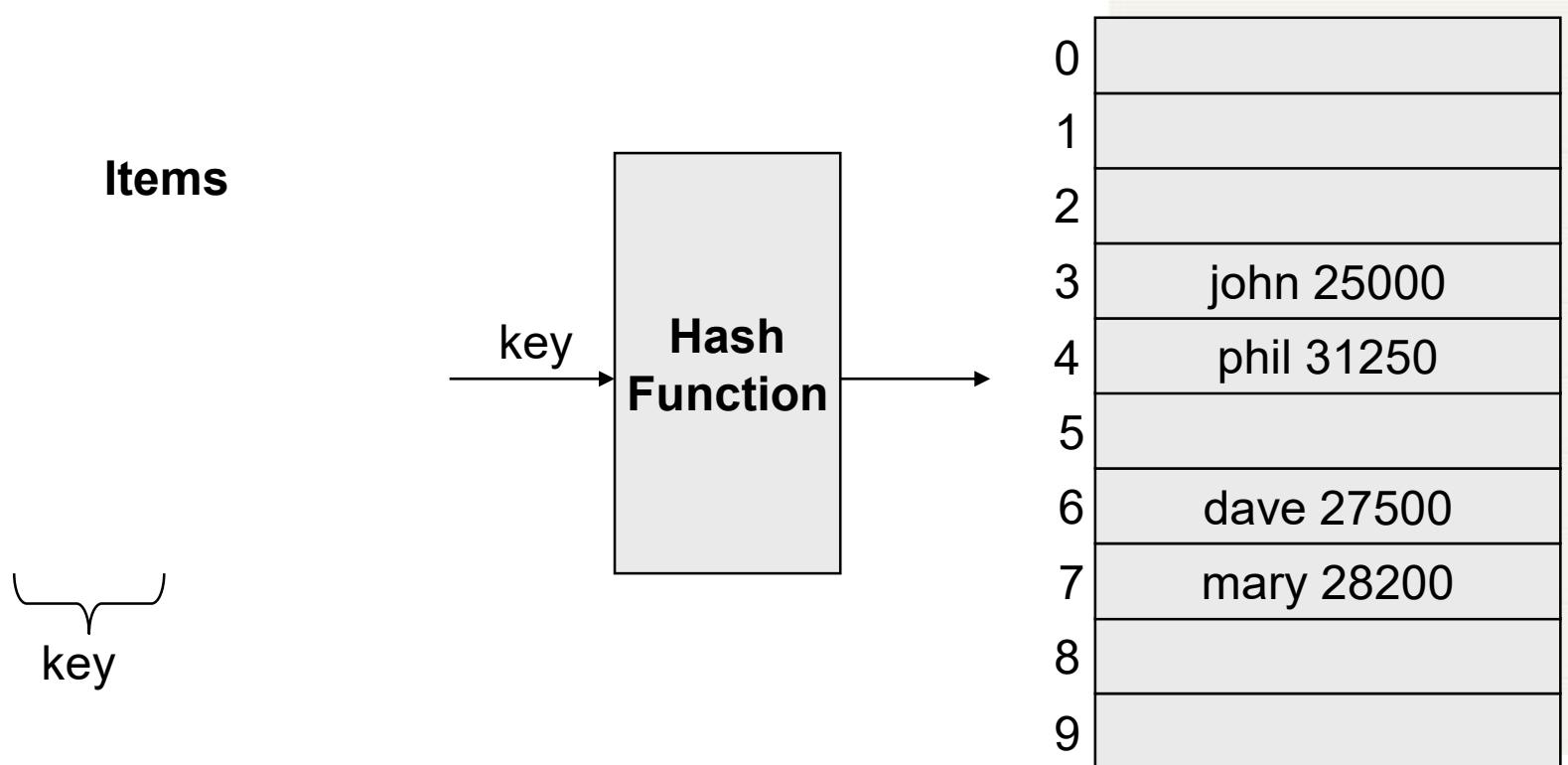
- **Compare Linear Search** – time for searching
 $O(n)$
- **Binary Search** – elements in the sorted order
Time of search $O(\log n)$
Disadvantages- ELEMENTS TO BE SORTED

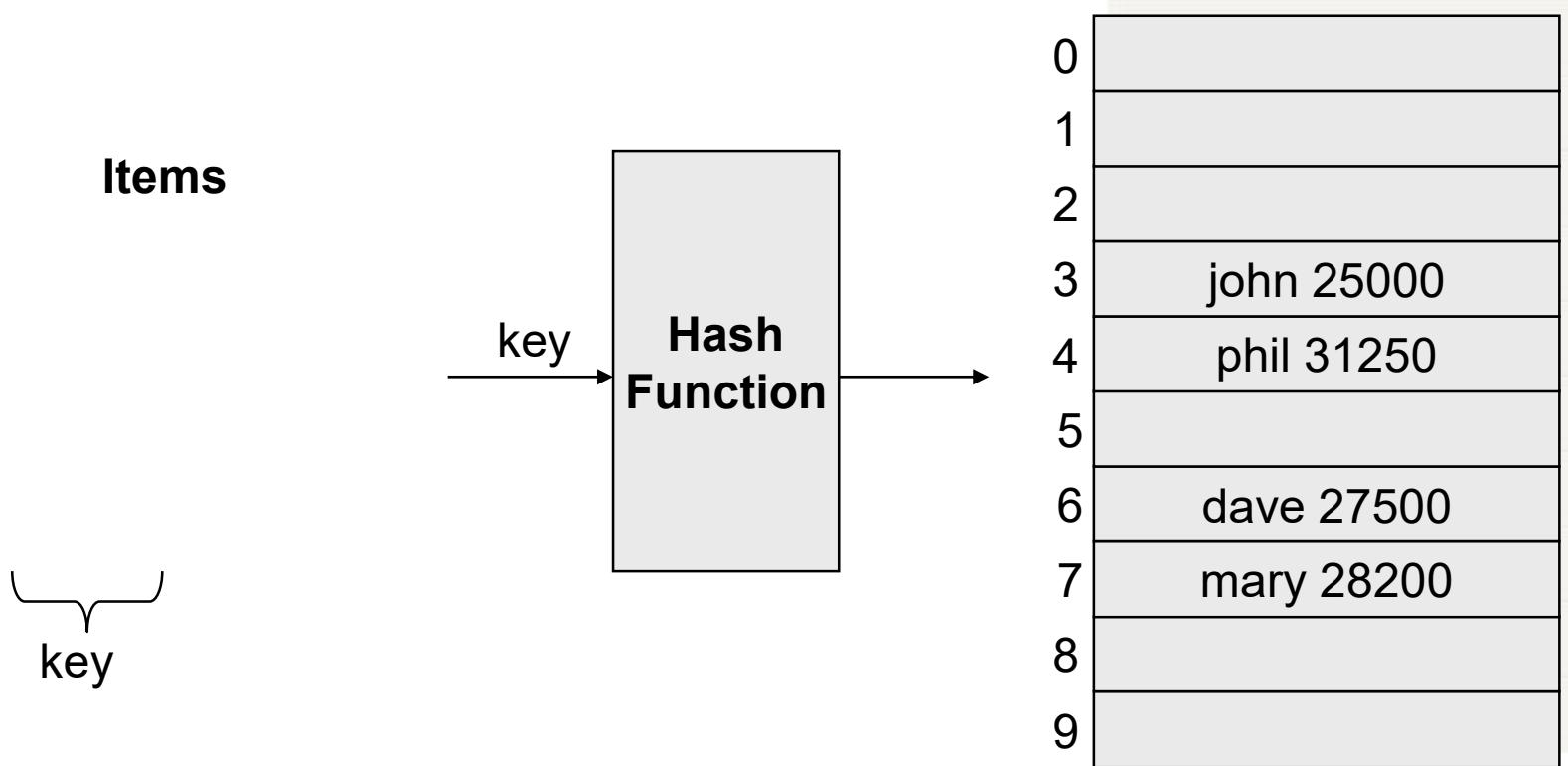
- A Hash table is a DS that implements an associated array Data type .

General Idea

- The ideal hash table structure is merely an array of some fixed size, containing the items.
- A stored item needs to have a data member, called ***key***, that will be used in computing the index value for the item.
 - Key could be an *integer*, a *string*, etc
 - e.g. a name or Id that is a part of a large employee structure
- The size of the array is *TableSize*.
- The items that are stored in the hash table are indexed by values from 0 to *TableSize* – 1.
- Each key is mapped into some number in the range 0 to *TableSize* – 1.
- The mapping is called a ***hash function***.

Example





Hash Function

- The hash function:
 - must be simple to compute.
 - must distribute the keys evenly among the cells.
- If we know which keys will occur in advance we can write *perfect* hash functions, but we don't.

If function is considered – one-to-one
Many to one

$h(x) = x$ is one to one function

- This may be many to one also
- Try to modify the function in such a way that
 - $h(x) \bmod \text{table size}$
 - $h(x) \bmod 10$

Given the value as 2, 32, 4, 5, 6, 7, 8, 45,

$$2 \bmod 10 = 2$$

$$32 \bmod 10 = 2$$

where there is a collision

There are various methods to avoid collision

The methods are – 1. Chaining

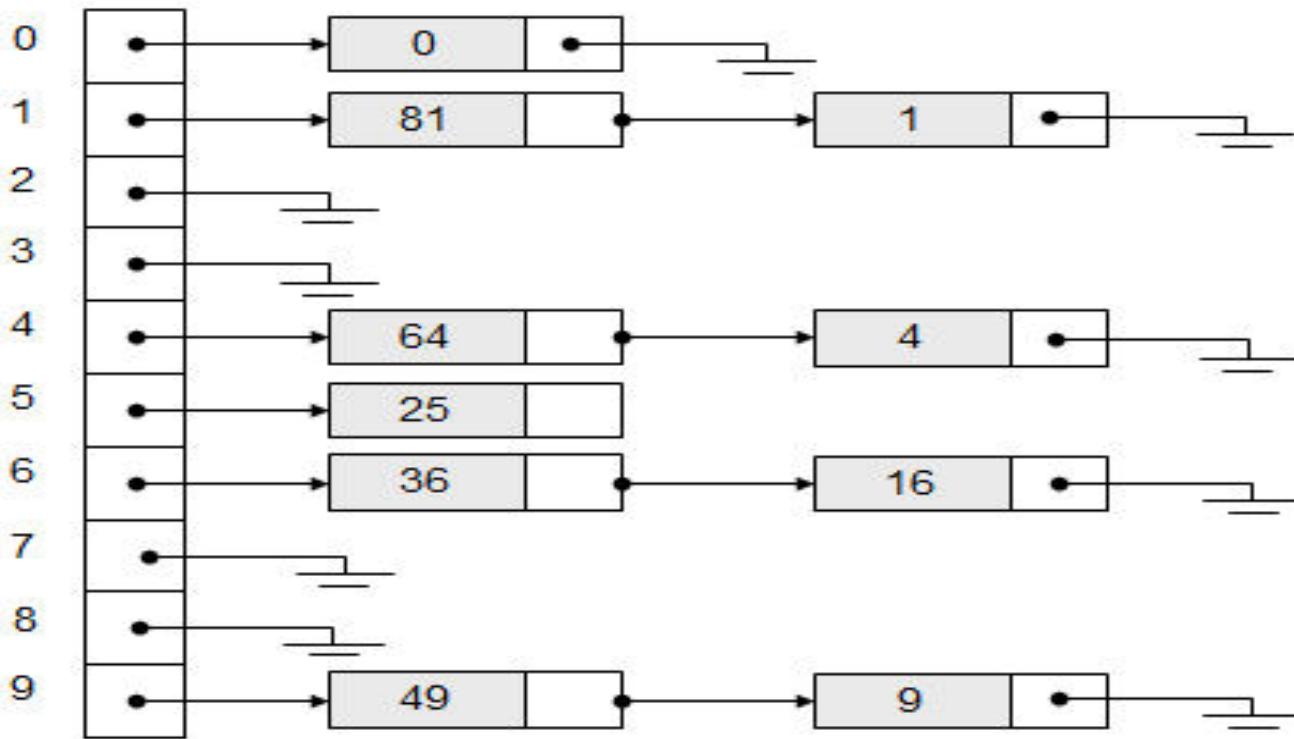
- 2. Open addressing
 - i) Linear probing
 - ii) Quadrating probing
 - iii) Double hashing

Chaining

Example

Keys: 0, 1, 4, 9, 16, 25, 36, 49, 64, 81

$$\text{hash(key)} = \text{key \% 10}.$$



Advantages of Chaining

- **Simple to Implement**
- Separate chaining is a **very simple technique to implement** compared to other data structures. **Input elements are just added to the corresponding linked list which the input has been hashed to. No collision occurs** in the table as the cells hold linked information.
- **Hash Table Capacity Not Limited**
- There is no issue with the table being filled to capacity, as **the table cells only hold linked information**. New elements are just added to the corresponding chain.
- The hash table can be of **theoretically infinite size**
- The hash table will not get polluted
- When searching, it will never search irrelevant information.

Disadvantages

- chaining has the disadvantage of using linked lists.
 - Requires the implementation of a second data structure.

LINEAR PROBING

- **Linear probing** is a scheme in computer programming for **resolving collisions** in hash tables, data structures for maintaining a **collection of key–value pairs** and looking up the value associated with a given key.

- linear probing is a form of open addressing.
- In these schemes, **each cell of a hash table stores a single key–value pair.**
- When the hash function causes a collision by mapping a new key to a cell of the hash table that is already occupied by another key, linear probing searches the table for the closest following free location and inserts the new key there.

h(x) =x mod size of array

if collision occurs

$$h(x) = [h(x) + f(i)] \text{ mod size}$$

where $f(i) = i$, where $i = 0, 1, 2, 3, \dots$

- In linear probing, collisions are resolved by sequentially scanning an array (with wrap around) until an empty cell is found.
 - i.e. f is a linear function of i , typically $f(i) = i$.
- **Example:**
 - Insert items with keys: 89, 18, 49, 58, 9 into an empty hash table.
 - Table size is 10.
 - **Hash function is $\text{hash}(x) = x \bmod 10$.**
 - $f(i) = i$;

```
hash ( 89, 10 ) = 9  
hash ( 18, 10 ) = 8  
hash ( 49, 10 ) = 9  
hash ( 58, 10 ) = 8  
hash ( 9, 10 ) = 9
```

After insert 89 After insert 18 After insert 49 After insert 58 After insert 9

0		49	49	49
1			58	
2				9
3				
4				
5				
6				
7				
8	18	18	18	18
9	89	89	89	89

- Lookups are performed in the same way, by **searching the table sequentially starting at the position given by the hash function($h(x) \bmod \text{size}$** , until finding a cell with a matching key or an empty cell.

Disadvantages:

- Clustering of element in particular location .The time taken to check will be more.
- As long as table is big enough, a free cell can always be found, but the time to do so can get quite large.
- Any key that hashes into the cluster will require several attempts to resolve the collision, and then it will add to the cluster
- So to avoid this we go for quadrating probing

- The average cost of a successful search is an average of the insertion costs over all smaller load factors.
- The average cost of finding the newly inserted item will be 2.5 no matter how many insertions follow.
- Ratio of number of elements (N) in a hash table to the hash *Table Size*.

Where the load factor is given by λ

- i.e. $\lambda = N/\text{Table Size}$

Quadratic probing:

- $h(x) = [h(x) + f(i)] \bmod \text{size}$
where $f(i) = i^2$, where $i = 0, 1, 2, 3, \dots$

2. Quadratic Probing

- Main Idea: Spread out the search for an empty slot –
Increment by i^2 instead of i
- $h_i(X) = (\text{Hash}(X) + i^2) \% \text{Table Size} , i=0,1,2,3.....,$
 $h_0(X) = \text{Hash}(X) \% \text{Table Size}$
 $h_1(X) = [\text{Hash}(X) + 1] \% \text{Table Size}$
 $h_2(X) = [\text{Hash}(X) + 4] \% \text{Table Size}$
 $h_3(X) = [\text{Hash}(X) + 9] \% \text{Table Size}$

Quadratic Probing

Example: insert 14,8,21,2,7 with size m=7

insert(14)

$$14 \% 7 = 0$$

insert(8)

$$8 \% 7 = 1$$

insert(21)

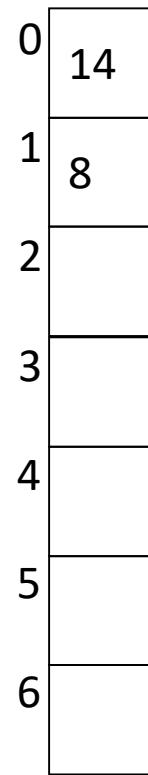
$$21 \% 7 = 0$$

insert(2)

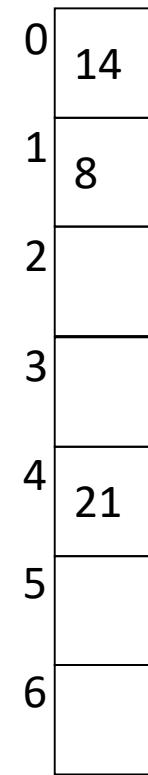
$$2 \% 7 = 2$$



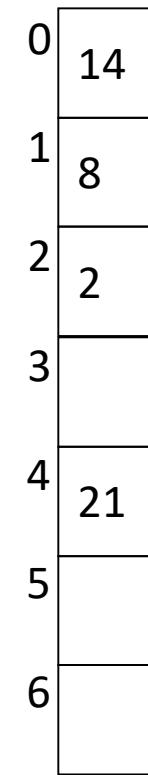
1



1



3

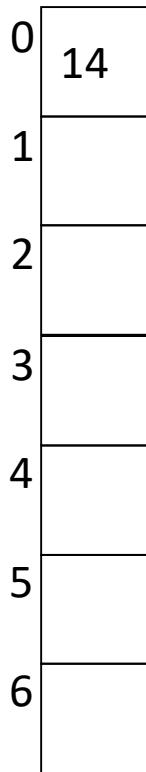


1

probes:

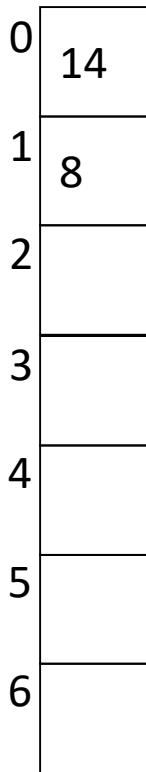
Problem With Quadratic Probing

insert(14)
 $14 \% 7 = 0$



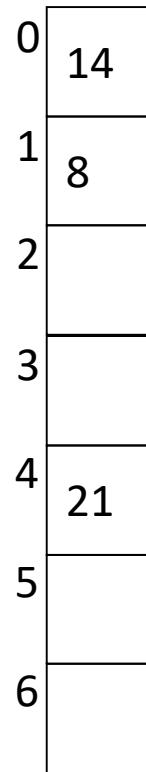
1

insert(8)
 $8 \% 7 = 1$



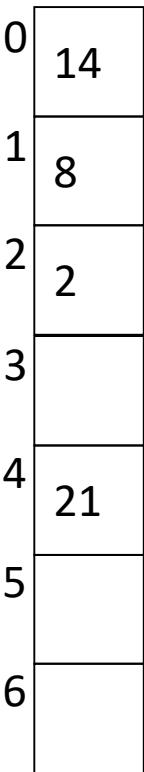
1

insert(21)
 $21 \% 7 = 0$



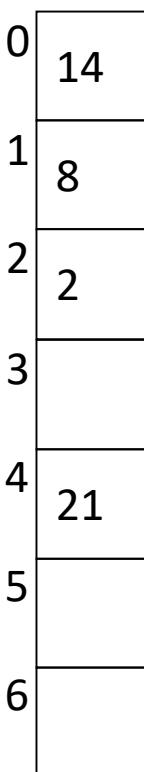
3

insert(2)
 $2 \% 7 = 2$



1

insert(7)
 $7 \% 7 = 0$



??

probes:

Limitations:

- In the Above example the process is never ending and **the no : of probes increases** but to identify a cell and to store the element is not possible
- In the case of quadratic probing, the situation is **even more drastic.**
- There is **no guarantee of finding an empty cell once the table gets more than half full**, or even before the table gets half full if the table size is not prime.
- So an empty key space can always be found as long as at most $(b / 2)$, (with size of the table is b) locations are filled, i.e., the hash table is not more than half full.

1. Get the key hash key k
2. Set counter j = 0
3. Compute hash function $h[k] = k \% \text{SIZE}$
4. If hash table[$h[k]$] is empty
 - (4.1) Insert key k at hash table[$h[k]$]
 - (4.2) Stop
- Else
 - (4.3) The key space at hash table[$h[k]$] is occupied, so we need to find the next available key space
 - (4.4) Increment j
 - (4.5) Compute new hash function $h[k] = (k + j * j) \% \text{SIZE}$
 - (4.6) Repeat Step 4 till j is equal to the SIZE of hash table
5. The hash table is full
6. Stop

Double Hashing

Double hashing is a collision resolving technique in Open Addressed Hash tables. Double hashing uses the idea of applying a second hash function to key when a collision occurs.

Double hashing can be done using :

$$\text{(hash1(key) + i * hash2(key)) \% TABLE_SIZE}$$

Here hash1() and hash2() are hash functions and TABLE_SIZE is size of hash table.

(We repeat by increasing i when collision occurs)

First Hash function is : $\text{hash1(key) = key \% TABLE_SIZE}$

Second Hash function is : $\text{hash2(key) = PRIME - (key \% PRIME)}$

where PRIME is a *prime smaller than the TABLE_SIZE*.

A good second Hash function is:

- It must never evaluate to zero
- Must make sure that all cells can be probed

Closed Hashing III: Double Hashing

- **Idea:** Spread out the search for an empty slot by using a second hash function
 - *No primary or secondary clustering*
- $h_i(X) = (\text{Hash}_1(X) + i * \text{Hash}_2(X)) \bmod \text{Table Size}$ for $i = 0, 1, 2, \dots$
- Good choice of $\text{Hash}_2(X)$ can guarantee does not get “stuck” as long as $\lambda < 1$
 - Integer keys:
 $\text{Hash}_2(X) = R - (X \bmod R)$
where R is a prime smaller than Table Size

Lets say, **Hash1 (key) = key % 13**

Hash2 (key) = 7 - (key % 7)

$$\text{Hash1}(19) = 19 \% 13 = 6$$

$$\text{Hash1}(27) = 27 \% 13 = 1$$

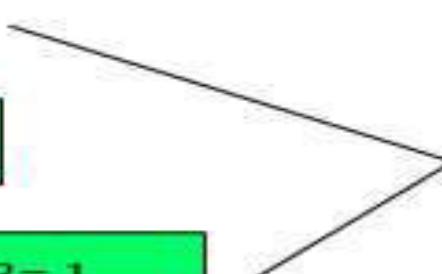
$$\text{Hash1}(36) = 36 \% 13 = 10$$

$$\text{Hash1}(10) = 10 \% 13 = 10$$

$$\text{Hash2}(10) = 7 - (10 \% 7) = 4$$

$$(\text{Hash1}(10) + 1 * \text{Hash2}(10)) \% 13 = 1$$

$$(\text{Hash1}(10) + 2 * \text{Hash2}(10)) \% 13 = 5$$



Collision

Double Hashing Example

insert(14)

$$14 \% 7 = 0$$

insert(8)

$$8 \% 7 = 1$$

insert(21)

$$21 \% 7 = 0$$

$$5 - (21 \% 5) = 4$$

insert(2)

$$2 \% 7 = 2$$

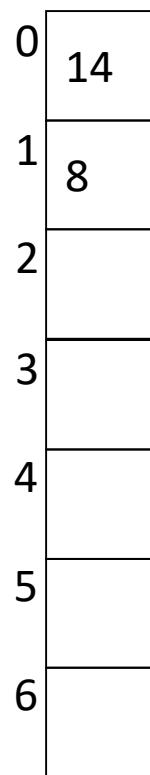
insert(7)

$$7 \% 7 = 0$$

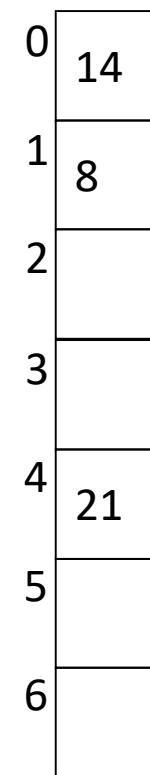
$$5 - (7 \% 5) = 3$$



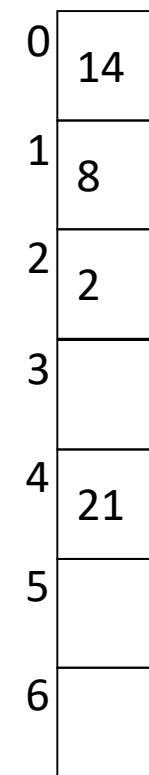
1



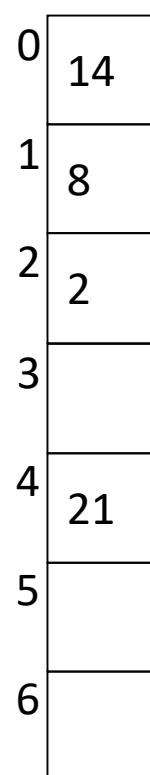
1



2



1



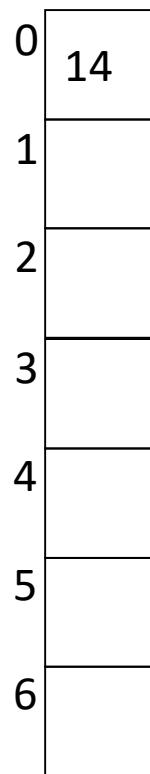
??

probes:

Double Hashing Example

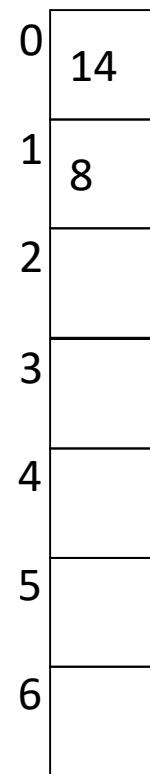
insert(14)

$$14 \% 7 = 0$$



insert(8)

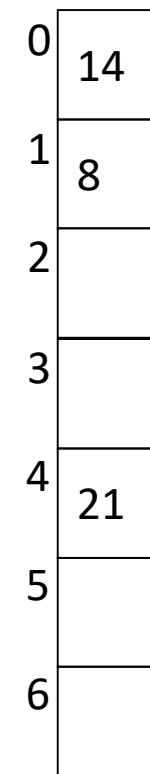
$$8 \% 7 = 1$$



insert(21)

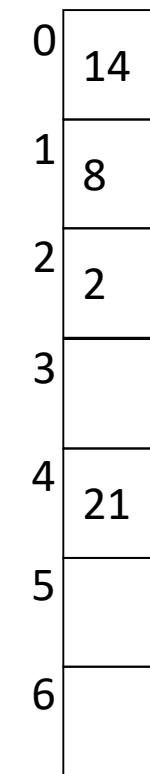
$$21 \% 7 = 0$$

$$5 - 1(21 \% 5) = 4$$



insert(2)

$$2 \% 7 = 2$$



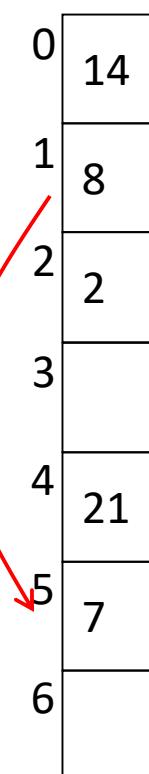
insert(7)

$$7 \% 7 = 0 +$$

$$5 - (7 \% 5) = 3$$

$$5 - 2(7 \% 5) = 1$$

$$5 - 3(7 \% 5) = -1$$



probes:

1

1

2

1

4

- **Advantages:**

skip values give different search paths for keys that collide.

- **Disadvantage:**

delete cumbersome to implement

Rehashing:

Steps to be followed:

- Build a new table that is about **twice as big** as old one
- Compute **new hash function**
- Compute **new hash value** for each non-deleted element
- Insert it in the new table

Example: 13 , 15 , 24 , 6 , 23

Table Size = 7

Hash Function $h(x) = x \bmod 7$

{Linear probing is used}

$$13 \% 7 = 6$$

$$6 \% 7 = 6$$

$$15 \% 7 = 1$$

$$24 \% 7 = 3$$

$$23 \% 7 = 2$$

0	6
1	15
2	
3	24
4	
5	
6	13

Insert element 23:

0	6
1	15
2	23
3	24
4	
5	
6	13

→ Table size is 70% Full {Create new table of size 17}

New hash Table:

$$h(x) = x \bmod 17$$

$$6 \% 17 = 6$$

$$15 \% 17 = 15$$

$$23 \% 17 = 6$$

$$24 \% 17 = 7$$

$$13 \% 17 = 13$$

0	
1	
.	
.	
6	6
7	23
8	24
.	
13	13
14	
15	15
16	

HASHING

WHY?

- Time taken for a search depends on the **size of the collection**
- Binary Search $O(\log n)$
- Linear Search $O(n)$

Hashing:

- Storage & retrieval of data in an **average time**, does not depend on the collection size { **$O(1)$** }

Hash Table:

- Array of fixed size, whose indices are the **range of hash function**

Hash Function:

- A **function** that takes an element {may be int, string} and **output an integer** in a certain range
- This integer is used to **determine the location** in the table for that element

$$h(k) = k \bmod m$$

The diagram illustrates the formula $h(k) = k \bmod m$. It features three red arrows pointing from labels below the equation to specific parts of the formula. The first arrow points from the label "Hash function" to the letter "h". The second arrow points from the label "Element" to the variable "k". The third arrow points from the label "Table Size" to the variable "m".

Hashing:

- Process of **mapping** a key value to a position in a table

Example: 7, 18 , 41 , 34

$$7 \% 10 = 7$$

$$18 \% 10 = 8$$

$$41 \% 10 = 1$$

$$34 \% 10 = 4$$

0	
1	41
2	
3	
4	34
5	
6	
7	7
8	18
9	

IDEAL Hashing:

- Hash Table is an array of some **fixed size**
- Search is performed based on the **key**
- Each key is mapped into some position in the range
0 to table size - 1

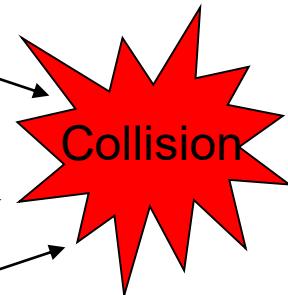
Drawbacks:

- **Waste too much space**, if universe is too **large** compared with actual number of elements to be stored
- **Collision** {i.e. Two keys may hash to same slot}

Example: To insert elements 80 , 40 , 65 , 24 , 58 , 35 , 98

Table Size = 11

$$80 \% 11 = 3$$
$$40 \% 11 = 7$$
$$65 \% 11 = 10$$
$$24 \% 11 = 2$$
$$58 \% 11 = 3$$
$$35 \% 11 = 2$$
$$98 \% 11 = 10$$



0	
1	
2	24
3	80
4	
5	
6	
7	40
8	
9	
10	65

Collision Handling Strategies

Separate
Chaining

Open
Addressing

- Linear Probing
- Quadratic Probing
- Double Hashing
- Rehashing

Separate Chaining:

- All Keys that maps to the **same hash value** are kept in a **list** (or ‘**Bucket**’)

Insertion:

- Compute **h (k)** to determine which **list** to traverse
- If $T [h (k)]$ contains **Null Pointer**, initialize this entry to point to a **linked list** that contains K-alone
- If $T [h (k)]$ is a **Non-empty list** then add k, at the **beginning** of this list

Example: 10 , 22 , 107 , 12 , 42

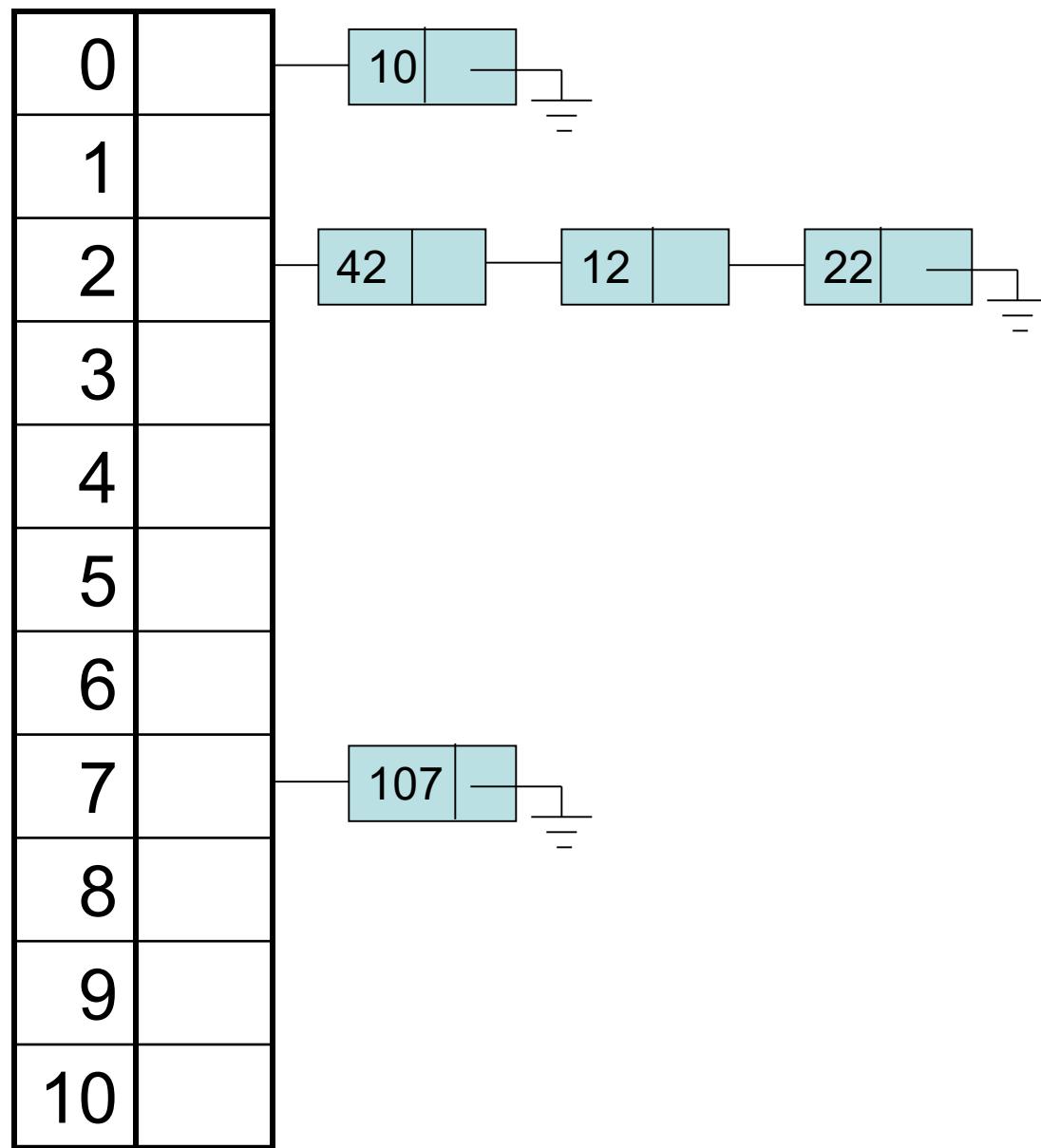
$$10 \% 10 = 0$$

$$22 \% 10 = 2$$

$$107 \% 10 = 7$$

$$12 \% 10 = 2$$

$$42 \% 10 = 2$$



Deletion:

- Compute $h(k)$ then search for k , within the list at $T[h(k)]$
- Delete if it found

Drawbacks:

- Memory allocation in linked list manipulation slow down program

Open Addressing:

- Relocate the key, if it collides with existing key { Store k at an entry different from $T [h (k)]$ }

Insertion: To insert key k, Compute $h_0 (k)$

- If $T [h_0 (k)] = \text{empty}$, insert key
- If Collision occurs, probe alternative cells $h_1 (k)$, $h_2 (K)$...
Until empty cell is found

$$h_i (k) = (\text{hash} (k) + F (i)) \bmod m$$

$m = \text{Table Size}$

$i = 1 \text{ to } m - 1$

$F (0) = 0$

Linear Probing:

- Cells are probed sequentially with wrap around

$$F(i) = i$$

- After searching cell hash (k) in the array look in hash (k) + 1 ,
hash (k) + 2 ...etc

$$h_i(k) = (\text{hash}(k) + i) \bmod m$$

Example: 89 , 18 , 49 , 58 , 69

$$h(k) = k \bmod 10$$

$$89 \% 10 = 9$$

$$18 \% 10 = 8$$

$$49 \% 10 = 9$$

$$58 \% 10 = 8$$

$$69 \% 10 = 9$$

0	49
1	58
2	69
3	
4	
5	
6	
7	
8	18
9	89

Drawbacks:

- Encounter **Primary Clustering** { blocks of contiguously occupied table entries}
- If $h(k)$ falls in Cluster , Cluster will **definitely grow in size by one**
- If **two Cluster** separated by one entry, inserting one key into cluster can merge two cluster, So cluster size increase drastically by single insertion
- Large Cluster are easy **targets for Collision**

(ii) Quadratic Probing:

$$F(i) = i^2$$

- After searching cell hash (k) in the array look in the 1st, 4th, 9th, 16th etc cell after hash (k)

$$h_i(k) = (\text{hash}(k) + i^2) \bmod m$$

Example: 89 , 18 , 49 , 58 , 69

$$89 \% 10 = 9$$

$$18 \% 10 = 8$$

$$49 \% 10 = 9$$

$$58 \% 10 = 8$$

$$69 \% 10 = 9$$

0	49
1	
2	58
3	69
4	
5	
6	
7	
8	18
9	89

Advantage:

- Less likely to encounter primary clustering

Drawbacks:

- **Secondary Clustering Problem:** Elements that hash to the same position will probe the same alternative cells
- **No guarantee** of finding an **empty cells**,
{once table gets more than **half full** (or) if table size is **not prime**}

Double Hashing:

- Items that hash to the same location with $\text{hash}(k)$, **won't** have same probe for $\text{hash}_2(k)$

$$F(i) = i \cdot \text{hash}_2(x)$$

Second hash function probe at a distance $\text{hash}_2(x)$, 2. $\text{hash}_2(x)$

$\text{hash}_2(x)$:

$$\text{hash}_2(x) = R - (x \bmod R)$$

{Prime Smaller than Table Size}

$$h_i(k) = (\text{hash}(k) + (i(R - (x \bmod R)))) \bmod M$$

Drawbacks:

- If table size is not prime , it is possible to run out of alternative locations prematurely { **Outer hash function must be a Prime** }

0	69
1	
2	
3	58
4	
5	
6	49
7	
8	18
9	89

Rehashing:

Steps to be followed:

- Build a new table that is about **twice as big** as old one
- Compute **new hash function**
- Compute **new hash value** for each non-deleted element
- Insert it in the new table

Example: 13 , 15 , 24 , 6

Table Size = 7

Hash Function $h(x) = x \bmod 7$

{Linear probing is used}

$$13 \% 7 = 6$$

$$6 \% 7 = 6$$

$$15 \% 7 = 1$$

$$24 \% 7 = 3$$

$$23 \% 7 = 2$$

0	6
1	15
2	
3	24
4	
5	
6	13

Insert element 23:

0	6
1	15
2	23
3	24
4	
5	
6	13

→ Table size is 70% Full {Create new table of size 17}

New hash Table:

$$h(x) = x \bmod 17$$

$$6 \% 17 = 6$$

$$15 \% 17 = 15$$

$$23 \% 17 = 6$$

$$24 \% 17 = 7$$

$$13 \% 17 = 13$$

0	
1	
.	
.	
6	6
7	23
8	24
.	
13	13
14	
15	15
16	

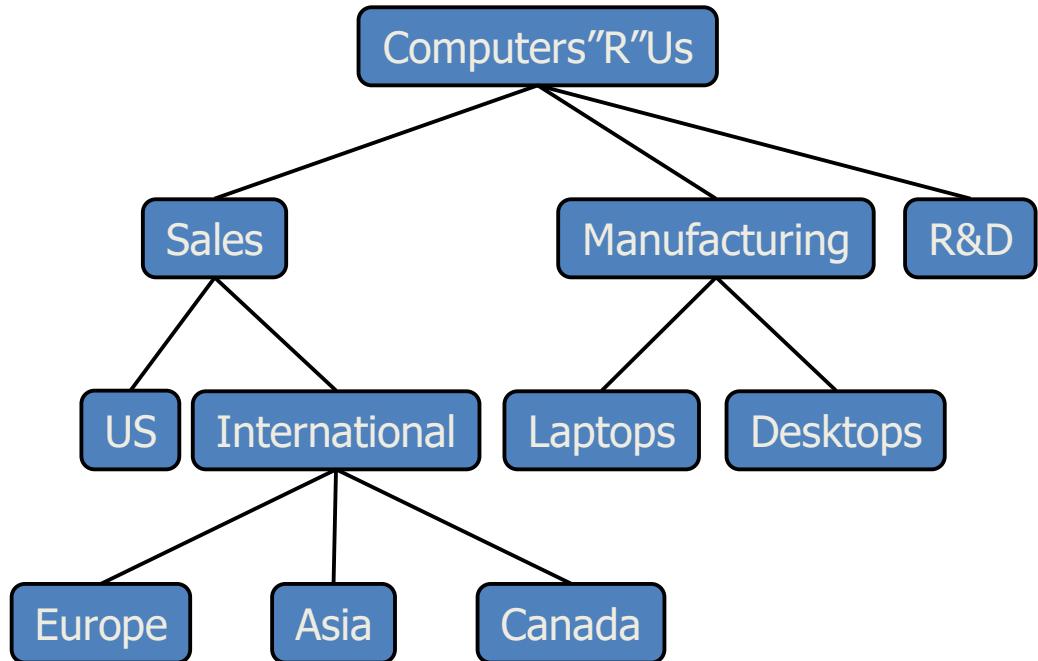
Trees

Trees

- Introduction to Trees
- Tree Terminology
- Binary tree
 - Implementation using Array or Linked list
- CONVERTING EXPRESSION INTO BINARY TREE
- Binary Tree Traversals
 - Inorder
 - Preorder
 - Postorder
 - Implementation of Binary Tree traversal
- Binary Search Trees(BST)
 - Insertion
 - Deletion
- Convert Binary Tree to BST by maintaining original structure

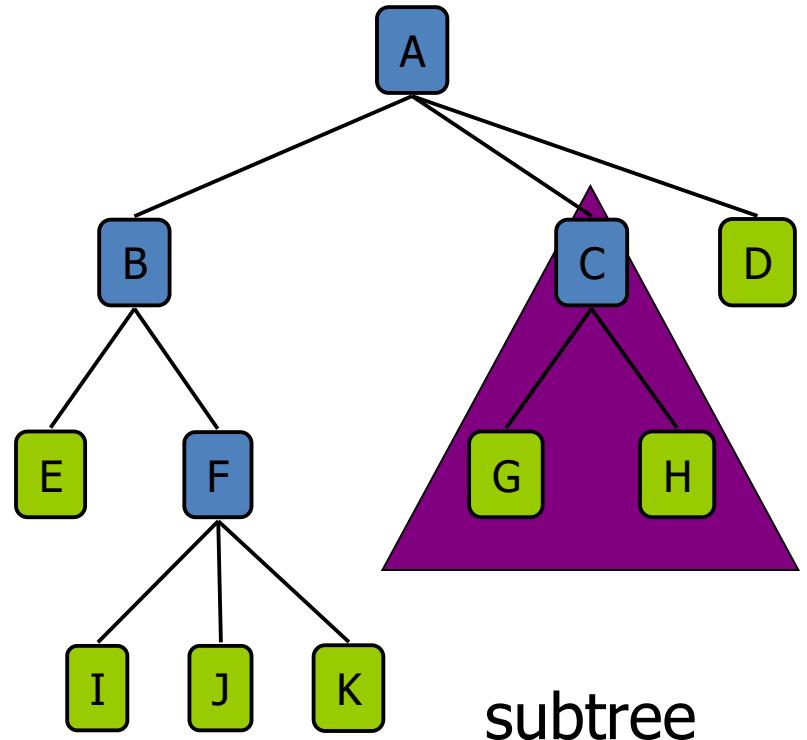
What is a Tree

- A tree is a **finite nonempty set of elements**.
- It is an **abstract model of a hierarchical structure**.
- consists of nodes with a **parent-child relation**.
- Applications:
 - Organization charts
 - File systems
 - Programming environments
 - Dictionary
 - Priority Queue

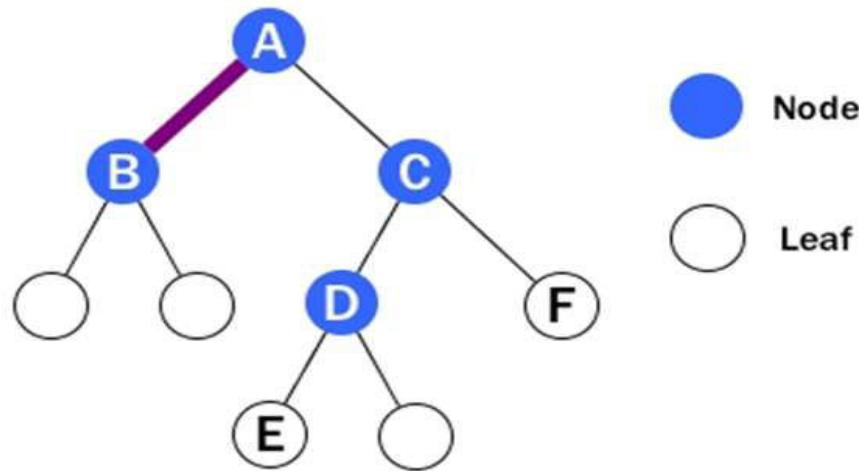


Tree Terminology

- **Root:** node without parent (A)
 - **Siblings:** nodes share the same parent
 - **Internal node:** node with at least one child (A, B, C, F). An intermediate node between the root and the leaf nodes is called an internal node.
 - **External node (leaf):** node without children (E, I, J, K, G, H, D)
 - **Ancestors** of a node: parent, grandparent, grand-grandparent, etc.
 - **Descendant** of a node: child, grandchild, grand-grandchild, etc.
 - **Depth** of a node: number of ancestors
 - **Height** of a tree: maximum depth of any node (3)
- **Subtree:** tree consisting of a node and its descendants



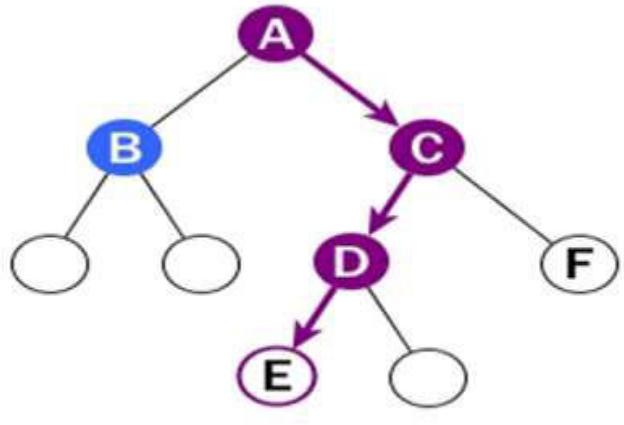
Edge – connection between one node to another.



About Edge

An example of edge is shown above between *A* and *B*. Basically, an edge is a line between two nodes, **or a node and a leaf**.

Path – a sequence of nodes and edges connecting a node with a descendant.



About Path

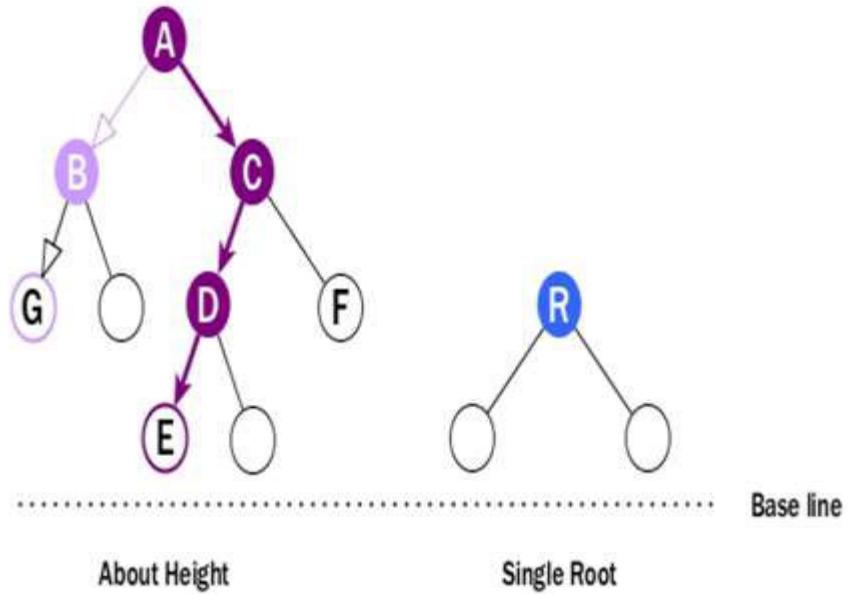
A path starts from a node and ends at another node or a leaf.

Please don't look over the following points:

- i) When we talk about a path, it includes all nodes and all edges along the path, *not just edges*.
- ii) In diagram, *we can't really talk about a path from B to F although B is above F*. Also there will be no path starting from a leaf or from a child node to a parent node. ([1])

Height of node – The height of a node is the number of edges on the longest downward path between that node and a leaf. When looking at height:

When looking at height:

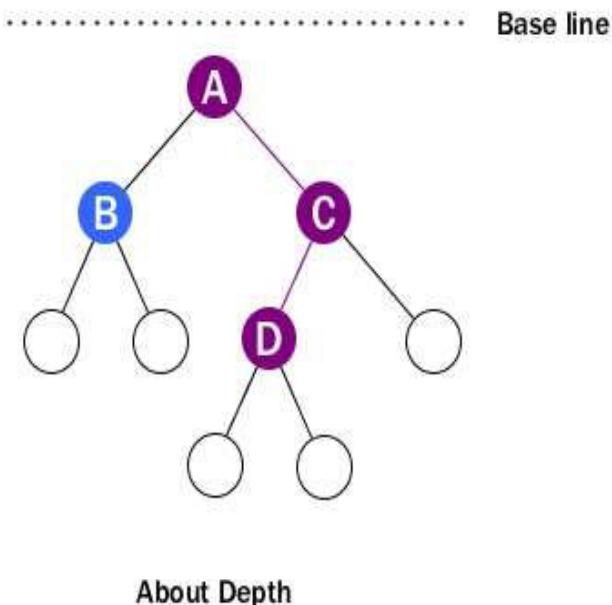


Every node has height.
So B can have height, so
does A , C and D .

**Leaf cannot have height as
there will be no path
starting from a leaf.**

It is the longest path from the node **to a leaf**. So A's height is the number of edges of the path to *E*, NOT to *G*. And its height is 3.

Depth –The depth of a node is the number of edges from the node to the tree's root node.



We don't care about path any more when depth pops in.

We just count how many edges between the targeting node and the root, ignoring directions. For example, D's depth is 2.

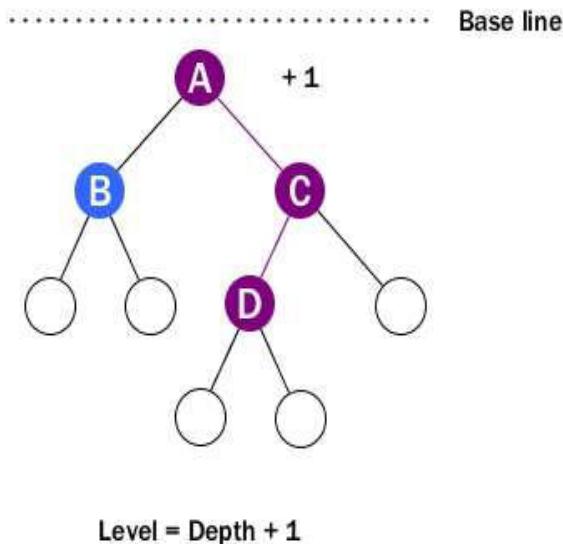
Recall that when talking about height, we actually imply a baseline located at bottom.

For depth, the baseline is at top which is root level. That's why we call it depth.

Note that **the depth of the root is 0**.

Level – The level of a node is defined by $1 + \text{the number of connections between the node and the root.}$

Simply, **level is depth plus 1.**



The important thing to remember is when talking about level, it **starts from 1** and **the level of the root is 1.**

We need to be careful about this when solving problems related to level.

Here Level of D is =No:of Connections between nodes
 $+1 = 2+1 = 3$

- Height and depth of a tree is equal
but height and depth of a node is not equal because the height is calculated by traversing from the given node to the deepest possible leaf. Depth is calculated from traversal from root to the given node.

A **root node** will have a **depth of 0**.

A **leaf node** will have a **height of 0**.

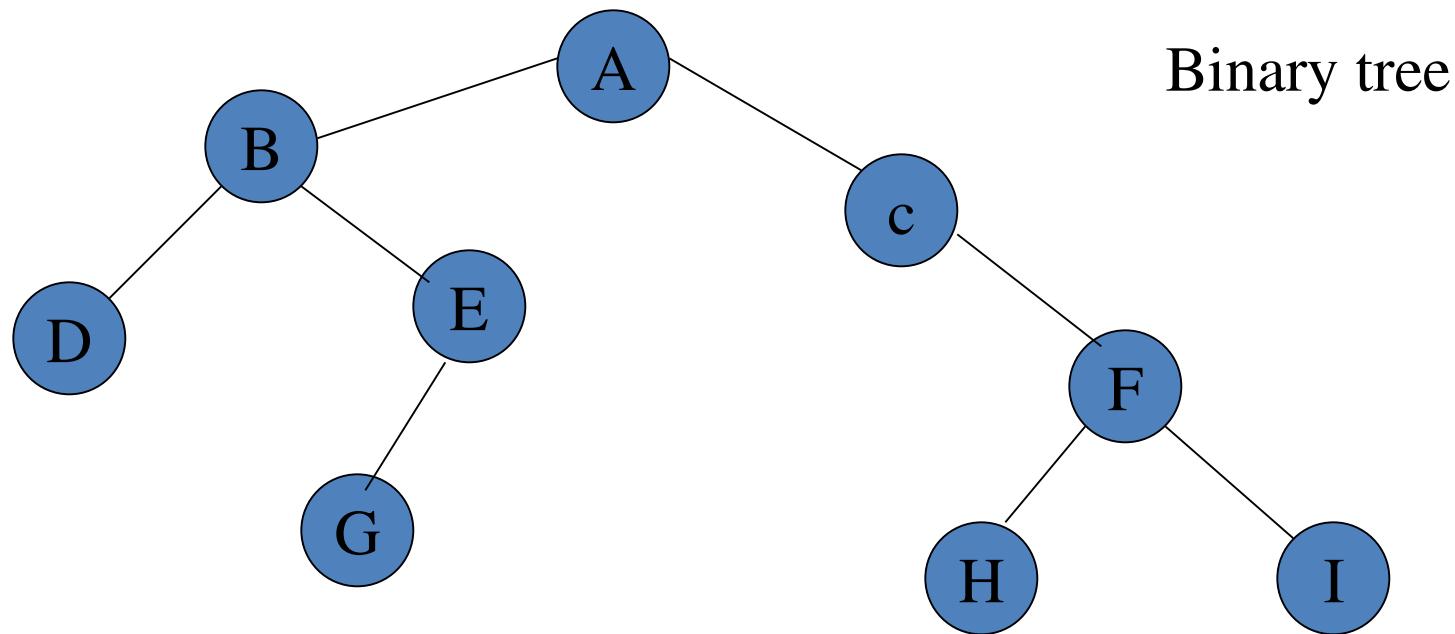
- A descendant node of a node is any node in the path from that node to the leaf node (including the leaf node). The immediate descendant of a node is the “child” node.
- An ancestor node of a node is any node in the path from that node to the root node (including the root node). The immediate ancestor of a node is the “parent” node.
- If node A is an ancestor of node B then node B is a descendant of node A.

Binary Tree

A binary tree is a finite set of elements that is either empty or is partitioned into three disjoint subsets.

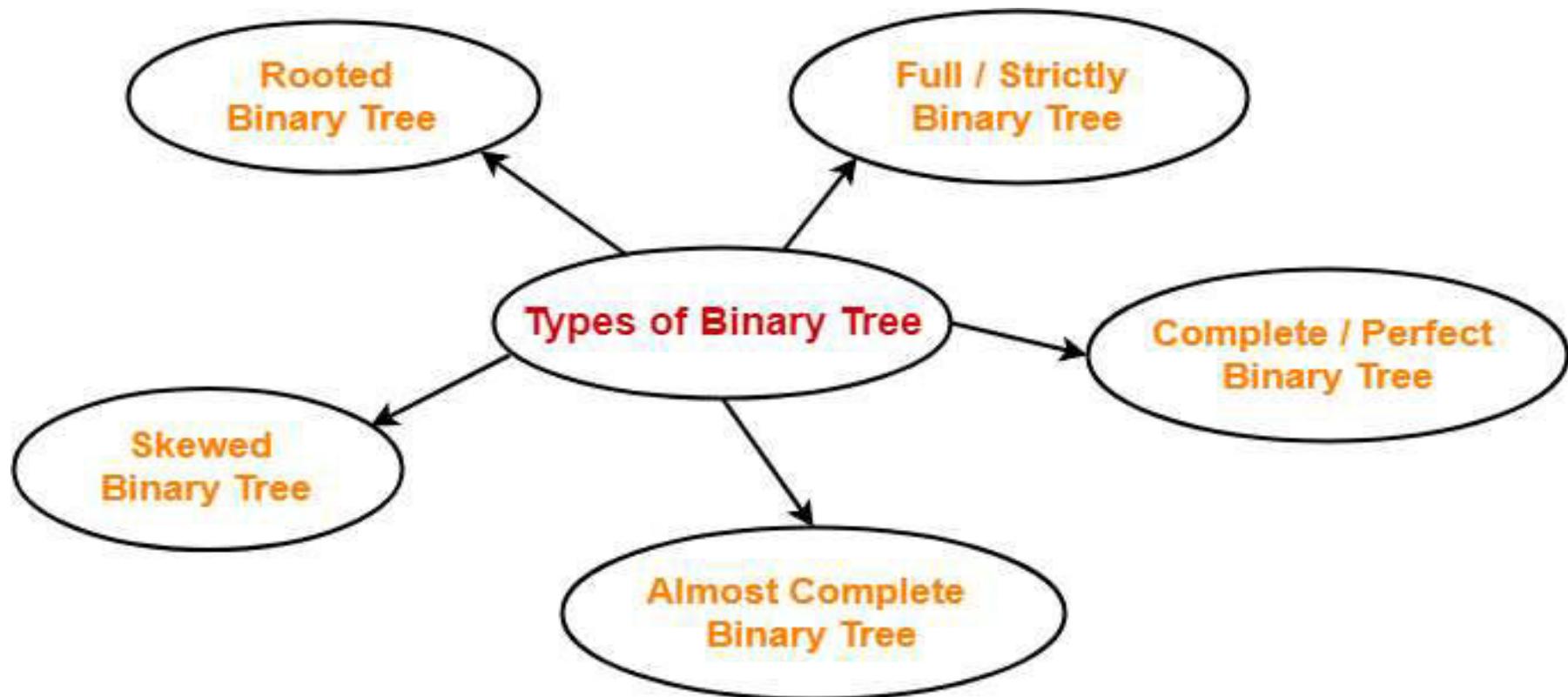
- The first subset contains a single element called the root of the tree.
- The other two subsets are themselves binary trees, called the left and right subtrees of the original tree. A left or right subtree can be empty.

Each element of a tree is called a node of the tree.



Binary tree

- A binary tree is composed of zero or more nodes
- Each may have 1 child, 2 children or no child (leaf node)
- Each node contains:
 - A value (some sort of data item)
 - A reference or pointer to a left child (may be null), and
 - A reference or pointer to a right child (may be null)
- A binary tree may be *empty* (contain no nodes)
- If not empty, a binary tree has a root node
 - Every node in the binary tree is reachable from the root node by a unique path
- A node with no left child and no right child is called a leaf
 - In some binary trees, only the leaves contain a value(Expression)



- A **full binary tree** (sometimes proper binary tree or 2-tree) is a tree in which *every node other than the leaves has two children*. Full binary tree is also called as **Strictly Binary Tree**.

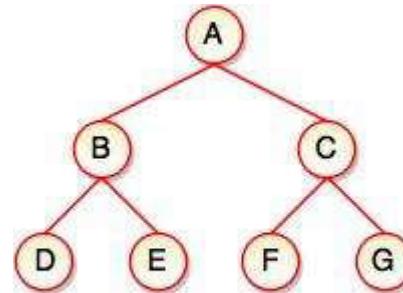
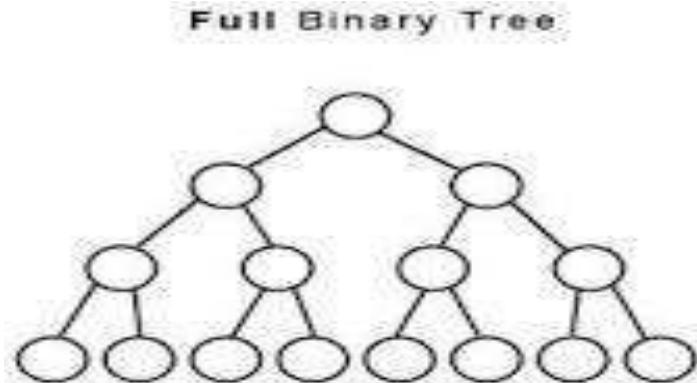
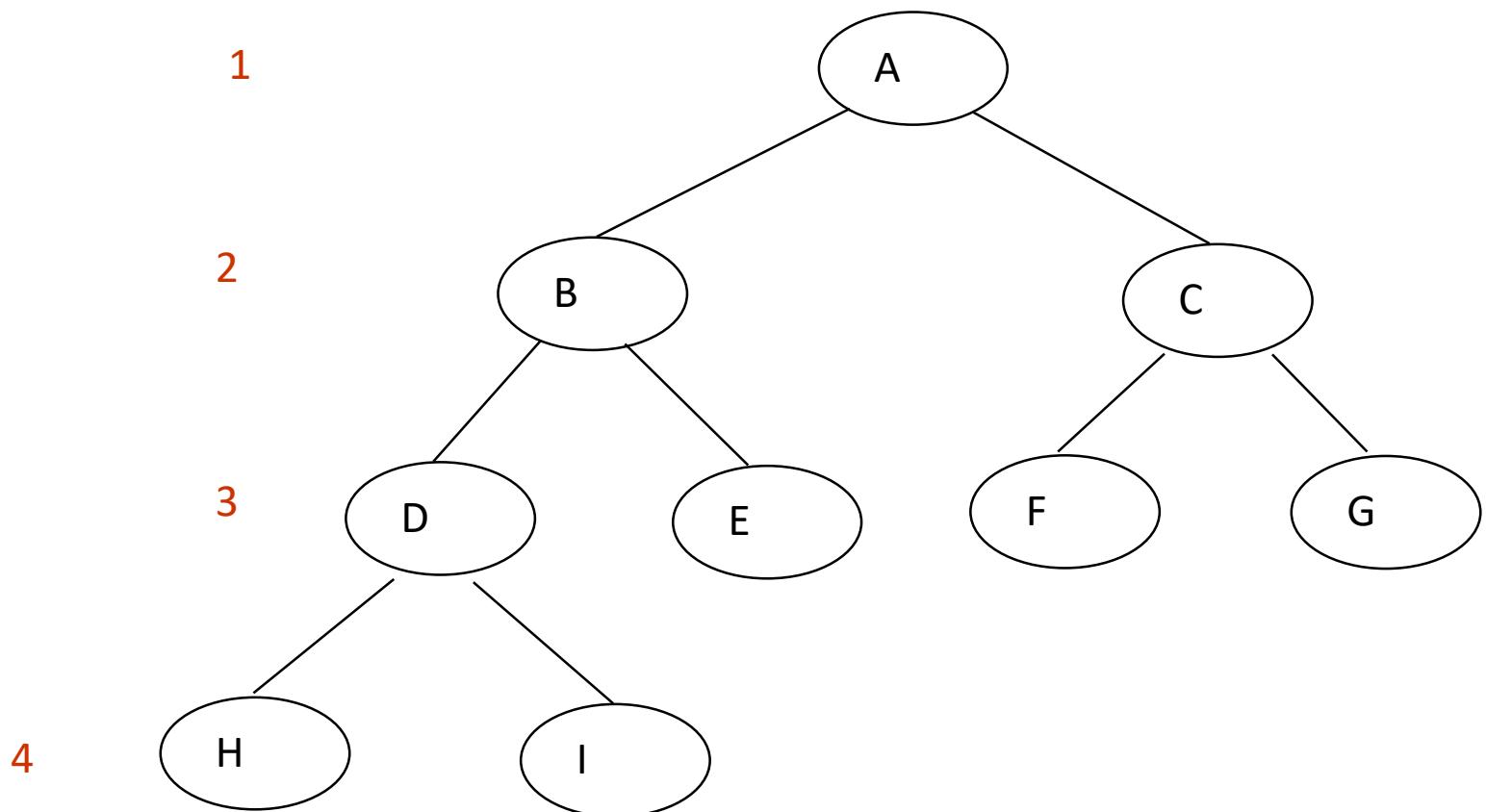


Fig. Full Binary Tree

- A **complete binary tree** is a **binary tree** in which **every level, except possibly the last, is completely filled, and all nodes are as far left as possible**.

Complete Binary Tree



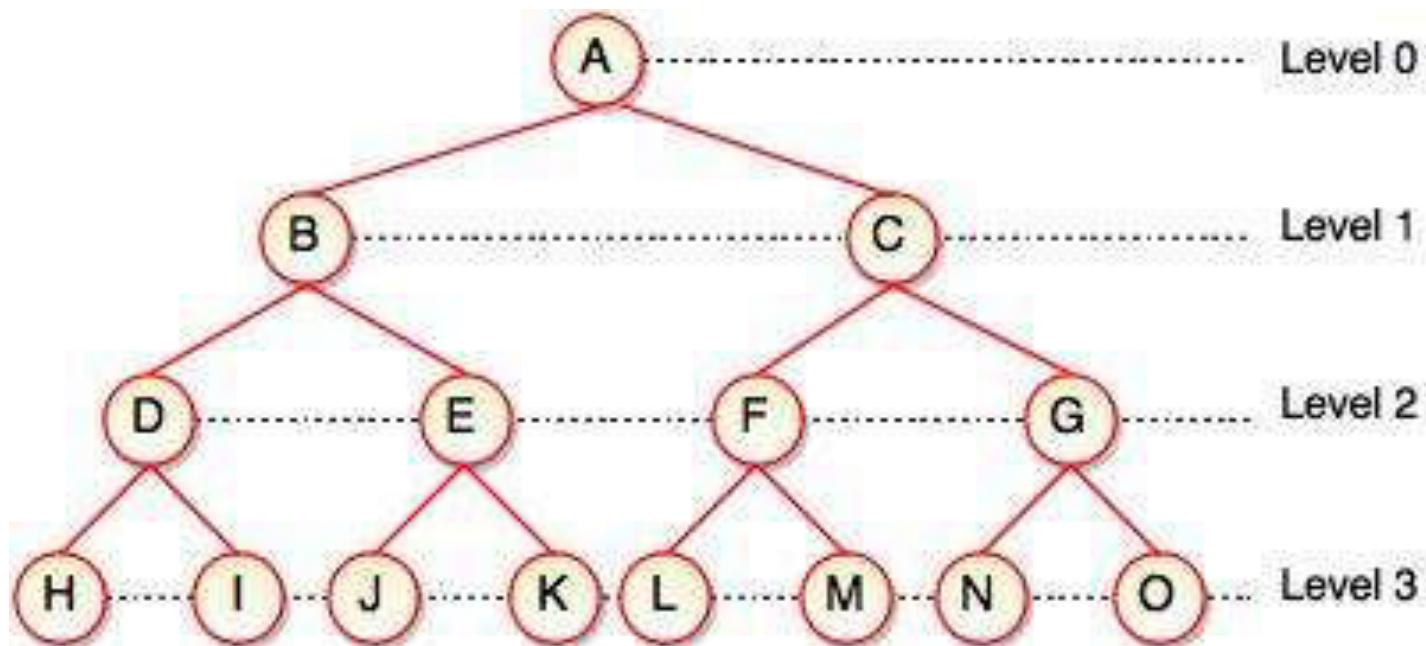


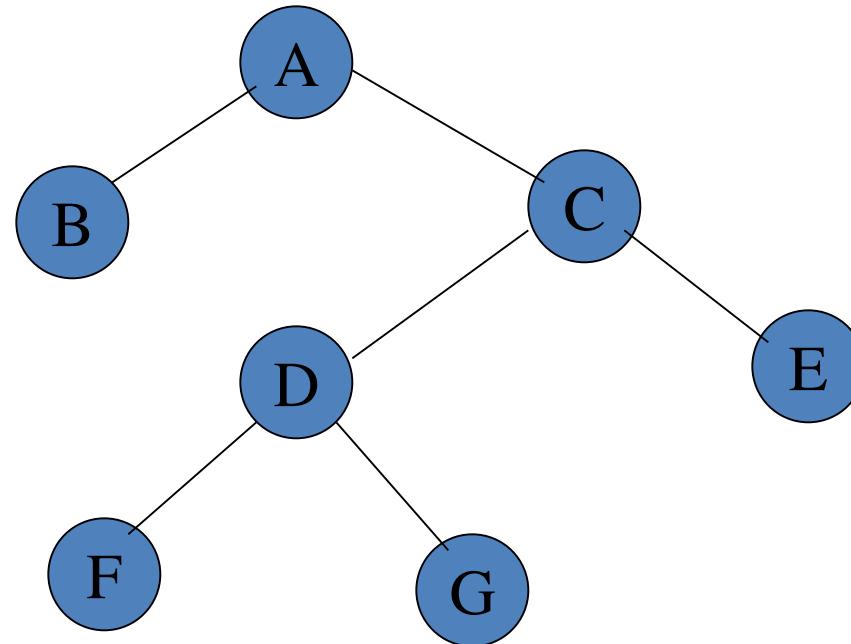
Fig. Complete Binary Tree

Binary Tree

Strictly binary trees: also called as Full binary tree

If every nonleaf node in a binary tree has nonempty left and right subtrees, the tree is called a strictly binary tree.

A strictly binary tree with n leaves always contains $2n - 1$ nodes.



Skewed Binary Tree

Skewed Binary Tree

If a tree which is dominated by left child node or right child node, is said to be a **Skewed Binary Tree**.

In a skewed binary tree, all nodes except one have only one child node. The remaining node has no child.

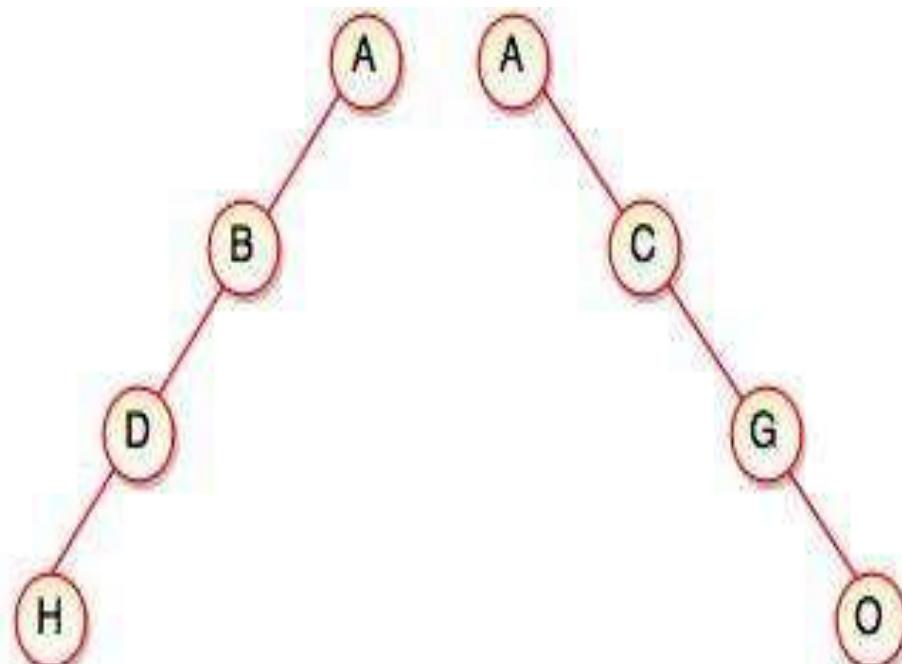


Fig. Left Skewed
Binary Tree

Fig. Right Skewed
Binary Tree

Extended Binary Tree

- Extended binary tree consists of **replacing every null subtree of the original tree with special nodes.**
- Empty circle represents internal node and filled circle represents external node.
- The nodes from the original tree are internal nodes and the special nodes are external nodes.
- Every internal node in the extended binary tree has exactly two children and every external node is a leaf. **It displays the result which is a complete binary tree.**

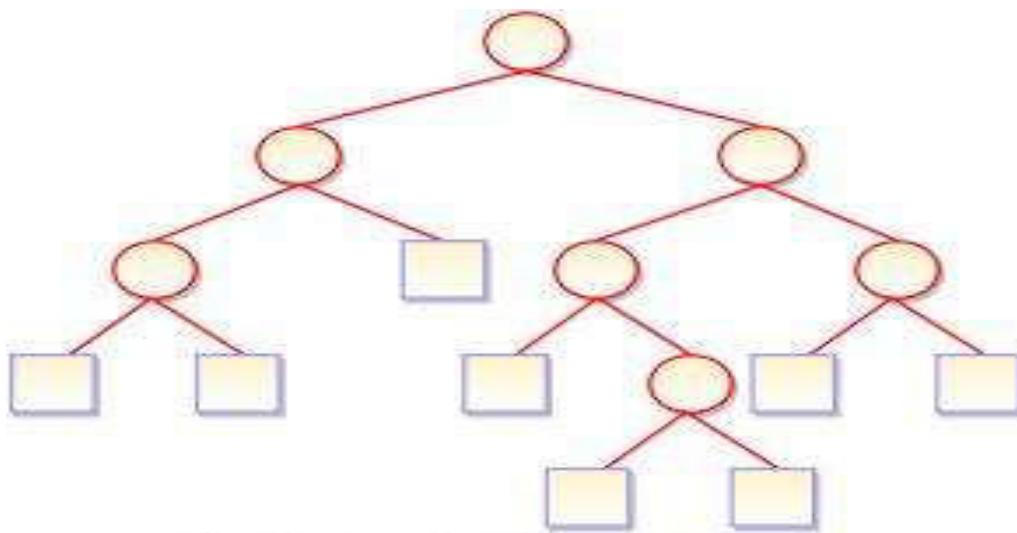
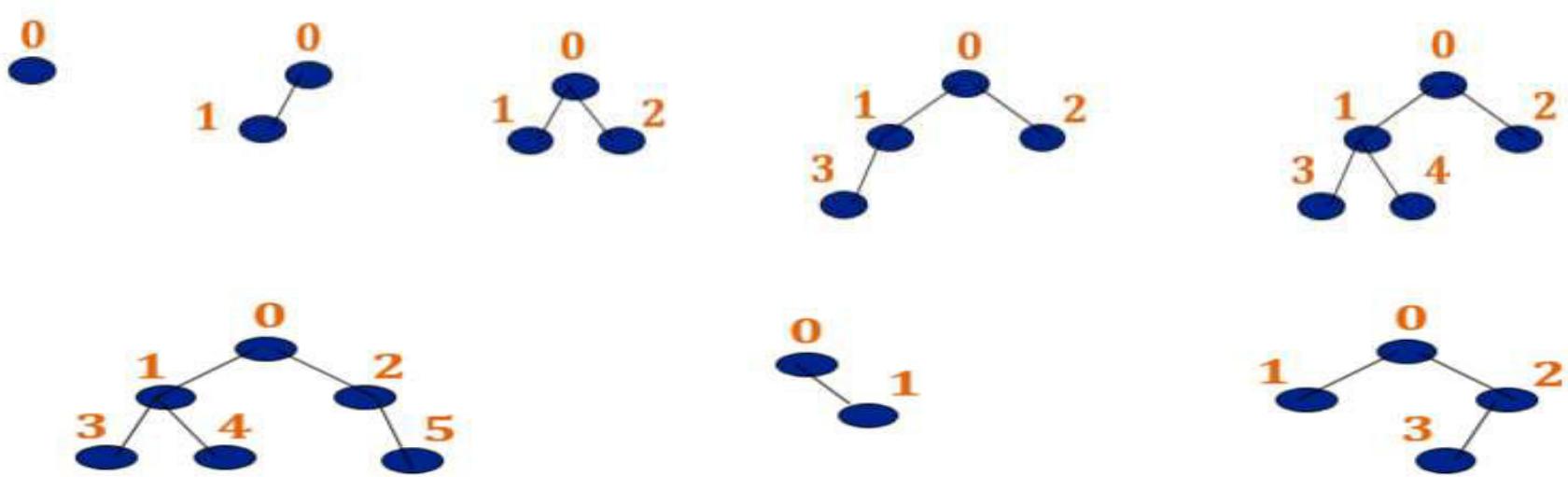


Fig. Extended Binary Tree

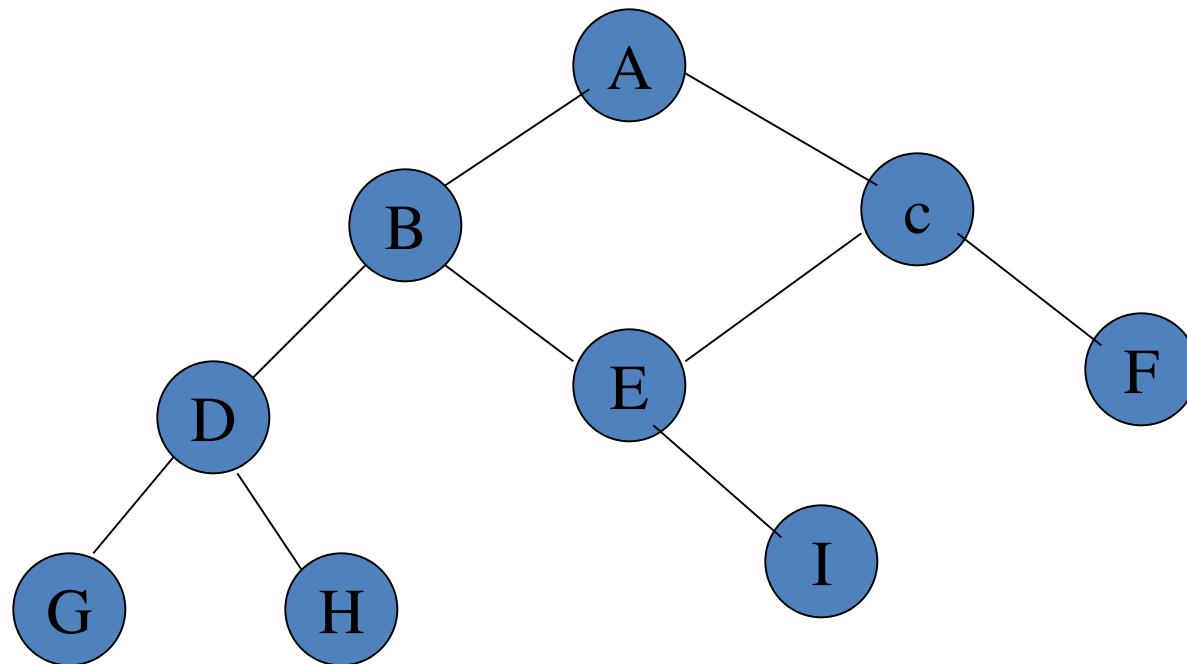
A the binary tree made up of the first n nodes of a canonically labeled complete Binary Tree is called **Almost complete Binary Tree**. Below are all almost complete binary trees,



Below trees are NOT almost complete binary trees (the labeling has no meaning, because there are gaps).

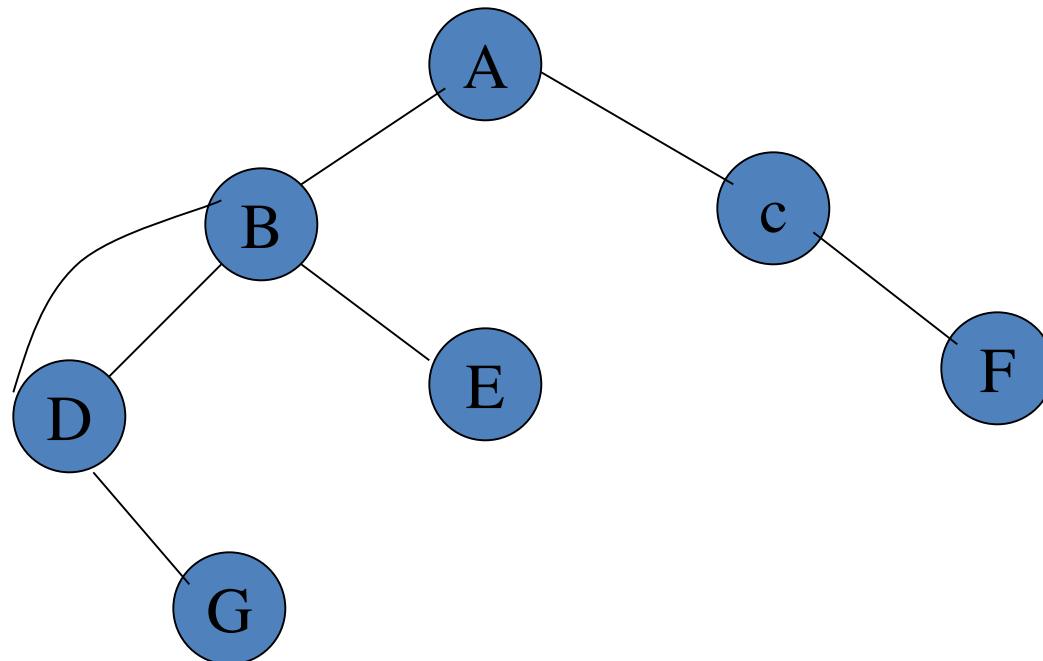
Binary Tree

Structures that are not binary trees



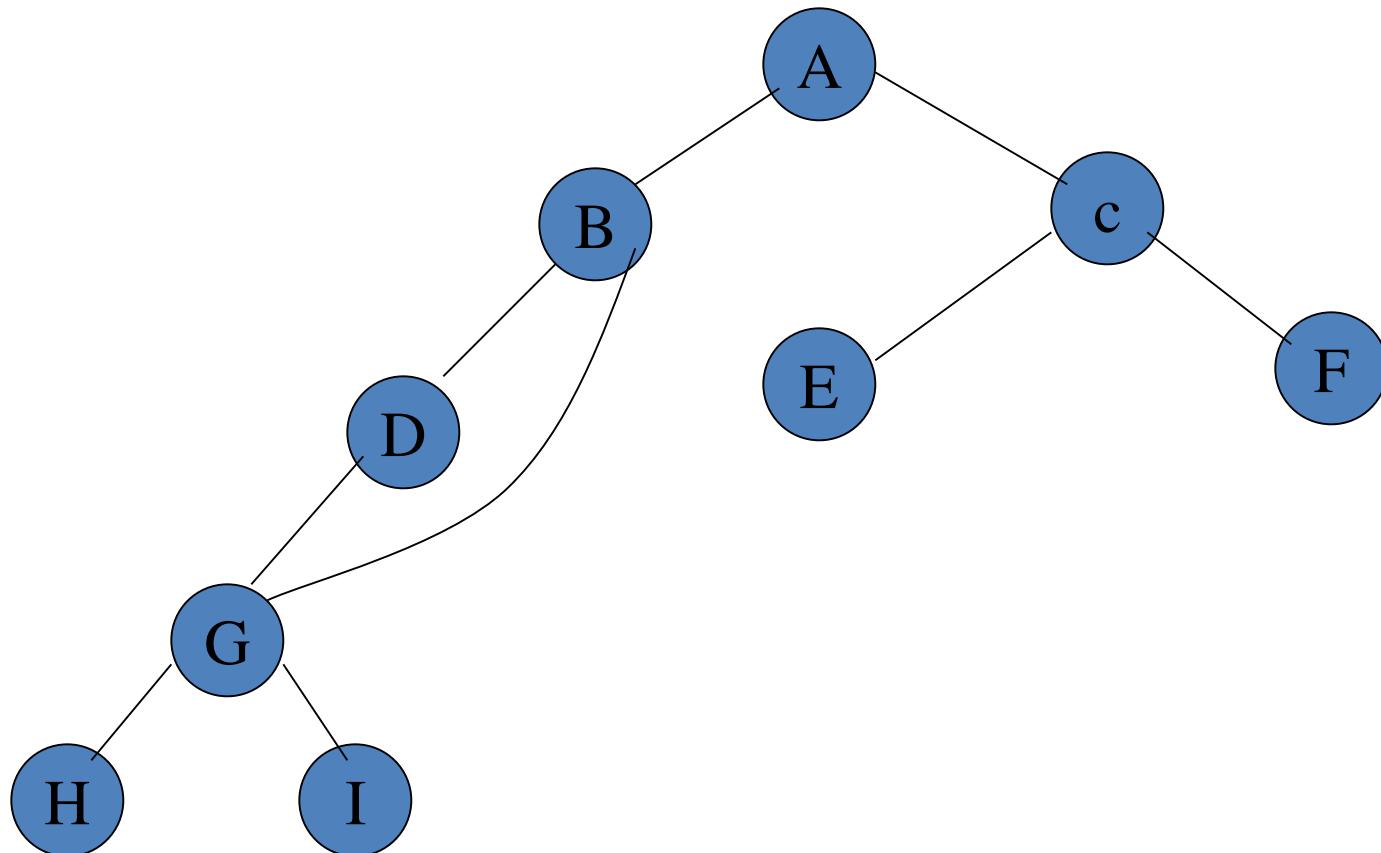
Binary Tree

Structures that are not binary trees



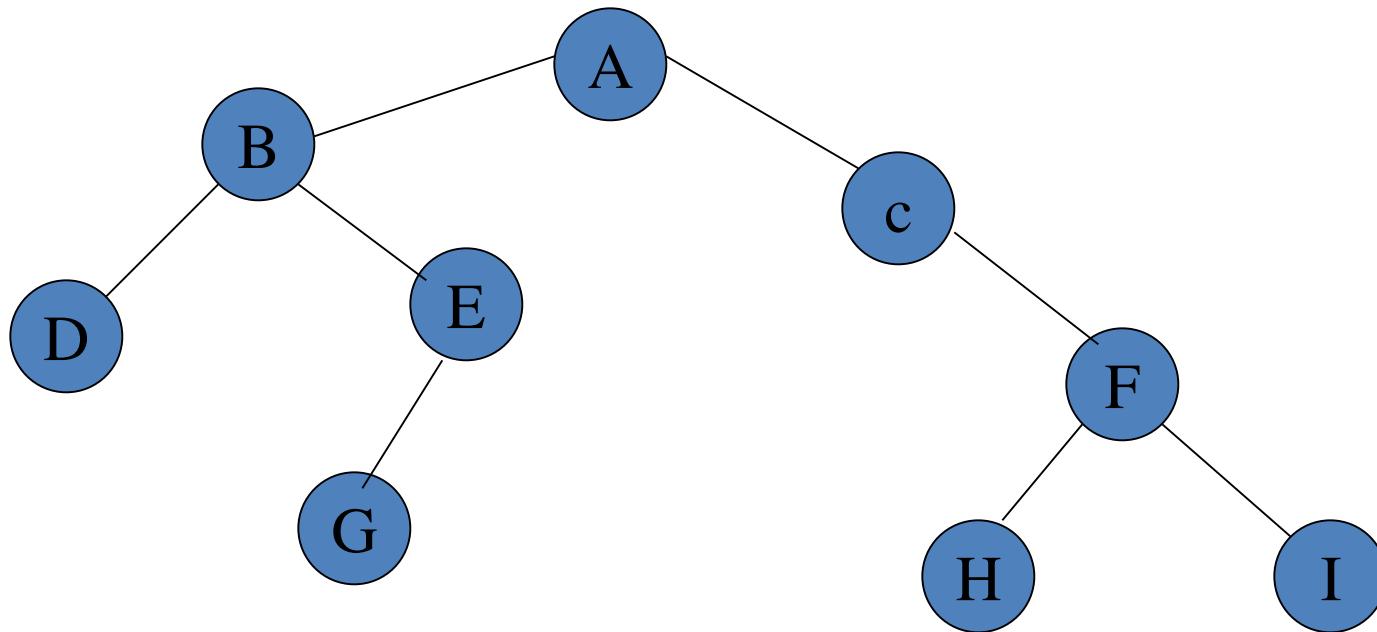
Binary Tree

Structures that are not binary trees



Binary tree

Structure that is not a strictly binary tree:
because nodes C and E have one son each.

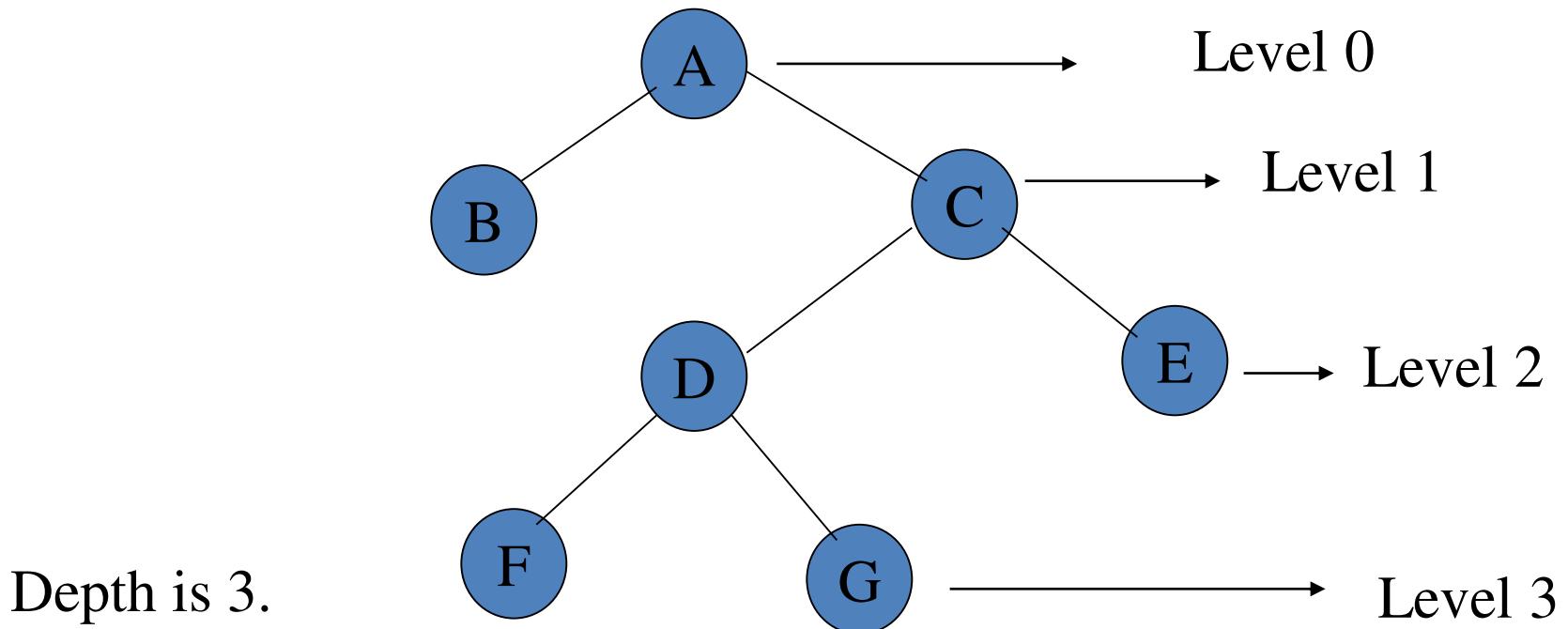


Level and depth of a binary Tree

The root of the tree has level 0(Some authors starts with Depth+1). And the level of any other node is one more than the level of its father.

Depth of a binary tree:

The depth of a binary tree is the maximum level of any leaf in the tree.

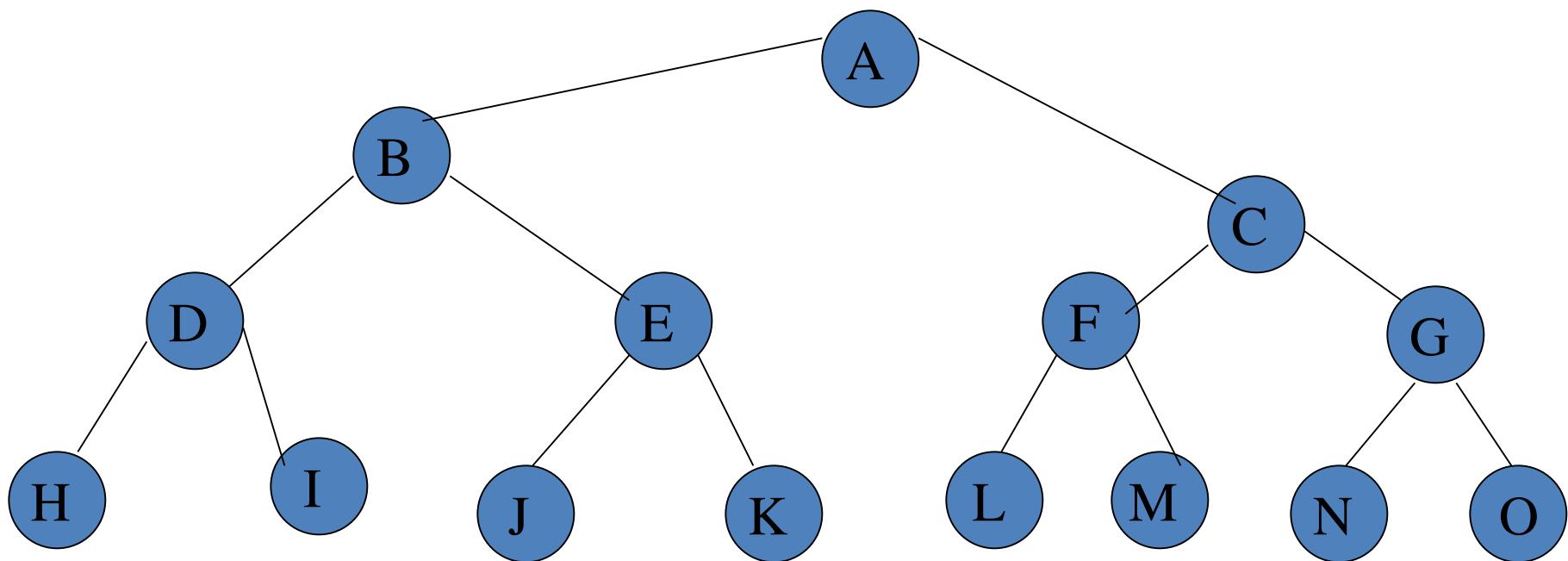


A complete binary tree

Complete binary tree of depth d is the strictly binary tree all of whose leaves are at level d.

A complete binary tree of depth d is the binary tree of depth d that contains exactly 2^1 nodes at each level l between 0 and d.

The total number of nodes = the sum of the number of nodes at each level between 0 and d.
= $2^{d+1} - 1$



Maximum Number of Nodes in a Binary Tree

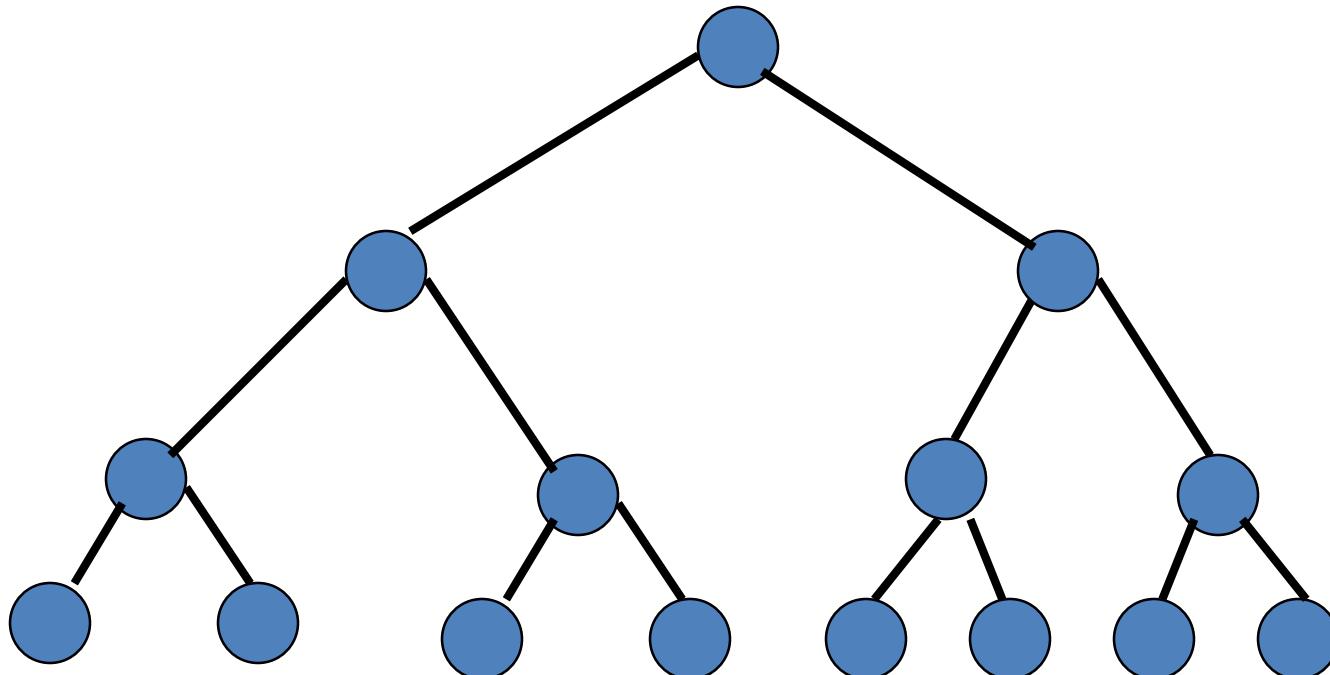
- ➊ The maximum number of nodes **on depth i** of a binary tree is 2^i , $i \geq 0$.
- ➋ The maximum number of nodes in a **binary tree of height k** is $2^{k+1} - 1$, $k \geq 0$.

Prove by induction.

$$\sum_{i=0}^k 2^i = 2^{k+1} - 1$$

Full Binary Tree

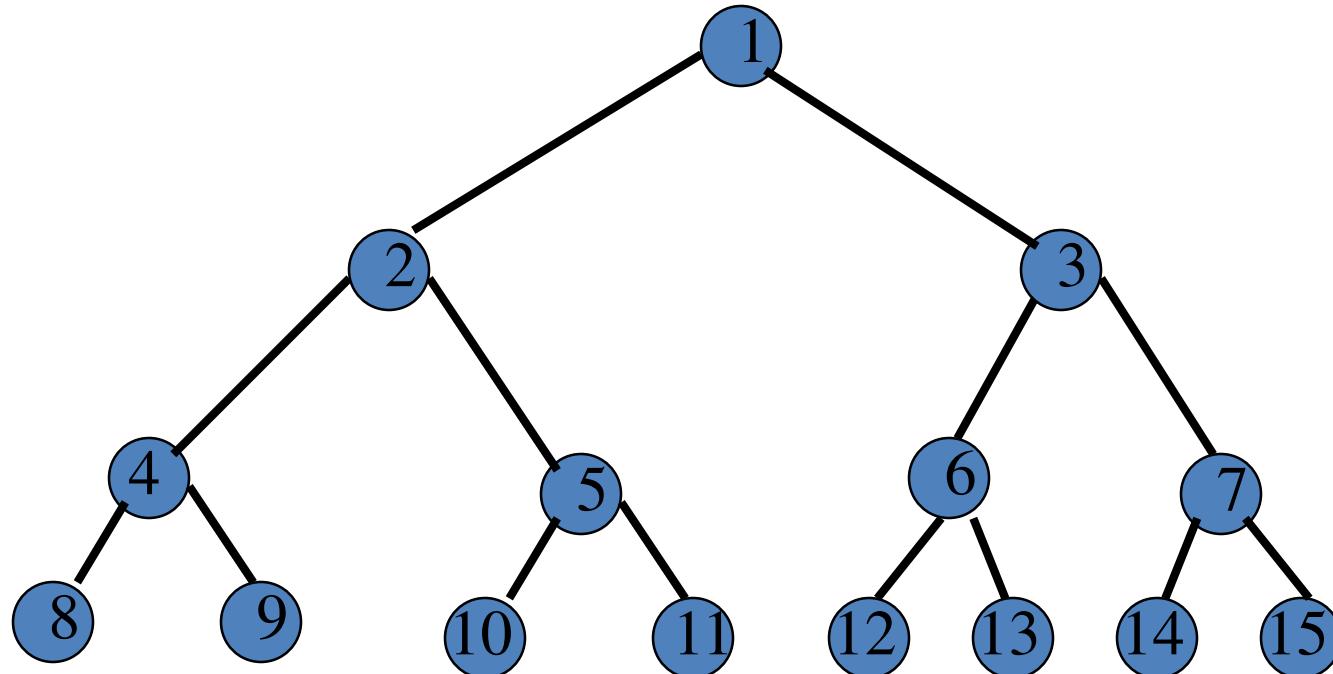
- A full binary tree of a **given height k** has $2^{k+1}-1$ nodes.



Height 3 full binary tree.

Labeling Nodes In A Full Binary Tree

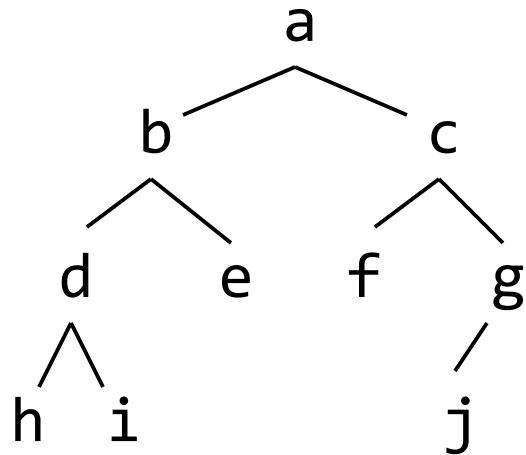
- Label the nodes **1** through **$2^{k+1} - 1$** .
- Label by levels from top to bottom.
- Within a level, label from left to right.



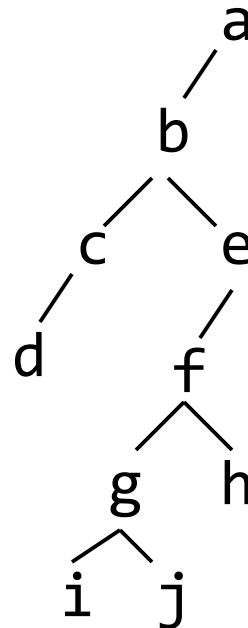
Special Trees

- An empty tree has a height of zero.
- A single node tree is a tree of height 1.
 - This is the only case where a node is both a root and a leaf.

Balance



A balanced binary tree



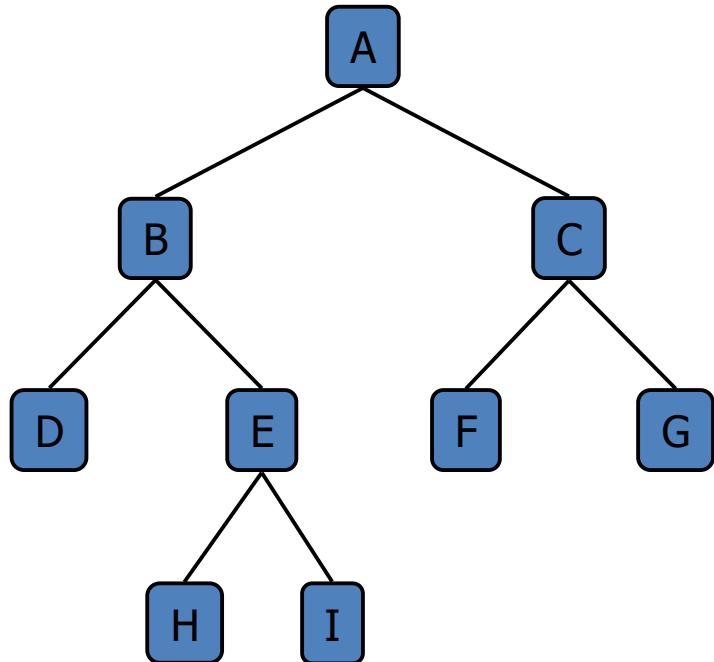
An unbalanced binary tree

- A **binary tree is balanced if every level above the lowest is “full”** (contains 2^n nodes)
- In most applications, a reasonably balanced binary tree is desirable

Binary Tree

- A binary tree is a tree with the following properties:
 - Each internal node has at most two children (degree of two)
 - The children of a node are an ordered pair
- We call the children of an internal node left child and right child
- Alternative recursive definition: a binary tree is either
 - a tree consisting of a single node, OR
 - a tree whose root has an ordered pair of children, each of which is a binary tree

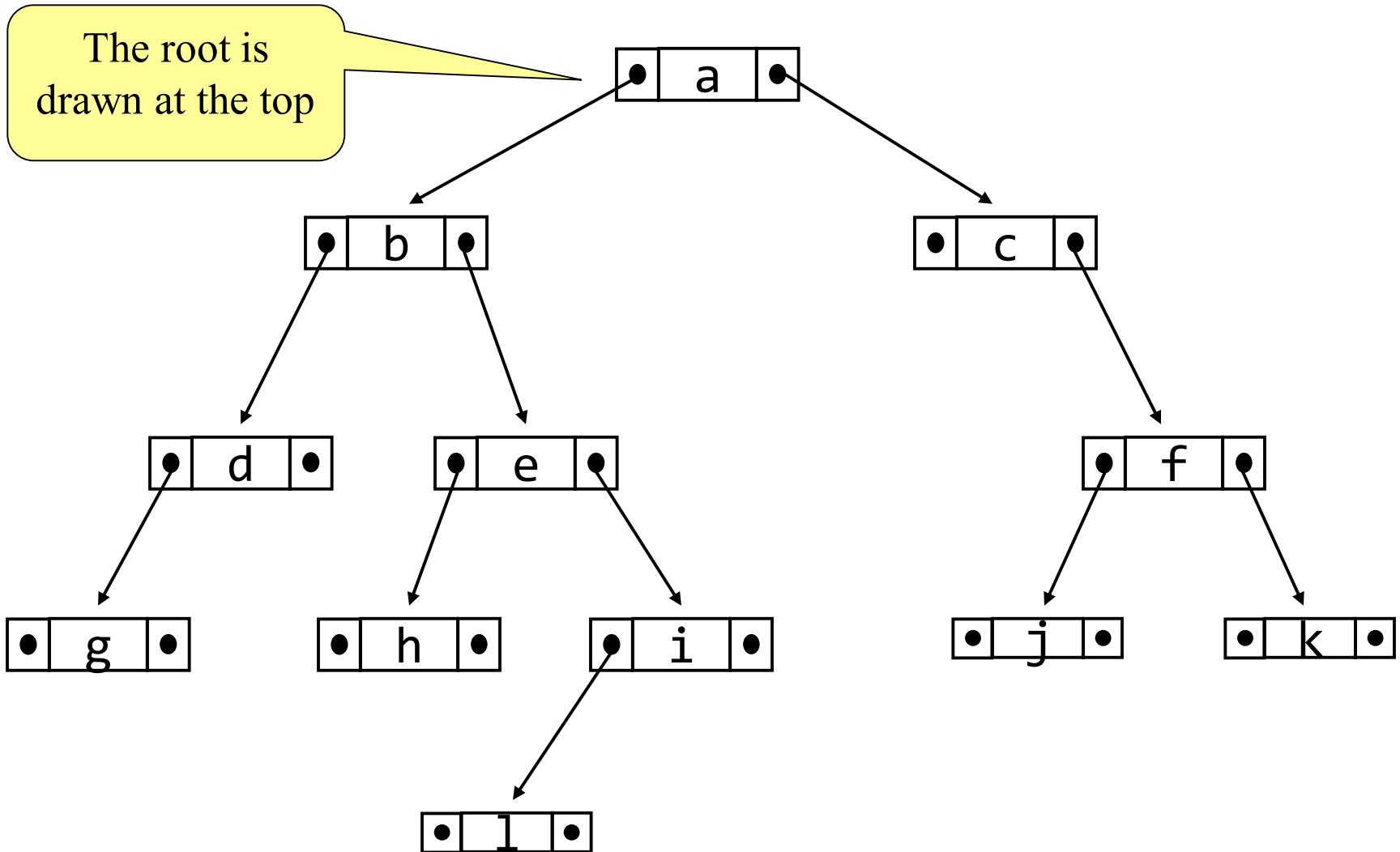
- Applications:
 - arithmetic expressions
 - decision processes
 - searching



Binary Trees

- According to the definition of trees, a node can have any number of children.
- A binary tree is restricted to only having 0, 1, or 2 children.
- A complete binary tree is one where all the levels are full with exception to the last level and it is filled from left to right.
- A full binary tree is one where if a node has a child, then it has two children.

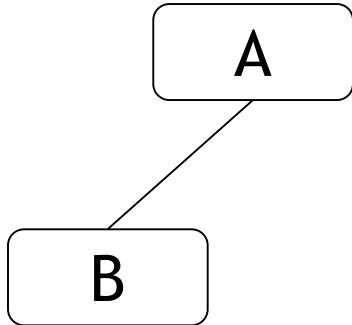
Picture of a binary tree



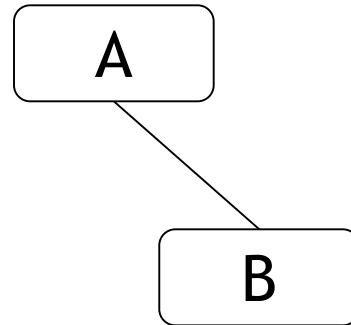
Left ≠ Right

- The following two binary trees are *different*:

LEFT CHILD



RIGHT CHILD



- In the first binary tree, **node A has a left child** but no right child; ***in the second, node A has a right child*** but no left child
- Put another way: Left and right are *not* relative terms

Binary Tree Traversals

- A traversal is where each node in a tree is visited and visited once
- For a tree of n nodes there are $n!$ traversals
- Of course most of those are hard to program
- **There are two very common traversals**
 - Breadth First
 - Depth First

Binary Tree Traversal- Breadth First

- In a breadth first traversal **all of the nodes on a given level are visited** and then all of the nodes on the next level are visited.
- Usually in a **left to right fashion**
- This is **implemented with a queue**

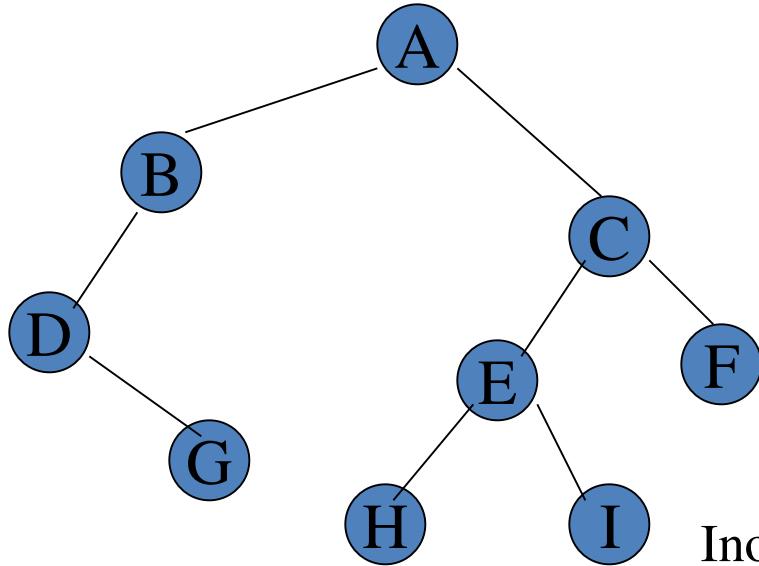
Binary Tree Traversal- Depth First

- In a depth first traversal **all the nodes on a branch are visited** before any others are visited
- **There are three common depth first traversals**
 - Inorder
 - Preorder
 - Postorder
- Each type has its use and specific application

Binary Tree traversals

- A binary tree is defined recursively: it consists of a **root**, a **left subtree**, and a **right subtree**
- To **traverse** (or **walk**) the binary tree is to **visit each node in the binary tree exactly once**
- **Tree traversals are naturally recursive**
- Since a binary tree has three “parts,” there are six possible ways to traverse the binary tree:
 - i) ROOT, LEFT, RIGHT
 - ii) LEFT, ROOT, RIGHT
 - iii) LEFT, RIGHT, ROOT
 - iv) root, right, left
 - v) right, root, left
 - vi) right, left, root

Traversing a binary tree



Preorder(Root Left Right)

1. Visit the **root**(visit the parent first and then left and right children;)
 2. Traverse the **left subtree** in preorder
 3. Traverse the **right subtree** in preorder
- preorder: ABDGCEHIF**

Inorder(Left Root Right)

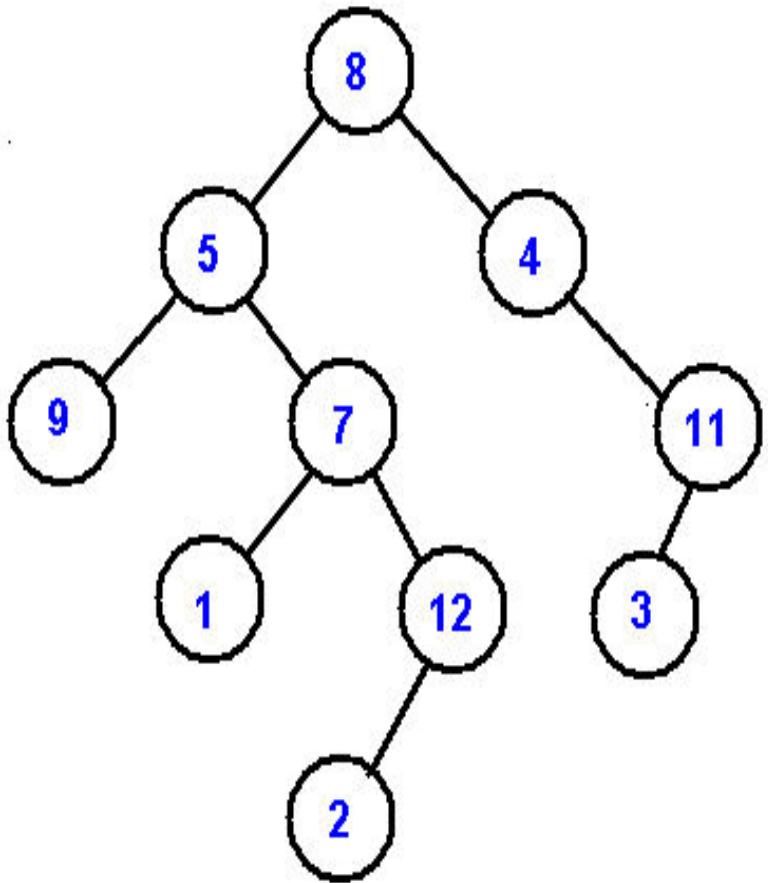
- 1.Traverse the **left subtree** in inorder
(visit the left child, then the parent and the right child;)
 2. Visit the **root**
 3. Traverse the **right subtree** in inorder
- inorder: DGBAHEICF**

Postorder(Left Right Root)

- 1.Traverse the **left subtree** in postorder
(visit left child, then the right child and then the parent;)
2. Traverse the **right subtree** in postorder
3. Visit the **root**

postorder: GDBHIEFCA

Example -1



PreOrder - 8, 5, 9, 7, 1, 12, 2, 4, 11, 3

InOrder - 9, 5, 1, 7, 2, 12, 8, 4, 3, 11

PostOrder - 9, 1, 2, 12, 7, 5, 3, 11, 4, 8

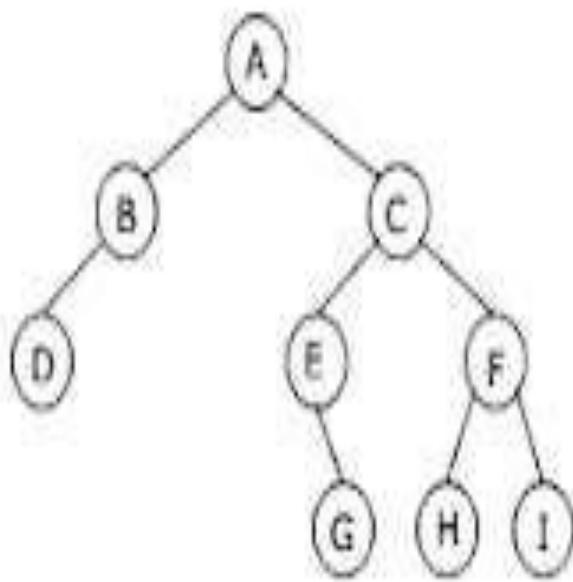
LevelOrder - 8, 5, 4, 9, 7, 11, 1, 12, 3, 2

Preorder(Root Left Right)

Inorder(Left Root Right)

Postorder(Left Right Root)

Example 2

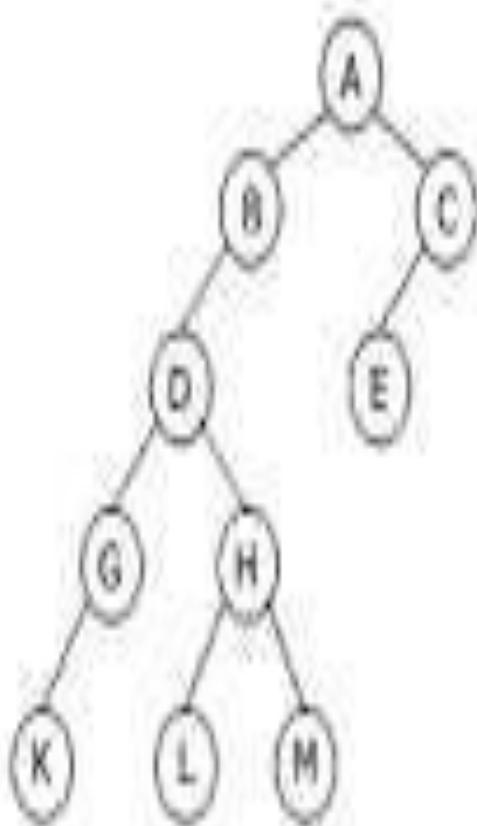


Binary Tree

- Preorder traversal yields:
A, B, D, C, E, G, F, H, I
- Postorder traversal yields:
D, B, G, E, H, I, F, C, A
- Inorder traversal yields:
D, B, A, E, G, C, H, F, I
- Levelorder traversal yields:
A, B, C, D, E, F, G, H, I

Pre, Post, Inorder and level order Traversing

Example-3



Binary Tree

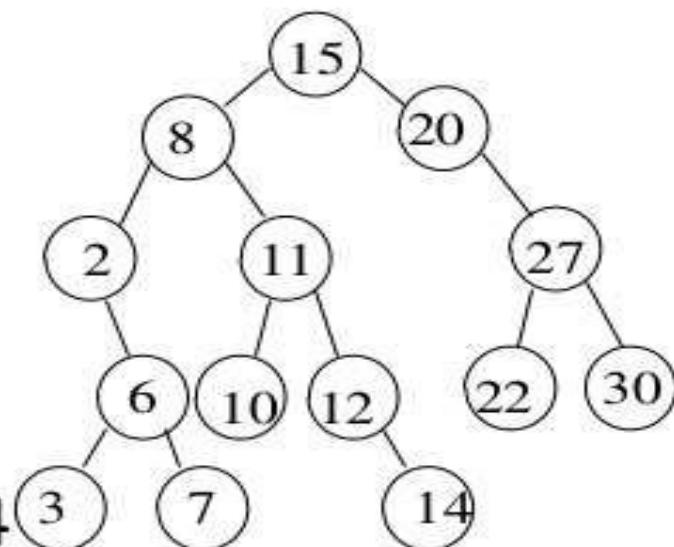
- Preorder traversal yields:
A, B, D, G, K, H, L, M, C, E
- Postorder traversal yields:
K, G, L, M, H, D, B, E, C, A
- Inorder traversal yields:
K, G, D, L, H, M, B, A, E, C

Pre, Post and Inorder Traversing

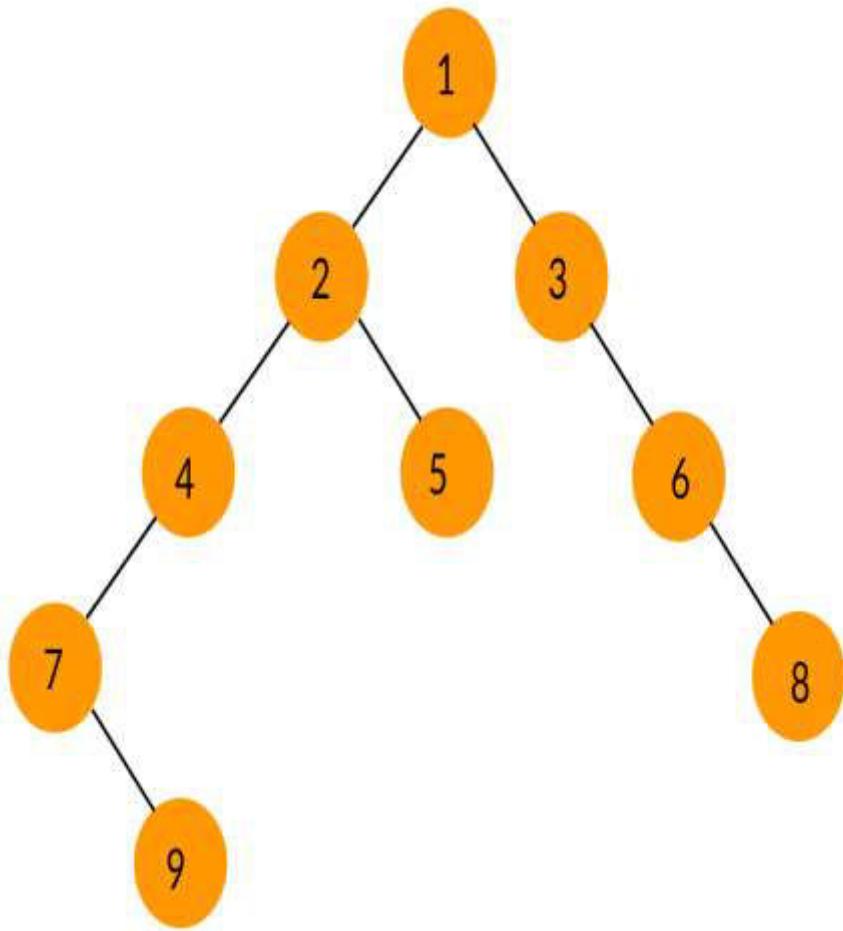
Example-4

Illustrations for Traversals (Contd.)

- Assume: visiting a node is printing its data
- **Preorder:** 15 8 2 6 3 7
11 10 12 14 20 27 22 30
- **Inorder:** 2 3 6 7 8 10 11
12 14 15 20 22 27 30
- **Postorder:** 3 7 6 2 10 14
12 11 8 22 30 27 20 15



Example 6



Inorder Traversal: 7 9 4 2 5 1 3 6 8

Preorder Traversal: 1 2 4 7 9 5 3 6 8

Postorder Traversal: 9 7 4 5 2 8 6 3 1

InOrder(root) visits nodes in the following order:

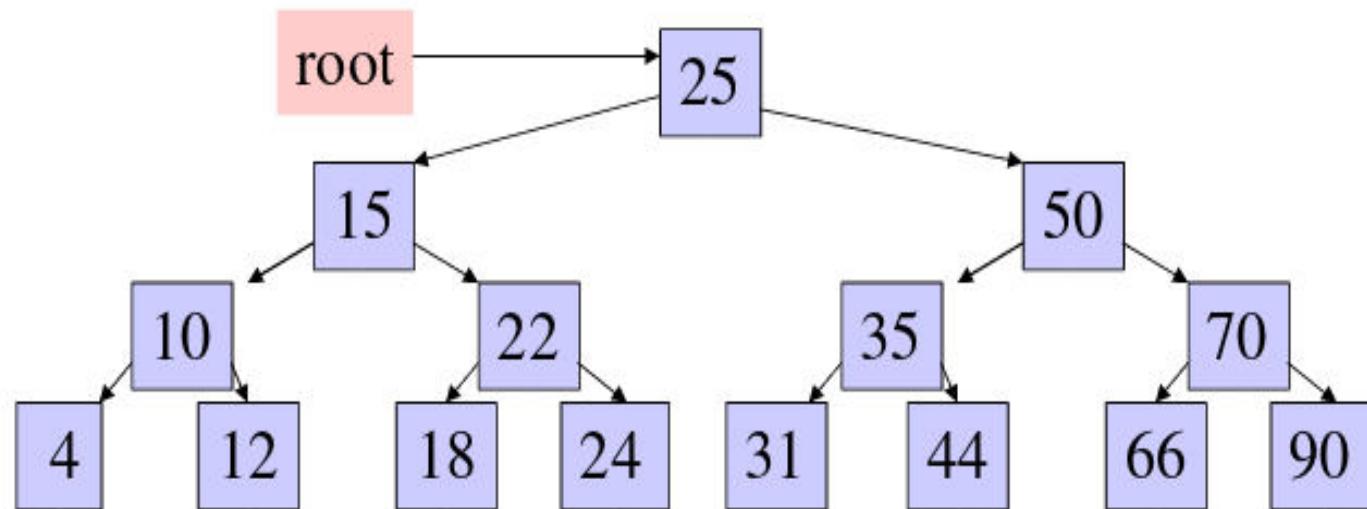
4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

A Pre-order traversal visits nodes in the following order:

25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

A Post-order traversal visits nodes in the following order:

4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25

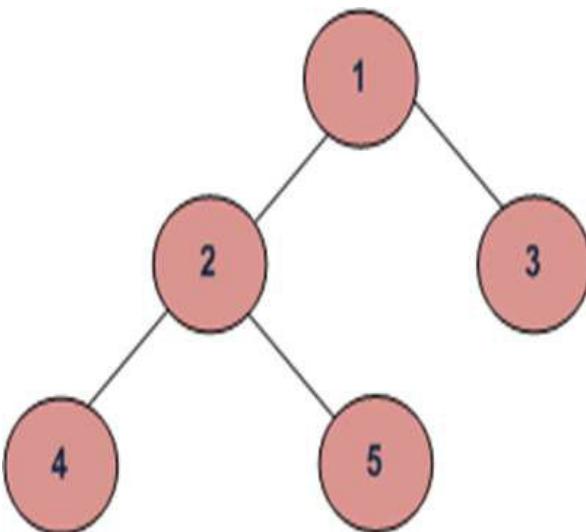


Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, **trees can be traversed in different ways. Following are the generally used ways for traversing trees**

Depth First TREE Traversals:

- (a) Inorder (Left, Root, Right) : 4 2 5 1 3
- (b) Preorder (Root, Left, Right) : 1 2 4 5 3
- (c) Postorder (Left, Right, Root) : 4 5 2 3 1

Breadth First or Level Order Traversal : 1 2
3 4 5



Depth First Traversals:

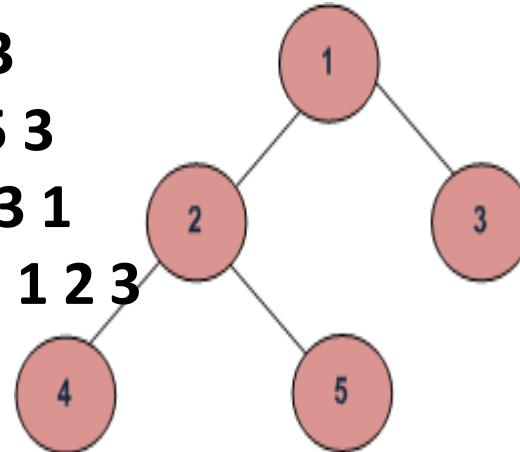
(a) Inorder (Left, Root, Right) : 4 2 5 1 3

(b) Preorder (Root, Left, Right) : 1 2 4 5 3

(c) Postorder (Left, Right, Root) : 4 5 2 3 1

Breadth First or Level Order Traversal : 1 2 3

4 5



Uses of Inorder

In case of *binary search trees (BST)*, *Inorder traversal gives nodes in non-decreasing order*.

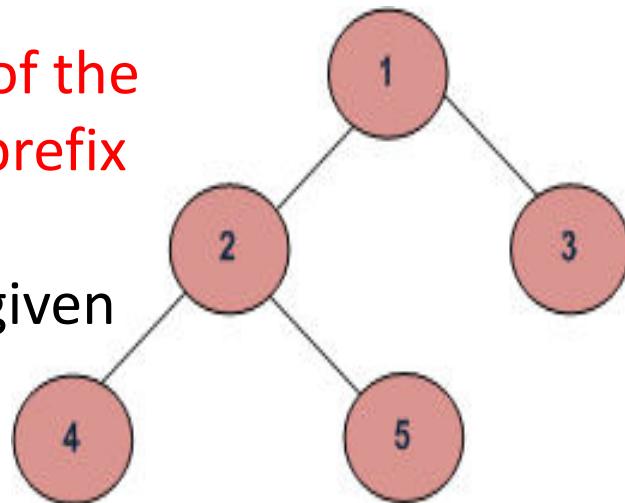
To get nodes of BST in non-increasing order, a variation of Inorder traversal where Inorder traversals reversed, can be used.

Example: Inorder traversal for the above given figure is 4 2 5 1 3.

Uses of Preorder

Preorder traversal is used **to create a copy of the tree**. Preorder traversal is also **used to get prefix expression on of an expression tree**.

Example: Preorder traversal for the above given figure is 1 2 4 5 3.



Uses of Postorder

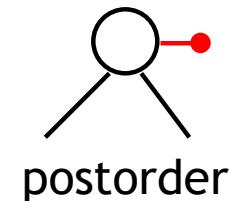
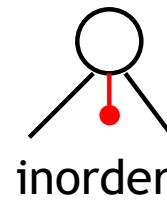
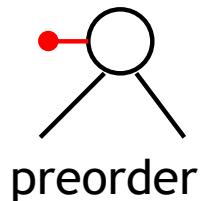
Postorder traversal is used **to delete the tree**.

Postorder traversal is also useful to get **the postfix expression of an expression tree**.

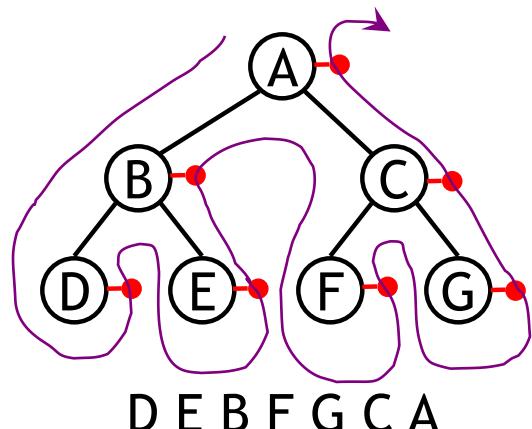
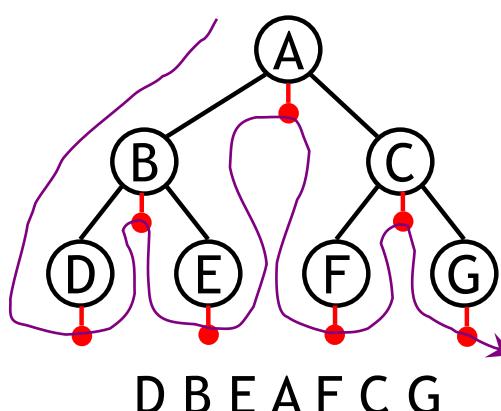
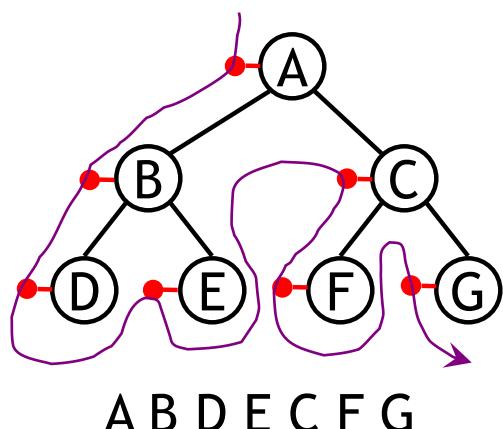
Example: Postorder traversal for the above given figure is 4 5 2 3 1.

Tree traversals using “flags”

- 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:



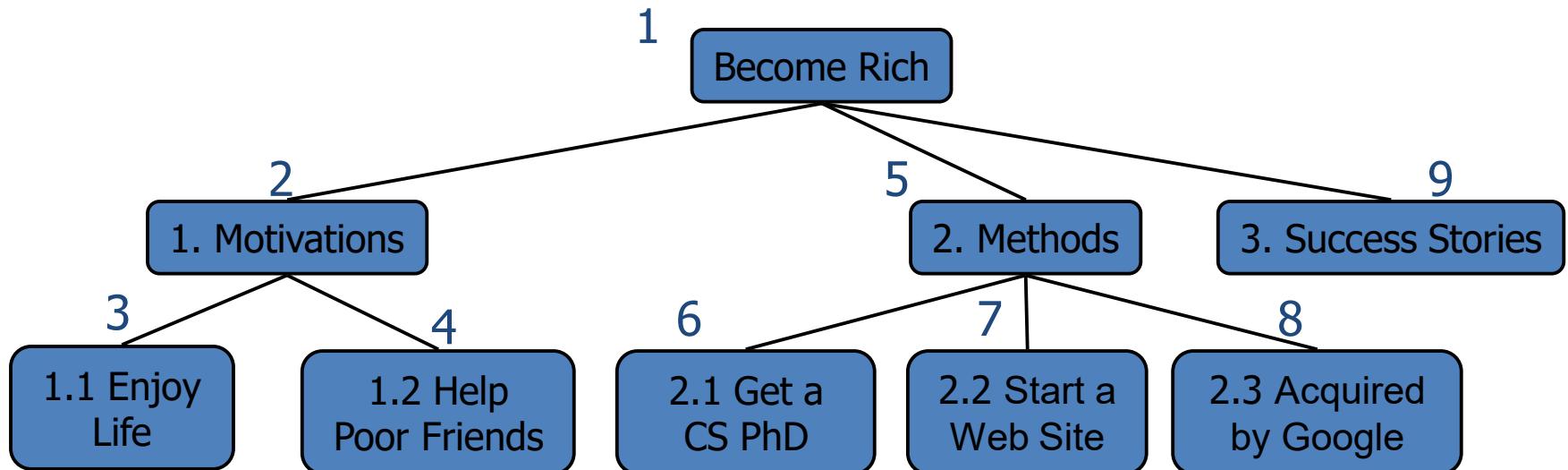
- To traverse the tree, collect the flags:



Preorder Traversal

- A traversal visits the nodes of a tree in a systematic manner
- In a preorder traversal, a node is visited before its descendants
- Application: print a structured document

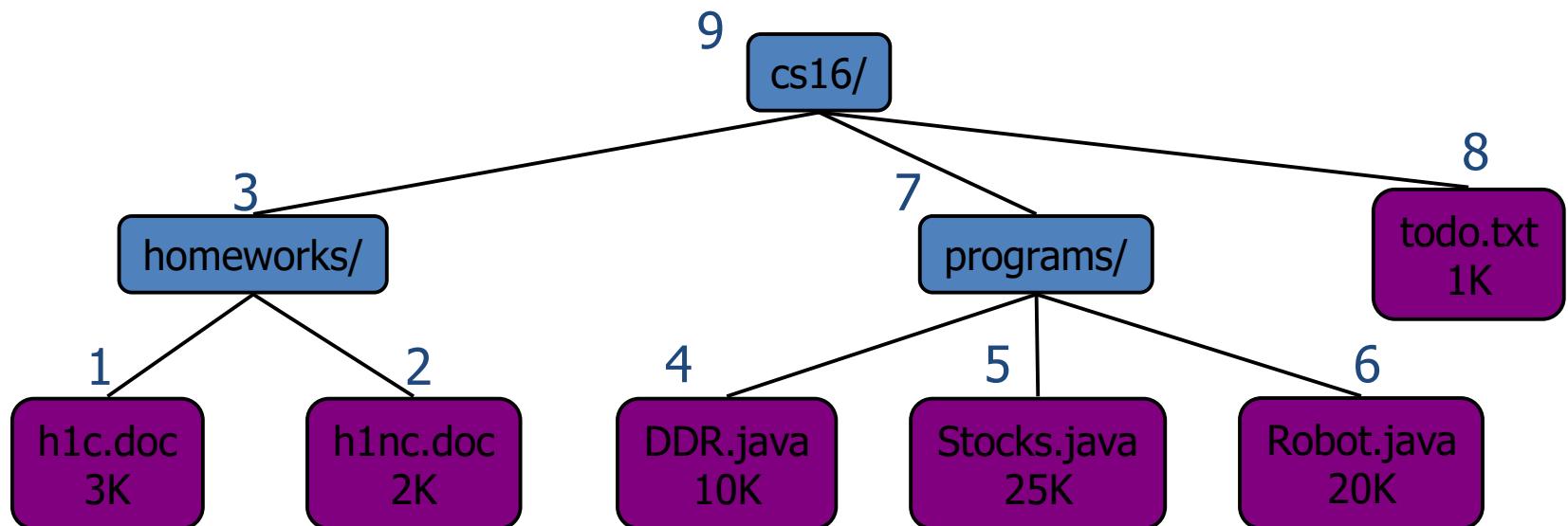
```
Algorithm preOrder(v)
    visit(v)
    for each child w of v
        preorder (w)
```



Postorder Traversal

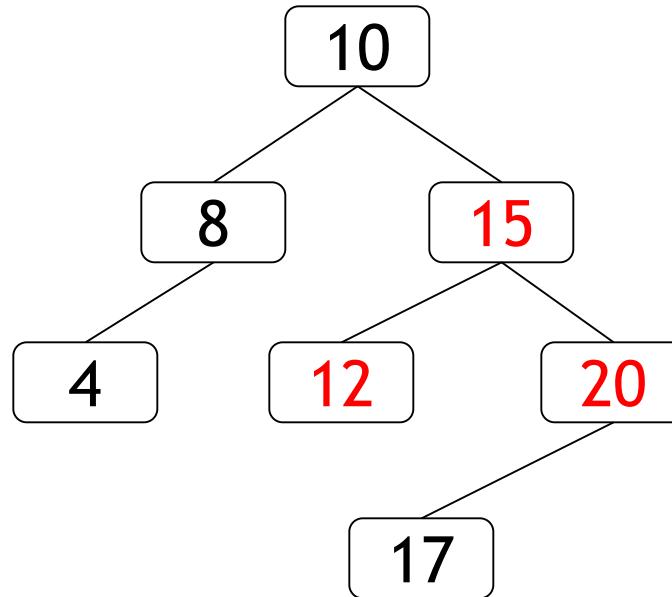
- In a postorder traversal, a node is visited after its descendants
- Application: compute space used by files in a directory and its subdirectories

```
Algorithm postOrder(v)
  for each child w of v
    postOrder (w)
    visit(v)
```



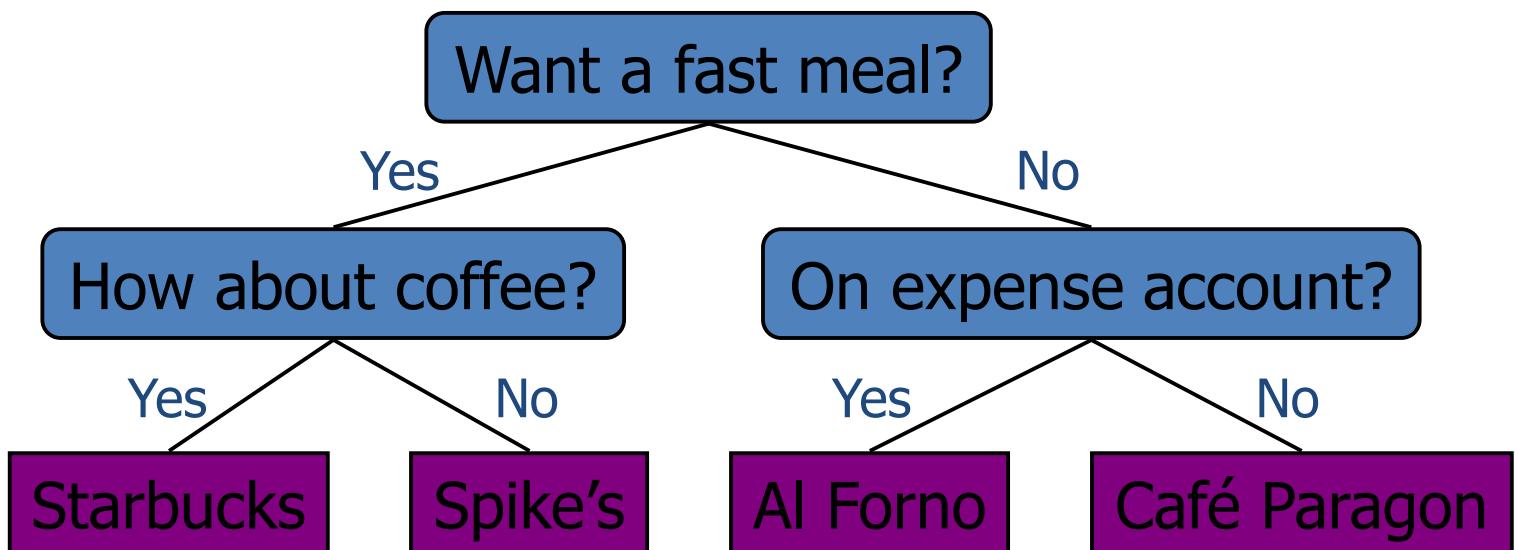
Sorted binary trees (Binary Search Trees, BST)

- A binary tree is sorted if **every node in the tree is larger than (or equal to) its left descendants, and smaller than (or equal to) its right descendants**
- Equal nodes can go either on the left or the right (but it has to be consistent)



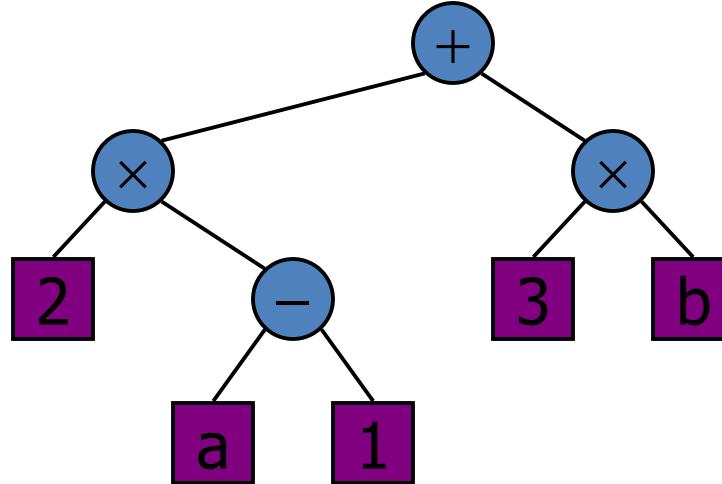
Decision Tree

- Binary tree associated with a decision process
 - internal nodes: questions with yes/no answer
 - external nodes: decisions
- Example: dining decision



Arithmetic Expression Tree

- Binary tree associated with an arithmetic expression
 - internal nodes: operators
 - external nodes: operands
- Example: arithmetic expression tree for the expression $(2 \times (a - 1)) + (3 \times b)$

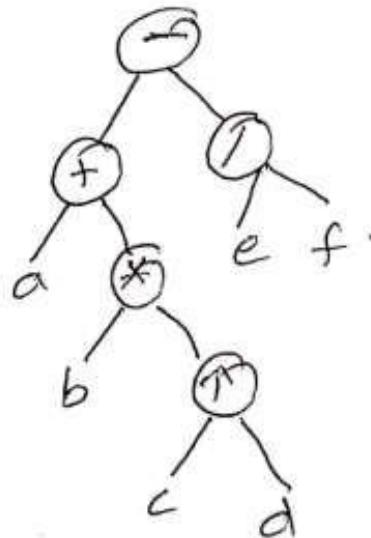
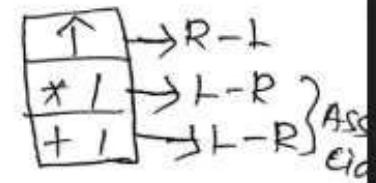
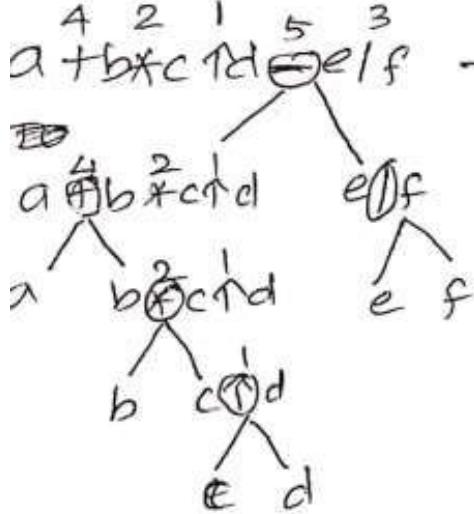


• CONVERTING EXPRESSION INTO BINARY TREE

Converting given expression into a tree.
 Given an expression $a+b*c\dagger d-e/f$

1. The Precedence of operator is given as

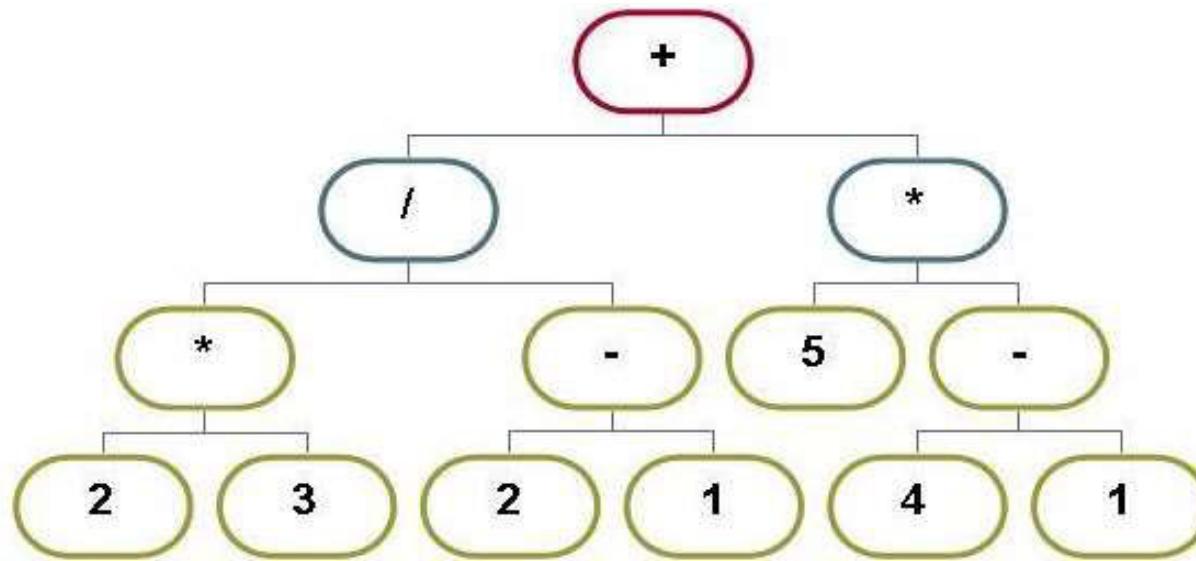
$\begin{matrix} 4 & 2 & 1 & 5 & 3 \\ a & + & b & * & c \end{matrix} \dagger d - e/f$ - [To construct tree - Start with highest number - i.e., 5] also root



CONVERTING EXPRESSION INTO BINARY TREE

$2 * 3 / (2 - 1) + 5 * (4 - 1)$

3 4 1 6 5 2



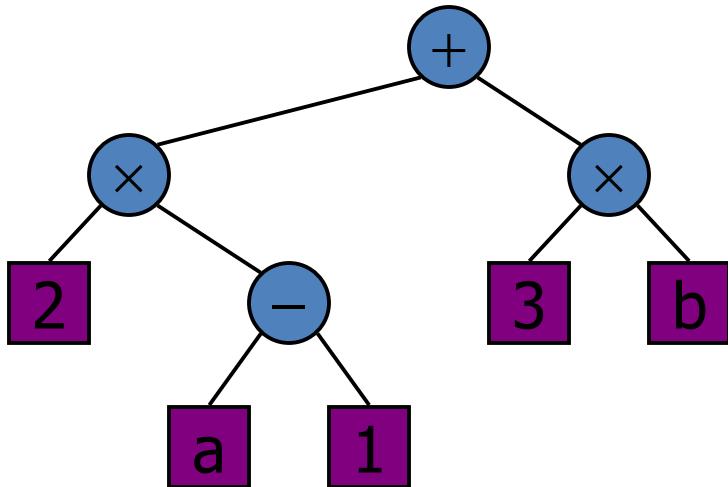
$\wedge \rightarrow R-L-1$

$*, / \rightarrow L-R-2$

$+, - \rightarrow L-R-3$

Print Arithmetic Expressions

- Specialization of an inorder traversal
 - print operand or operator when visiting node
 - print "(" before traversing left subtree
 - print ")" after traversing right subtree



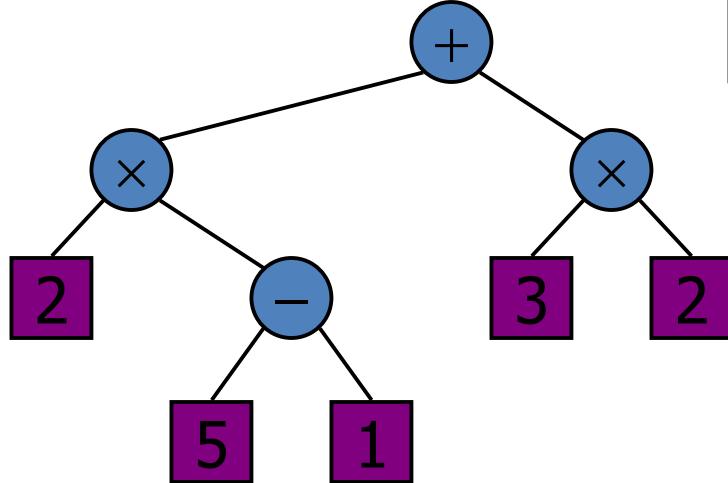
Algorithm *inOrder* (*v*)

```
if isInternal (v){  
    print("(")  
    inOrder (leftChild (v)))  
    print(v.element ())  
    if isInternal (v){  
        inOrder (rightChild (v)))  
        print ("")})
```

$$((2 \times (a - 1)) + (3 \times b))$$

Evaluate Arithmetic Expressions

- recursive method returning the value of a subtree
- when visiting an internal node, combine the values of the subtrees



Algorithm *evalExpr(v)*

if *isExternal (v)*

return *v.element ()*

else

x \leftarrow *evalExpr(leftChild (v))*

y \leftarrow *evalExpr(rightChild (v))*

\diamond \leftarrow operator stored at *v*

return *x* \diamond *y*

Binary Search Trees(BST)

- A binary search tree (BST) is a binary tree that has the following property:

For each node n of the tree, all values stored in its left subtree are less than value v stored in n , and all values stored in the right subtree are greater than v .

- This definition excludes the case of duplicates. They can be included and would be put in the right subtree.

Binary Search Tree(BST)

The following is definition of Binary Search Tree(BST) :

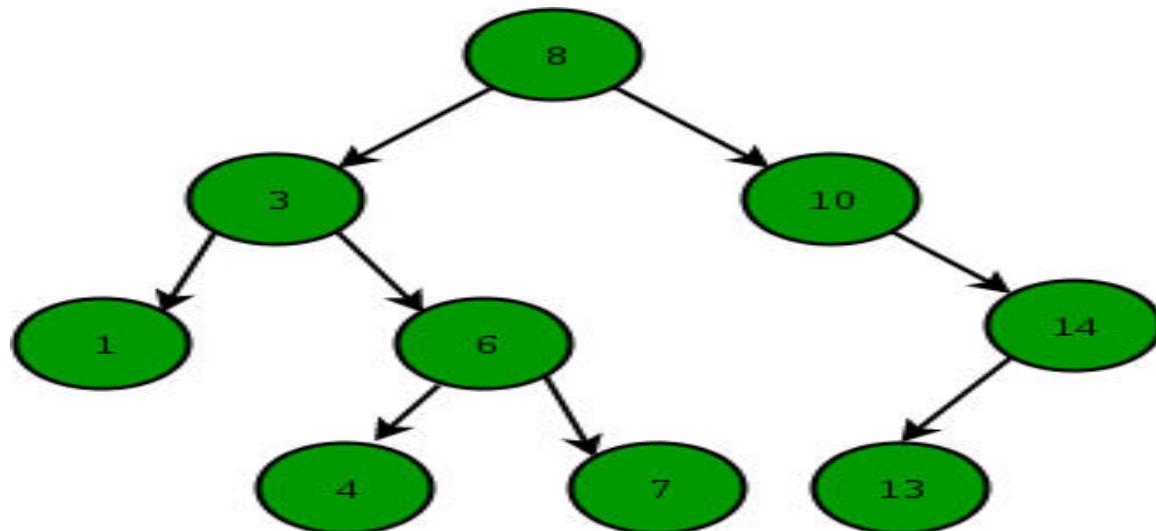
- Binary Search Tree, is a node-based binary tree data structure which has the following properties:
- The **left subtree** of a node contains only nodes with keys lesser than the node's key.
- The **right subtree** of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.

There must be no duplicate nodes.

• **Searching a key**

To search a given key in Binary Search Tree, we first compare it with root, if the key is present at root, we return root. If key is greater than root's key, we recur for right subtree of root node. Otherwise we recur for left subtree.

- Searching in binary tree has **worst case complexity of $O(n)$.**
- In general, **time complexity is $O(h)$ where h is height of BST.**



Given an array of elements – construct an BST

- Let's begin by first establishing some rules for Binary Search Trees:
- A parent node has, at most, 2 child nodes.
- The **left child node is always less than the parent node.**
- The **right child node is always greater than or equal to the parent node.**

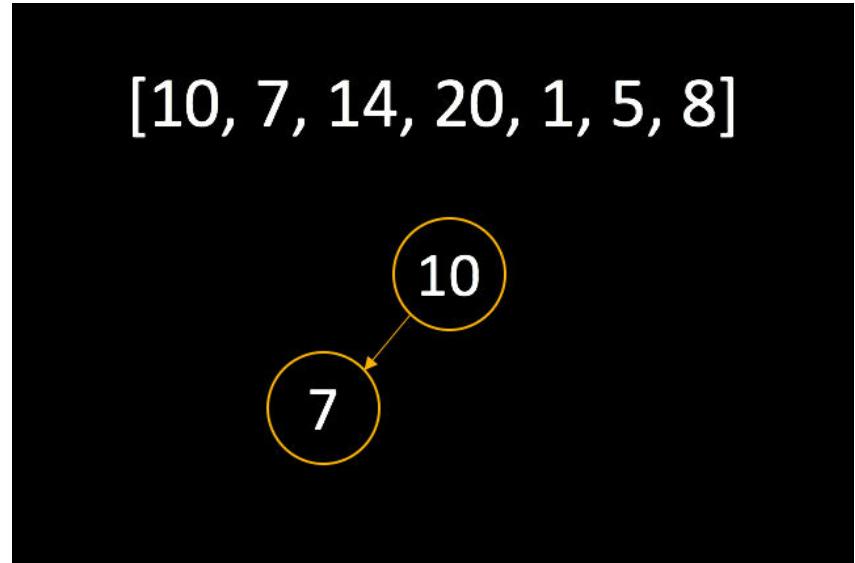
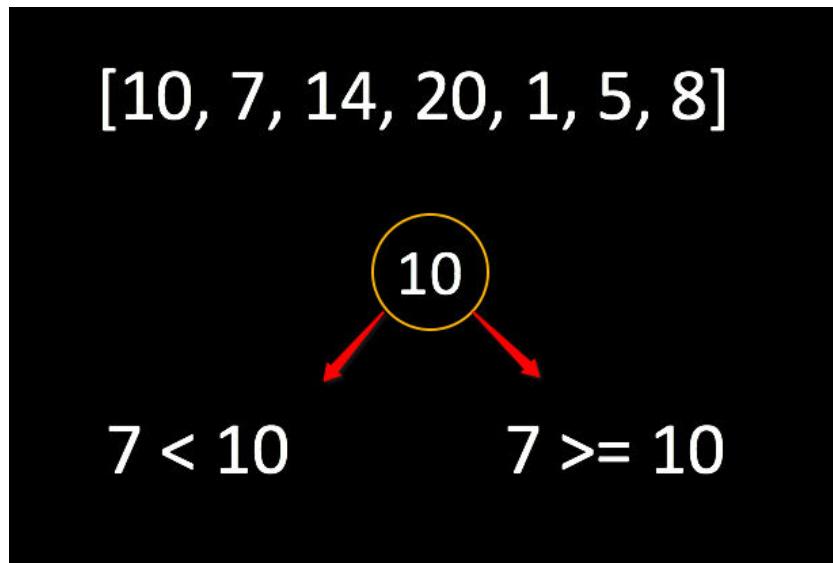
[10, 7, 14, 20, 1, 5, 8]

- Here is the array of elements
- create a Binary Search Tree
- The first value in the array is 10, so the first step in constructing the tree will be to make 10 the root node, as shown here:

[10, 7, 14, 20, 1, 5, 8]

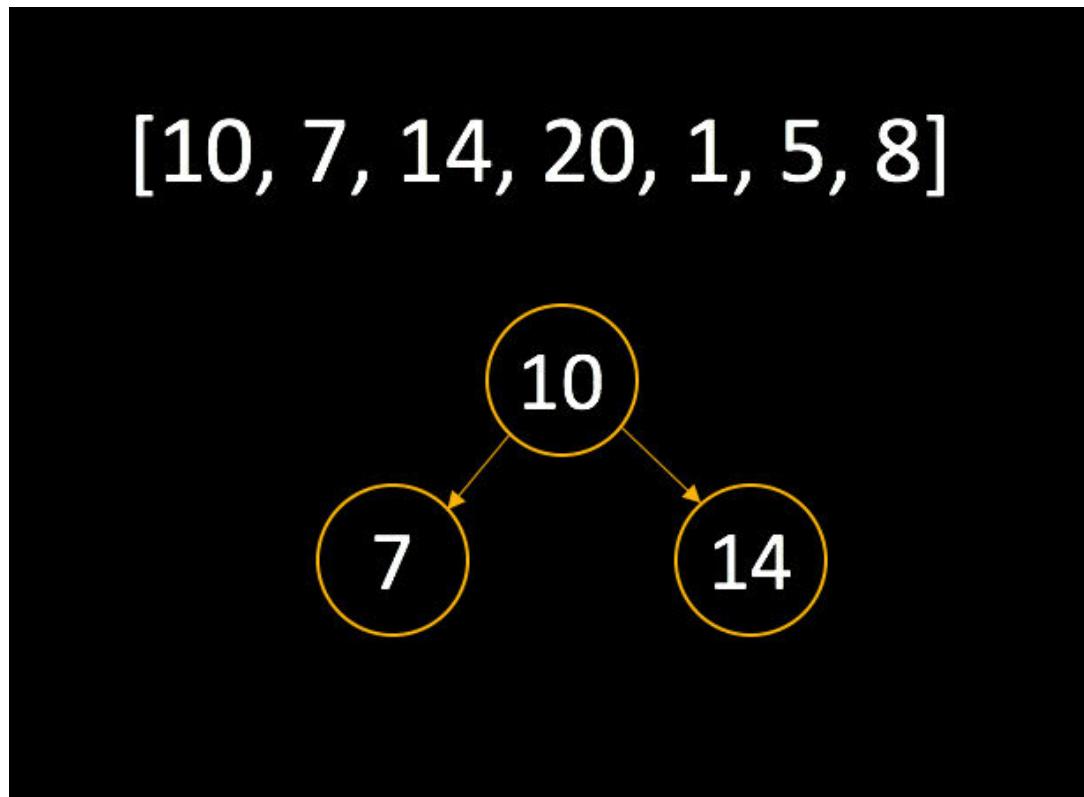


- The first step we'll take for adding the 7 to the tree will be to compare it to the root node: Since we know that the 7 is less than 10 we designate it as the left child node, as shown here.



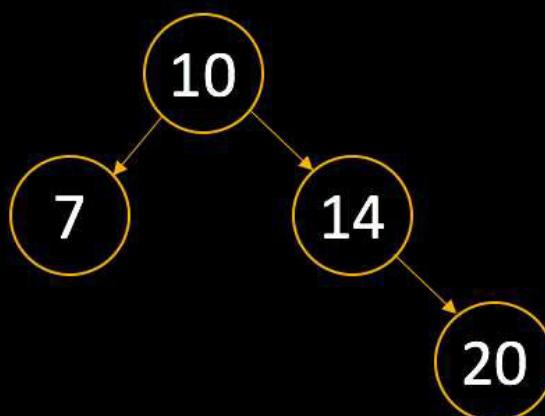
Recursively Perform Comparisons for Each Element

- Following the same pattern, we perform the same comparison with the 14 value in the array. Comparing the value of 14 to the root node of 10 we know that 14 is the right child.

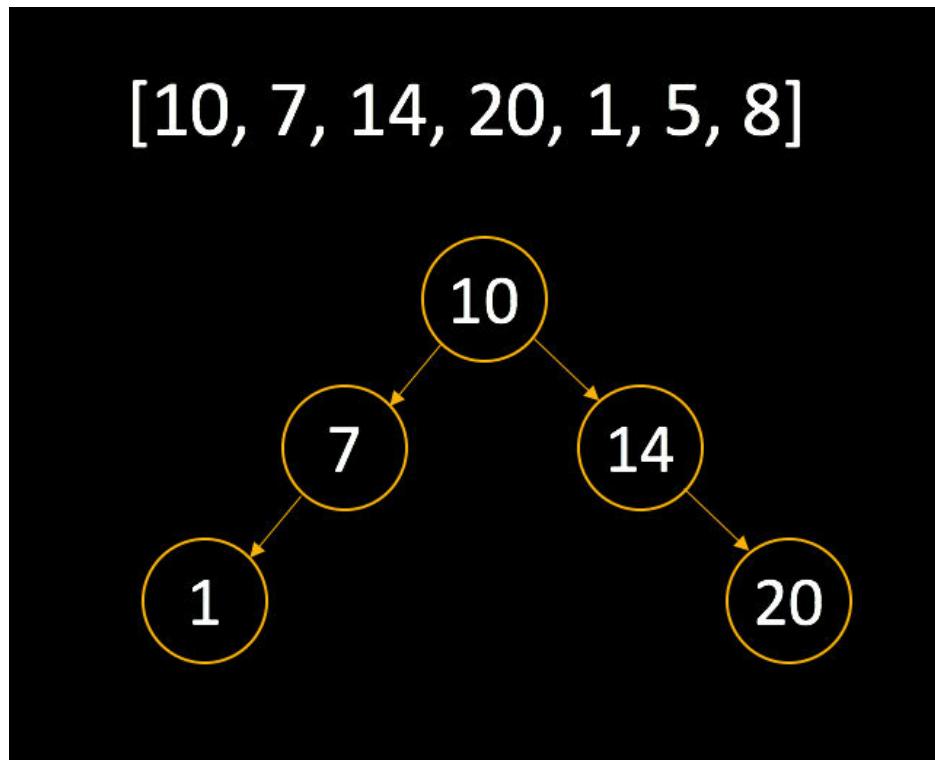


- Making our way through the array we come to the 20. We'll start with comparing the array to 10, which it's greater than. So we move to the right and compare it with 14, it's greater than 14 and 14 doesn't have any children to the right, so we make 20 the right child of 14.

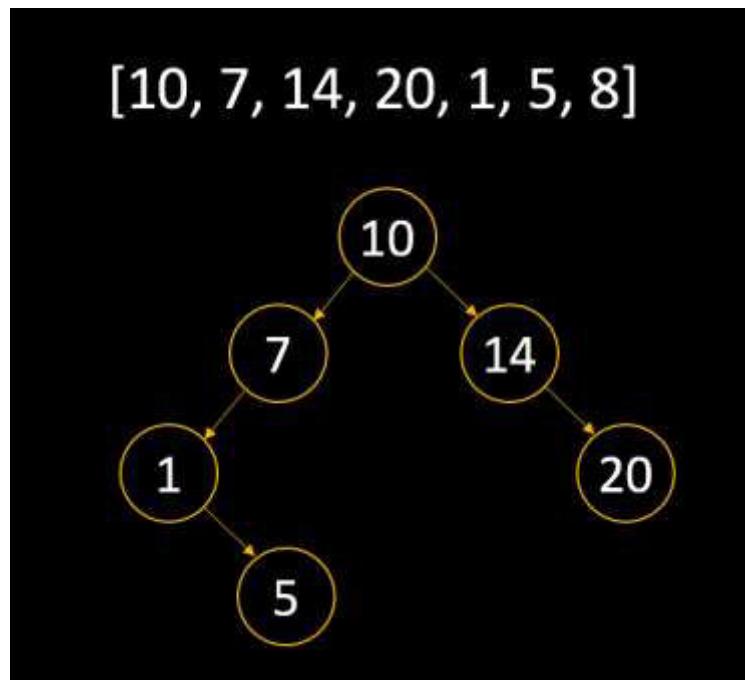
[10, 7, 14, 20, 1, 5, 8]



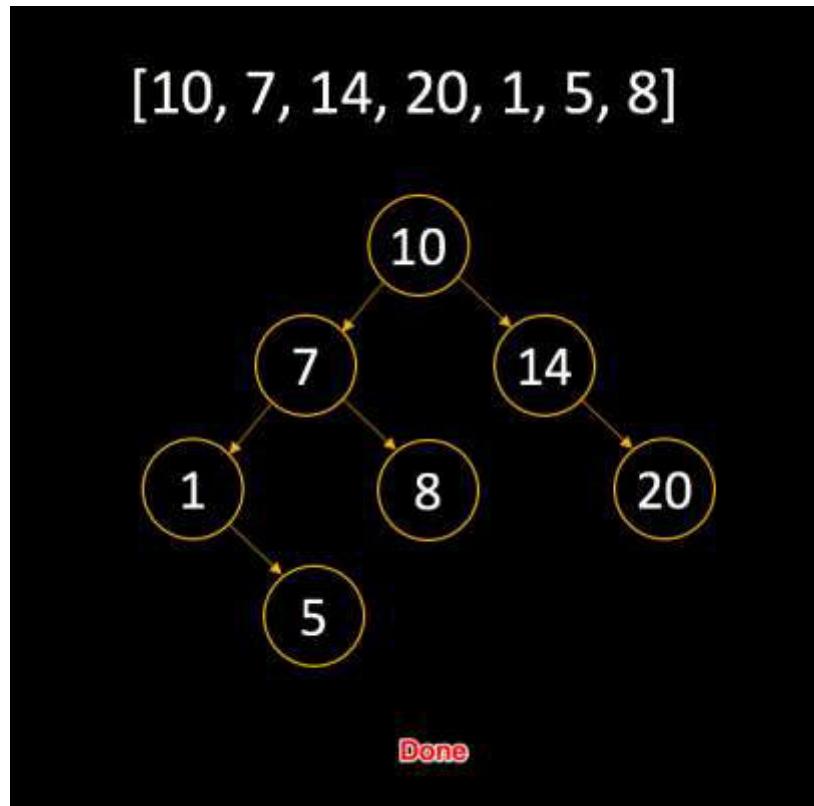
- Now we have the value 1. Following the same pattern as the other values we will compare 1 to 10, move to the left and compare it with 7, and finally make 1 the left child of the 7 node.



- With the 5 value we compare it to the 10. Since 5 is less than 10 we traverse to the left and compare it with 7. Since we know that 5 is less than 7 we continue down the tree and compare 5 to the 1 value. With 1 having no child nodes and 5 being greater than 1 we know to make 5 the right child of the 1 node.



- Lastly we will insert the value 8 into the tree. With 8 being less than 10 we move it to the left and compare it with 7. 8 is greater than 7, so we move it to the right and complete the tree making 8 the right child of 7.

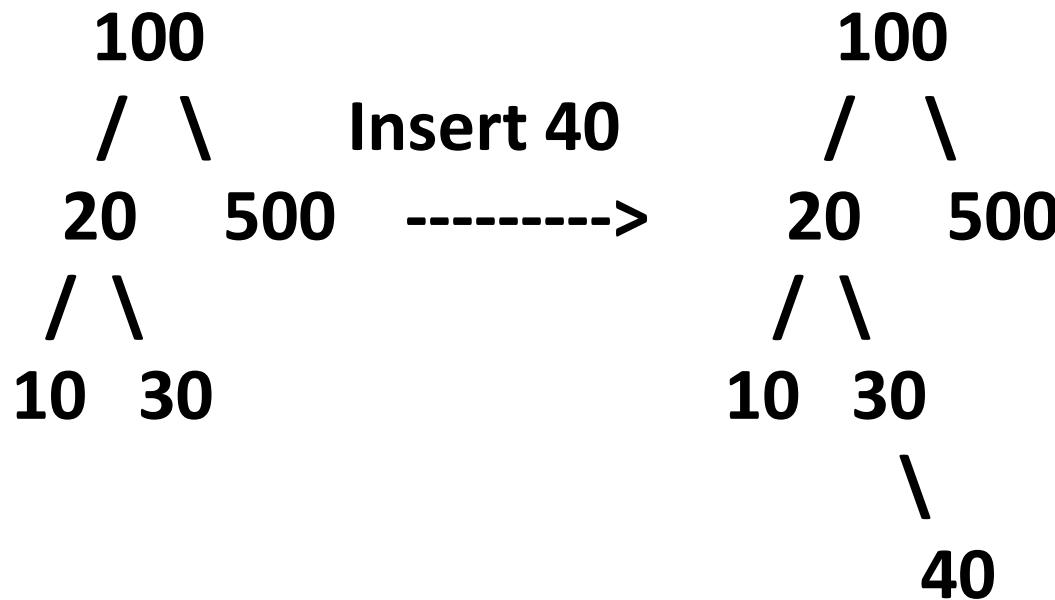


Inserting an element in BST

Insertion of a key

A new key is always inserted at leaf. We start searching a key from root till we hit a leaf node. Once a leaf node is found, the new node is added as a child of the leaf node.

worst case complexity of $O(n)$. In general, time complexity is $O(h)$.



Here is the general pseudocode for BSTINSERT(V,T) (assume V is not in T):

```
BSTINSERT(V,T) {  
    if T is empty  
        then T = create_singleton(V)  
    else if V > rootvalue(T)  
        then if T's right subtree exists  
            then BSTINSERT(V,T's right subtree)  
            else T's right subtree = create_singleton(V)  
        else if T's left subtree exists  
            then BSTINSERT(V,T's left subtree)  
            else T's left subtree = create_singleton(V) }
```

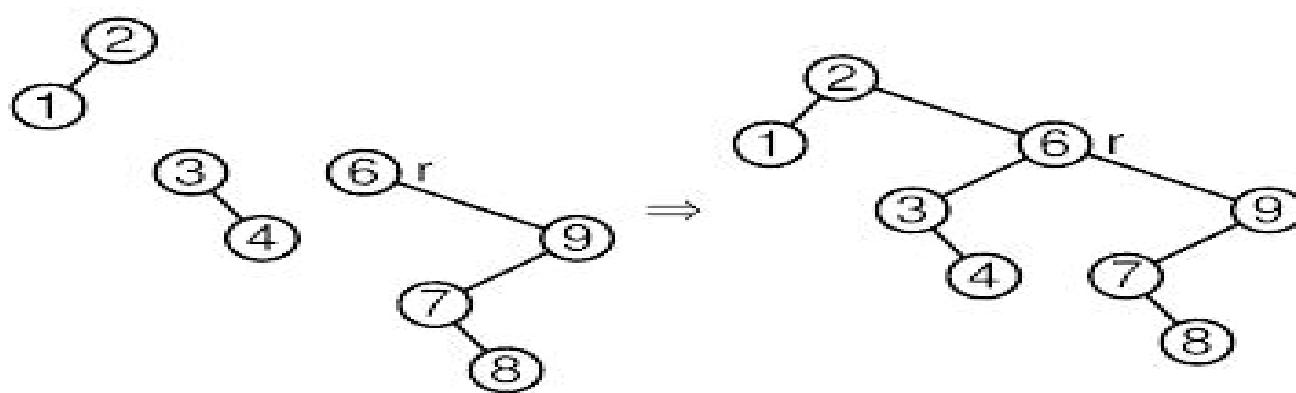
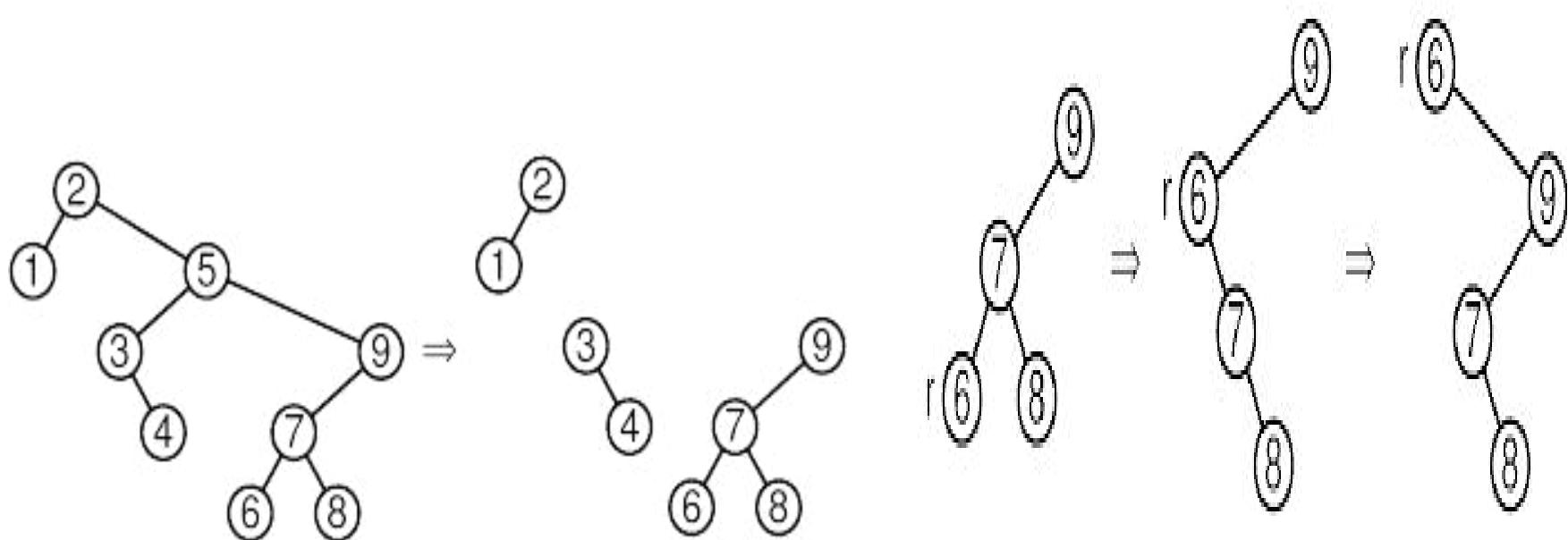
```
static Node Insert(Node root,int value) {  
    if(root == null){  
        Node node = new Node();  
        node.data = value;  
        root = node;  
    }  
  
    else{  
        if(value < root.data){  
            root.left = Insert(root.left, value);  
        }  
        else if(value > root.data)  
            root.right = Insert(root.right, value);  
    }  
    return root;  
}
```

Deletion

- Deletion poses a bigger problem
- When we delete we normally have two choices
 - 1. Deletion by merging**
 - 2. Deletion by copying**

Deletion by Merging

- Deletion by merging takes two subtrees and merges them together into one tree
- The idea is you have a node n to delete
 n can have two children
- So you find the smallest element in n 's right subtree
- we take the minimum node r in the former right subtree and repeatedly perform a right rotation on its parent, until it is the root of its subtree.
- The root of the left subtree replaces n



Deletion by copying

- This will simply swap values and reduce a difficult case to an easier one

1. If the node n to be deleted has no children,

- easy blow it away

2. If it has one child

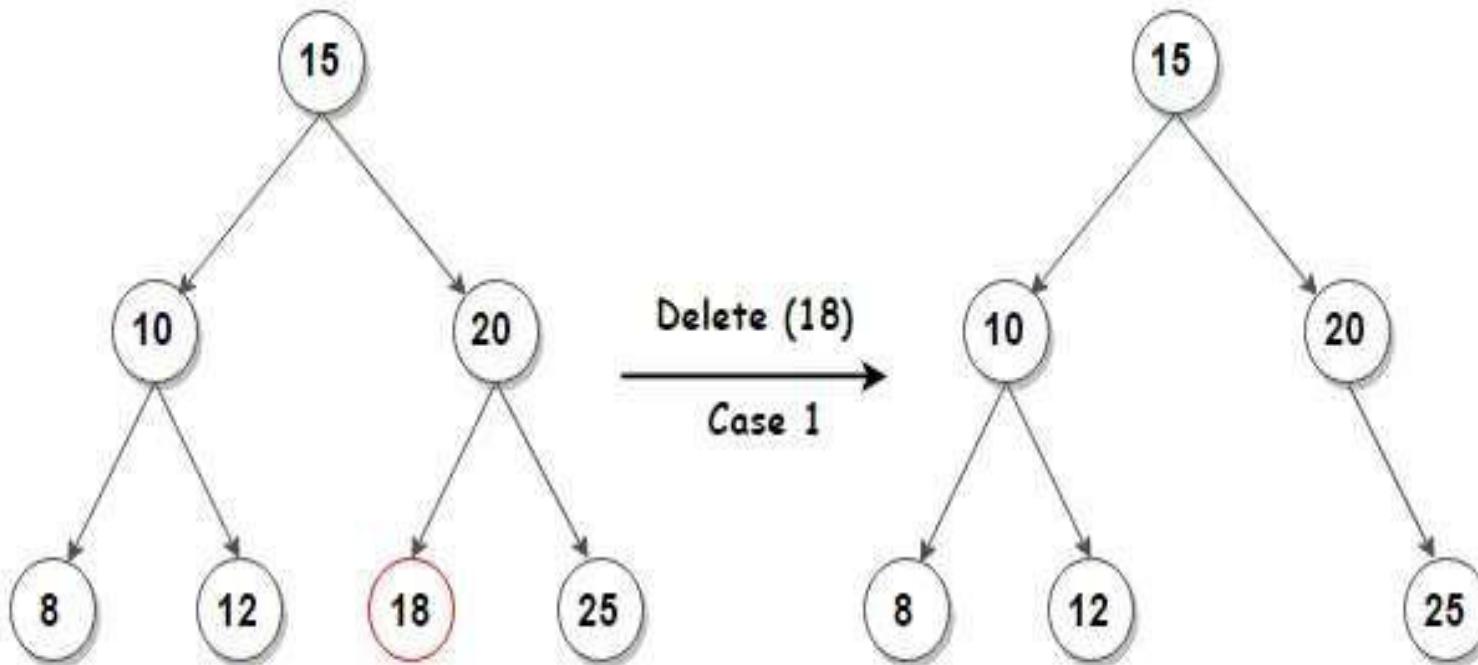
- Easy simply pass n's child pointer up, make n's parent point to n's child and blow n away

3. If n has two child,

- Now we have deletion by copying

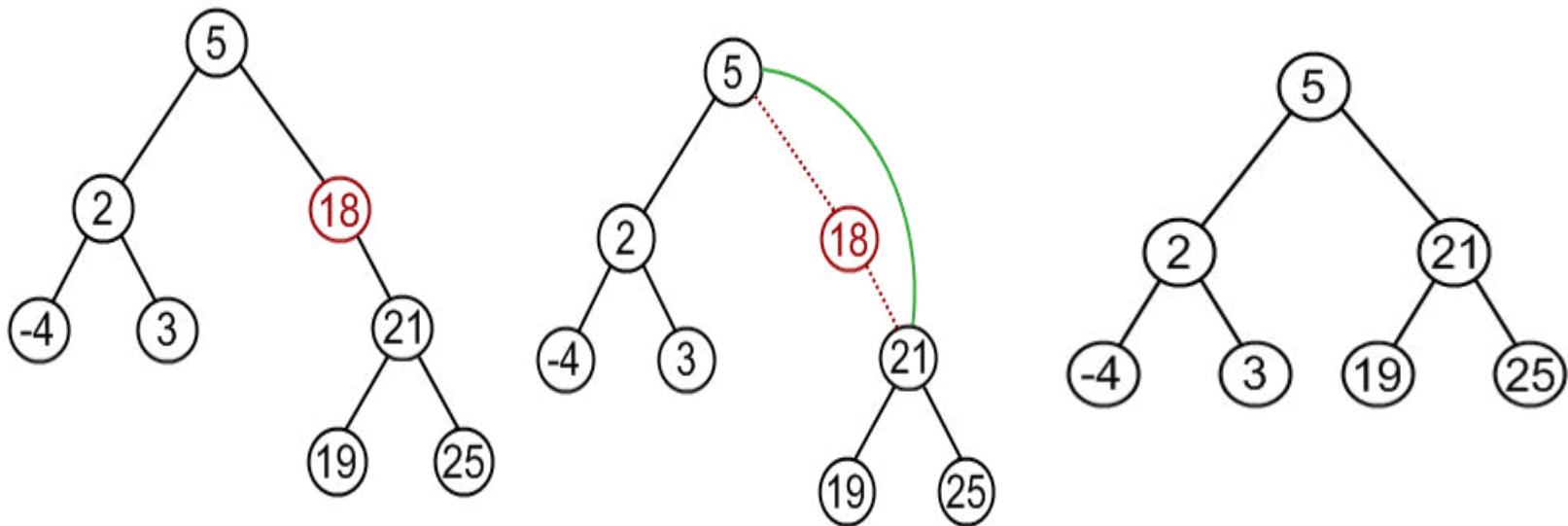
The worst case time complexity of delete operation is $O(h)$ where h is height of Binary Search Tree. In worst case, we may have to travel from root to the deepest leaf node.

- *1. If the node n to be deleted has no children(Leaf node)*



2. Node to be removed has one child.

- In this case, node is cut from the tree and algorithm links single child (with it's subtree) directly to the parent of the removed node.
- **Example.** Remove 18 from a BST.

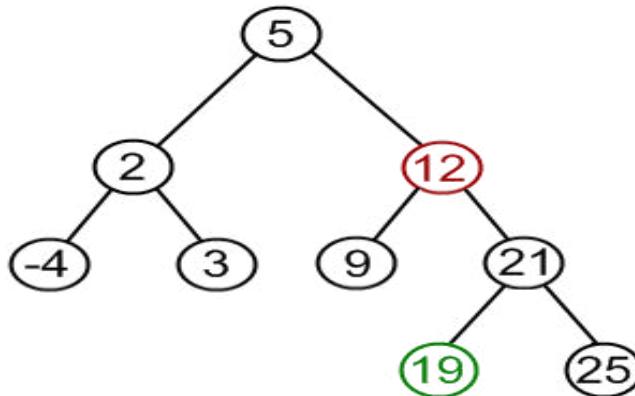
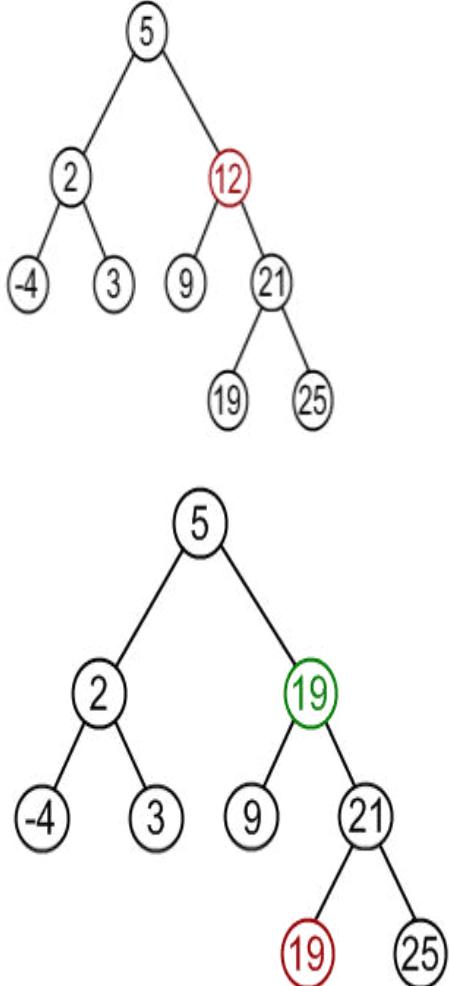


3. Node to be removed has two children.

- The approach used to remove a node, which has two children:
 - i) find a **minimum value** in the **right subtree of the node to be removed** [We also use inorder successor in deleting a node ie., **(smallest in the right subtree)**]
 - ii) replace value of the node to be removed with found minimum. Now, right subtree contains a duplicate!
 - iii) apply remove() to the right subtree to remove a duplicate.
- Notice, that the node with minimum value has no left child and, therefore, its removal may result in first or second cases only.

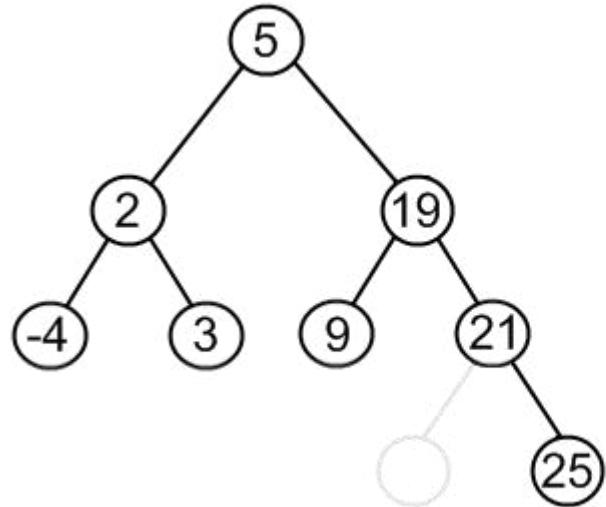
Example: Remove 12 from a BST.

Find minimum element in the right subtree of the node to be removed. **In current example it is 19.** We also use inorder successor in deleting a node



Replace 12 with 19. Notice, that **only values are replaced, not nodes.** Now we have two nodes with the same value.

Remove 19 from the left subtree.



Procedure :

1. At first locate the node **to be deleted**.

2. If the node is a leaf node :

- i. If the node is left child of the parent , make null the left pointer of its parent node and free the space for the node.
- ii. If the node is right child of the parent , make null the right pointer of its parent node and free the space for the node.

3. If the node has one child

- i. If the node to be deleted is a left chile of its parent , then make link between the left pointer of its parent node and the child node to be deleted. Then delete the node.
- ii.If the node to be deleted is a right child of its parent , then make link between the right pointer of its parent node and the child node to be deleted. Then delete the node.

4. If the node to be deleted has two child :

- i. Locate the node with Minimum value from the Right sub-tree of the node to be deleted.
 - ii. Replace the node value to be deleted by the node found in step 4(i)
5. Now we get updated output

```
/* Given a binary search tree and a key, this function deletes the key  
and returns the new root */  
struct node* deleteNode(struct node* root, int key)  
{  
    // base case  
    if (root == NULL) return root;  
  
    // If the key to be deleted is smaller than the root's key,  
    // then it lies in left subtree  
    if (key < root->key)  
        root->left = deleteNode(root->left, key);  
  
    // If the key to be deleted is greater than the root's key,  
    // then it lies in right subtree  
    else if (key > root->key)  
        root->right = deleteNode(root->right, key);  
  
    // if key is same as root's key, then This is the node  
    // to be deleted
```

```
else
```

```
{  
    // node with only one child or no child  
    if (root->left == NULL)  
    {  
        struct node *temp = root->right;  
        free(root);  
        return temp;  
    }  
    else if (root->right == NULL)  
    {  
        struct node *temp = root->left;  
        free(root);  
        return temp;  
    }
```

**// node with two children: Get the inorder successor
(smallest in the right subtree)**

```
struct node* temp = minValueNode(root->right);
```

// Copy the inorder successor's content to this node

```
root->key = temp->key;
```

// Delete the inorder successor

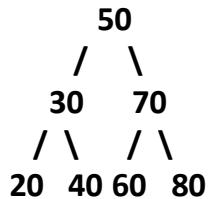
```
root->right = deleteNode(root->right, temp->key);
```

```
}
```

```
return root;
```

```
}
```

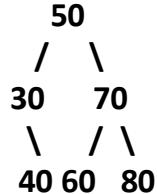
Let us create following BST



Inorder traversal of the given tree

20 30 40 50 60 70 80

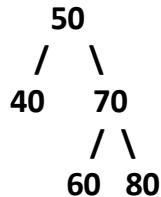
Delete 20 //Leaf node



Inorder traversal of the modified tree

30 40 50 60 70 80

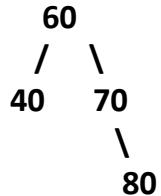
Delete 30 // deletion with one child



Inorder traversal of the modified tree

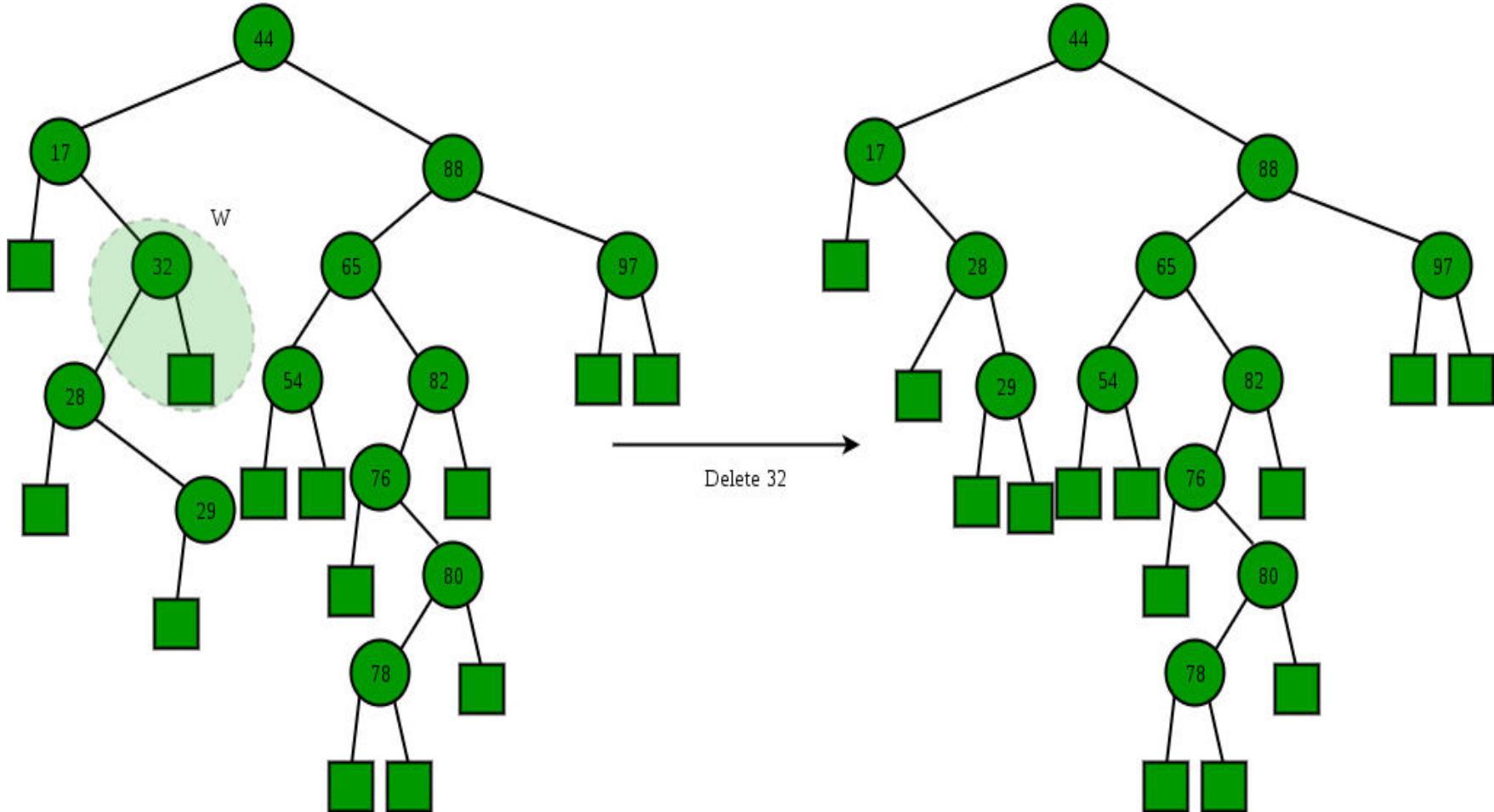
40 50 60 70 80

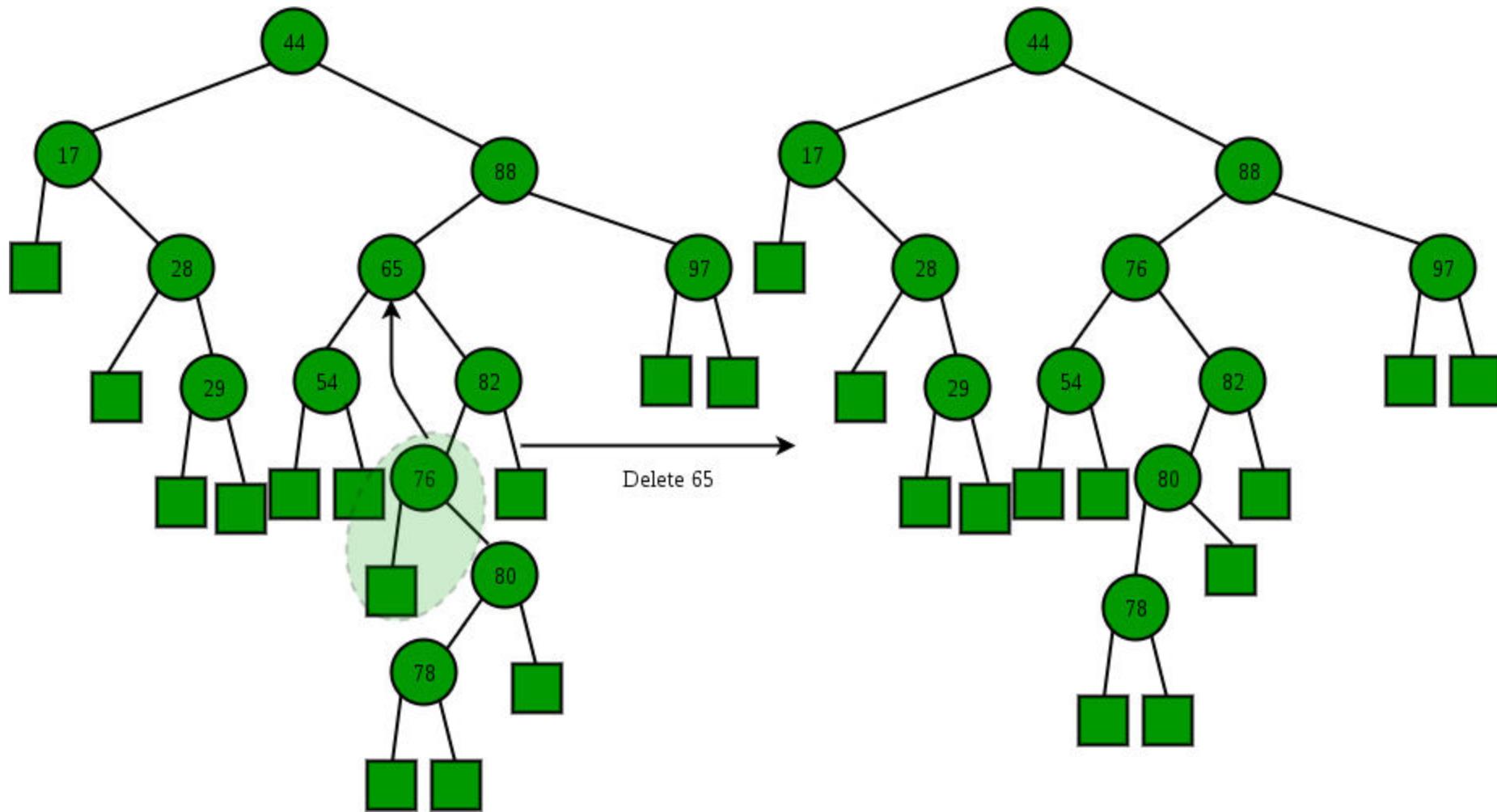
Delete 50 // Delete a node with two child



Inorder traversal of the modified tree

40 60 70 80

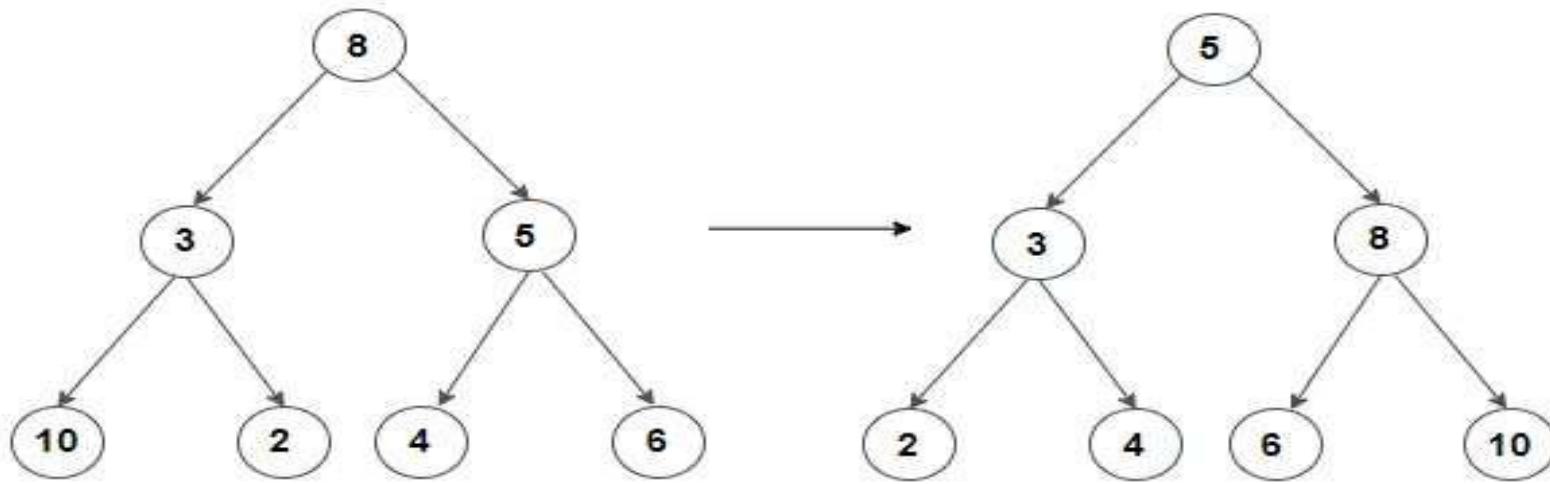




Convert Binary Tree to BST by maintaining original structure

- **Algorithm:**
 1. Traverse the tree in inorder and store the values in array.
 2. Sort the Array.
 3. Now it again traverses given binary tree in inorder fashion and simultaneously it traverses sorted inorder array. At each node visit of binary tree, the value of that node is changed to corresponding element in the inorder array. That is value of first node visited during inorder traversal is changed to value of element at 0th index in sorted inorder array, value of second node visited during inorder traversal is changed to value of element at 1st index and so on..

In the above algorithm, Step #1 and #3 take $O(n)$ time and step #2 which is a sorting step takes $O(n\log n)$ time using heap sort/merge sort method. **Therefore, overall time complexity of this method is $O(n\log n)$.**



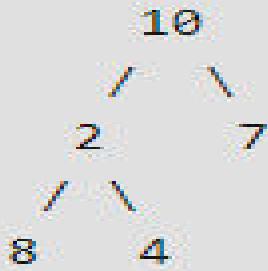
INORDER: 10 2 3 8 4 5 6

SORT INORDER : 2 3 4 5 6 8 10

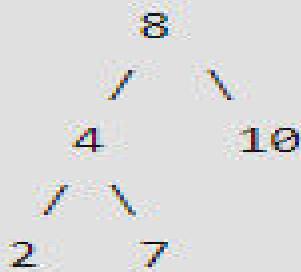
STORE IN position based on INORDER:

Example 1

Input:

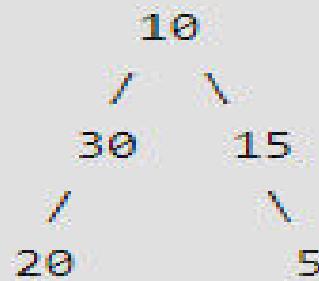


Output:

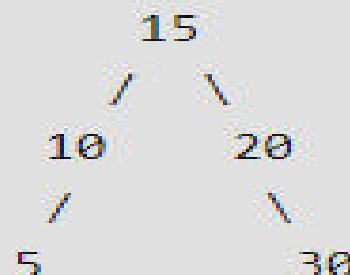


Example 2

Input:



Output:



Inorder: 8 2 4 10 7

Sort: 2 4 7 8 10

Inorder in place:

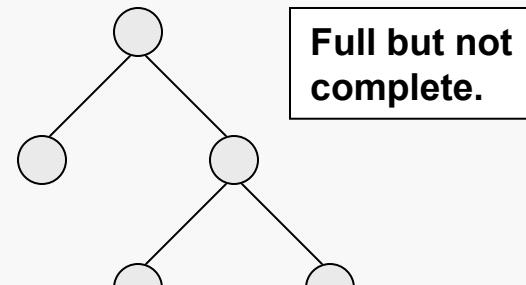
Inorder: 20 30 10 15 5

Sort: 5 10 15 20 30

Inorder in place:

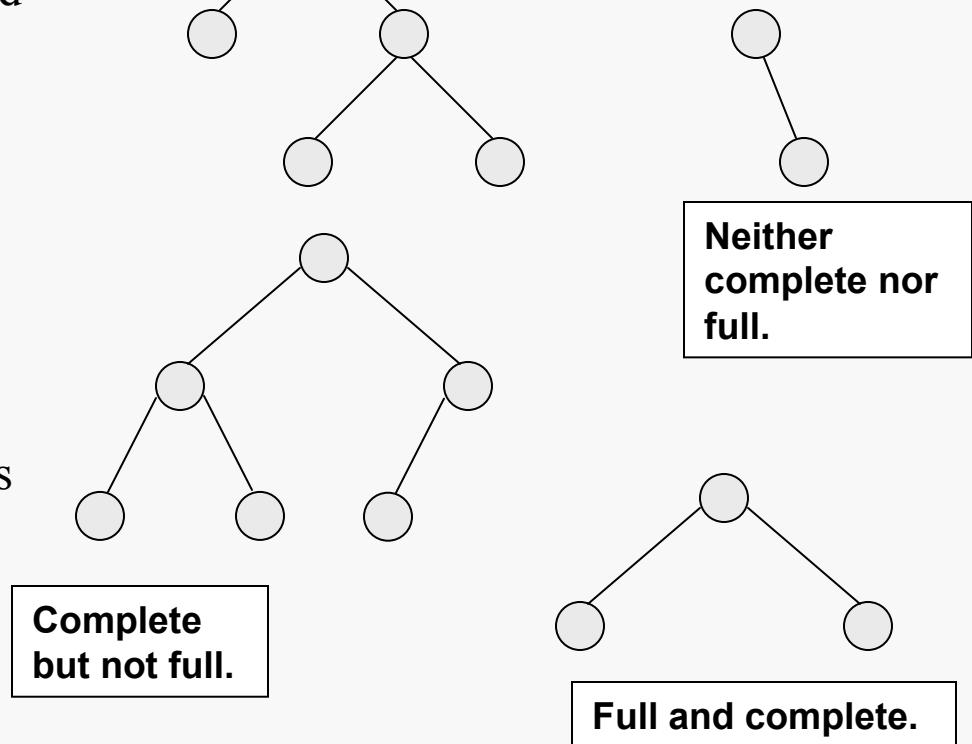
Here are two important types of binary trees. Note that the definitions, while similar, are logically independent.

Definition: a binary tree T is *full* if each node is either a leaf or possesses exactly two child nodes.



Full but not complete.

Definition: a binary tree T with n levels is *complete* if all levels except possibly the last are completely full, and the last level has all its nodes to the left side.



Complete but not full.

Full and complete.

Theorem: Let T be a nonempty, full binary tree Then:

- (a) If T has I internal nodes, the number of leaves is $L = I + 1$.
- (b) If T has I internal nodes, the total number of nodes is $N = 2I + 1$.
- (c) If T has a total of N nodes, the number of internal nodes is $I = (N - 1)/2$.
- (d) If T has a total of N nodes, the number of leaves is $L = (N + 1)/2$.
- (e) If T has L leaves, the total number of nodes is $N = 2L - 1$.
- (f) If T has L leaves, the number of internal nodes is $I = L - 1$.

Basically, this theorem says that the number of nodes N , the number of leaves L , and the number of internal nodes I are related in such a way that if you know any one of them, you can determine the other two.

proof of (a): We will use induction on the number of internal nodes, I . Let S be the set of all integers $I \geq 0$ such that if T is a full binary tree with I internal nodes then T has $I + 1$ leaf nodes.

For the base case, if $I = 0$ then the tree must consist only of a root node, having no children because the tree is full. Hence there is 1 leaf node, and so $0 \in S$.

Now suppose that for some integer $K \geq 0$, every I from 0 through K is in S . That is, if T is a nonempty binary tree with I internal nodes, where $0 \leq I \leq K$, then T has $I + 1$ leaf nodes.

Let T be a full binary tree with $K + 1$ internal nodes. Then the root of T has two subtrees L and R ; suppose L and R have I_L and I_R internal nodes, respectively. Note that neither L nor R can be empty, and that every internal node in L and R must have been an internal node in T , and T had one additional internal node (the root), and so $K + 1 = I_L + I_R + 1$.

Now, by the induction hypothesis, L must have $I_L + 1$ leaves and R must have $I_R + 1$ leaves. Since every leaf in T must also be a leaf in either L or R , T must have $I_L + I_R + 2$ leaves.

Therefore, doing a tiny amount of algebra, T must have $K + 2$ leaf nodes and so $K + 1 \in S$. Hence by Mathematical Induction, $S = [0, \infty)$.

QED

Theorem: Let T be a binary tree with λ levels. Then the number of leaves is at most $2^{\lambda-1}$.

proof: We will use strong induction on the number of levels, λ . Let S be the set of all integers $\lambda \geq 1$ such that if T is a binary tree with λ levels then T has at most $2^{\lambda-1}$ leaf nodes.

For the base case, if $\lambda = 1$ then the tree must have one node (the root) and it must have no child nodes. Hence there is 1 leaf node (which is $2^{\lambda-1}$ if $\lambda = 1$), and so $1 \in S$.

Now suppose that for some integer $K \geq 1$, all the integers 1 through K are in S . That is, whenever a binary tree has M levels with $M \leq K$, it has at most 2^{M-1} leaf nodes.

Let T be a binary tree with $K + 1$ levels. If T has the maximum number of leaves, T consists of a root node and two nonempty subtrees, say S_1 and S_2 . Let S_1 and S_2 have M_1 and M_2 levels, respectively. Since M_1 and M_2 are between 1 and K , each is in S by the inductive assumption. Hence, the number of leaf nodes in S_1 and S_2 are at most 2^{K-1} and 2^{K-1} , respectively. Since all the leaves of T must be leaves of S_1 or of S_2 , the number of leaves in T is at most $2^{K-1} + 2^{K-1}$ which is 2^K . Therefore, $K + 1$ is in S .

Hence by Mathematical Induction, $S = [1, \infty)$.

QED

Theorem: Let T be a binary tree. For every $k \geq 0$, there are no more than 2^k nodes in level k .

Theorem: Let T be a binary tree with λ levels. Then T has no more than $2^\lambda - 1$ nodes.

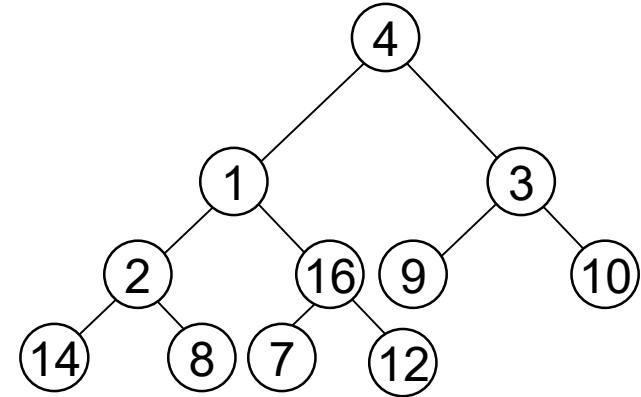
Theorem: Let T be a binary tree with N nodes. Then the number of levels is at least $\lceil \log(N + 1) \rceil$.

Theorem: Let T be a binary tree with L leaves. Then the number of levels is at least $\lceil \log L \rceil + 1$.

Heapsort

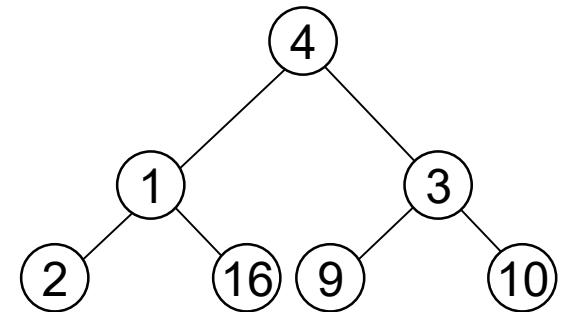
Special Types of Trees

- *Def:* Full binary tree = a binary tree in which each node is either a leaf or has degree exactly 2.



Full binary tree

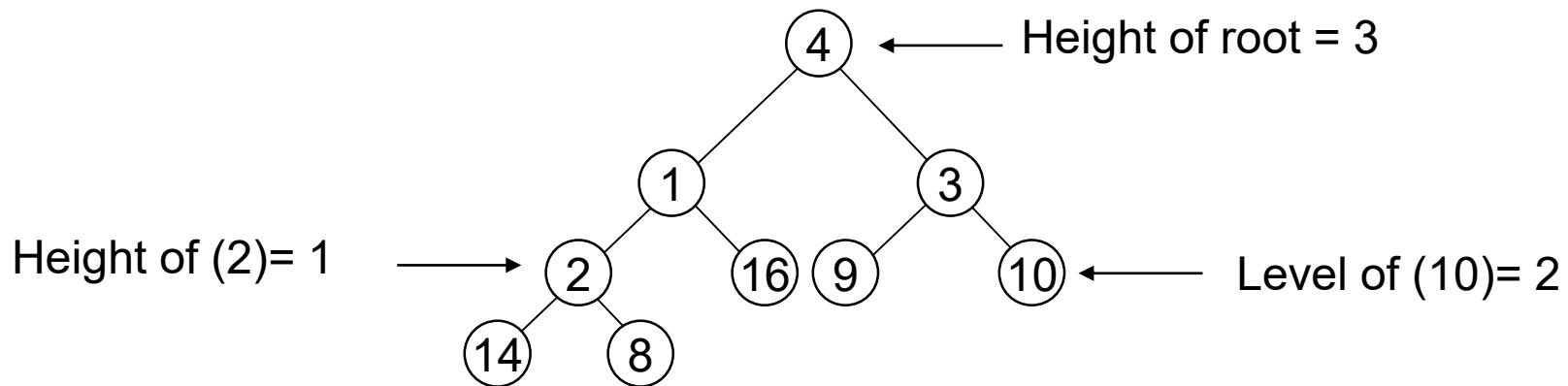
- *Def:* Complete binary tree = a binary tree in which all leaves are on the same level and all internal nodes have degree 2.



Complete binary tree

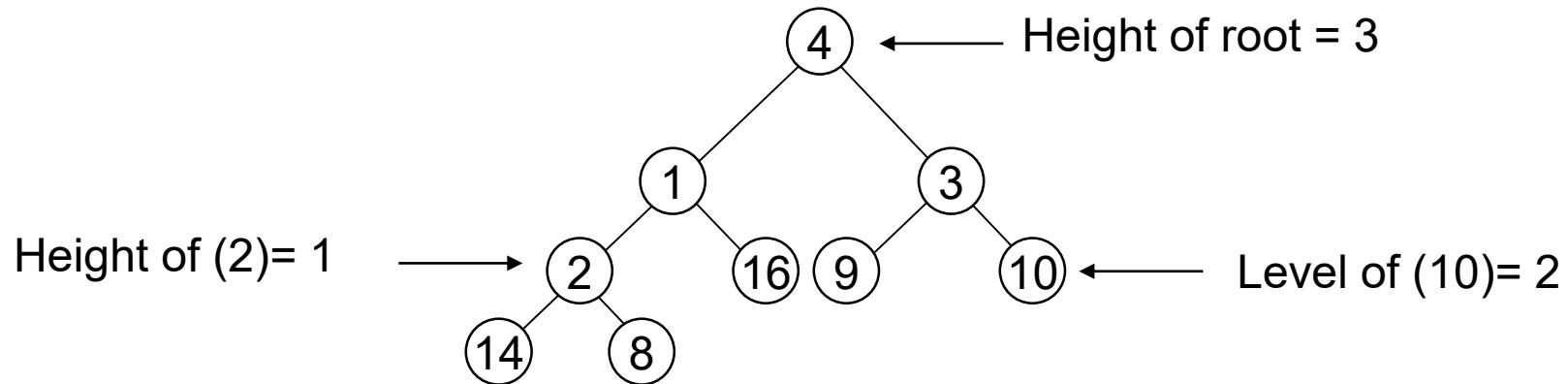
Definitions

- **Height** of a node = the number of edges on the longest simple path from the node down to a leaf
- **Level** of a node = the length of a path from the root to the node
- **Height** of tree = height of root node



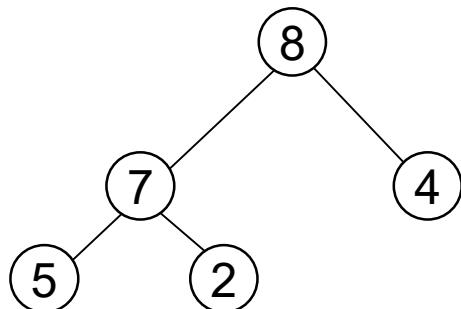
Useful Properties

- There are **at most** 2^l nodes at level (or depth) l of a binary tree
- A binary tree with $\text{height } d$ has **at most** $2^{d+1} - 1$ nodes
- A binary tree with n nodes has $\text{height } \text{at least } \lfloor \lg n \rfloor$



The Heap Data Structure

- *Def:* A **heap** is a all most complete binary tree with the following two properties:
 - **Structural property:** all levels are full, except possibly the last one, which is filled from left to right
 - **Order (heap) property:** for any node x
 $\text{Parent}(x) \geq x$



Heap

From the heap property, it follows that:

“The root is the maximum element of the heap!”

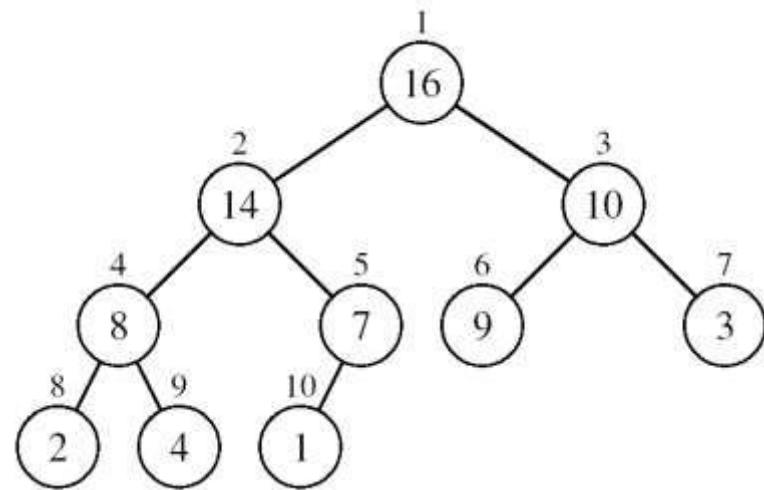
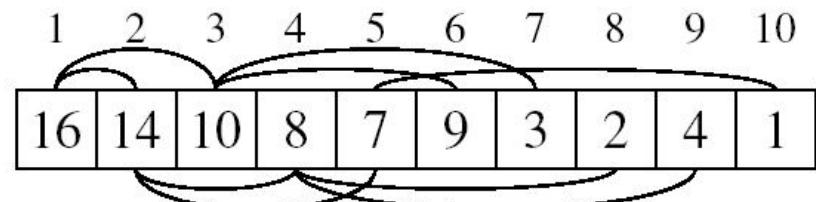
A heap is a binary tree that is filled in order

Array Representation of Heaps

- A heap can be stored as an array

A.

- Root of tree is $A[1]$
 - Left child of $A[i] = A[2i]$
 - Right child of $A[i] = A[2i + 1]$
 - Parent of $A[i] = A[\lfloor i/2 \rfloor]$
 - Heapsize[A] \leq length[A]
- The elements in the subarray $A[\lfloor n/2 \rfloor + 1] .. n]$ are leaves
- Here $n=10$
- Therefore $A[6]$ to $A[10]$ are leaves.
- Moreover $A[1]$ to $A[\lfloor n/2 \rfloor]$ are non leaves



Heap Types

- **Max-heaps** (largest element at root), have the *max-heap property*:

- for all nodes i , excluding the root:

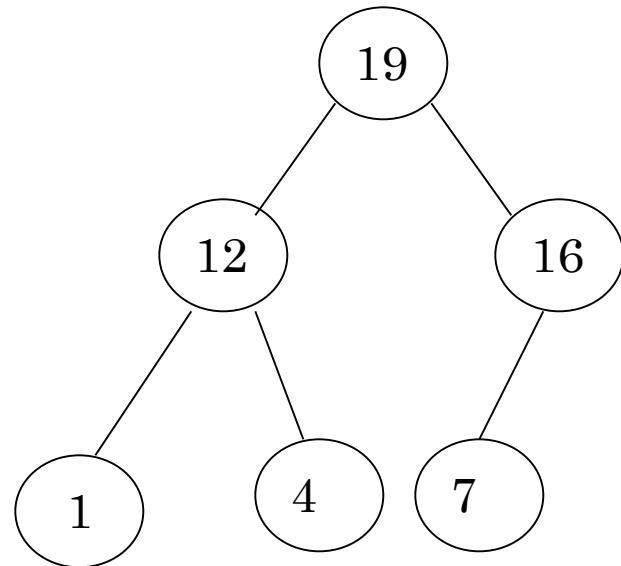
$$A[\text{PARENT}(i)] \geq A[i]$$

- **Min-heaps** (smallest element at root), have the *min-heap property*:

- for all nodes i , excluding the root:

$$A[\text{PARENT}(i)] \leq A[i]$$

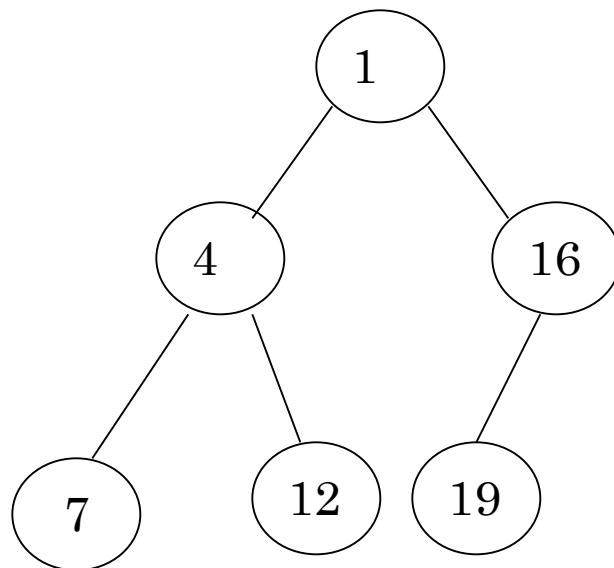
Max Heap Example



19	12	16	1	4	7
----	----	----	---	---	---

Array A

Min heap example

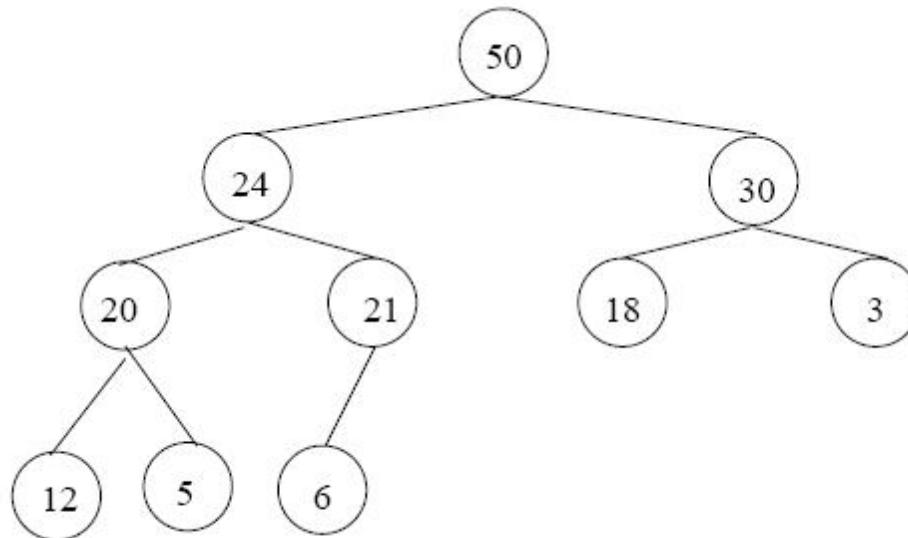


1	4	16	7	12	19
---	---	----	---	----	----

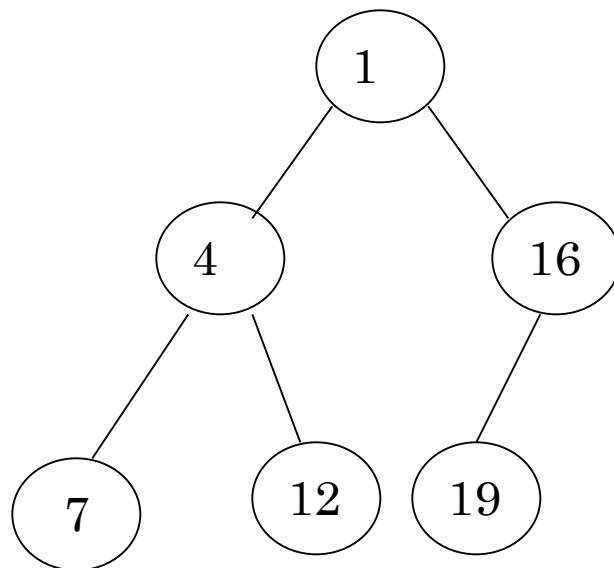
Array A

Adding/Deleting Nodes

- New nodes are always inserted at the bottom level (left to right)
- Nodes are removed from the bottom level (right to left)



Min heap example



1	4	16	7	12	19
---	---	----	---	----	----

Array A

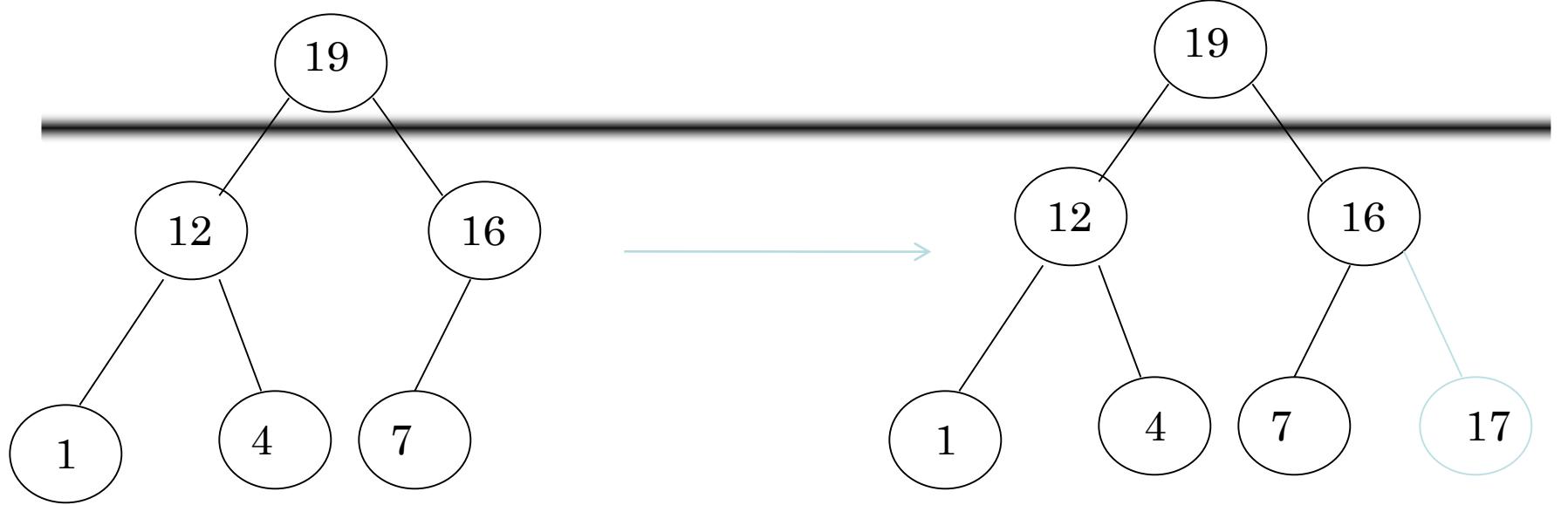
Insertion

- **Algorithm**
 1. Add the new element to the next available position at the lowest level
 2. Restore the max-heap property if violated
 - General strategy is percolate up (or bubble up): if the parent of the element is smaller than the element, then interchange the parent and child.

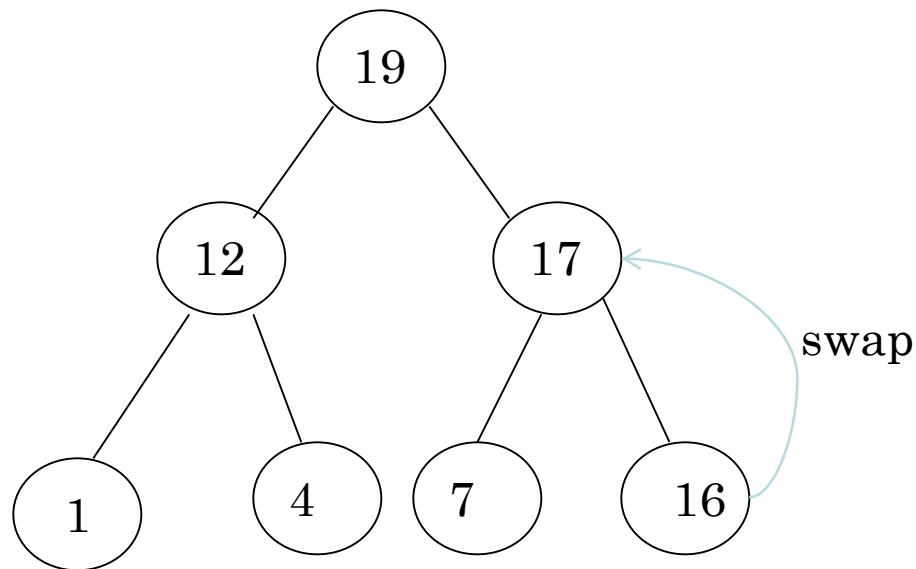
OR

Restore the min-heap property if violated

- General strategy is percolate up (or bubble up): if the parent of the element is larger than the element, then interchange the parent and child.

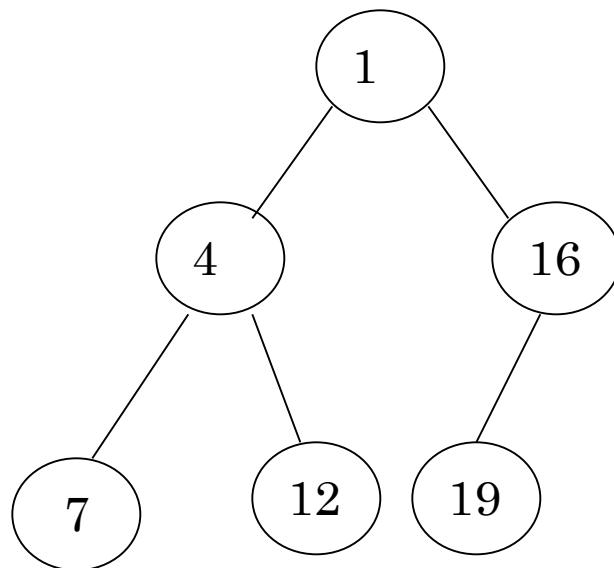


Insert 17



Percolate up to maintain the
heap property

Min heap example



1	4	16	7	12	19
---	---	----	---	----	----

Array A

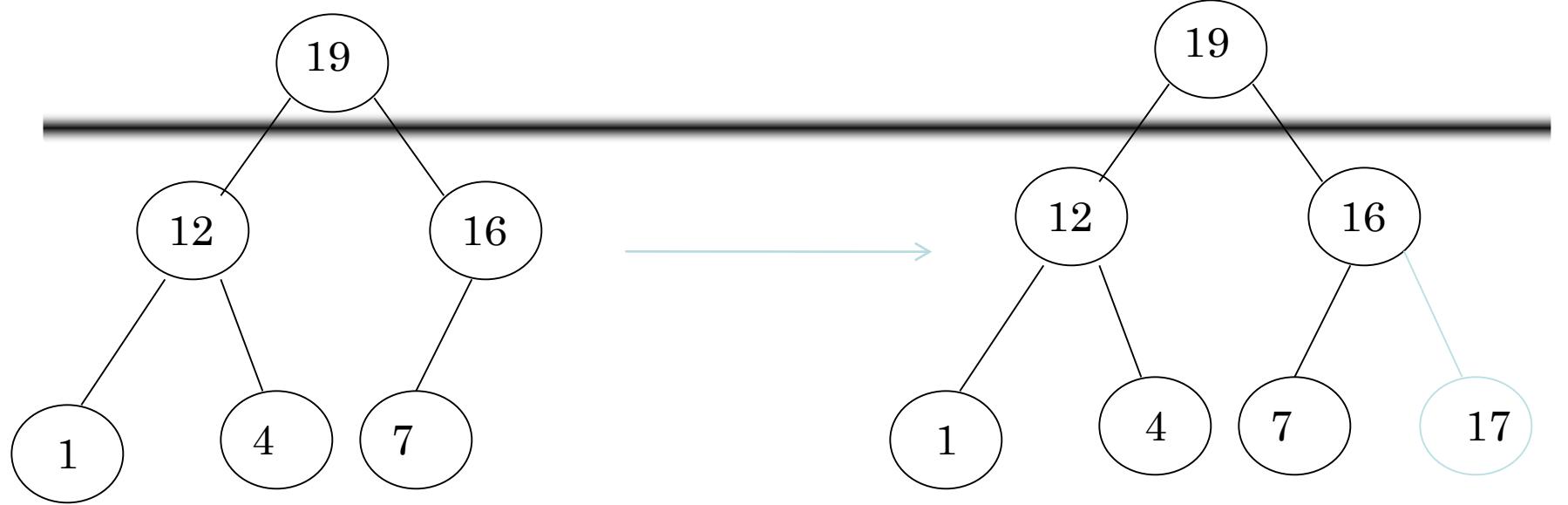
Insertion

- **Algorithm**
 1. Add the new element to the next available position at the lowest level
 2. Restore the max-heap property if violated
 - General strategy is percolate up (or bubble up): if the parent of the element is smaller than the element, then interchange the parent and child.

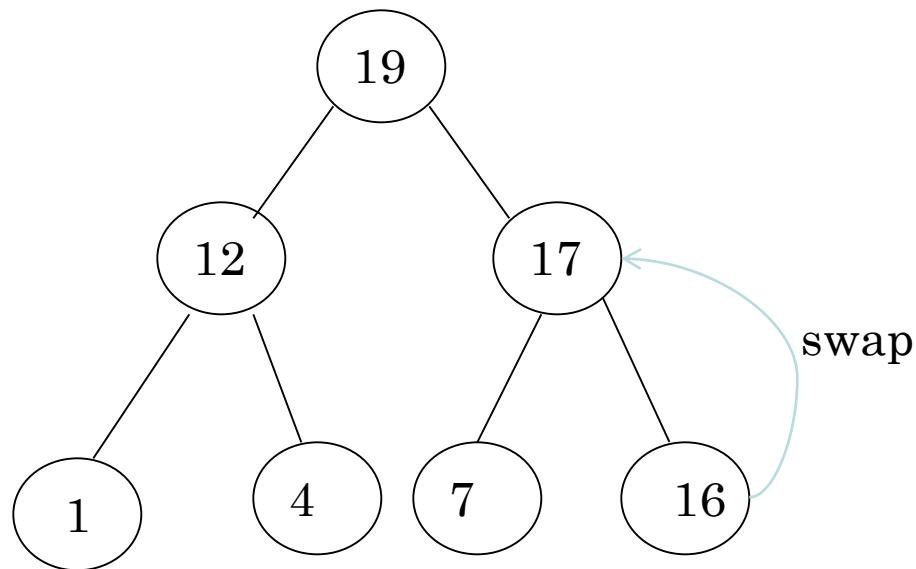
OR

Restore the min-heap property if violated

- General strategy is percolate up (or bubble up): if the parent of the element is larger than the element, then interchange the parent and child.



Insert 17



Percolate up to maintain the
heap property

Deletion

- Delete max
 - Copy the last number to the root (overwrite the maximum element stored there).
 - Restore the max heap property by percolate down.
- Delete min
 - Copy the last number to the root (overwrite the minimum element stored there).
 - Restore the min heap property by percolate down.

Heap Sort

A sorting algorithm that works by first organizing the data to be sorted into a special type of binary tree called a heap

Procedures on Heap

- Heapify
- Build Heap
- Heap Sort

Heapify

- Heapify picks the largest child key and compare it to the parent key. If parent key is larger than heapify quits, otherwise it swaps the parent key with the largest child key. So that the parent is now becomes larger than its children.

Heapify(A, i)

{

l \leftarrow left(*i*)
 r \leftarrow right(*i*)
 if *l* \leq heapsize[A] and A[i] > A[l]
 then largest \leftarrow *l*

 else largest \leftarrow *i*
 if *r* \leq heapsize[A] and A[r] > A[largest]
 then largest \leftarrow *r*

 if largest != *i*
 then swap A[i] \leftrightarrow A[largest]
 Heapify(A, largest)

}

BUILD HEAP

- We can use the procedure 'Heapify' in a bottom-up fashion to convert an array $A[1 \dots n]$ into a heap. Since the elements in the subarray $A[n/2 + 1 \dots n]$ are all leaves, the procedure BUILD_HEAP goes through the remaining nodes of the tree and runs 'Heapify' on each one. The bottom-up order of processing node guarantees that the subtree rooted at children are heap before 'Heapify' is run at their parent.

Buildheap(A)

{

 heapsize[A] \leftarrow length[A]
 for i \leftarrow |length[A]/2 down to 1
 do Heapify(A, i)

}

Heap Sort Algorithm

- The heap sort algorithm starts by using procedure BUILD-HEAP to build a heap on the input array $A[1 \dots n]$. Since the maximum element of the array stored at the root $A[1]$, it can be put into its correct final position by exchanging it with $A[n]$ (the last element in A). If we now discard node n from the heap than the remaining elements can be made into heap. Note that the new element at the root may violate the heap property. All that is needed to restore the heap property.

Heapsort(A)

{

Buildheap(A)

for $i \leftarrow \text{length}[A]$ //down to 2

do swap $A[1] \leftrightarrow A[i]$

heapsize[A] $\leftarrow \text{heapsize}[A] - 1$

Heapify(A, 1)

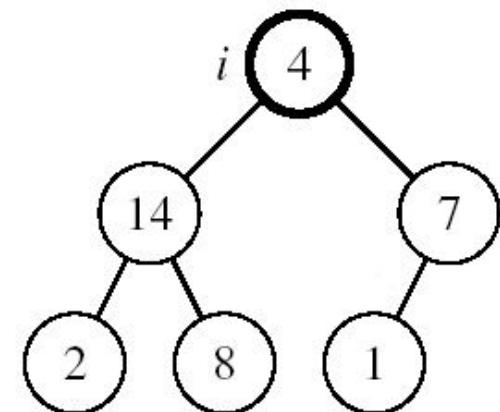
}

Operations on Heaps

- Maintain/Restore the max-heap property
 - MAX-HEAPIFY
- Create a max-heap from an unordered array
 - BUILD-MAX-HEAP
- Sort an array in place
 - HEAPSORT

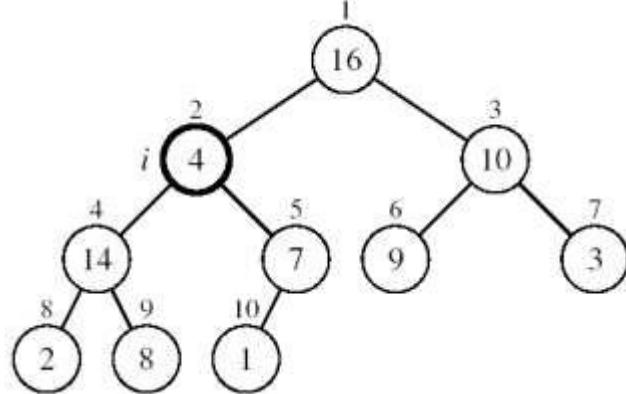
Maintaining the Heap Property

- Suppose a node is smaller than a child
 - Left and Right subtrees of i are max-heaps
- To eliminate the violation:
 - Exchange with larger child
 - Move down the tree
 - Continue until node is not smaller than children



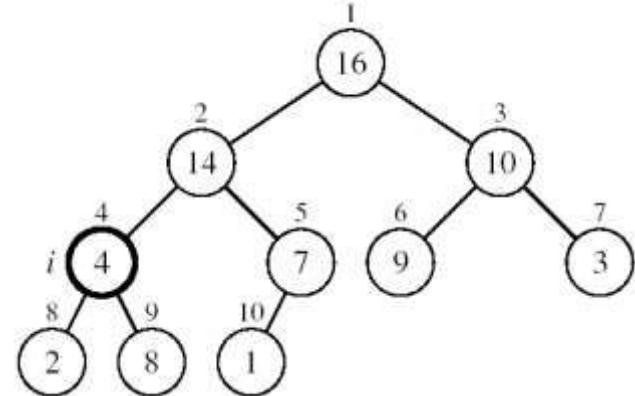
Example

MAX-HEAPIFY(A, 2, 10)



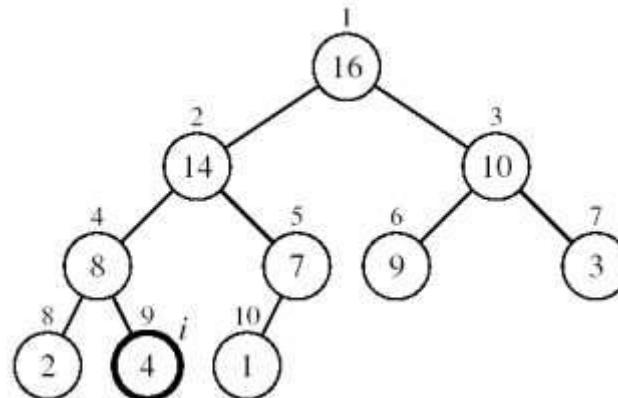
$A[2] \leftrightarrow A[4]$

$A[2]$ violates the heap property



$A[4]$ violates the heap property

$A[4] \leftrightarrow A[9]$

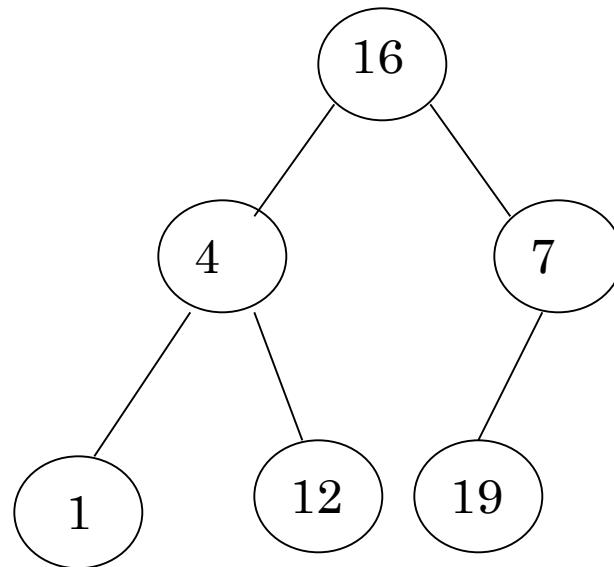


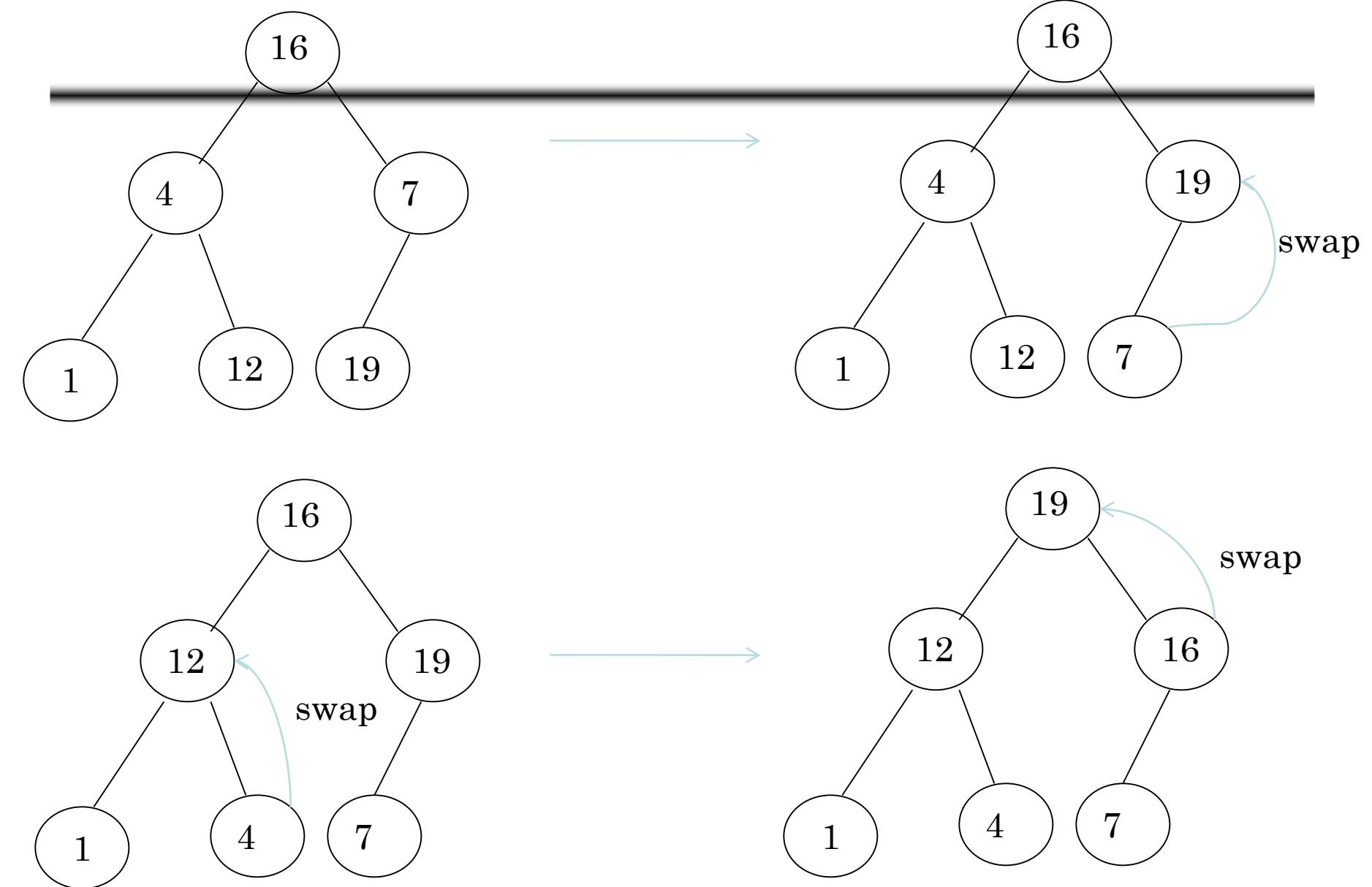
Heap property restored

Example: Convert the following array to a heap

16	4	7	1	12	19
----	---	---	---	----	----

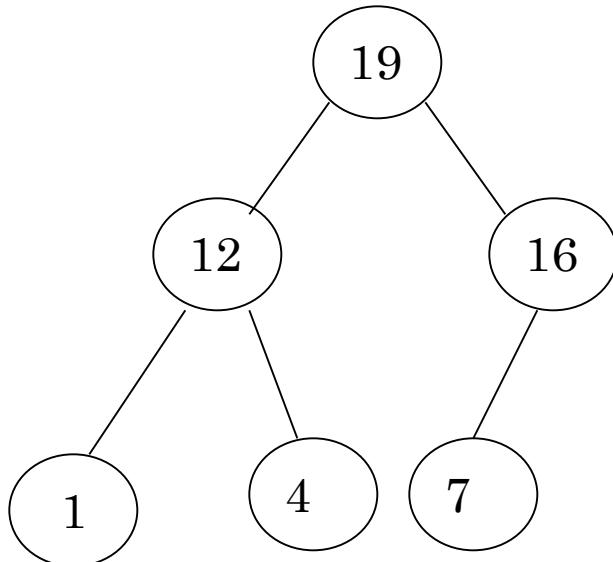
Picture the array as a complete binary tree:





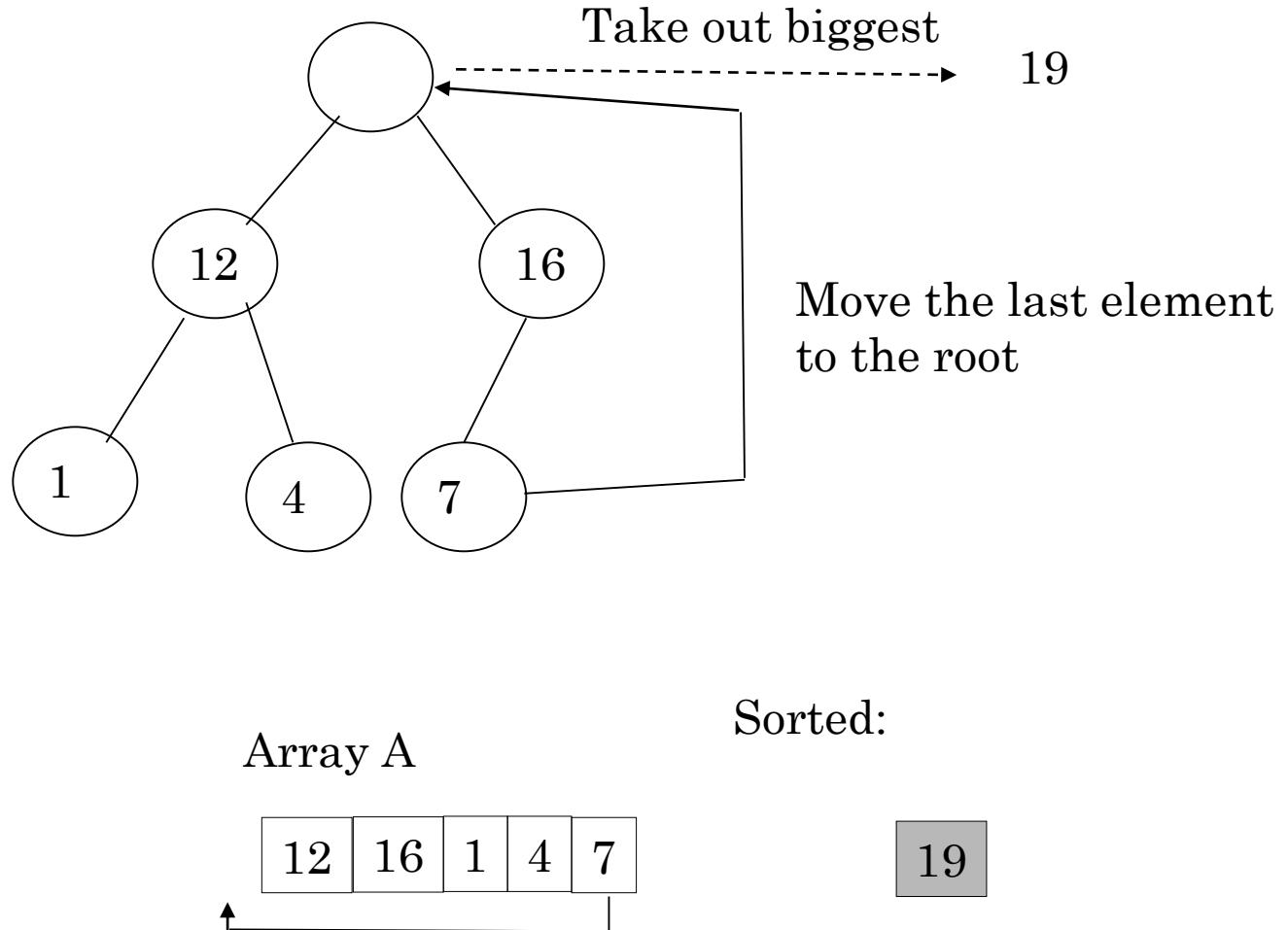
Heap Sort

- The heapsort algorithm consists of two phases:
 - build a heap from an arbitrary array
 - use the heap to sort the data
- To sort the elements in the **decreasing order**, use a **min heap**
- To sort the elements in the **increasing order**, use a **max heap**

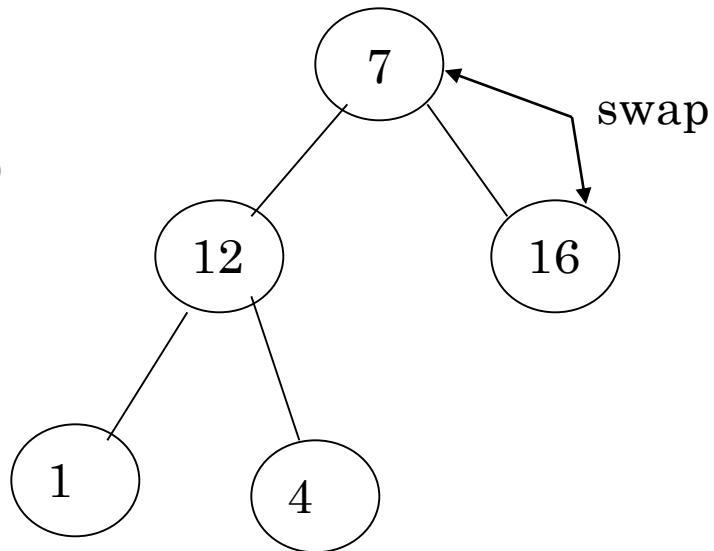


- CAT-II syllabus updated in VTOP
- Reference material in VTOP (B1 slot course page)
- DA – Uploaded for both set of students
- i) <25 out of 50 in CAT_I should do the problems uploaded in VTOP. This is only for CAT_ii syllabus(Submitted before CAT_II exam) and one more assignment for CAT_I syllabus to be done after CAT_ii and Submitted on or before 17 th March.
- ii) For those who have secured ≥ 25 in CAT_I will have separate assignment which is uploaded in VTOP in DA upload (Submitted on or before 17 th March)

Example of Heap Sort



HEAPIFY()

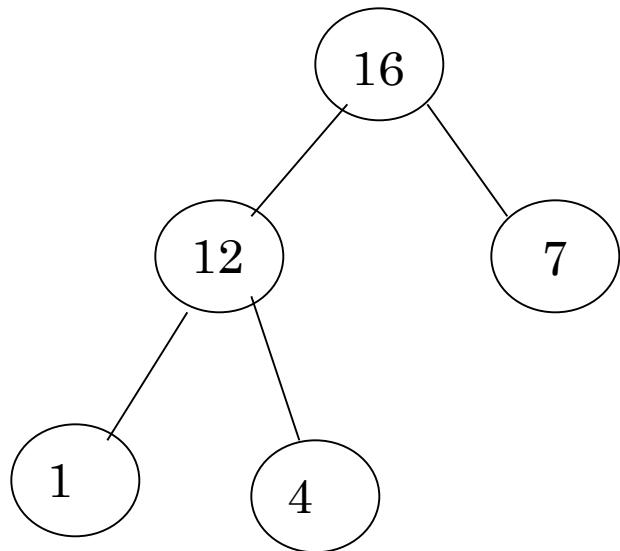


Array A

7	12	16	1	4
---	----	----	---	---

Sorted:

19



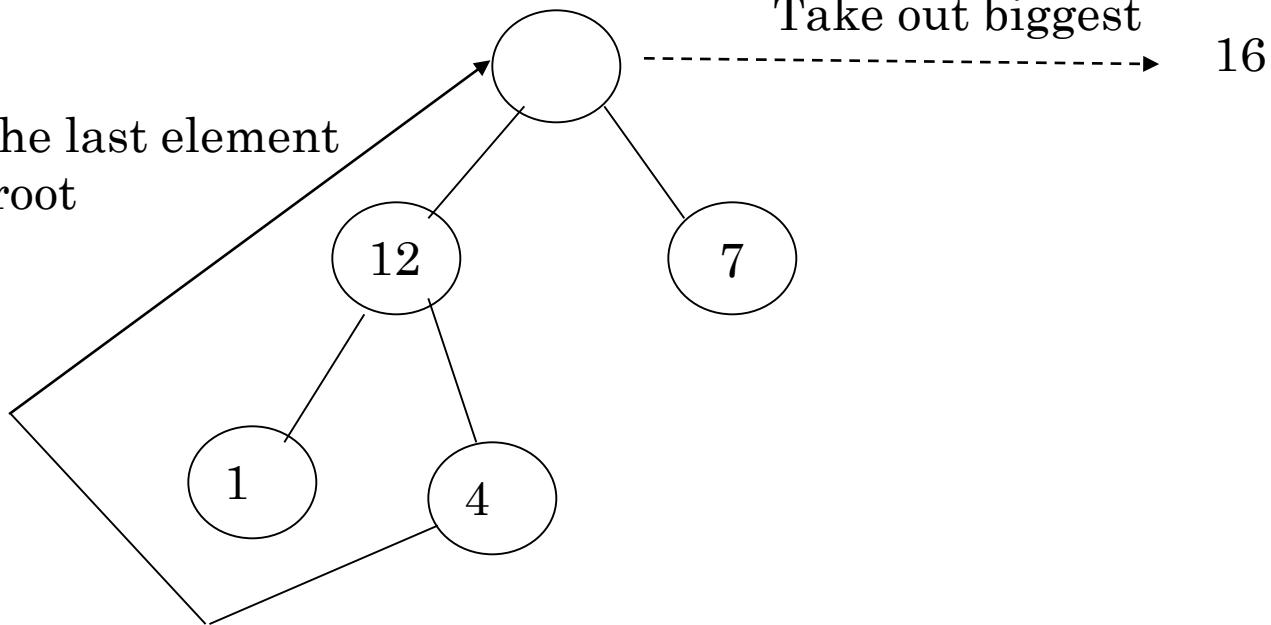
Array A

16	12	7	1	4
----	----	---	---	---

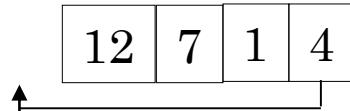
Sorted:

19

Move the last element
to the root

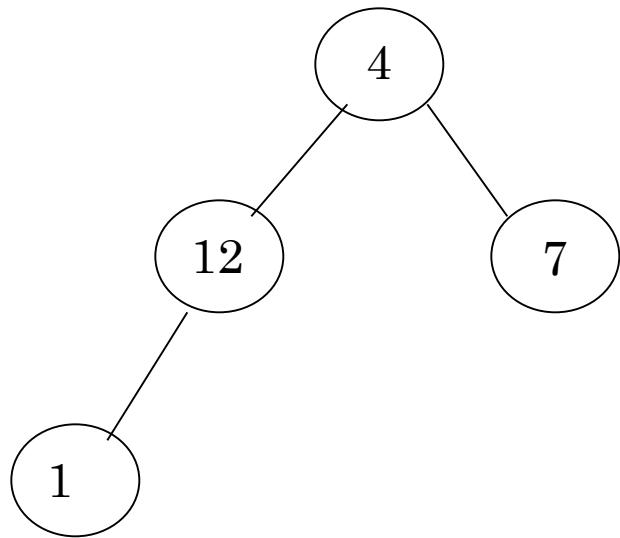


Array A



Sorted:



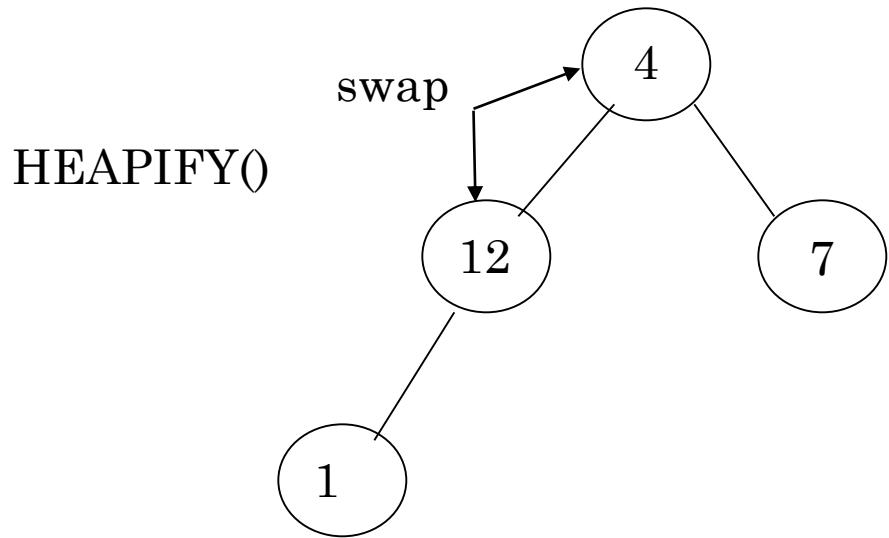


Array A

4	12	7	1
---	----	---	---

Sorted:

16	19
----	----

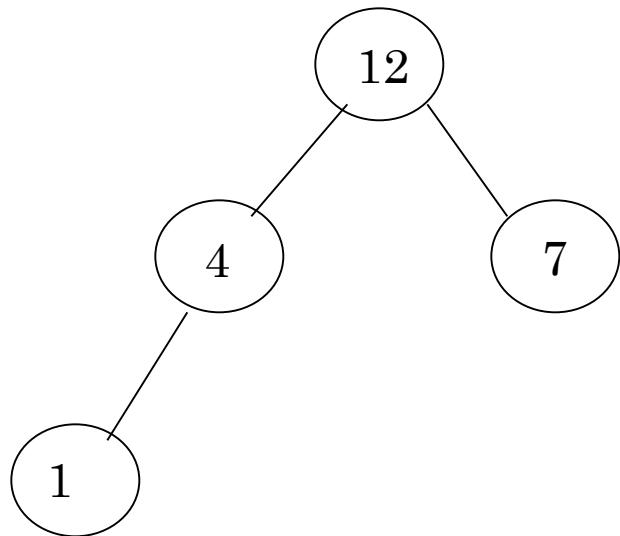


Array A

4	12	7	1
---	----	---	---

Sorted:

16	19
----	----

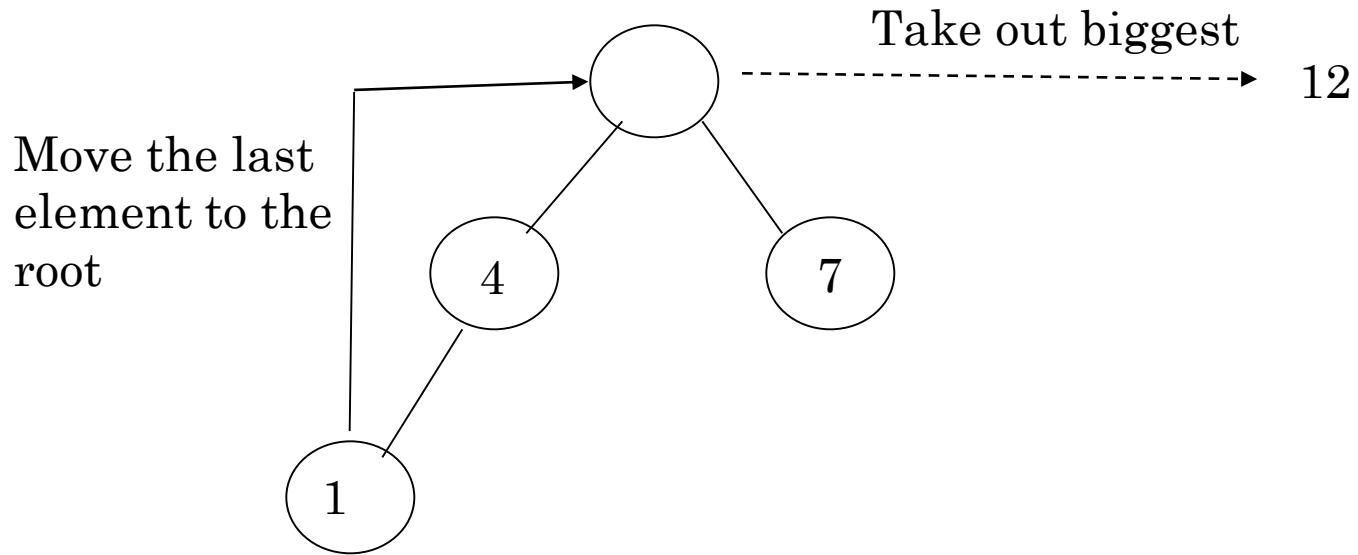


Array A

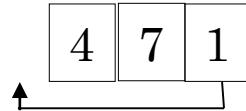
12	4	7	1
----	---	---	---

Sorted:

16	19
----	----

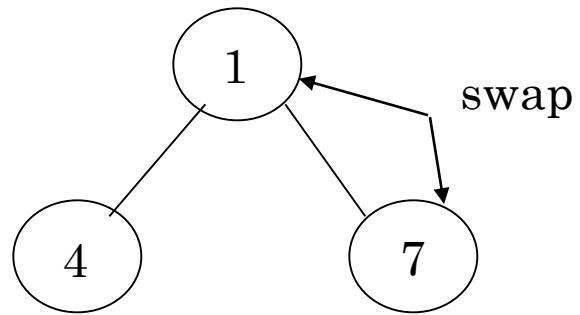


Array A



Sorted:



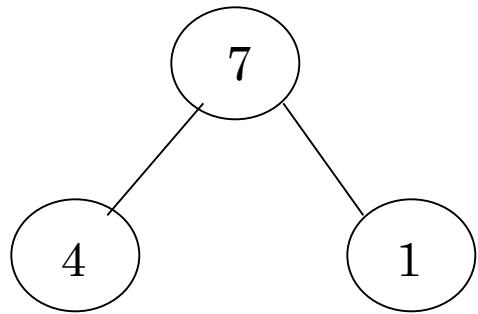


Array A

1	4	7
---	---	---

Sorted:

12	16	19
----	----	----

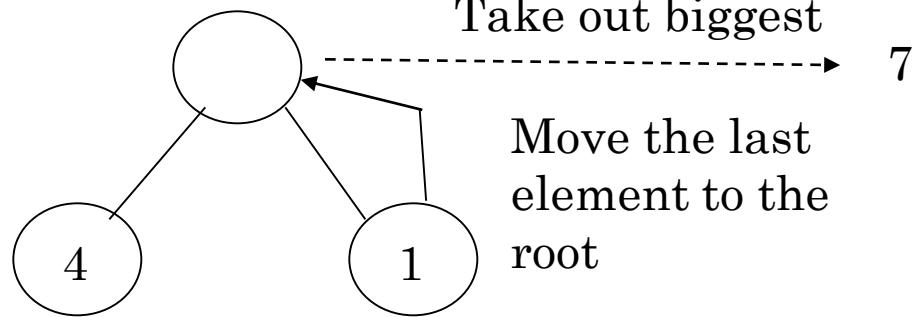


Array A

7	4	1
---	---	---

Sorted:

12	16	19
----	----	----



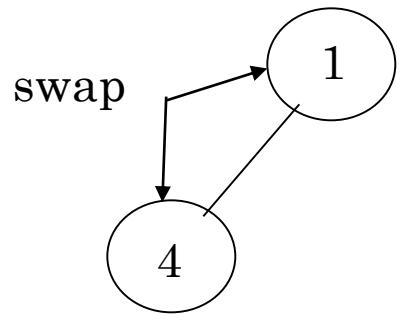
Array A

1	4
---	---

Sorted:

7	12	16	19
---	----	----	----

HEAPIFY()



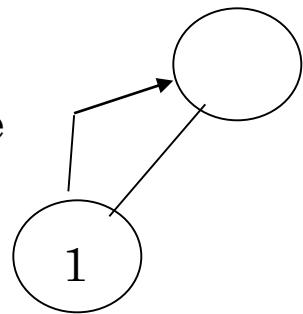
Array A

4	1
---	---

Sorted:

7	12	16	19
---	----	----	----

Move the last
element to the
root



Take out biggest

4

Array A

1

Sorted:

4 7 12 16 19

1

Take out biggest

Array A

Sorted:

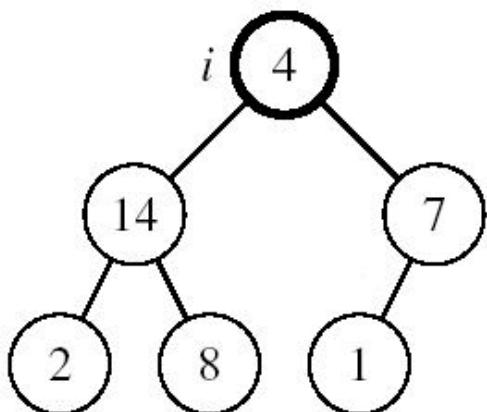
1	4	7	12	16	19
---	---	---	----	----	----

Sorted:

1	4	7	12	16	19
---	---	---	----	----	----

Maintaining the Heap Property

- Assumptions:
 - Left and Right subtrees of i are max-heaps
 - $A[i]$ may be smaller than its children



Alg: MAX-HEAPIFY(A, i, n)

1. $l \leftarrow \text{LEFT}(i)$
2. $r \leftarrow \text{RIGHT}(i)$
3. **if** $l \leq n$ and $A[l] > A[i]$
4. **then** $\text{largest} \leftarrow l$
5. **else** $\text{largest} \leftarrow i$
6. **if** $r \leq n$ and $A[r] > A[\text{largest}]$
7. **then** $\text{largest} \leftarrow r$
8. **if** $\text{largest} \neq i$
9. **then** exchange $A[i] \leftrightarrow A[\text{largest}]$
10. MAX-HEAPIFY($A, \text{largest}, n$)

MAX-HEAPIFY Running Time

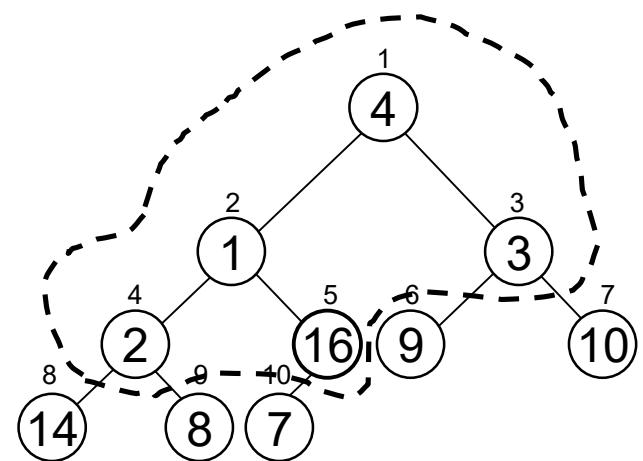
- Intuitively:
 - It traces a path from the root to a leaf (longest path length: h)
 - At each level, it makes exactly 2 comparisons
 - Total number of comparisons is $\lceil 2h \rceil$
 - Running time is $O(h)$ or $O(\lg n)$
- Running time of MAX-HEAPIFY is $O(\lg n)$
- Can be written in terms of the height of the heap,
as being $O(h)$
 - Since the height of the heap is $\lfloor \lg n \rfloor$

Building a Heap

- Convert an array $A[1 \dots n]$ into a max-heap ($n = \text{length}[A]$)
- The elements in the subarray $A[\lfloor n/2 \rfloor + 1 \dots n]$ are leaves
- Apply MAX-HEAPIFY on elements between 1 and $\lfloor n/2 \rfloor$

Alg: BUILD-MAX-HEAP(A)

1. $n = \text{length}[A]$
2. **for** $i \leftarrow \lfloor n/2 \rfloor$ **downto** 1
3. **do** MAX-HEAPIFY(A, i, n)



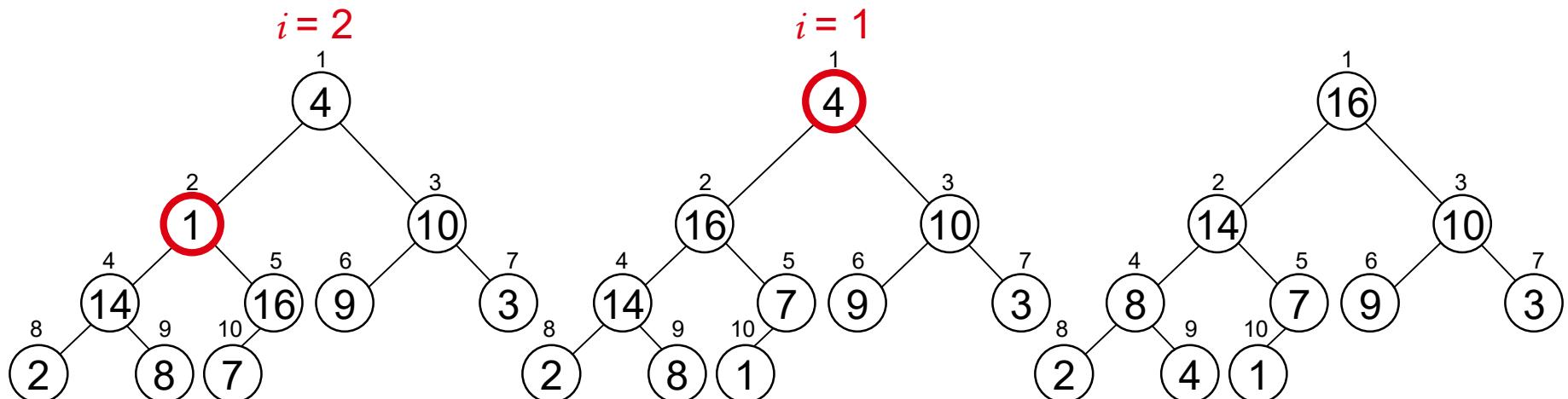
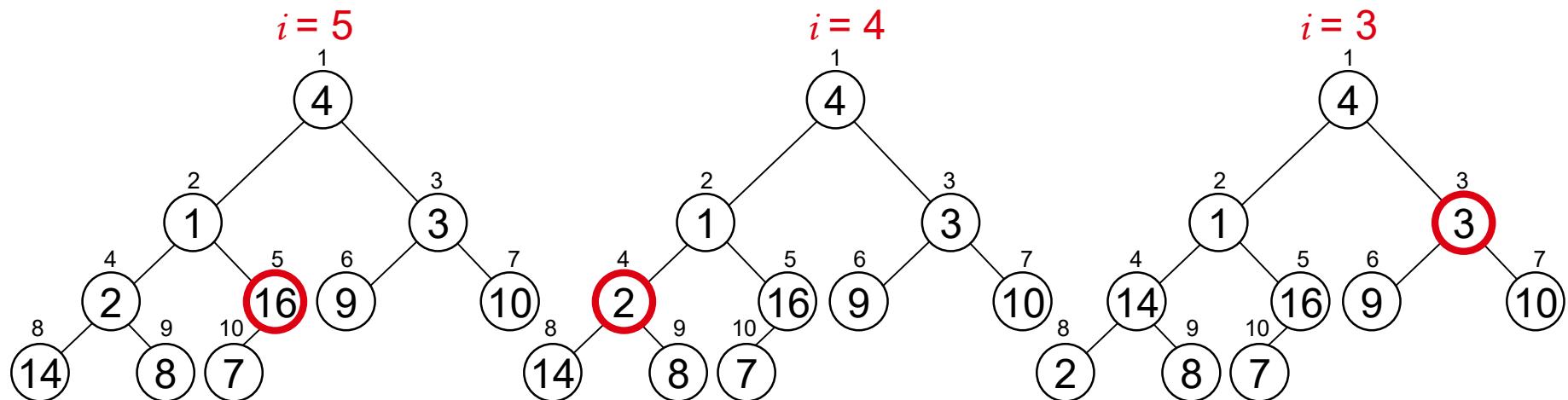
A:

4	1	3	2	16	9	10	14	8	7
---	---	---	---	----	---	----	----	---	---

Example:

A

4	1	3	2	16	9	10	14	8	7
---	---	---	---	----	---	----	----	---	---



Running Time of BUILD MAX HEAP

Alg: BUILD-MAX-HEAP(A)

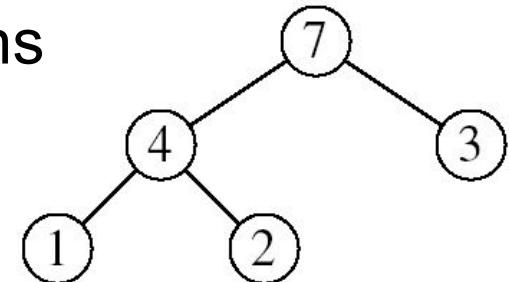
1. $n = \text{length}[A]$
 2. **for** $i \leftarrow \lfloor n/2 \rfloor$ **downto** 1
 3. **do** $\text{MAX-HEAPIFY}(A, i, n)$
- $O(\lg n)$ } $O(n)$

\Rightarrow Running time: $O(n \lg n)$

- This is not an asymptotically tight upper bound

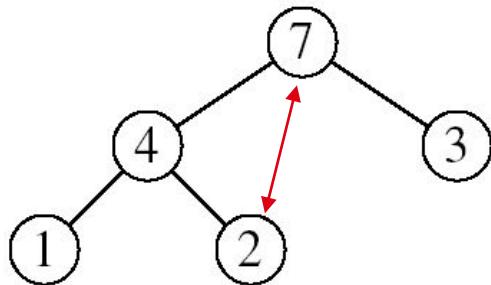
Heapsort

- Goal:
 - Sort an array using heap representations
- Idea:
 - Build a **max-heap** from the array
 - Swap the root (the maximum element) with the last element in the array
 - “Discard” this last node by decreasing the heap size
 - Call MAX-HEAPIFY on the new root
 - Repeat this process until only one node remains

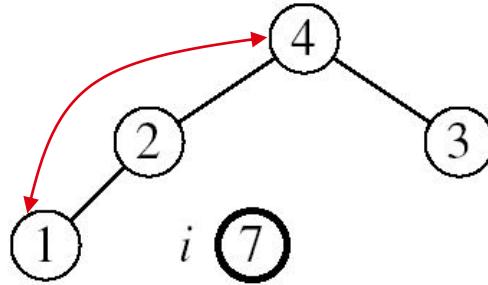


Example:

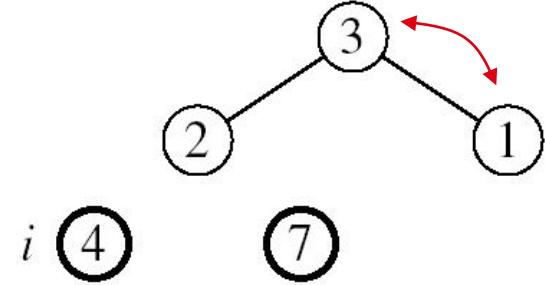
$A=[7, 4, 3, 1, 2]$



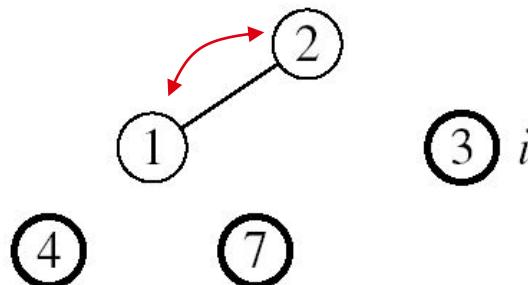
MAX-HEAPIFY(A , 1, 4)



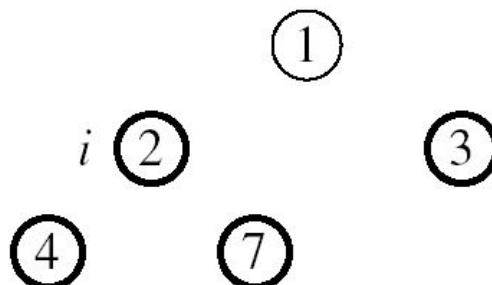
MAX-HEAPIFY(A , 1, 3)



MAX-HEAPIFY(A , 1, 2)



MAX-HEAPIFY(A , 1, 1)



A

1	2	3	4	7
---	---	---	---	---

Alg: HEAPSORT(A)

1. BUILD-MAX-HEAP(A) $O(n)$
 2. **for** $i \leftarrow \text{length}[A]$ **downto** 2
 3. **do** exchange $A[1] \leftrightarrow A[i]$
 4. MAX-HEAPIFY($A, 1, i - 1$) $O(\lg n)$
- $\left. \begin{matrix} \\ \\ \\ \end{matrix} \right\} n-1 \text{ times}$

- Running time: $O(n \lg n)$ --- Can be shown to be $\Theta(n \lg n)$

Time Analysis

- Build Heap Algorithm will run in $O(n)$ time
- There are $n-1$ calls to Heapify each call requires $O(\log n)$ time
- Heap sort program combine Build Heap program and Heapify, therefore it has the running time of $O(n \log n)$ time
- Total time complexity: $O(n \log n)$

Heap

Jianye Hao

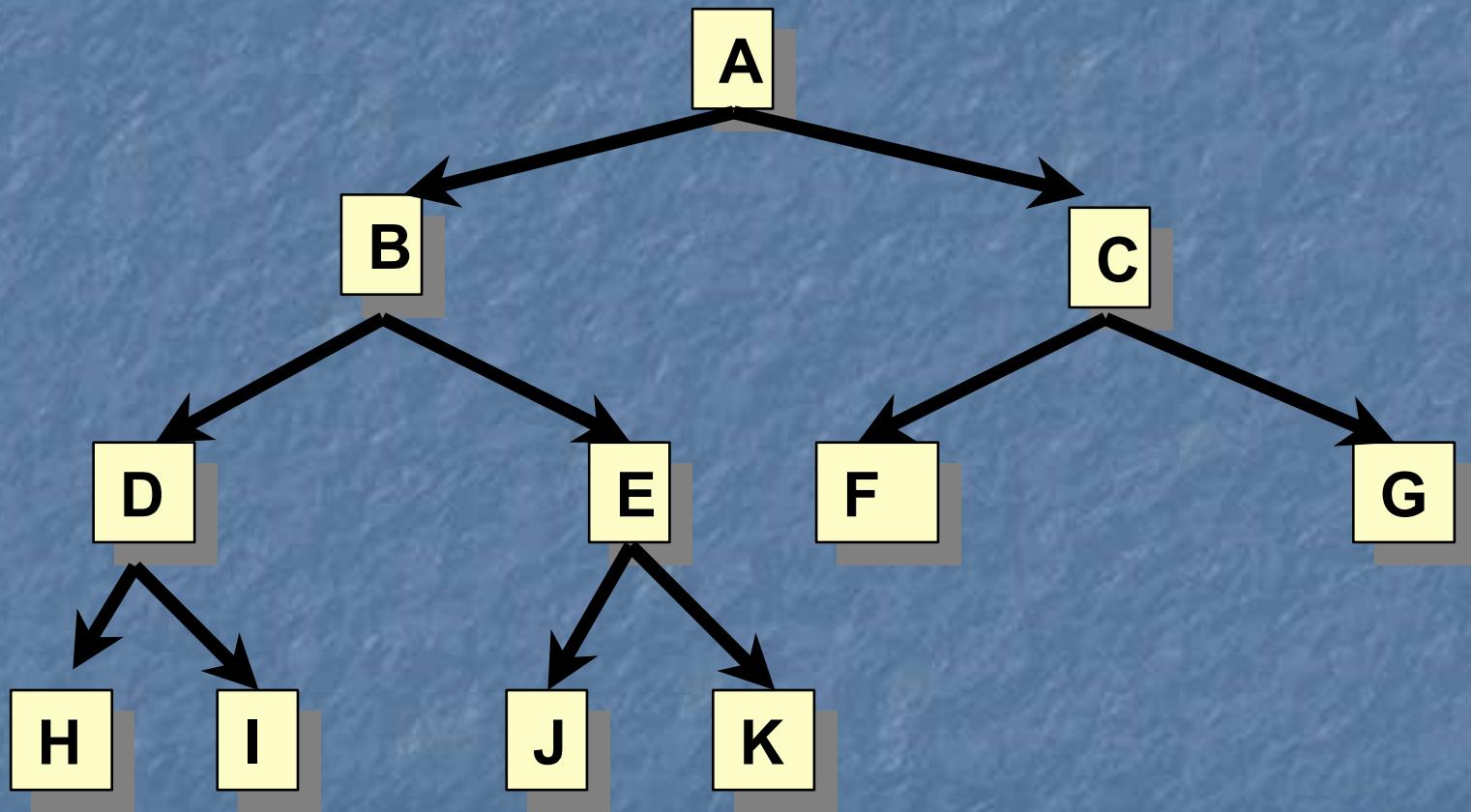
Heap

- Definition in Data Structure
 - Heap: A special form of **complete binary tree** that key value of each node is no smaller (larger) than the key value of its children (if any).
- Max-Heap: root node has the largest key.
 - A **max tree** is a tree in which the key value in each node is **no smaller than** the key values in its children. A **max heap** is a **complete binary tree** that is also a max tree.
- Min-Heap: root node has the smallest key.
 - A **min tree** is a tree in which the key value in each node is **no larger than** the key values in its children. A **min heap** is a **complete binary tree** that is also a min tree.

Complete Binary Tree

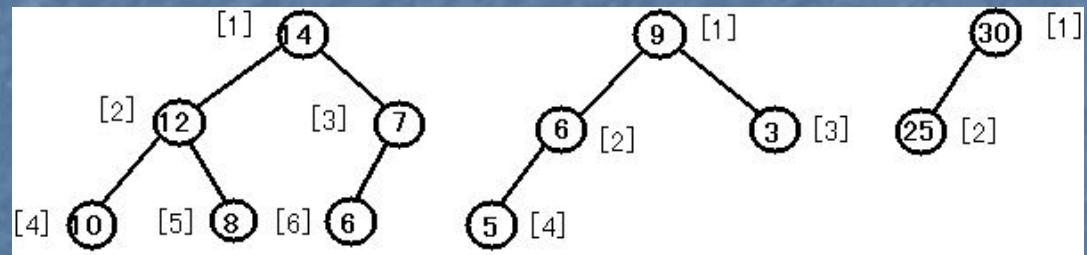
- A **complete binary tree** is a binary tree in which every level, *except possibly the last*, is completely filled, and all nodes are as far left as possible.

Complete Binary Trees - Example

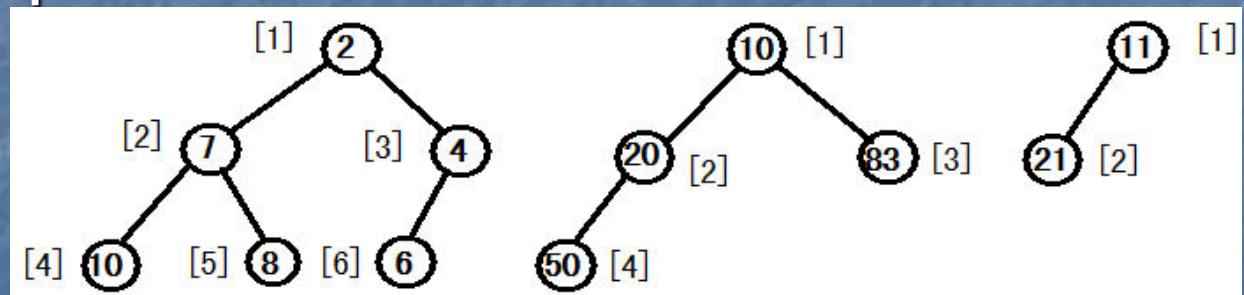


Heap

- Example:
 - Max-Heap



- Min-Heap



Heap

- Notice:
 - Heap in data structure is a **complete binary tree!** (**Nice representation in Array**)
 - Heap in C program environment is an array of memory.



- Stored using array in C
- | index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|-------|----|----|----|----|----|----|----|----|---|----|
| value | 77 | 61 | 59 | 48 | 19 | 11 | 26 | 15 | 1 | 5 |

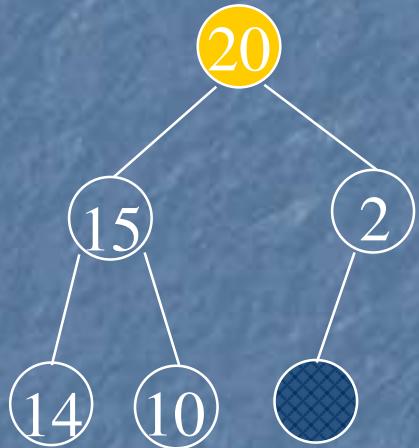
Heap

- Operations
 - Creation of an empty heap
 - Insertion of a new element into the heap
 - Deletion of the largest(smallest) element from the heap
- Heap is complete binary tree, can be represented by array. So the complexity of inserting any node or deleting the root node from Heap is $O(\text{height}) = O(\log_2 n)$.

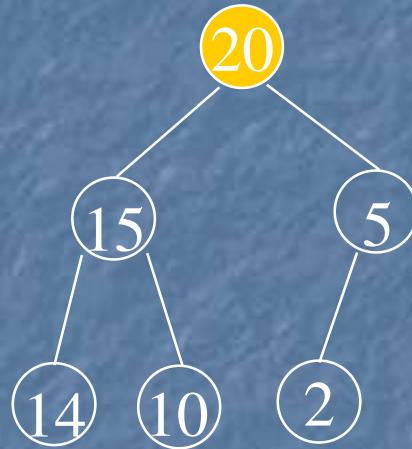
Heap

- Given the index i of a node
- $\text{Parent}(i)$
 - return $i/2$
- $\text{LeftChild}(i)$
 - return $2i$
- $\text{RightChild}(i)$
 - Return $2i+1$

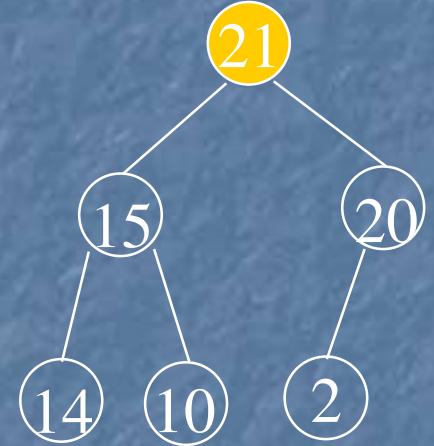
Example of Insertion to Max Heap



initial location of new node



insert 5 into heap



insert 21 into heap

Insertion into a Max Heap

```
void insert_max_heap(element item, int *n)
{
    int i;
    if (HEAP_FULL(*n)) {
        fprintf(stderr, "the heap is full.\n");
        exit(1);
    }
    i = ++(*n);
    while ((i!=1)&&(item.key>heap[i/2].key)) {
        heap[i] = heap[i/2];
        i /= 2;
    }
    heap[i]= item;
}
```

Example of Deletion from Max Heap



Deletion from a Max Heap

```
element delete_max_heap(int *n)
{
    int parent, child;
    element item, temp;
    if (HEAP_EMPTY(*n)) {
        fprintf(stderr, "The heap is empty\n");
        exit(1);
    }
    /* save value of the element with the
       highest key */
    item = heap[1];
    /* use last element in heap to adjust heap */
    temp = heap[(*n)--];
    parent = 1;
    child = 2;
```

Deletion from a Max Heap

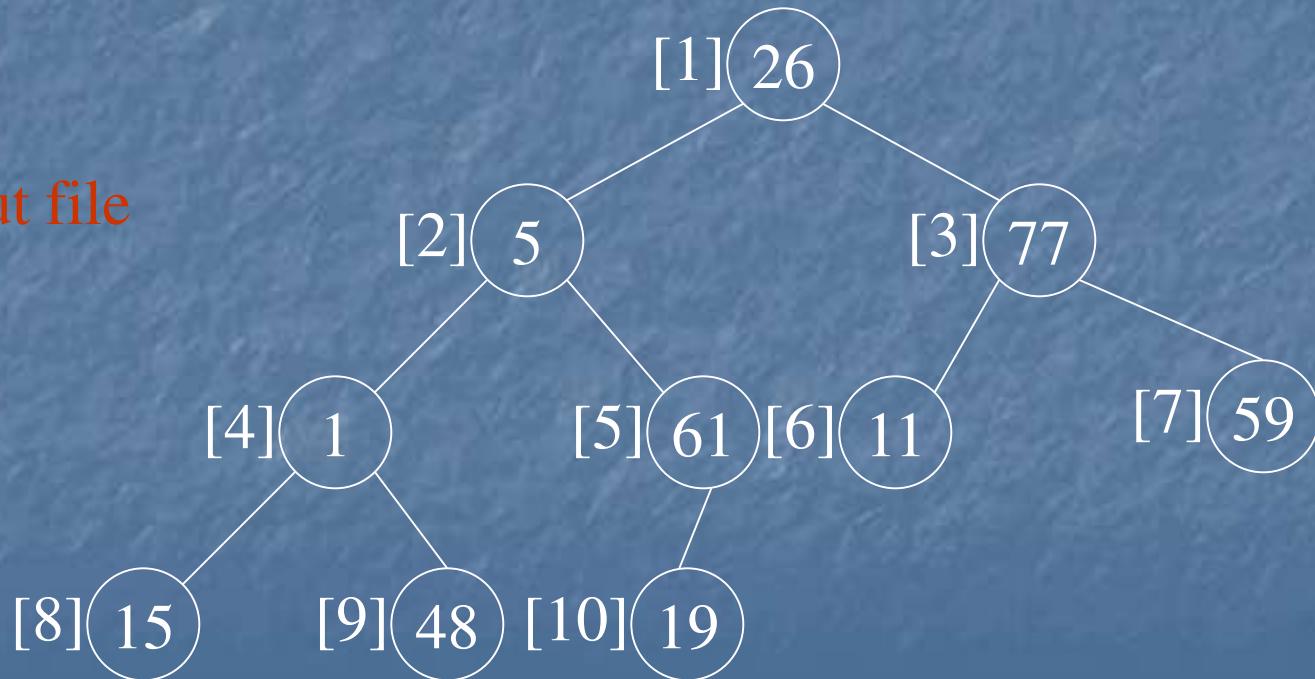
```
while (child <= *n) {  
    /* find the larger child of the current  
    parent */  
    if ((child < *n) &&  
        (heap[child].key < heap[child+1].key))  
        child++;  
    if (temp.key >= heap[child].key) break;  
    /* move to the next lower level */  
    heap[parent] = heap[child];  
    parent = child  
    child *= 2;  
}  
heap[parent] = temp;  
return item;  
}
```

Application On Sorting: Heap Sort

- See an illustration first
 - Array interpreted as a binary tree

1	2	3	4	5	6	7	8	9	10
26	5	77	1	61	11	59	15	48	19

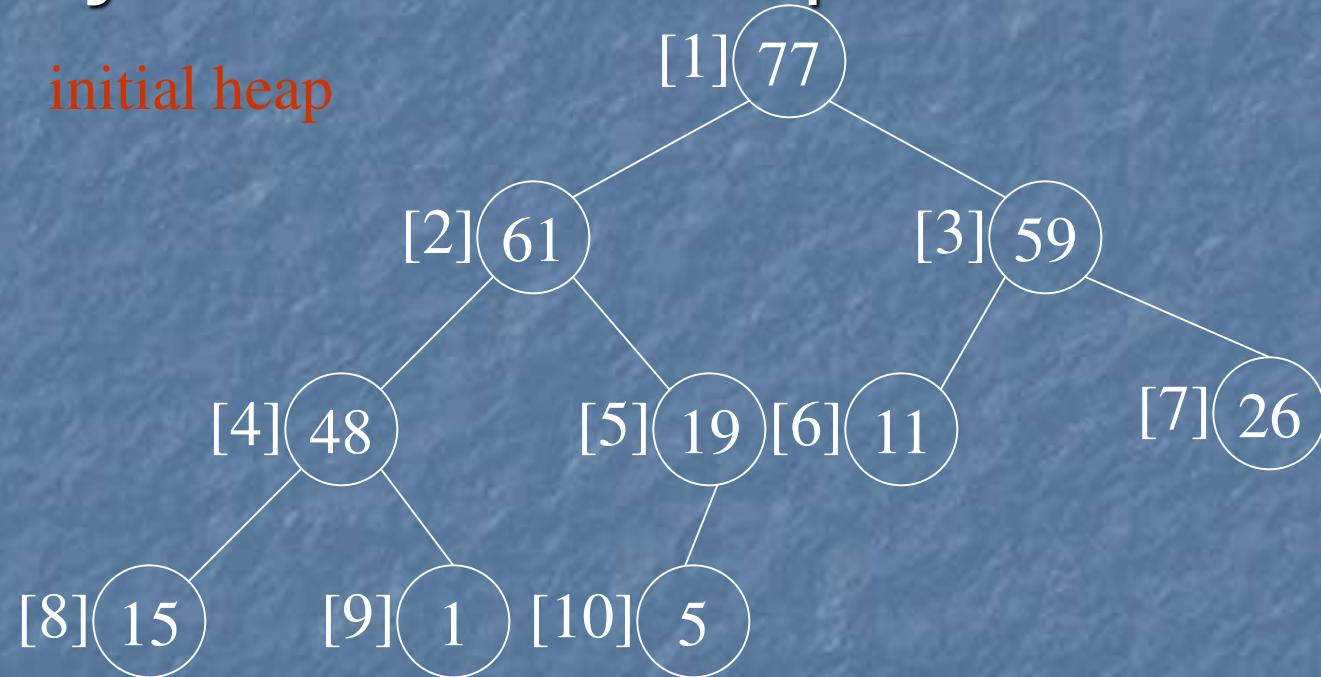
input file



Heap Sort

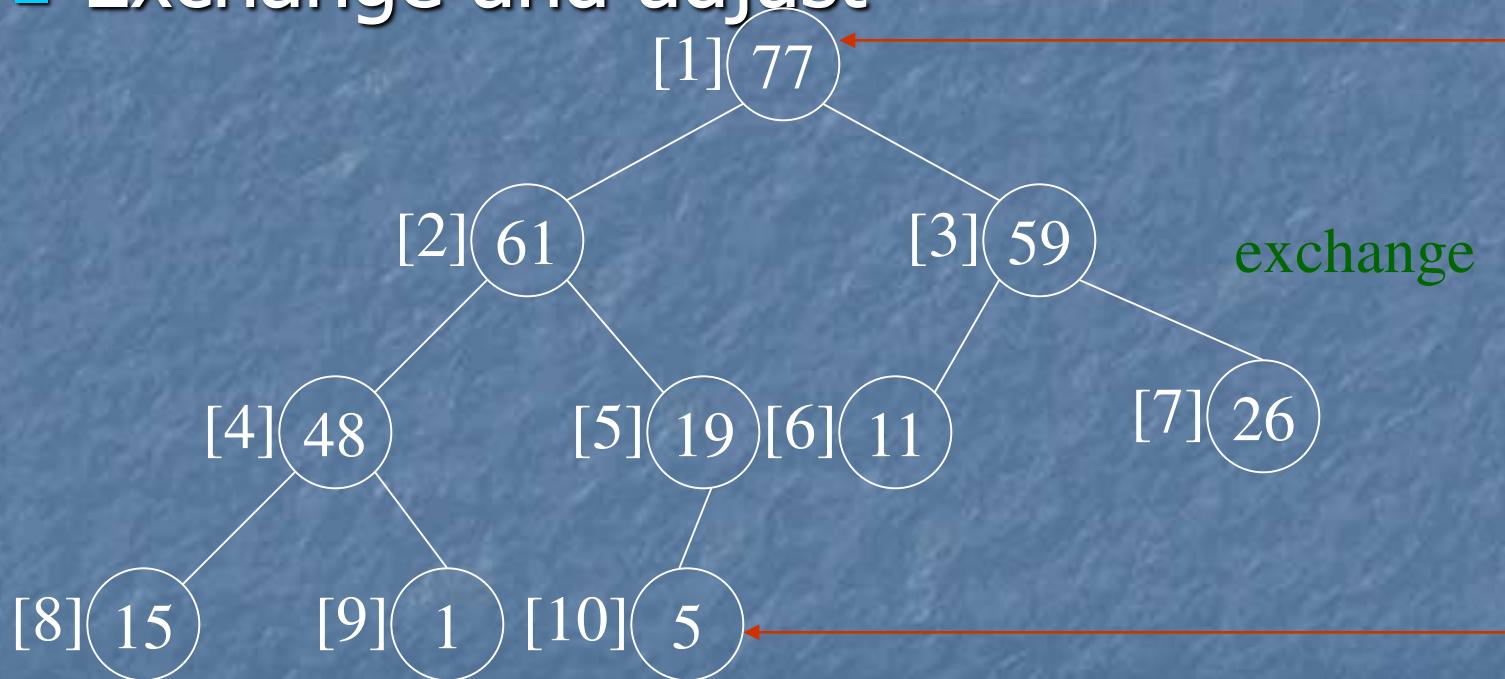
- Adjust it to a MaxHeap

initial heap

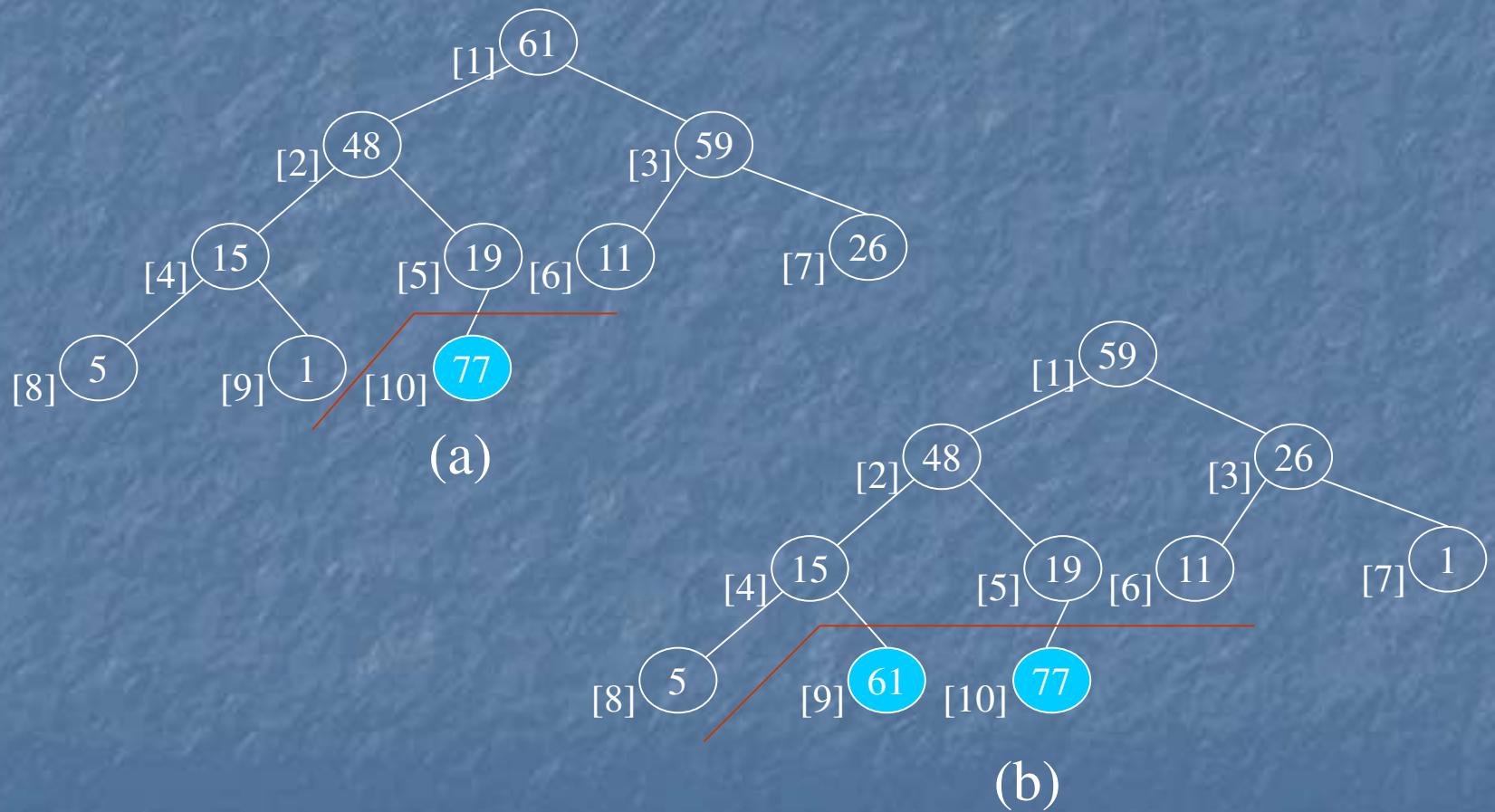


Heap Sort

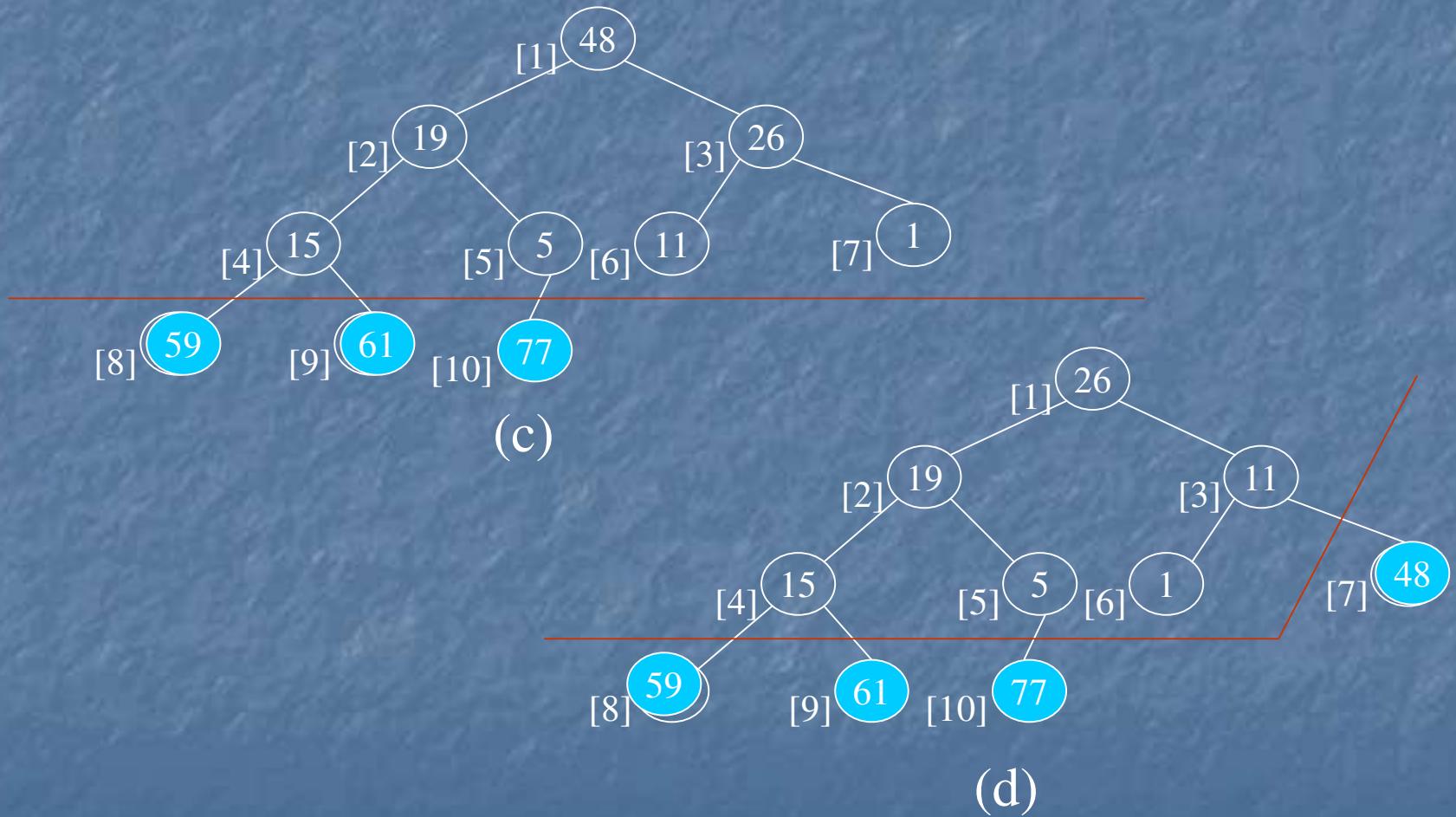
- Exchange and adjust



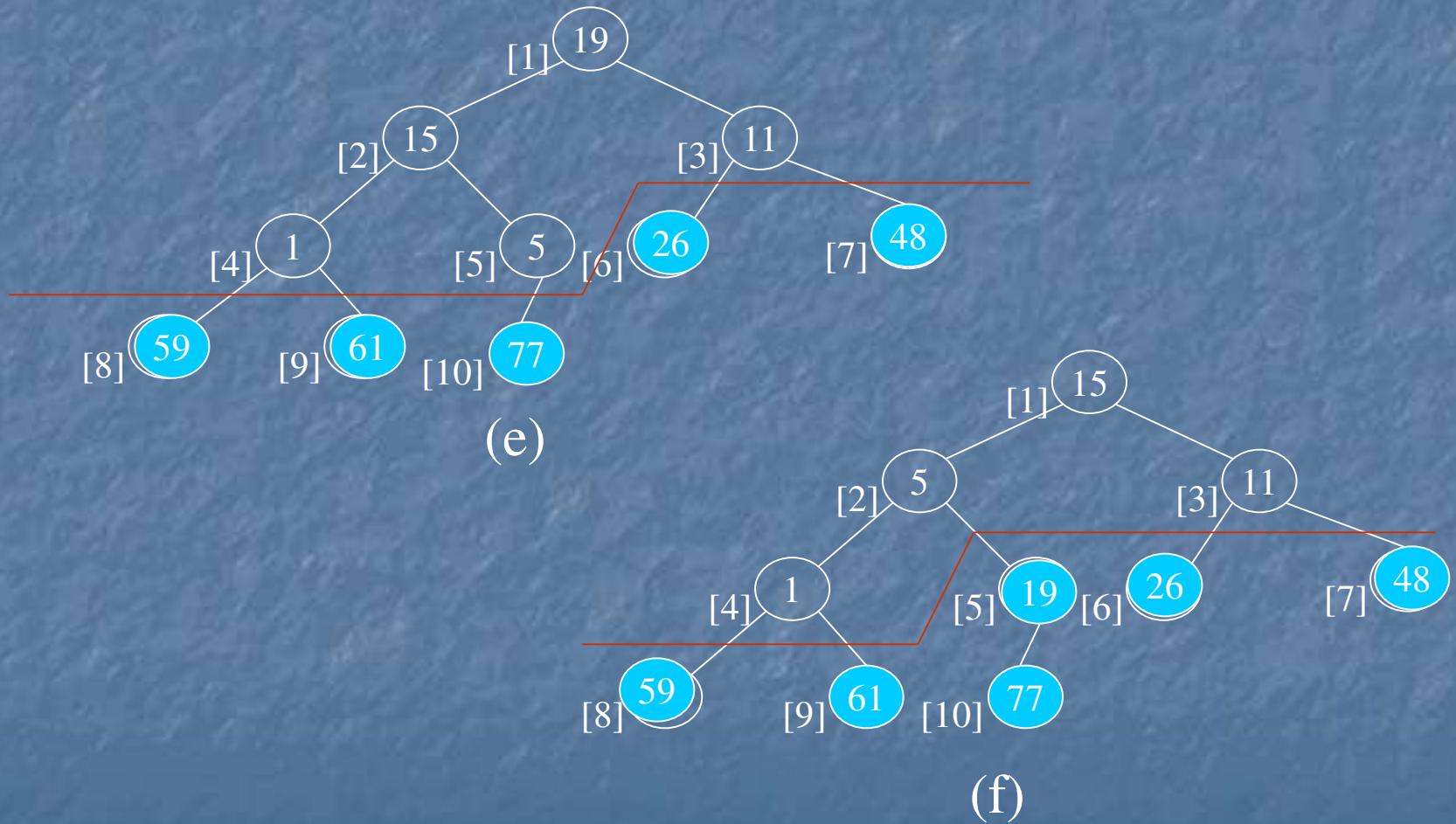
Heap Sort



Heap Sort



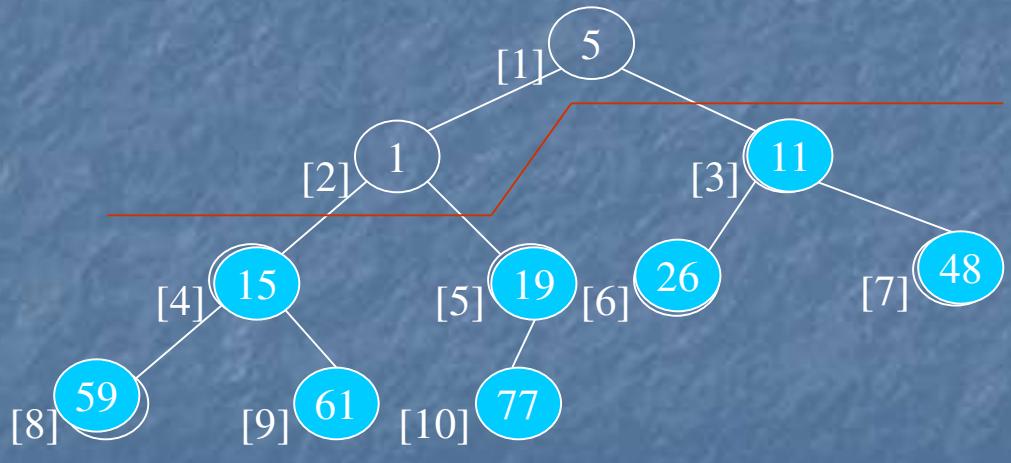
Heap Sort



Heap Sort

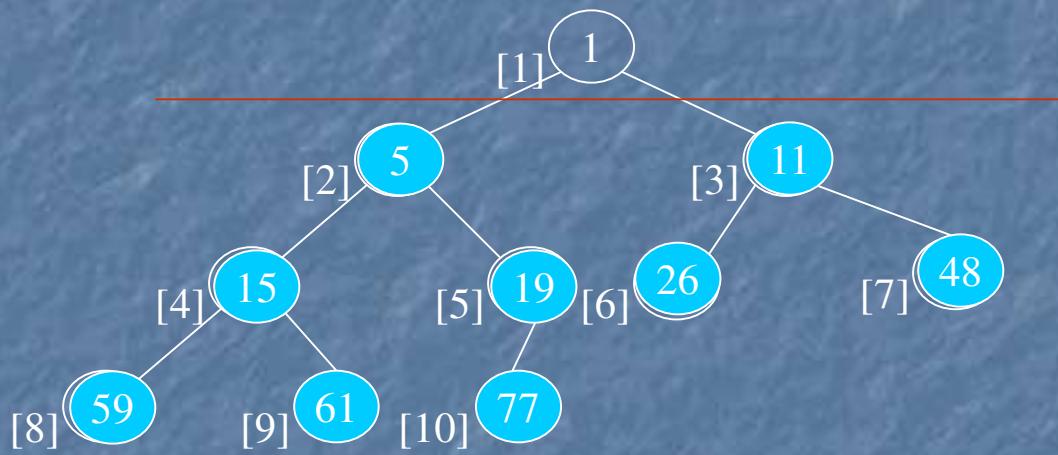


(g)



(h)

Heap Sort



- So results (i)

77 61 59 48 26 19 15 11 5 1

Questions

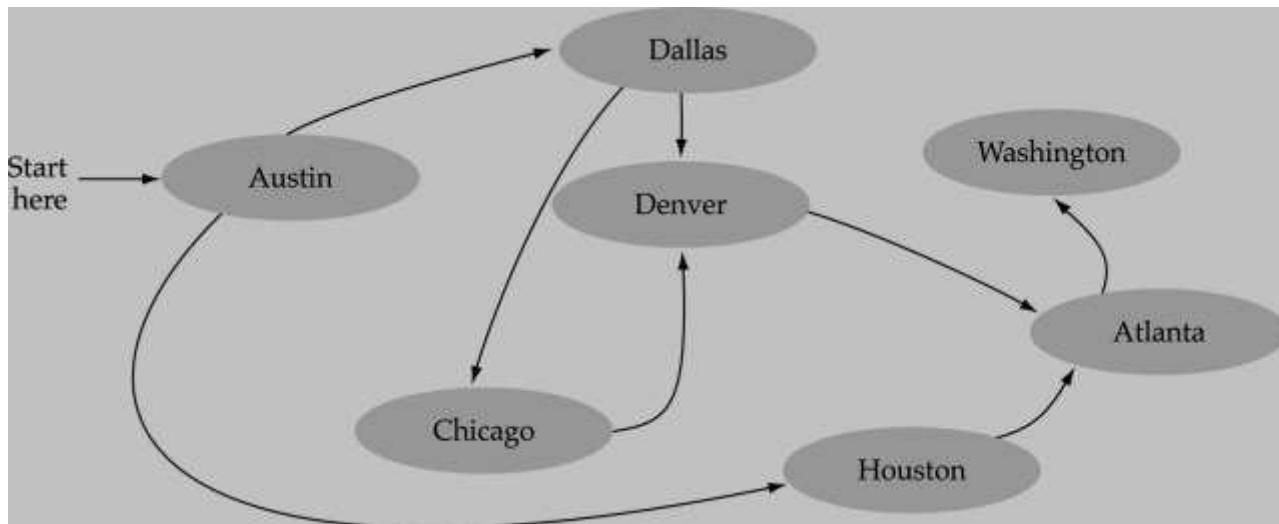
?

Graph

Data Structure

What is a graph?

- A data structure that consists of a **set of nodes (*vertices*)** and a **set of edges** that relate the nodes to each other
- The set of edges describes relationships among the vertices



Formal definition of graphs

- A graph G is defined as follows:

$$G=(V,E)$$

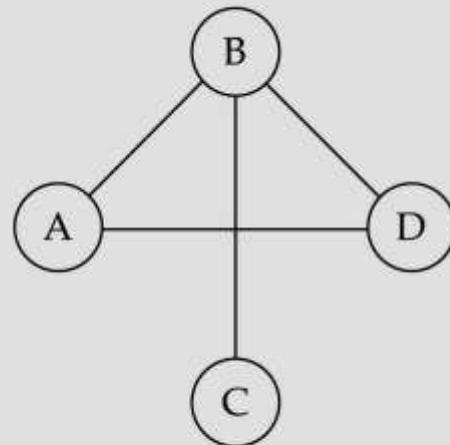
$V(G)$: a finite, nonempty set of vertices

$E(G)$: a set of edges (pairs of vertices)

Undirected graph

- When the edges in a graph have no direction, the graph is called *undirected*

ected graph.

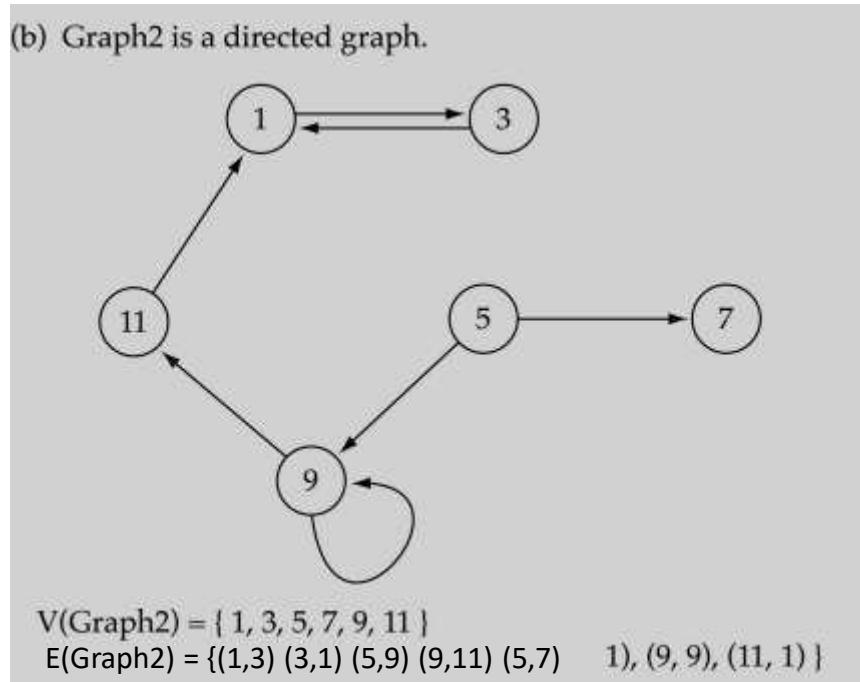


$$V(\text{Graph1}) = \{ A, B, C, D \}$$

$$E(\text{Graph1}) = \{ (A, B), (A, D), (B, C), (B, D) \}$$

Directed Graph

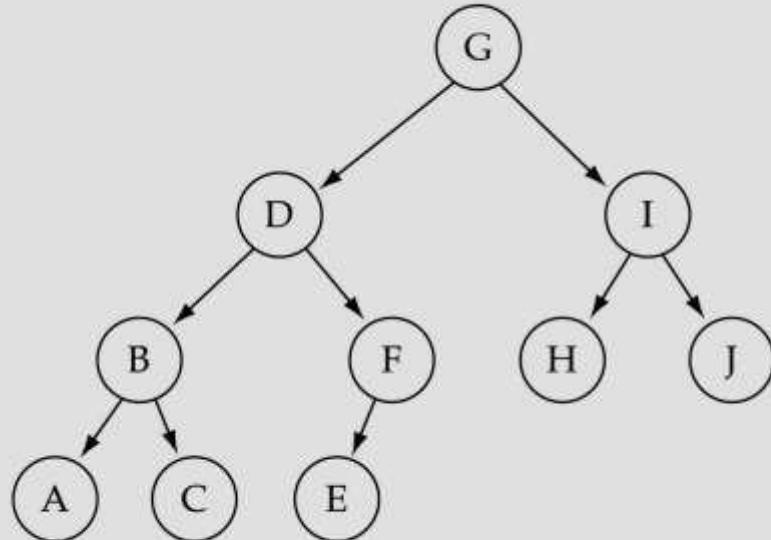
- When the edges in a graph have a direction, the graph is called *directed* (or *digraph*)



Trees vs graphs

- Trees are special cases of graphs!!

(c) Graph3 is a directed graph.



$$V(\text{Graph3}) = \{ A, B, C, D, E, F, G, H, I, J \}$$

$$E(\text{Graph3}) = \{ (G, D), (G, I), (D, B), (D, F), (I, H), (I, J), (B, A), (B, C), (F, E) \}$$

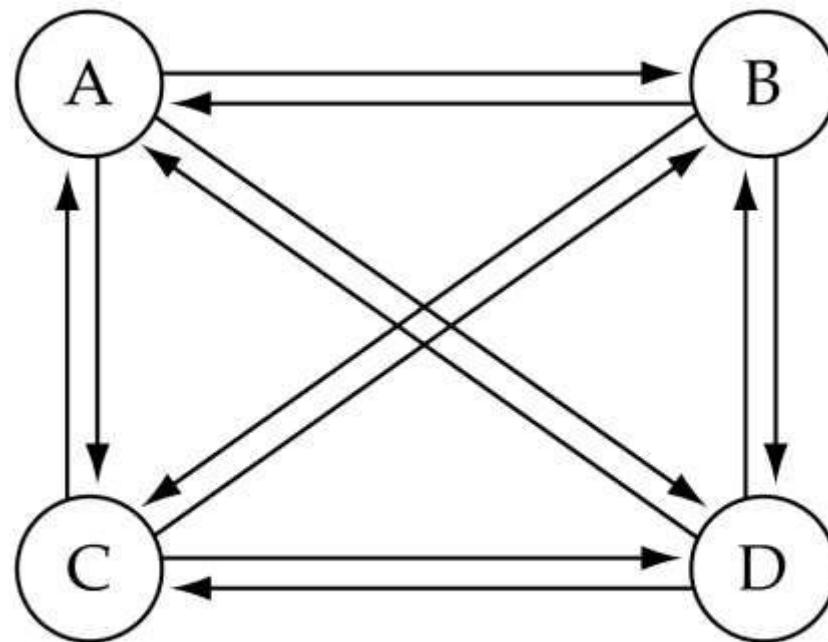
Graph terminology

- Adjacent nodes: two nodes are adjacent if they are connected by an edge
- Path: a sequence of vertices that connect two nodes in a graph
- Complete graph: a graph in which every vertex is directly connected to every other vertex

Graph terminology (cont.)

- What is the number of edges in a complete directed graph with N vertices?

$$N * (N-1)$$

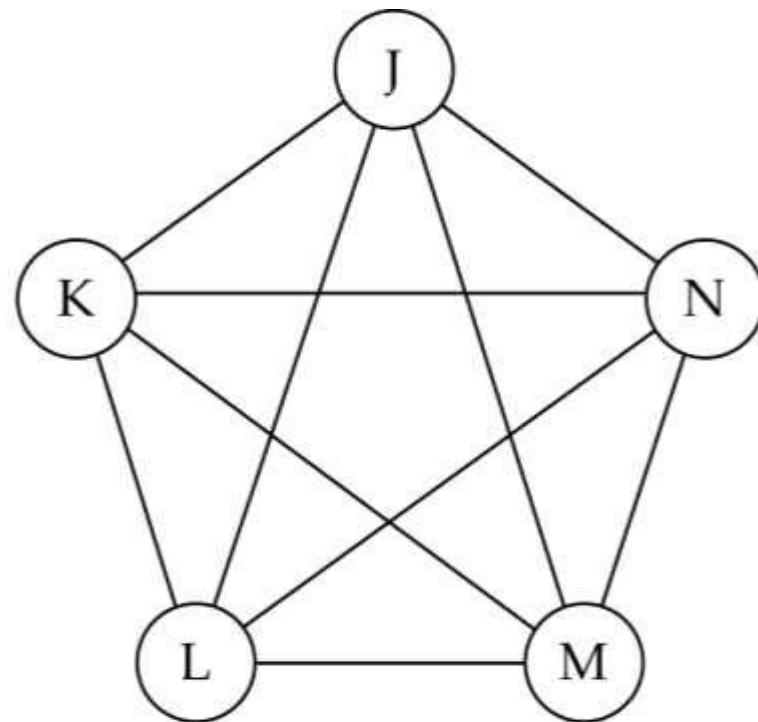


(a) Complete directed graph.

Graph terminology (cont.)

- What is the number of edges in a complete undirected graph with N vertices?

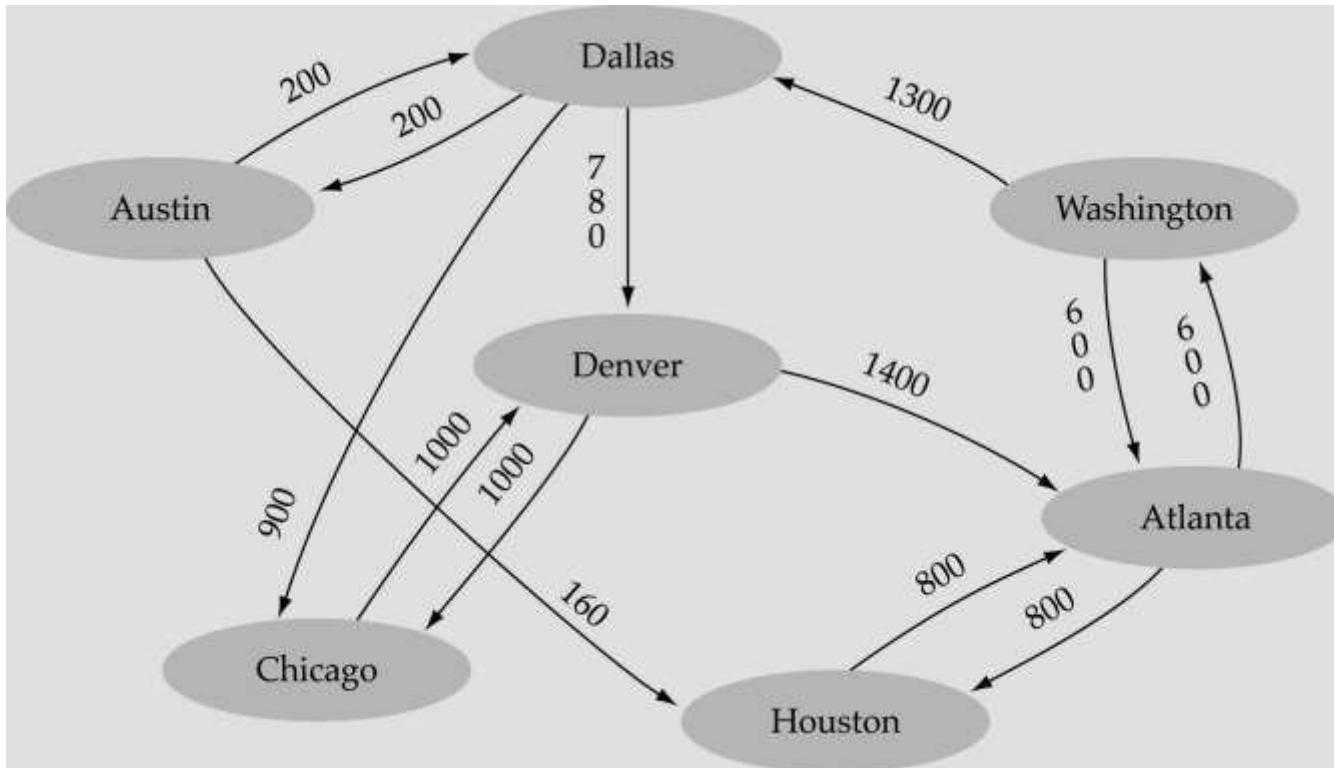
$$N * (N-1) / 2$$



(b) Complete undirected graph.

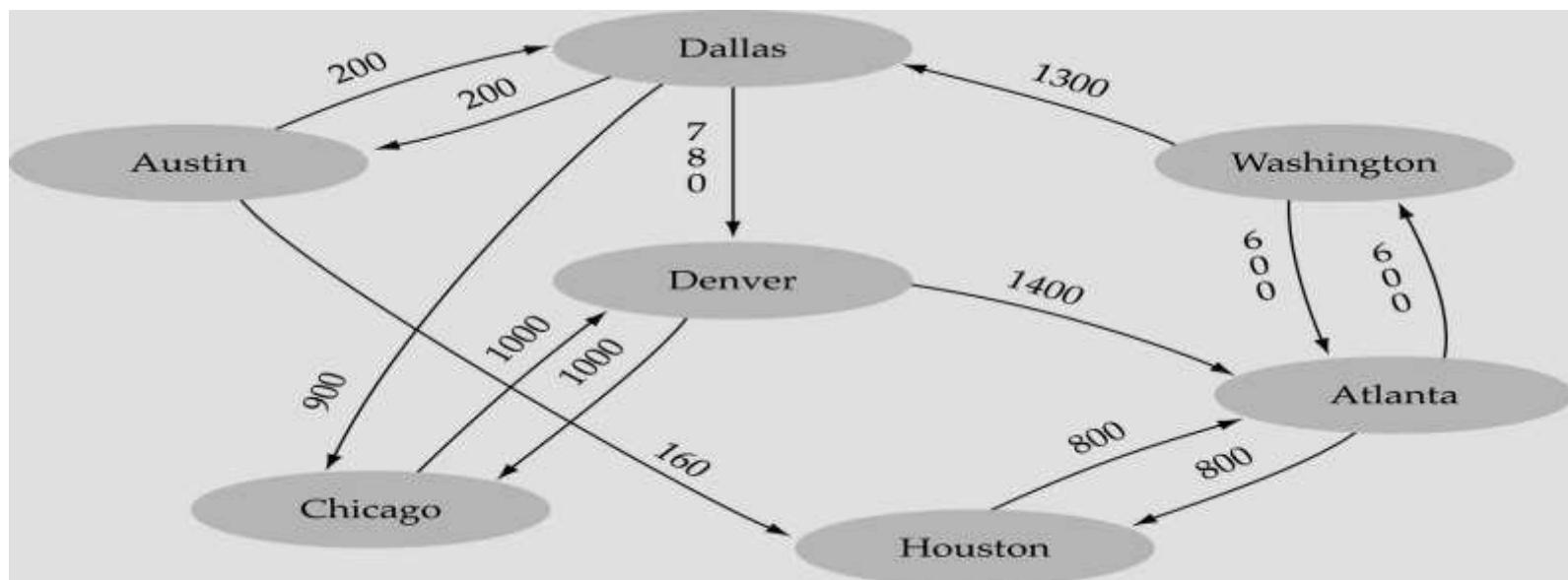
Graph terminology (cont.)

- Weighted graph: a graph in which each edge carries a value



Graph implementation

- Array-based implementation
 - A 1D array is used to represent the vertices
 - A 2D array (adjacency matrix) is used to represent the edges



```

graph LR
    .numVertices 7
    .vertices
    .edges

```

[0]	"Atlanta"
[1]	"Austin"
[2]	"Chicago"
[3]	"Dallas"
[4]	"Denver"
[5]	"Houston"
[6]	"Washington"
[7]	
[8]	
[9]	

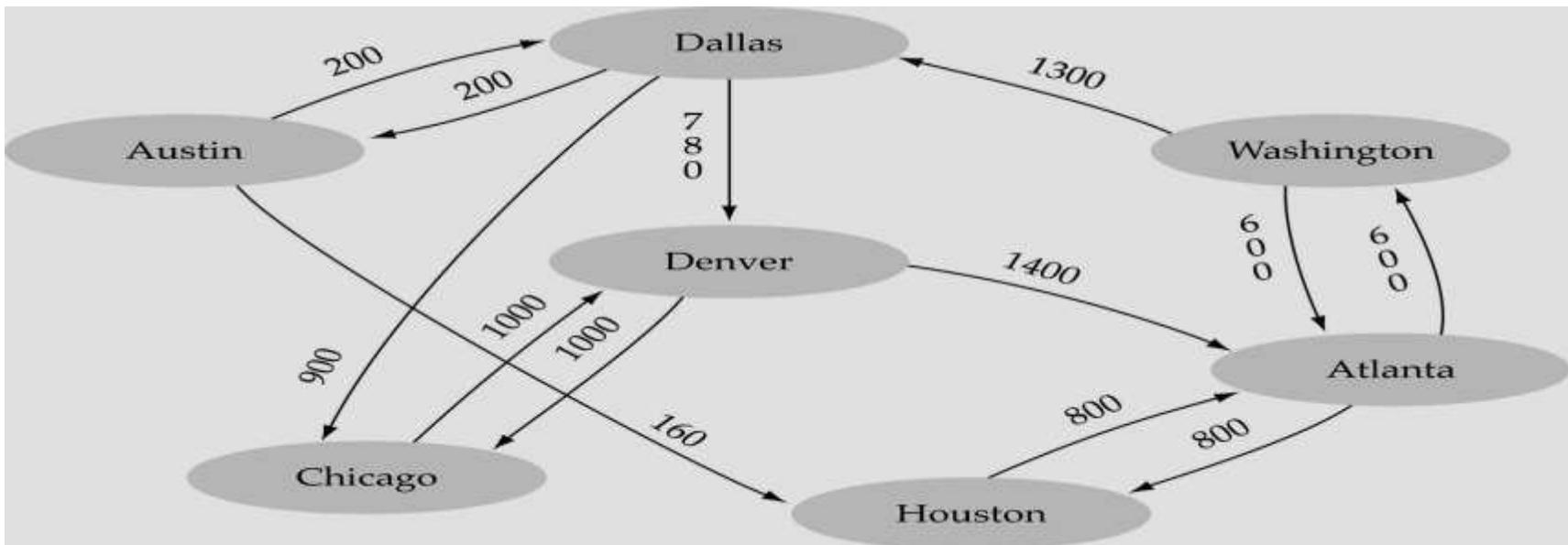
[0]	0	0	0	0	0	800	600	•	•	•
[1]	0	0	0	200	0	160	0	•	•	•
[2]	0	0	0	0	1000	0	0	•	•	•
[3]	0	200	900	0	780	0	0	•	•	•
[4]	1400	0	1000	0	0	0	0	•	•	•
[5]	800	0	0	0	0	0	0	•	•	•
[6]	600	0	0	1300	0	0	0	•	•	•
[7]	•	•	•	•	•	•	•	•	•	•
[8]	•	•	•	•	•	•	•	•	•	•
[9]	•	•	•	•	•	•	•	•	•	•

[0] [1] [2] [3] [4] [5] [6] [7] [8] [9]

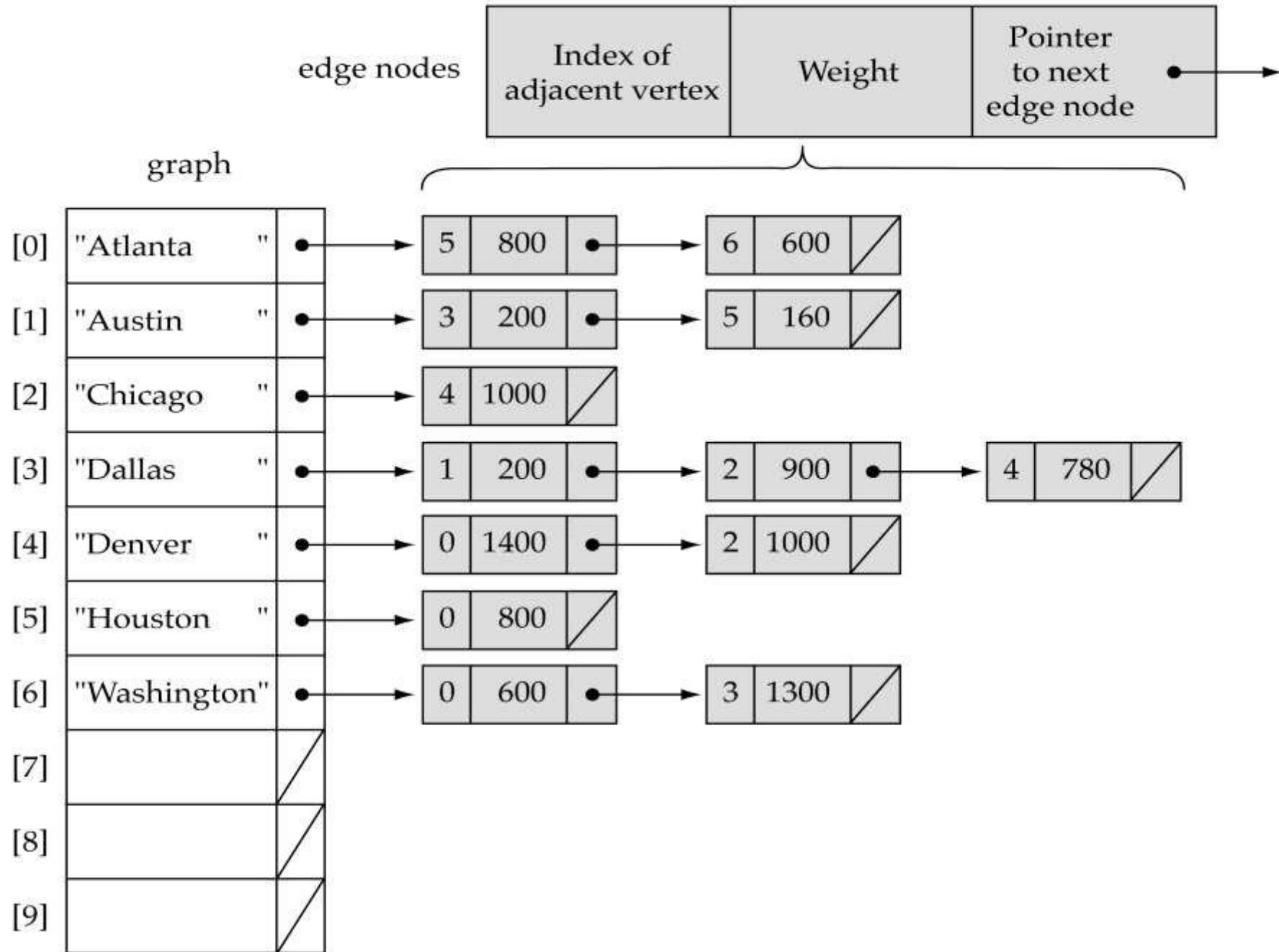
(Array positions marked '•' are undefined)

Graph implementation (cont.)

- Linked-list implementation
 - A 1D array is used to represent the vertices
 - A list is used for each **vertex v** which contains the vertices which are adjacent from v (adjacency list)



(a)



Adjacency matrix vs. adjacency list representation

- **Adjacency matrix**
 - Good for dense graphs -- $|E| \sim O(|V|^2)$
 - Memory requirements: $O(|V| + |E|) = O(|V|^2)$
 - Connectivity between two vertices can be tested quickly
- **Adjacency list**
 - Good for sparse graphs -- $|E| \sim O(|V|)$
 - Memory requirements: $O(|V| + |E|) = O(|V|)$
 - Vertices adjacent to another vertex can be found quickly

Graph Traversal Techniques

- A) Depth-First-Traversal/ Depth-First-Search (DFS)
- B) Breadth-First-Traversal/ Breadth-First-Search (BFS)

A) Depth-First-Traversal/ Depth-First-Search (DFS)

- DFS visits the child vertices before visiting the sibling vertices
- Traverses the depth of any particular path before exploring its breadth
- A stack is generally used when implementing DFS
- The algorithm begins with a chosen "root" vertex;
- It then iteratively transitions from the current vertex to an adjacent, unvisited vertex, until it can no longer find an unexplored vertex to transition to from its current location.

DFS algorithm

A standard DFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

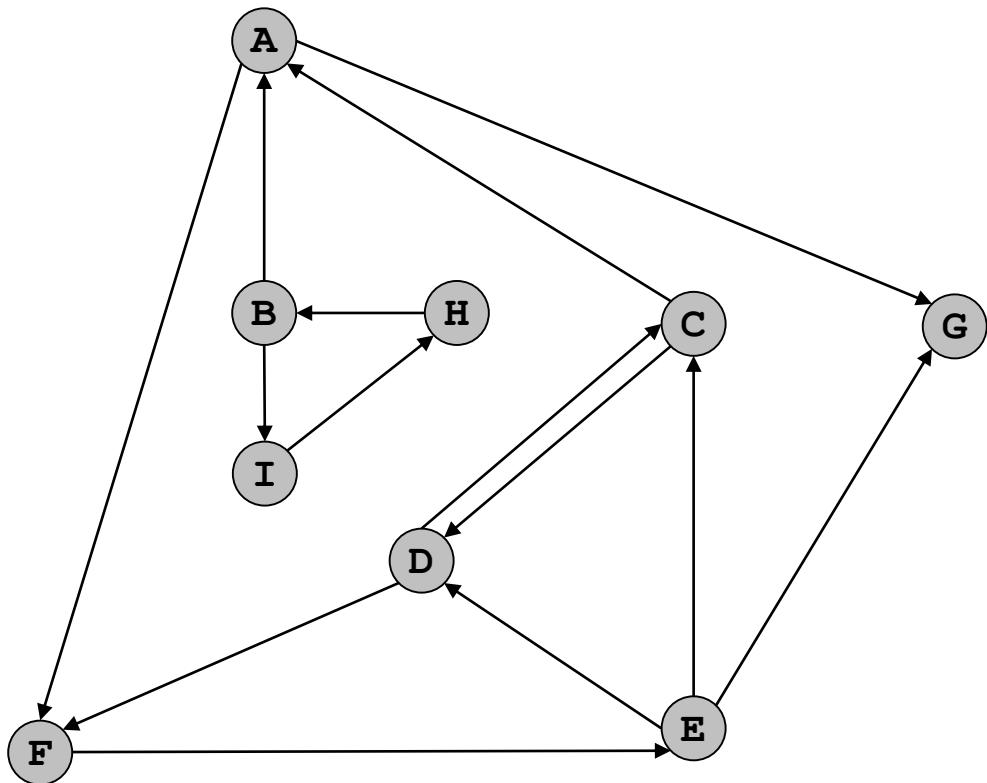
The DFS algorithm works as follows:

- i. Start by putting any one of the graph's vertices on top of a stack.
- ii. Take the top item of the stack and add it to the visited list.
- iii. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of stack.
- iv. Keep repeating steps 2 and 3 until the stack is empty.

Algorithm for DFS

1. Push Starting Vertex
 2. While (Stack is Empty)
 - { i) P=top()
 - ii) Push only one adjacent vertex of P and Print
If no valid adjacent Vertex for P , Then
Pop()
- }

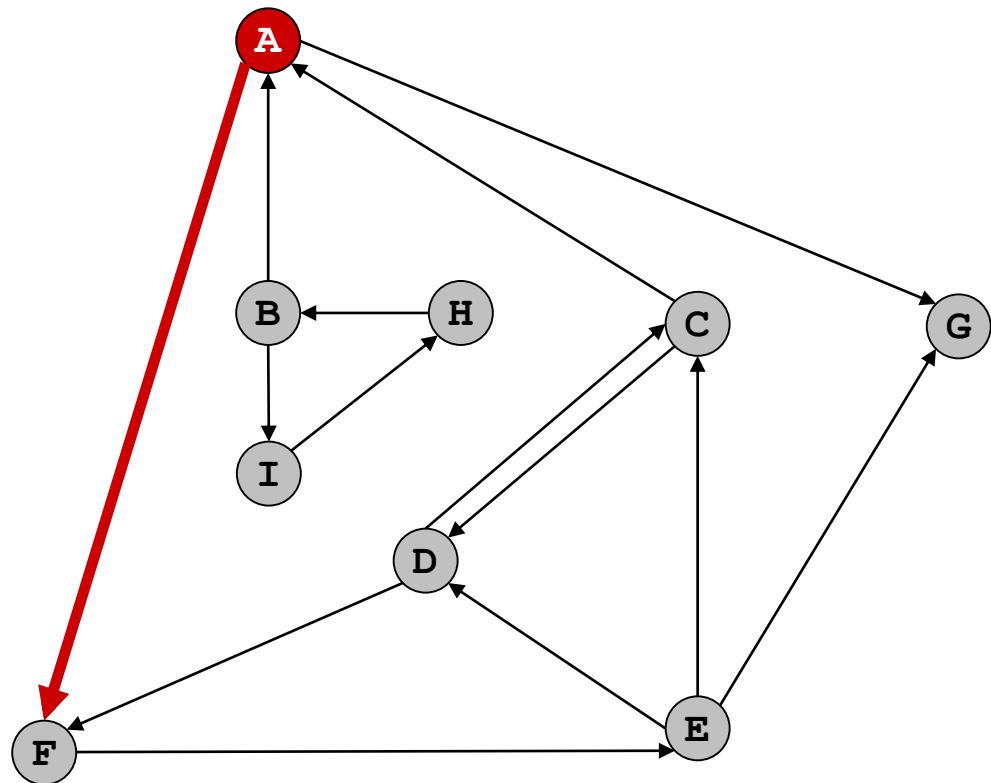
Directed Depth First Search



Adjacency Lists

A:	F	G	
B:	A	I	
C:	A	D	
D:	C	F	
E:	C	D	G
F:	E		
G:			
H:	B		
I:	H		

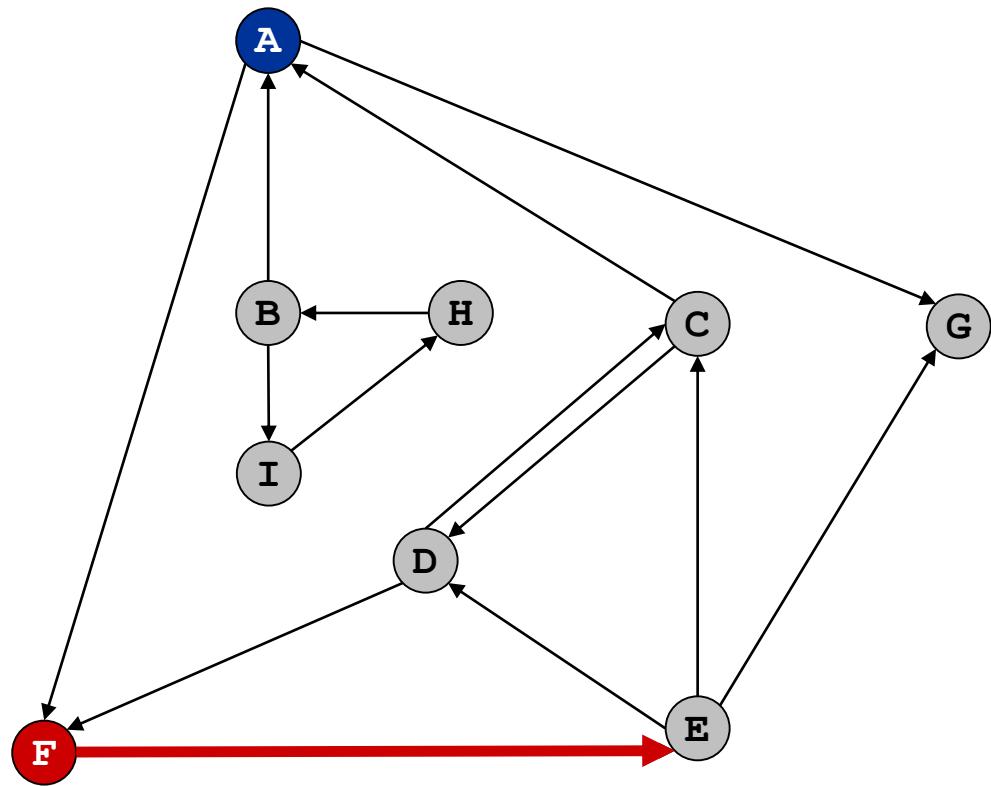
Directed Depth First Search



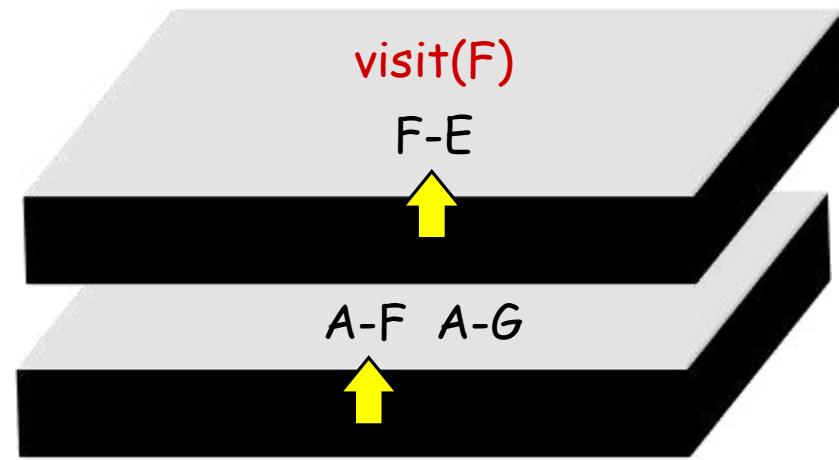
Function call stack:

dfs(A)
A-F A-G

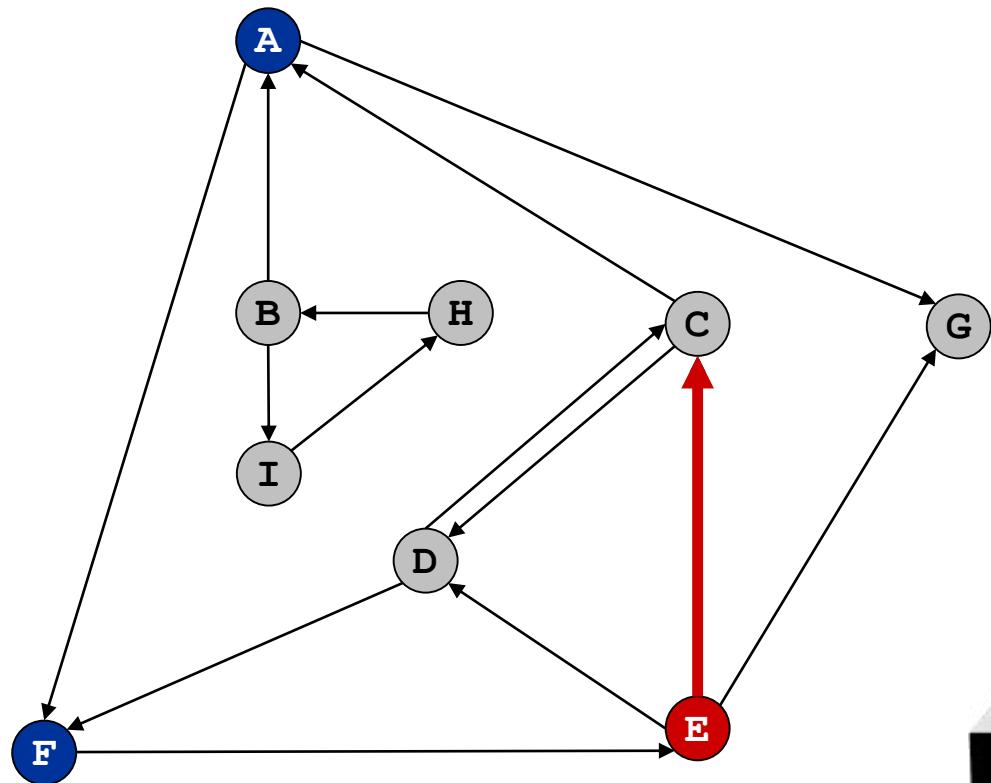
Directed Depth First Search



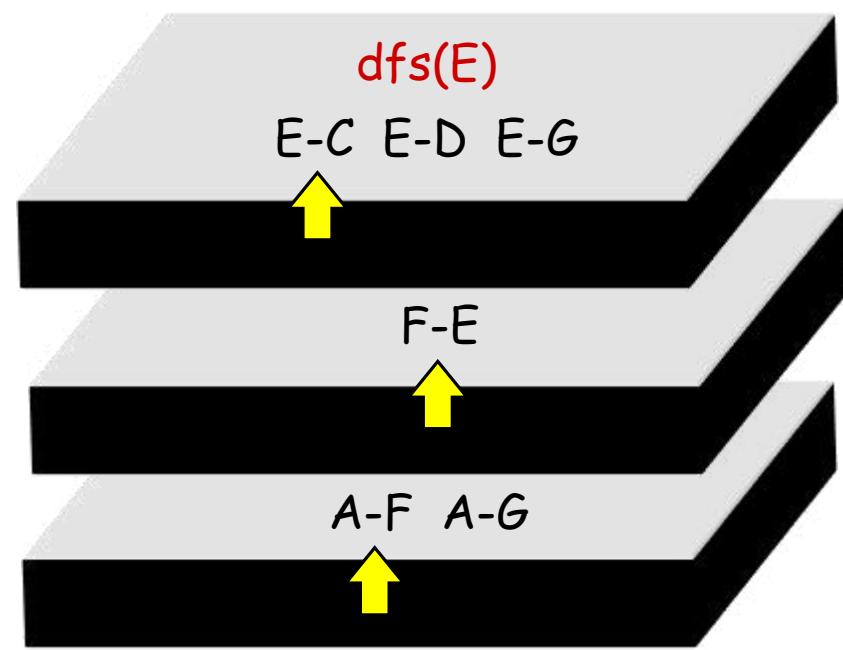
Function call stack:



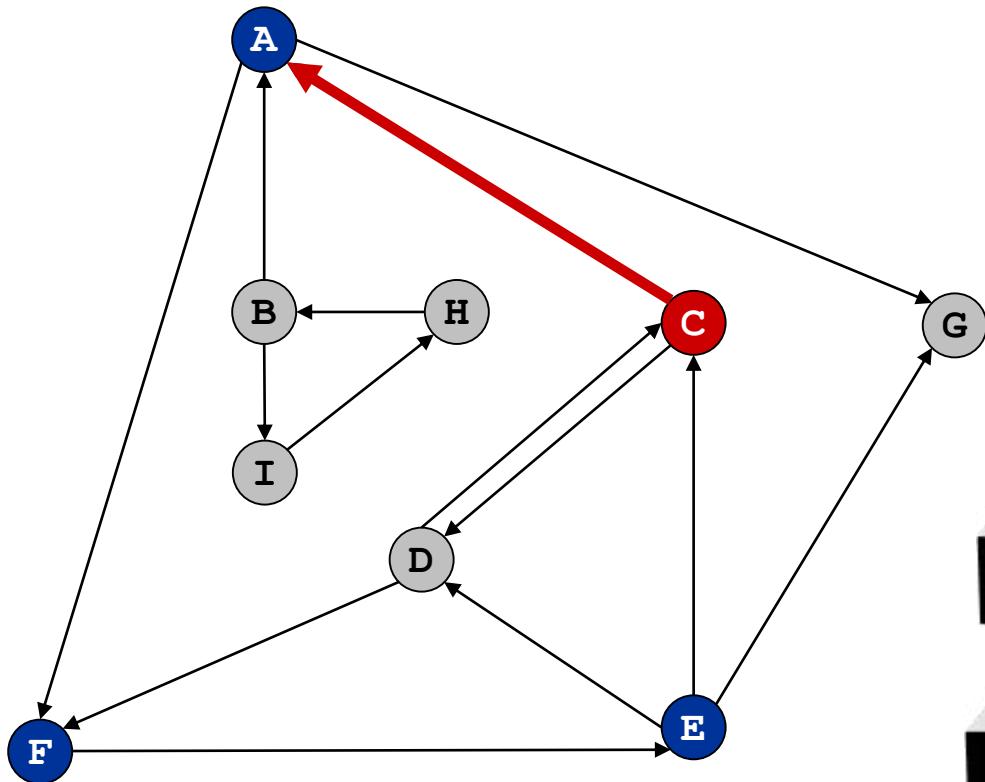
Directed Depth First Search



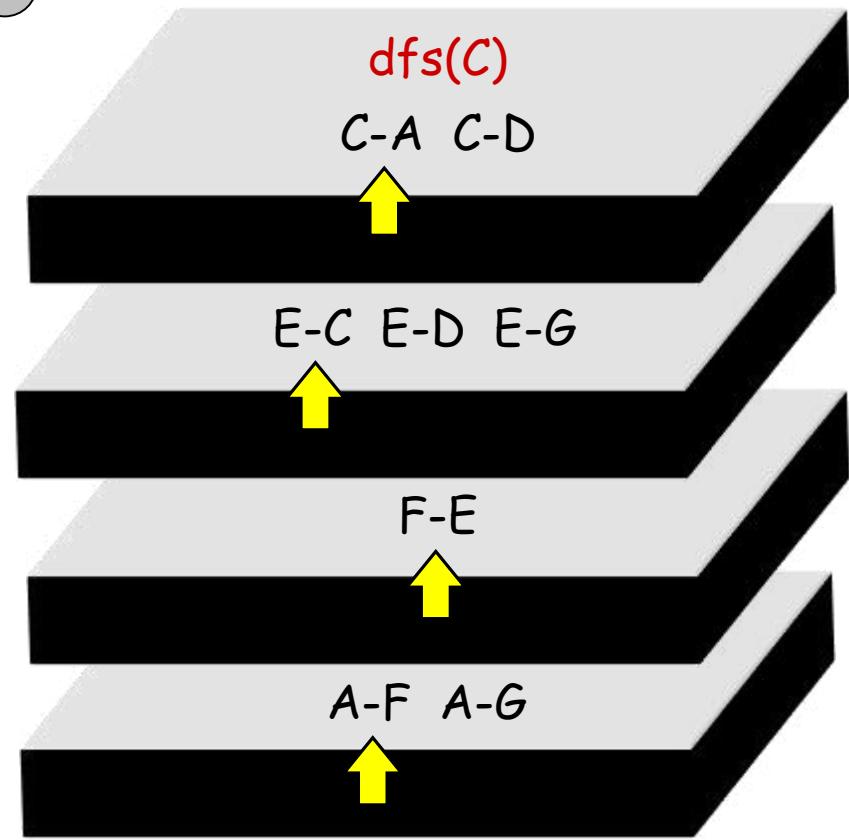
Function call stack:



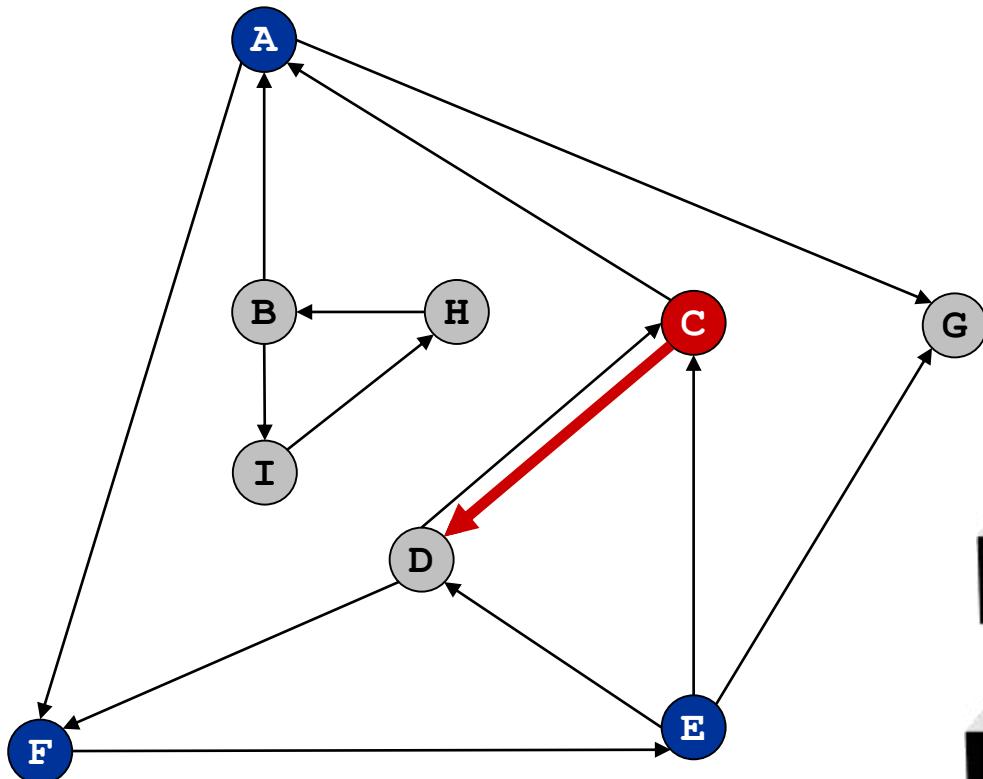
Directed Depth First Search



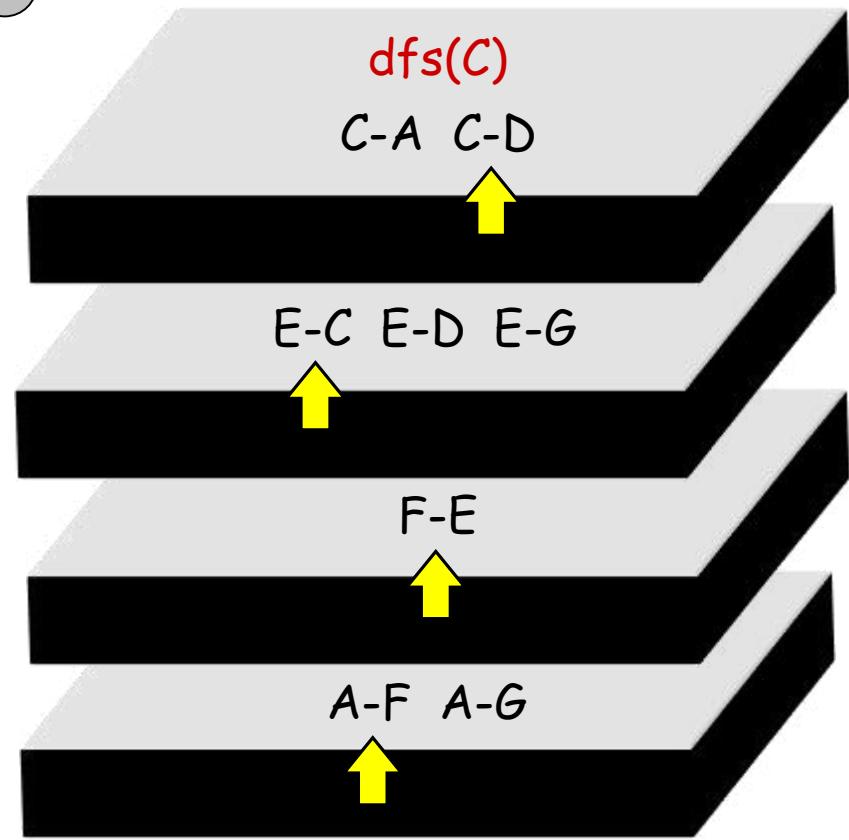
Function call stack:



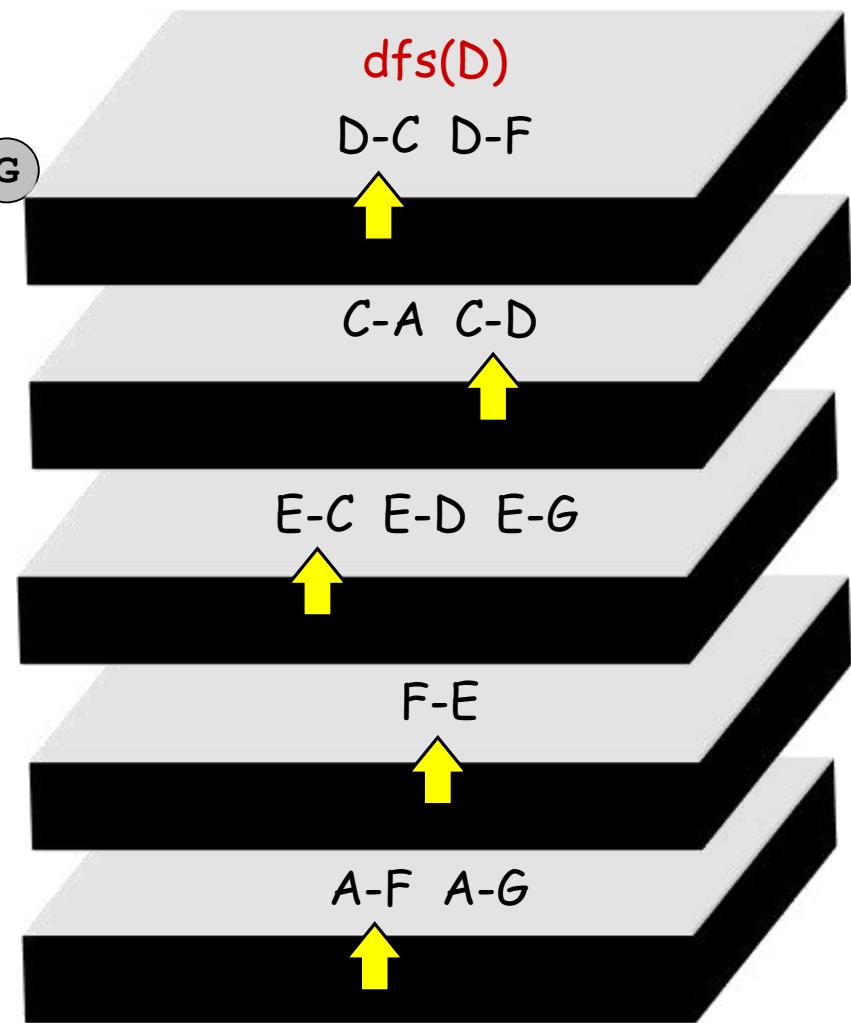
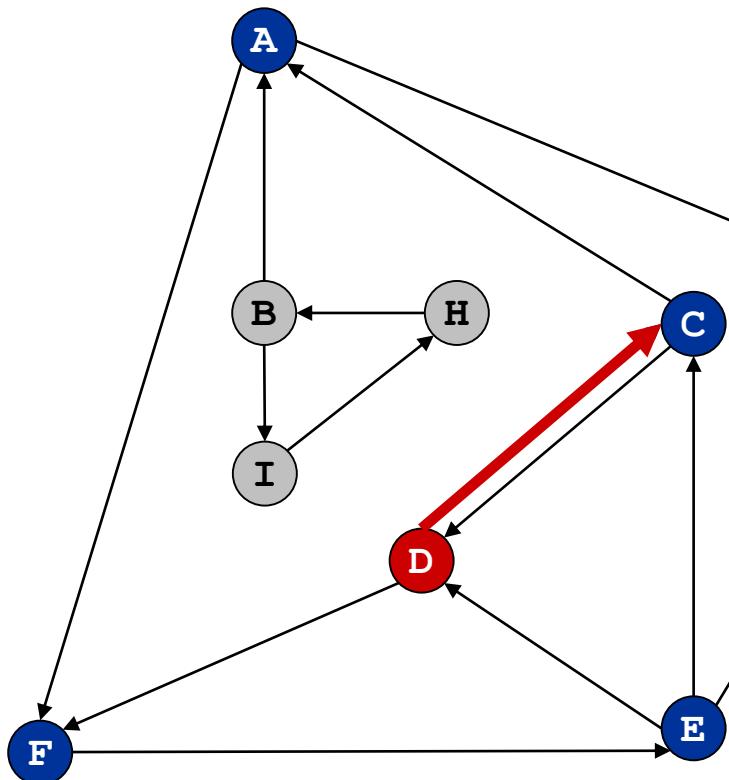
Directed Depth First Search



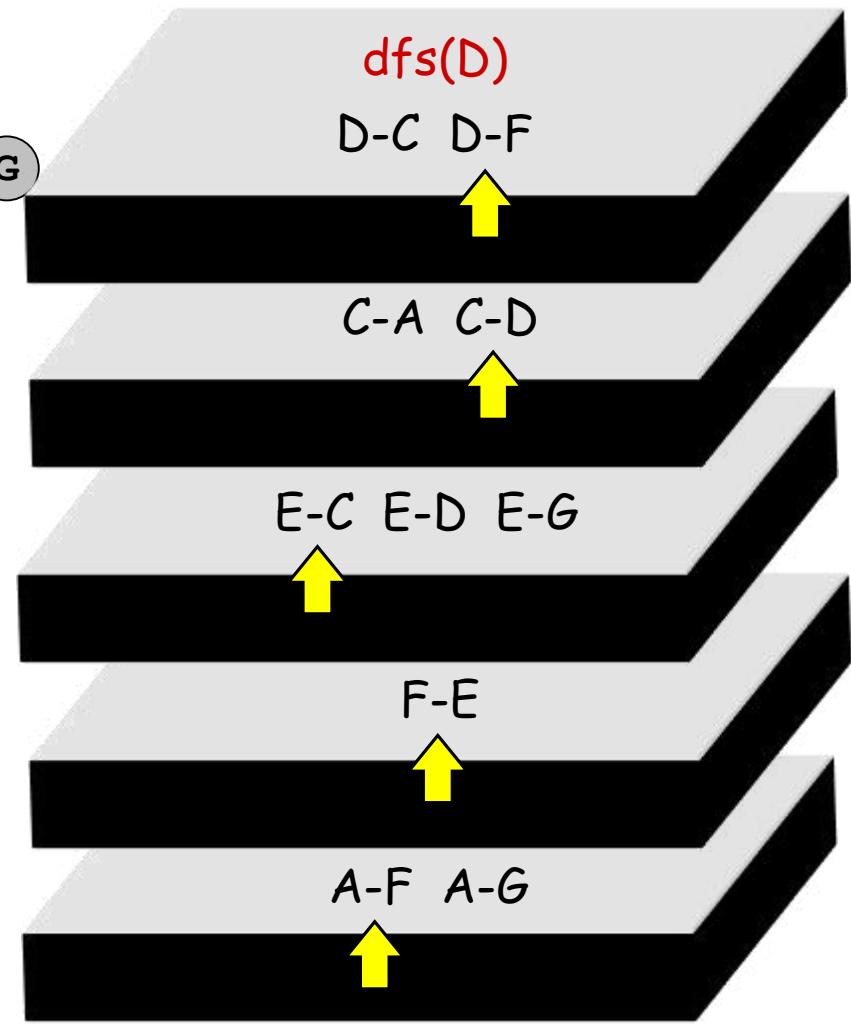
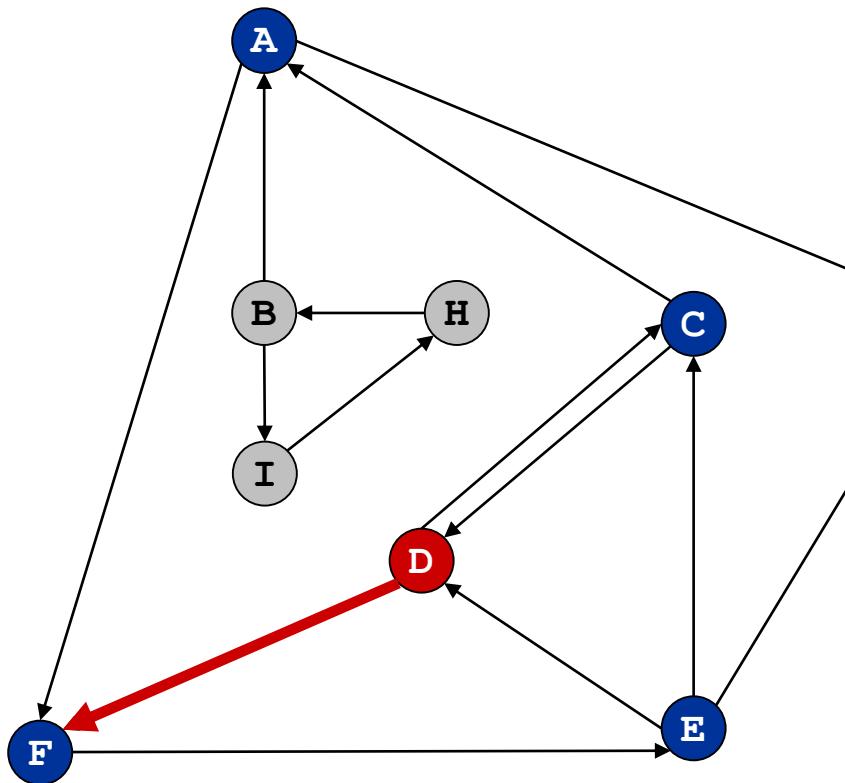
Function call stack:



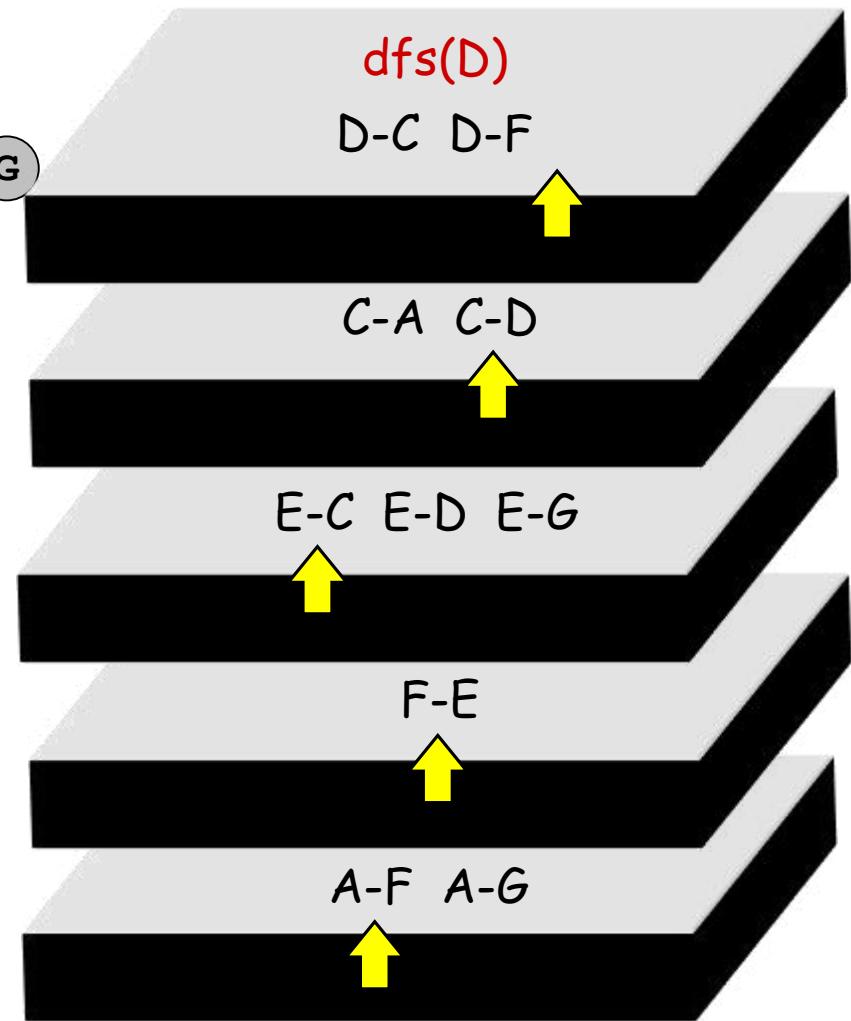
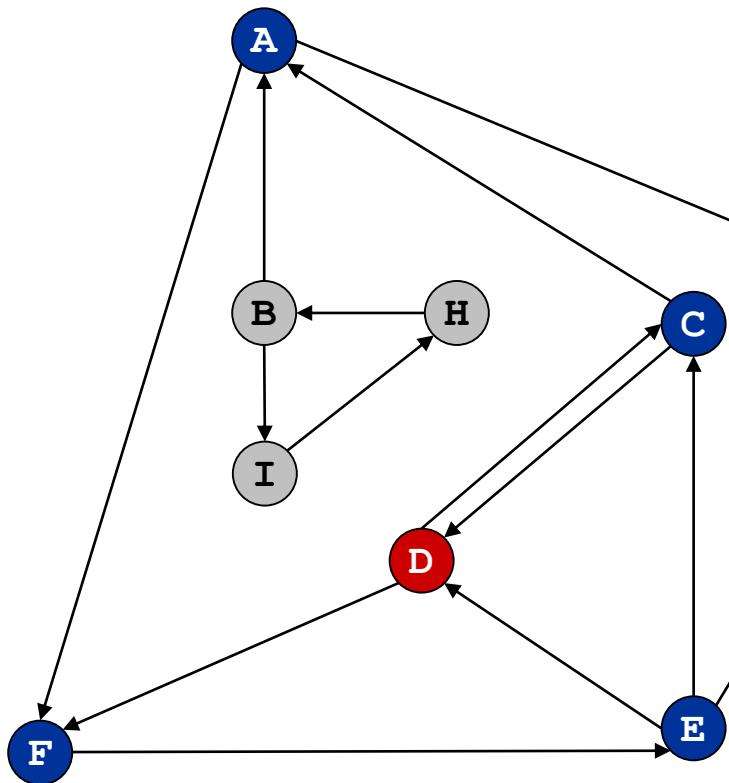
Directed Depth First Search



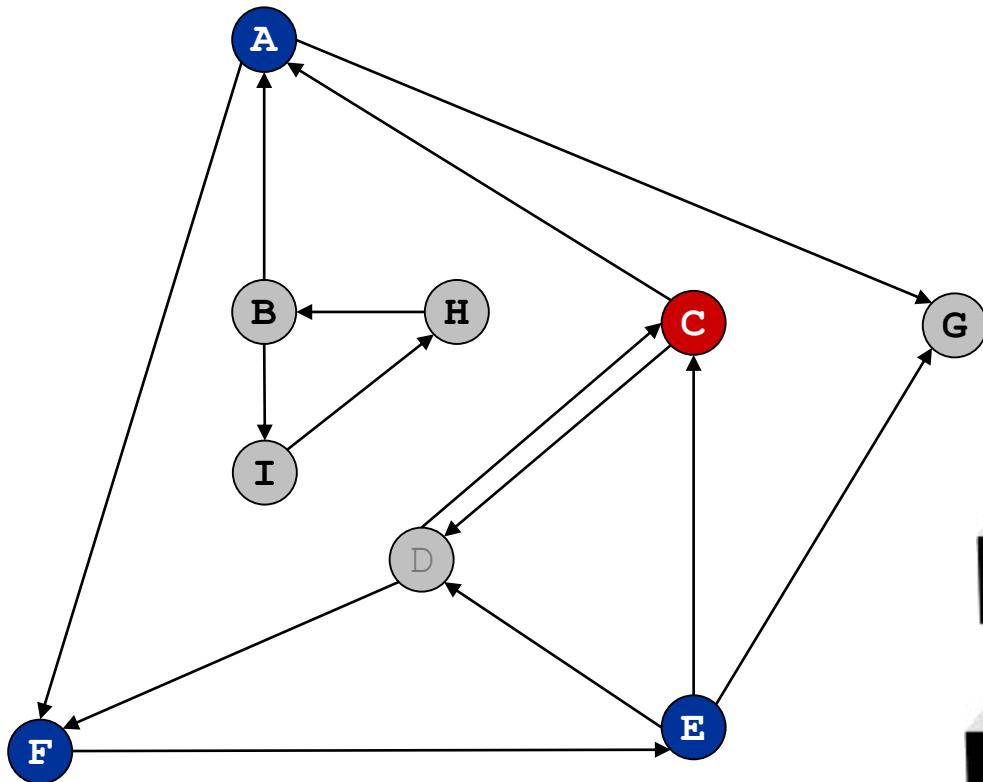
Directed Depth First Search



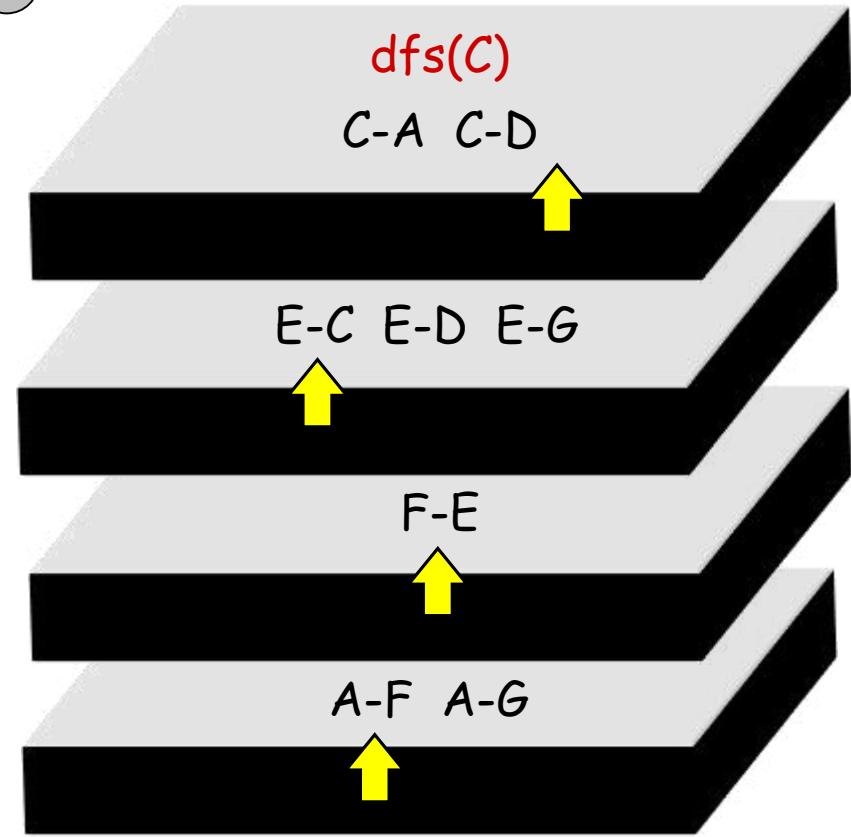
Directed Depth First Search



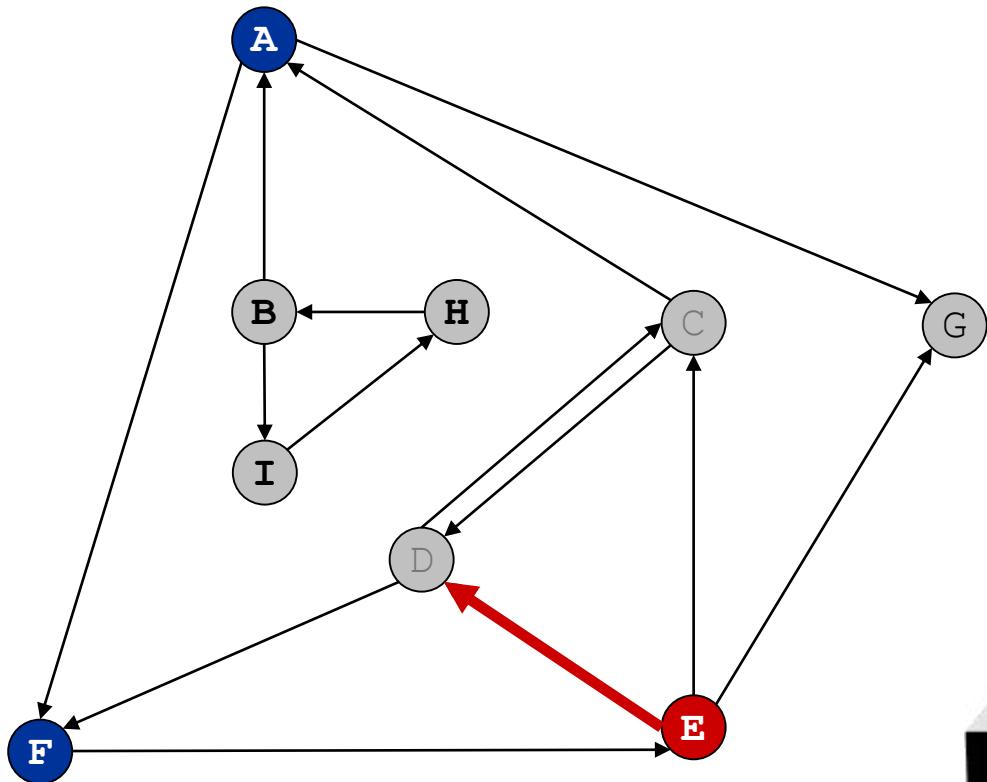
Directed Depth First Search



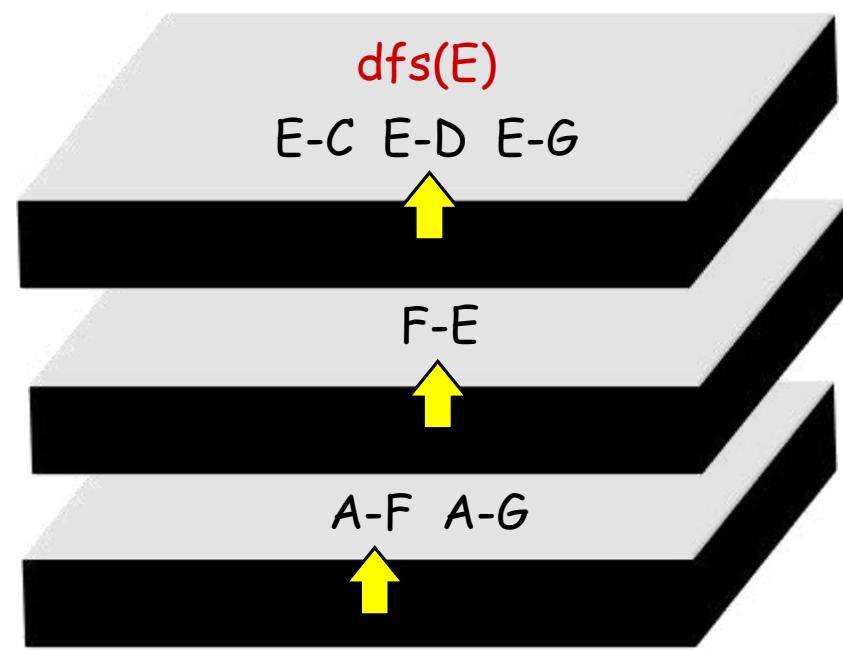
Function call stack:



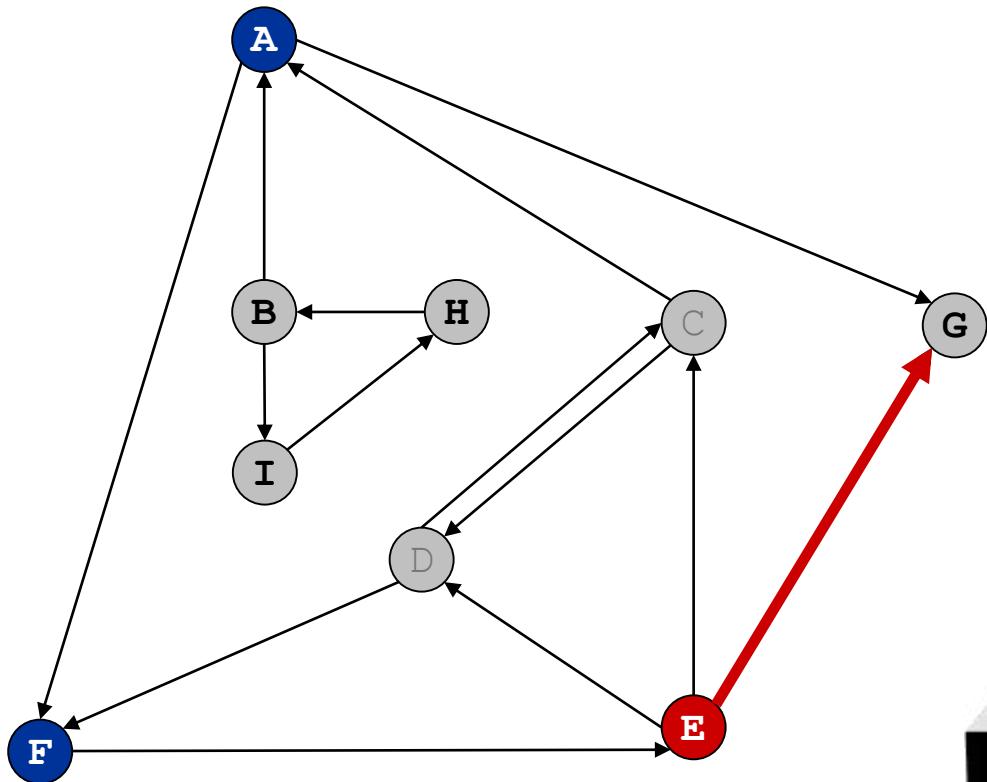
Directed Depth First Search



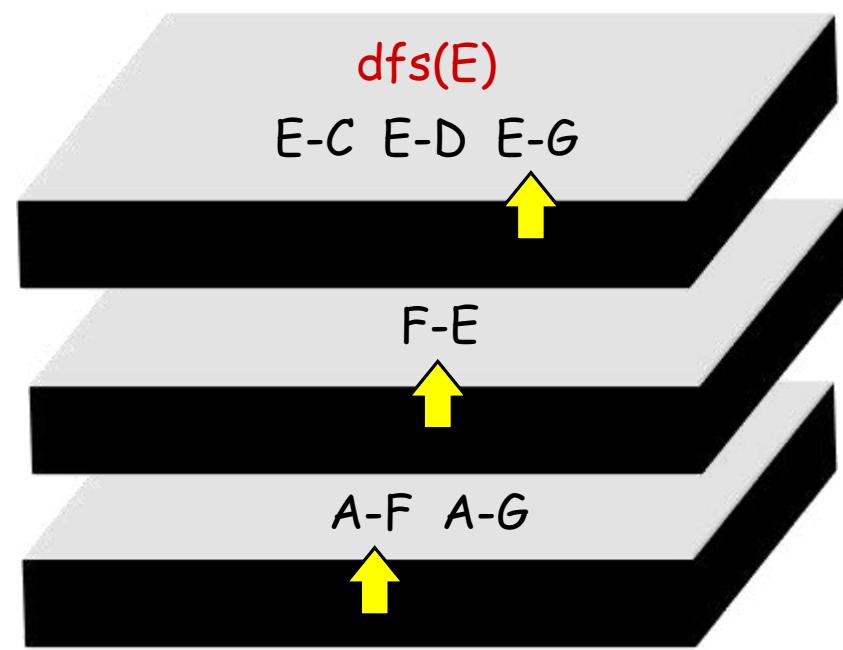
Function call stack:



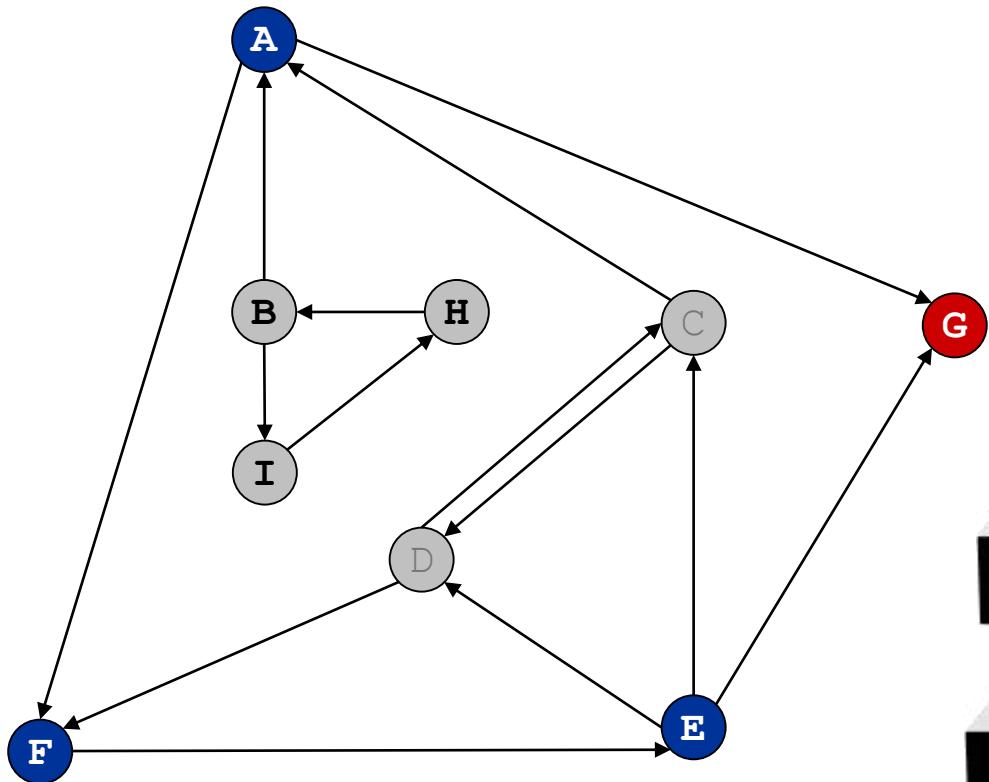
Directed Depth First Search



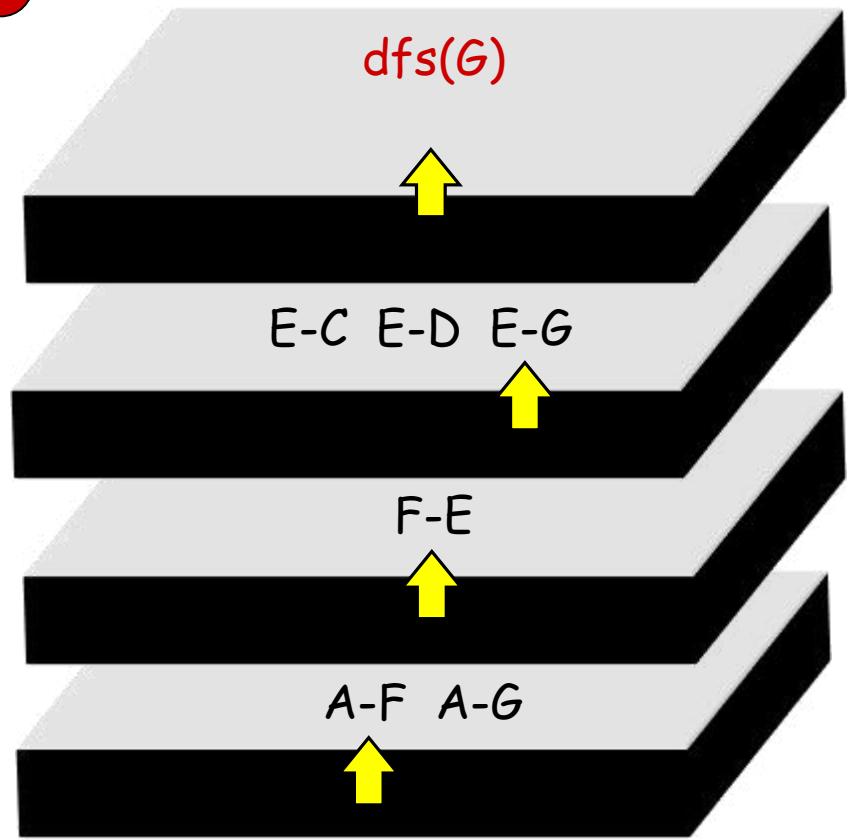
Function call stack:



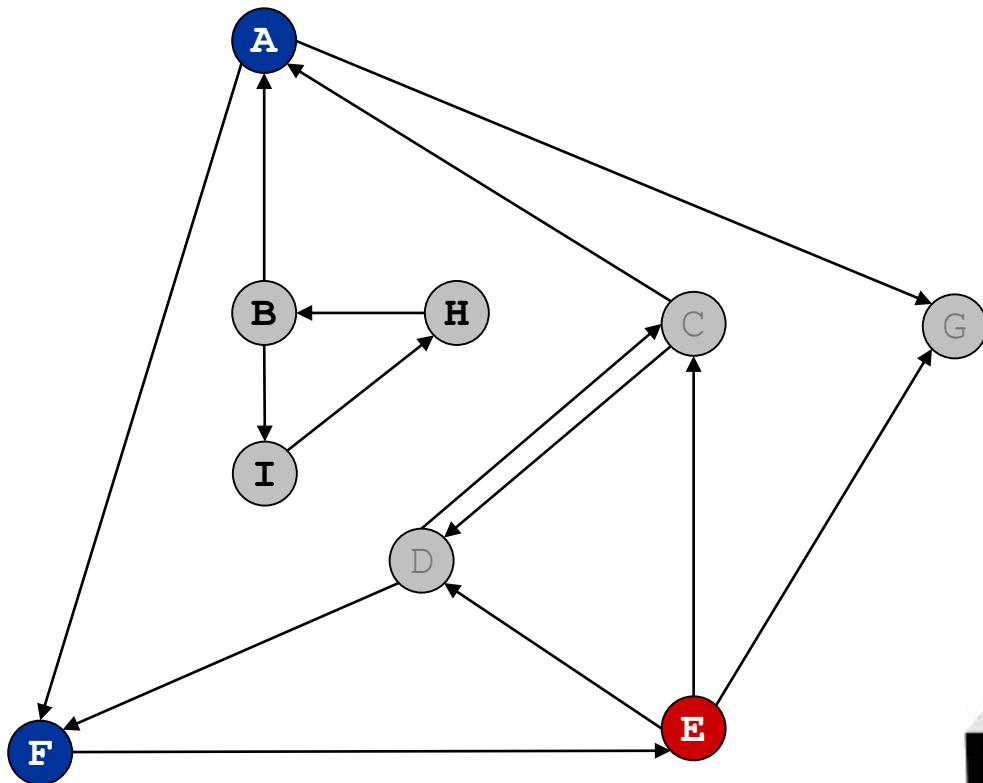
Directed Depth First Search



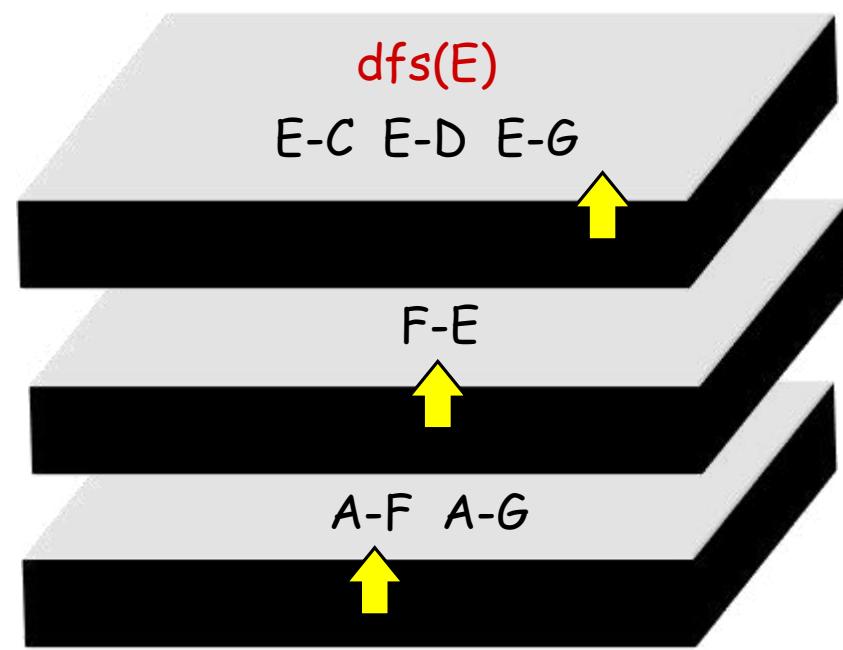
Function call stack:



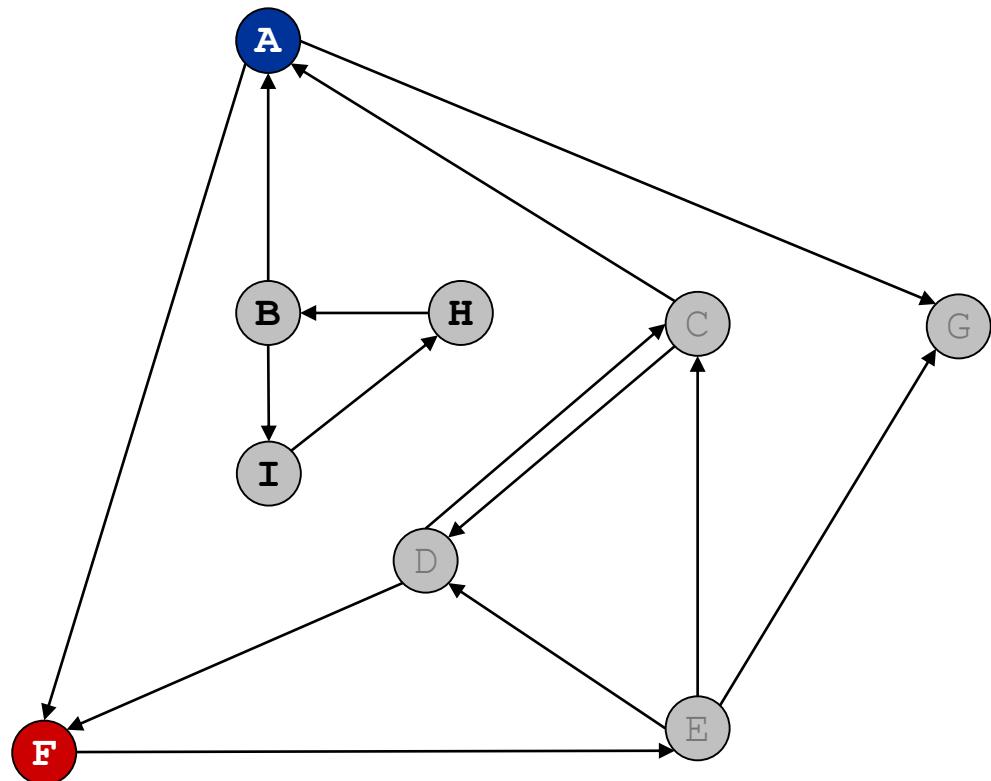
Directed Depth First Search



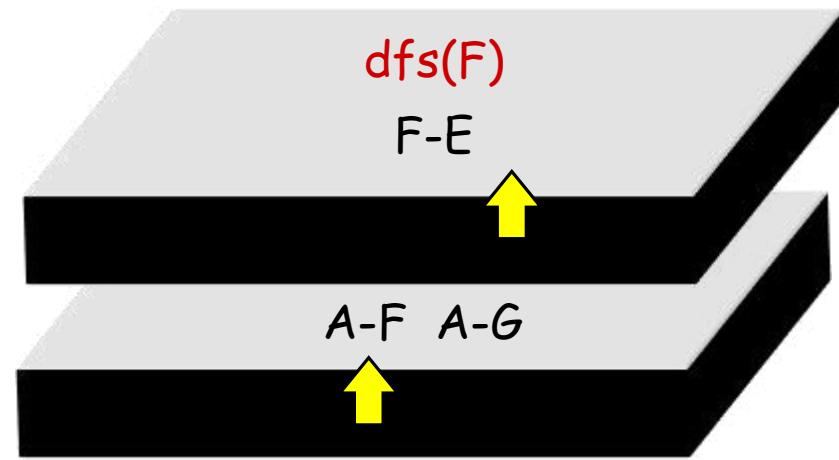
Function call stack:



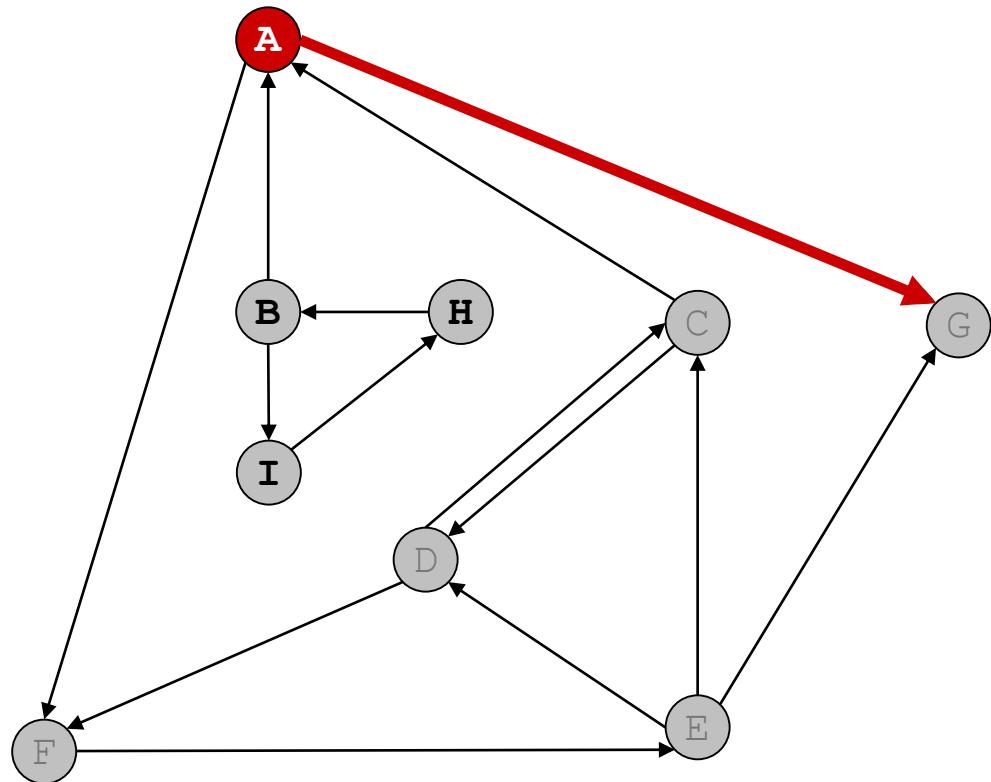
Directed Depth First Search



Function call stack:



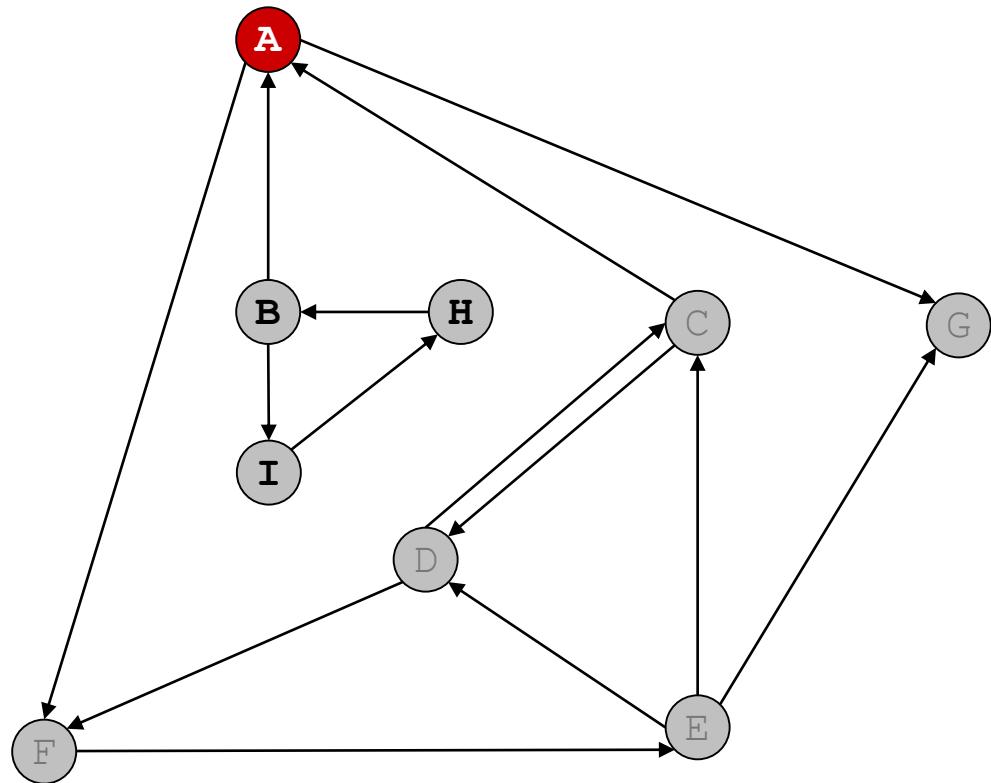
Directed Depth First Search



Function call stack:

dfs(A)
A-F A-G

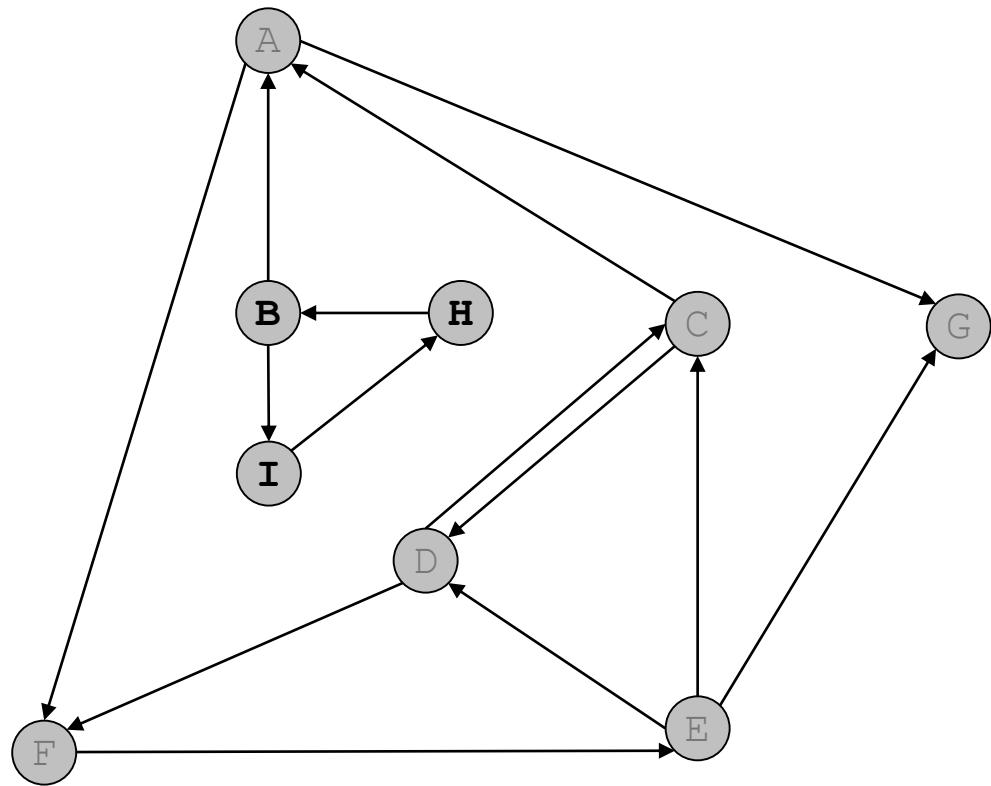
Directed Depth First Search



Function call stack:

$\text{dfs}(A)$
A-F A-G

Directed Depth First Search



Nodes reachable from A: A, C, D, E, F, G

Graph Traversal Techniques (cont'd)

DFS-Algo(G,s)

for all v **in** $V[G]$ **do**

visited[v] := **false**

end for

S := **EmptyStack**

visited[s] := **true** //print s

Push(S,s)

while not **Empty**(S) **do**

u := **Pop**(S)

if there is at least one unvisited vertex in $\text{Adj}[u]$ **then**

Pick w **to be any unvisited vertex in** $\text{Adj}[u]$

Push(S,u)

visited[w] := **true** //print w

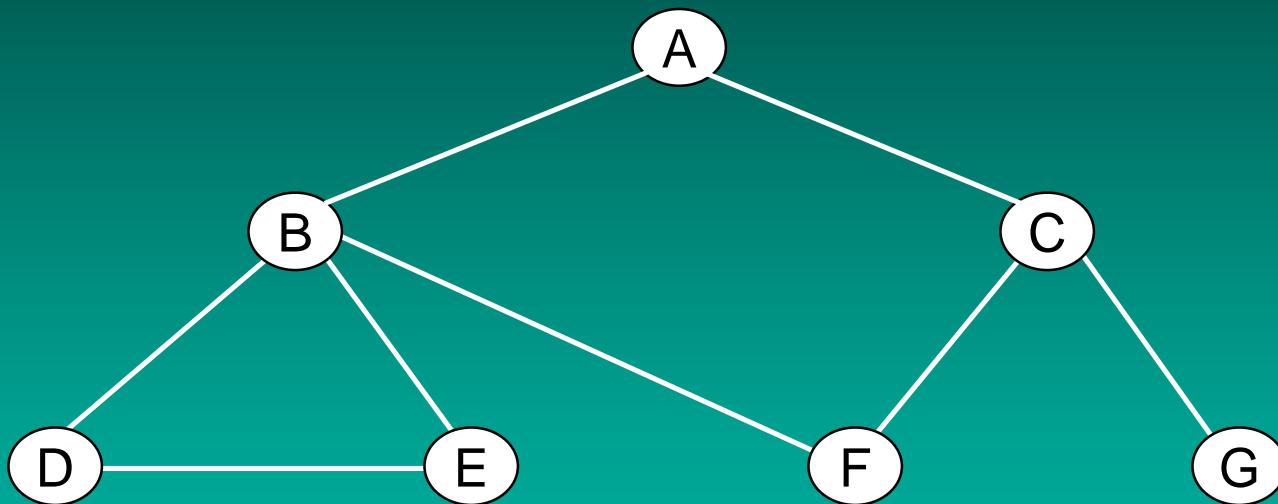
Push(S,w)

end if

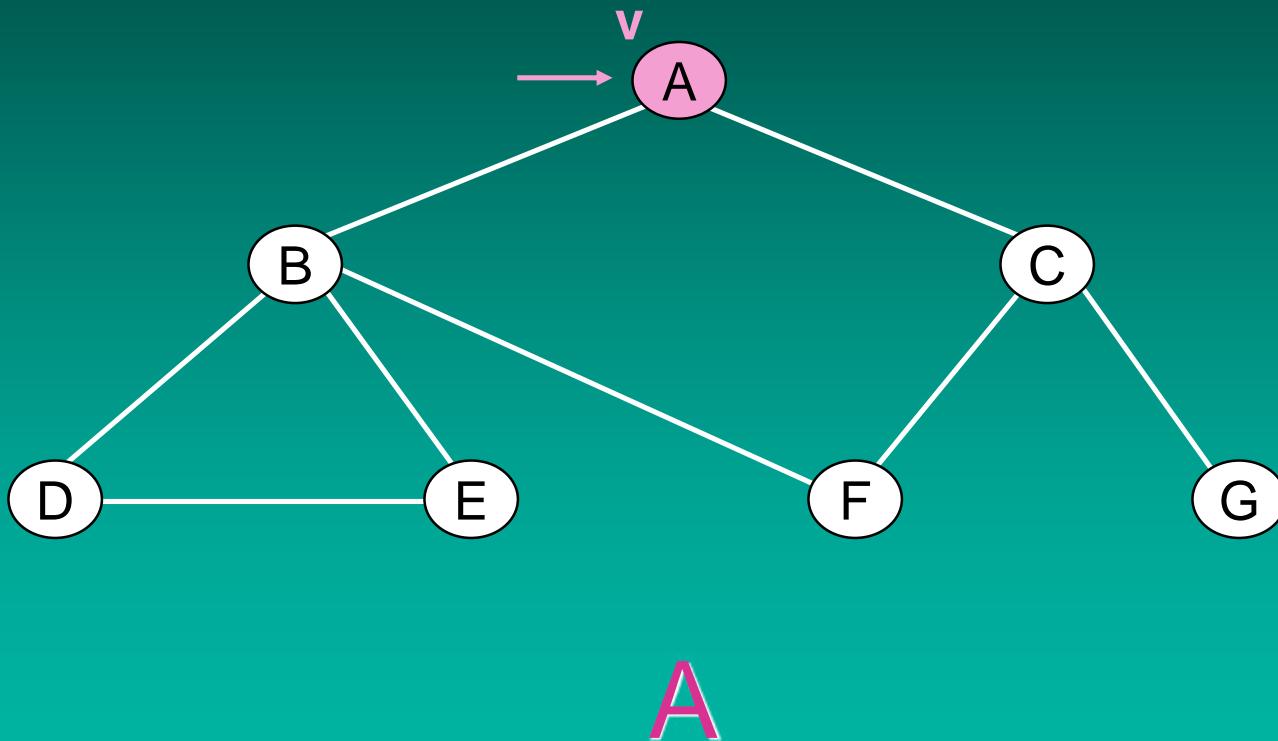
end while

- **Performance:**
 - DFS-Visit called once for each Edge: $\Theta(|E|)$
 - Initialization is $\Theta(|V|)$
 - Total: $\Theta(|V| + |E|)$
-
- **Uses:**
 - Topological sort: sort vertices so that for all edges $(u, v) \in G, u < v$
 - Useful for ordering events, for example

Depth-First Search

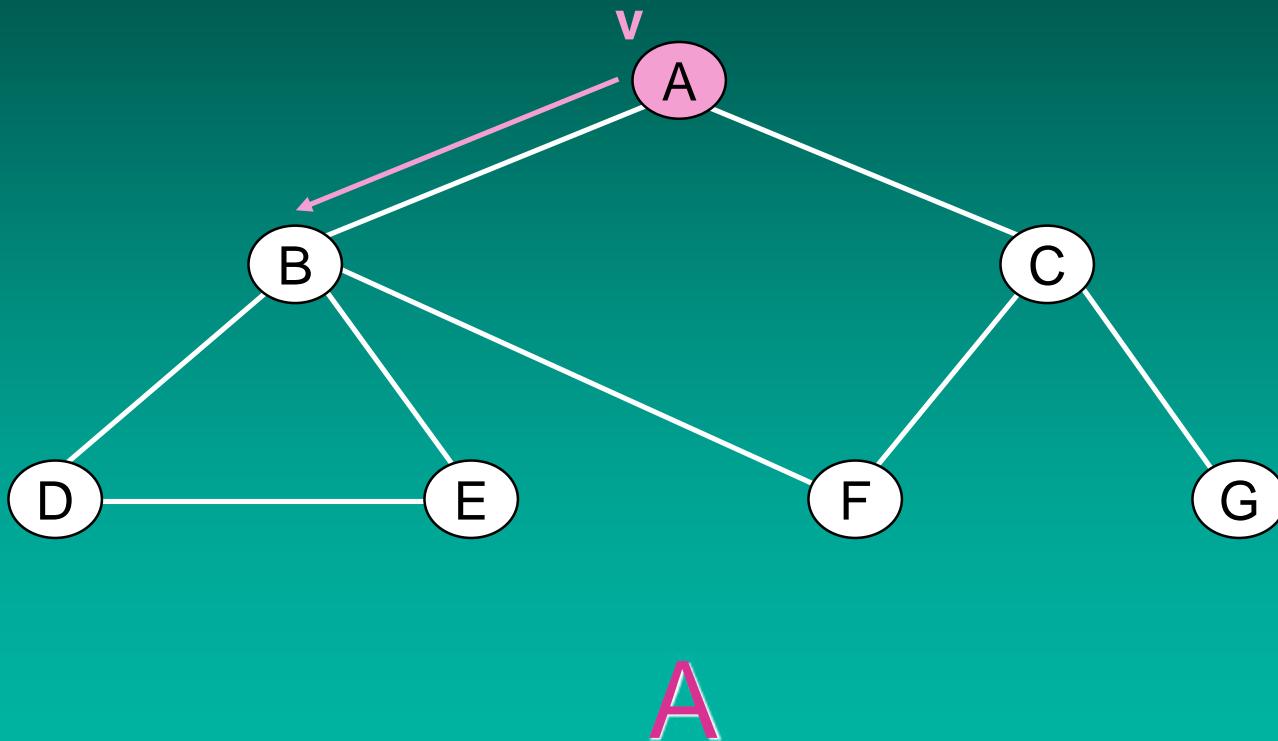


Depth-First Search

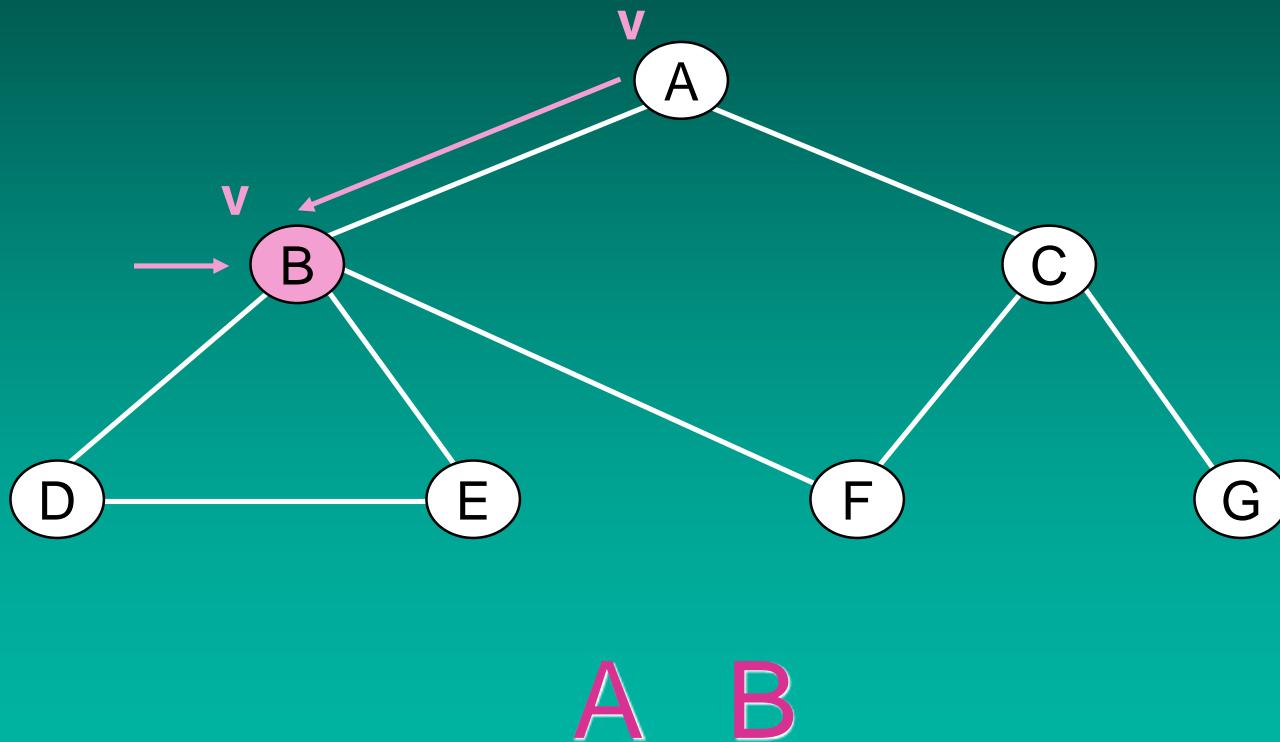


A

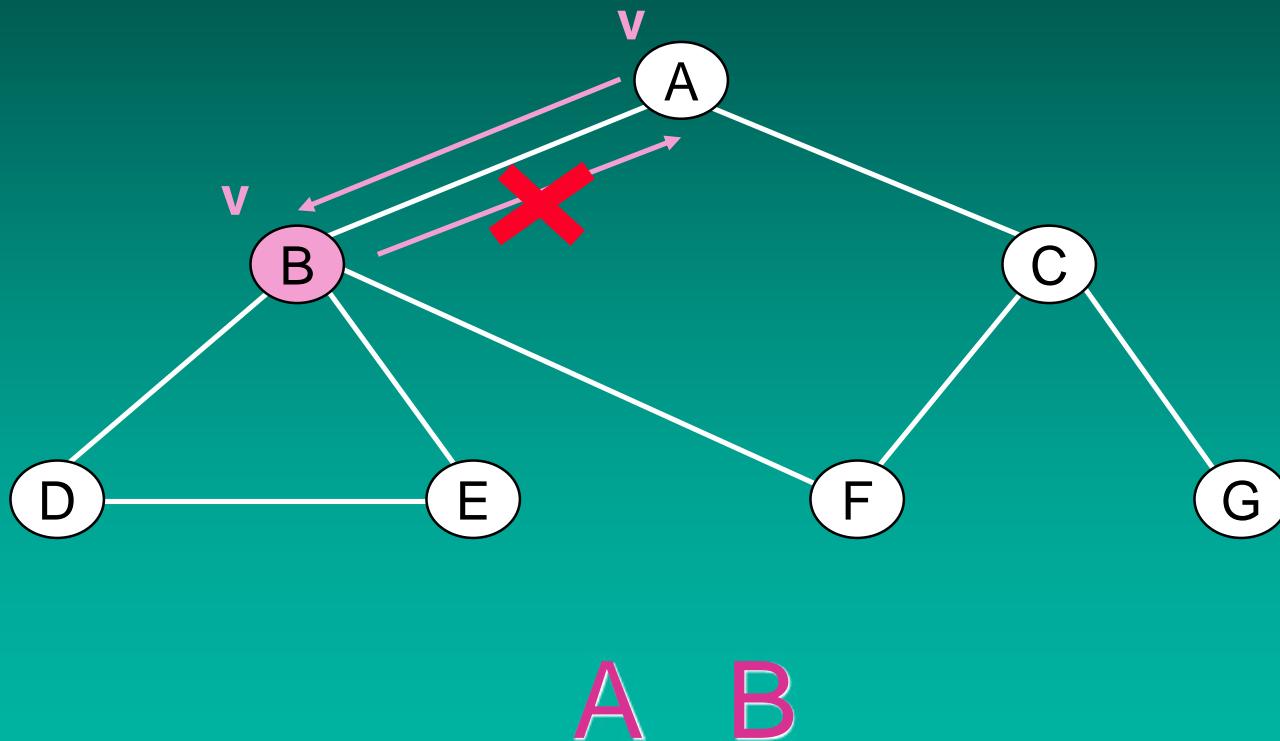
Depth-First Search



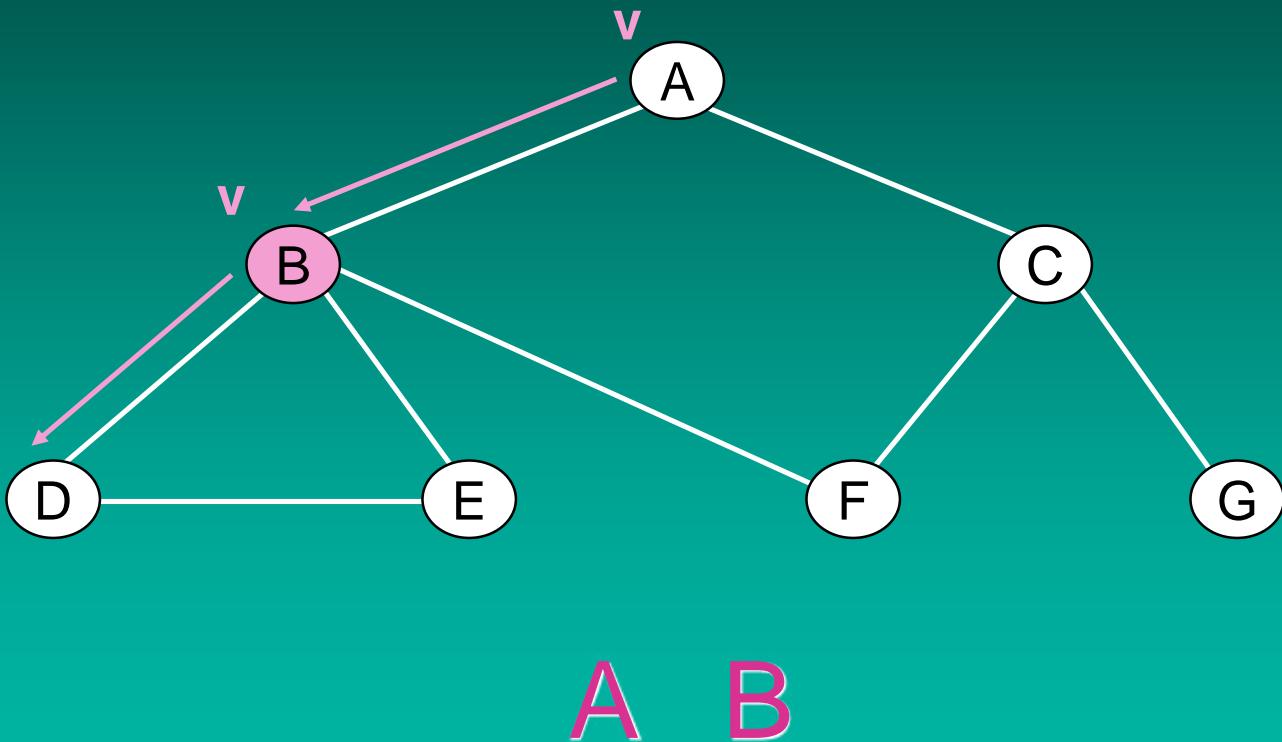
Depth-First Search



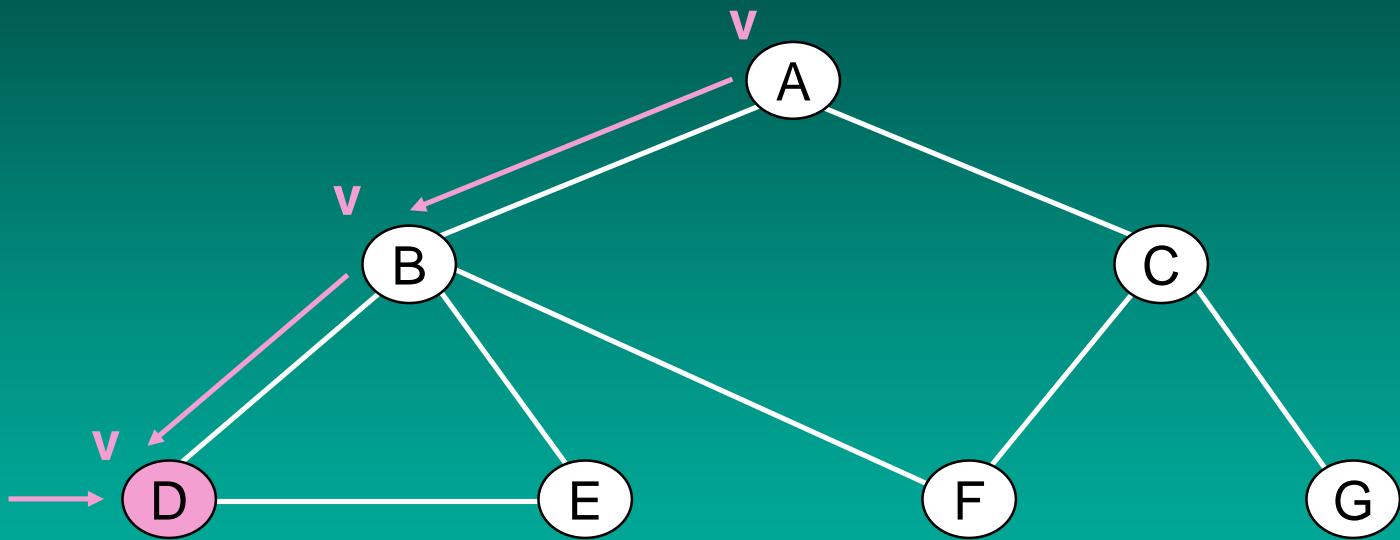
Depth-First Search



Depth-First Search

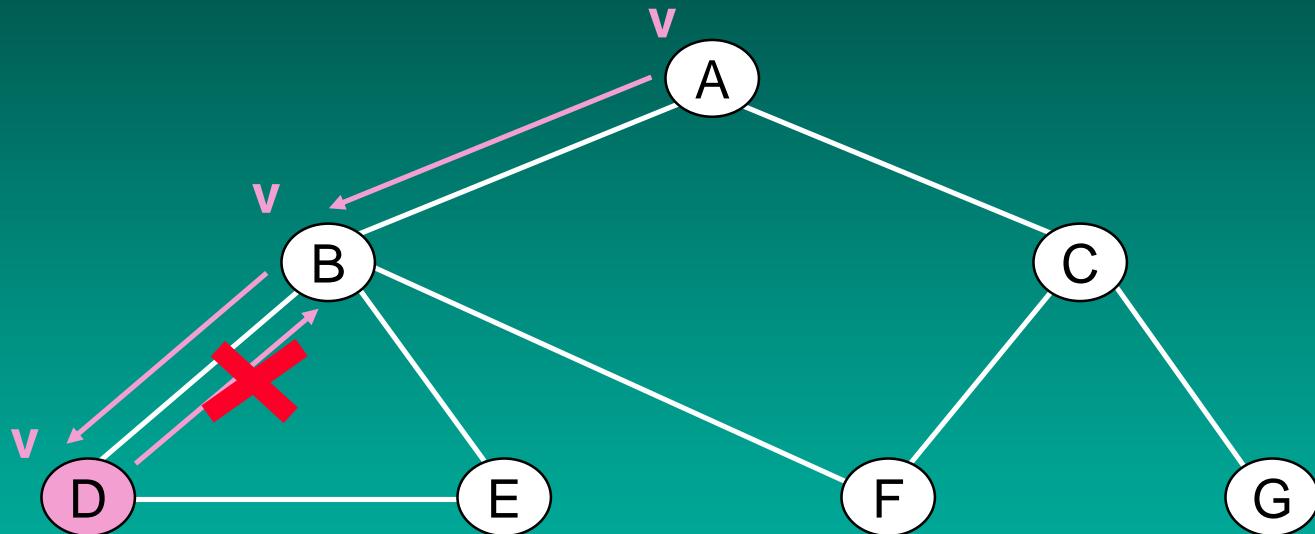


Depth-First Search



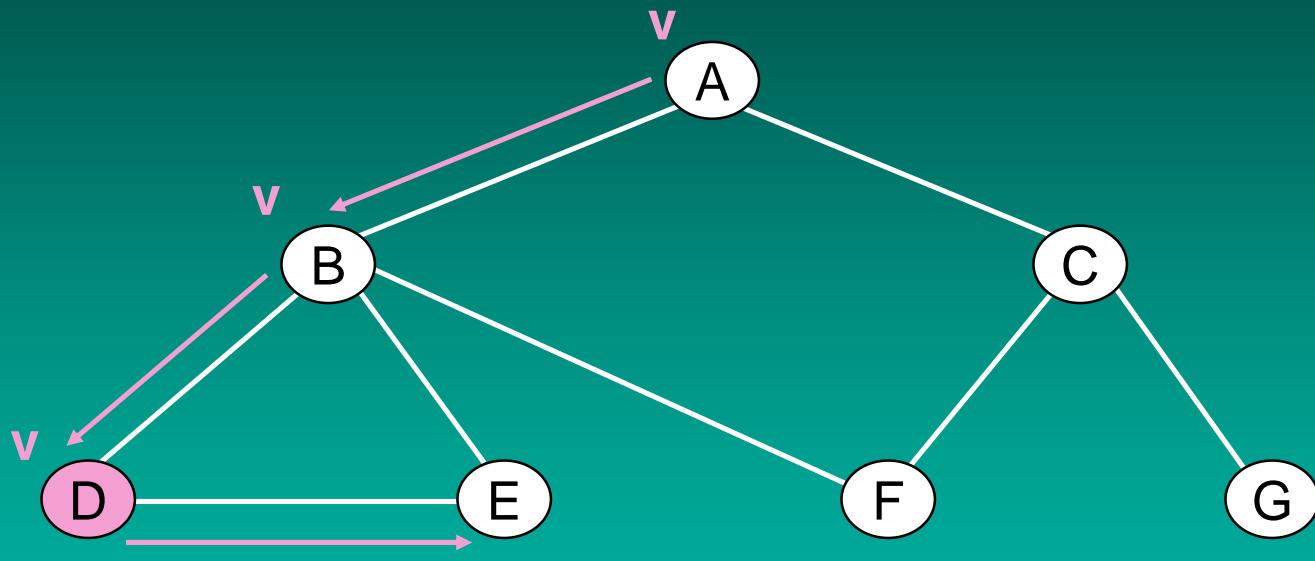
A B D

Depth-First Search



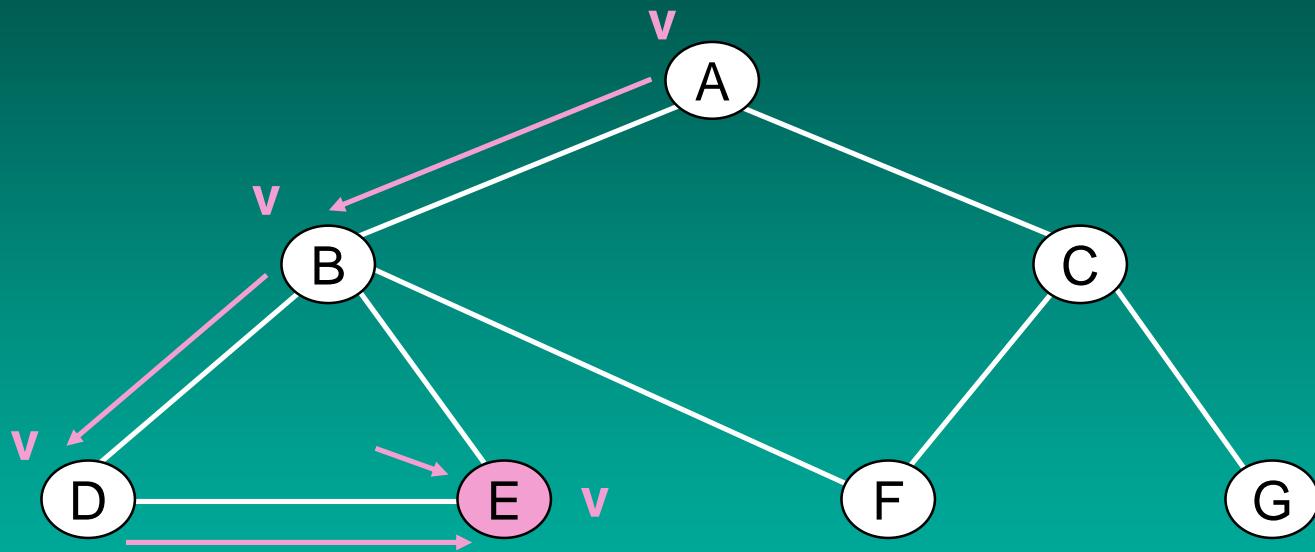
A B D

Depth-First Search



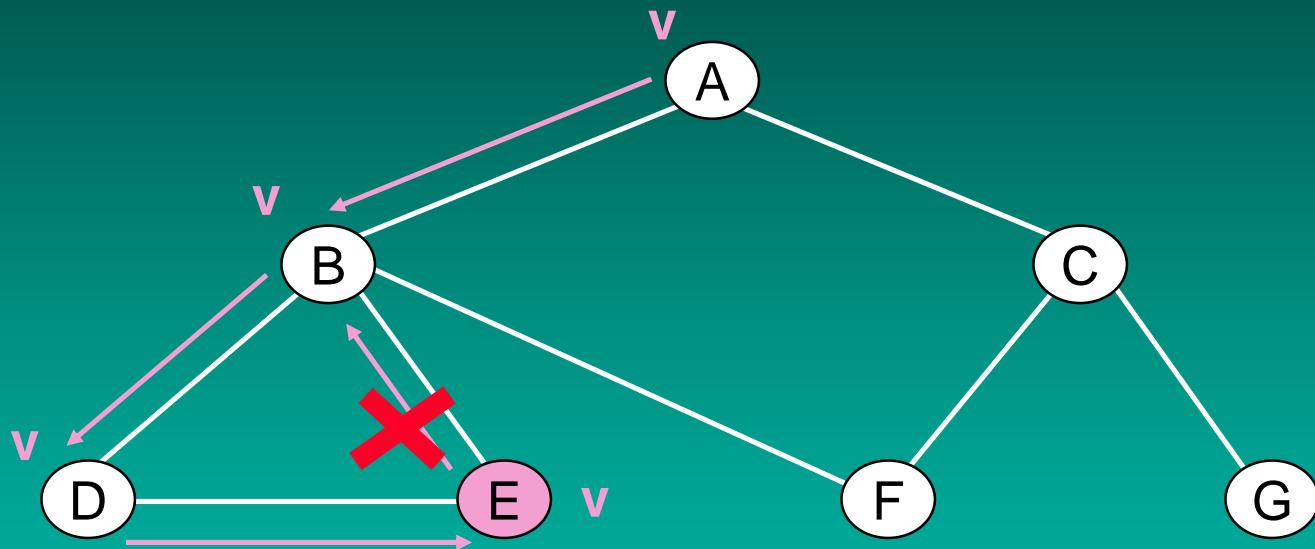
A B D

Depth-First Search



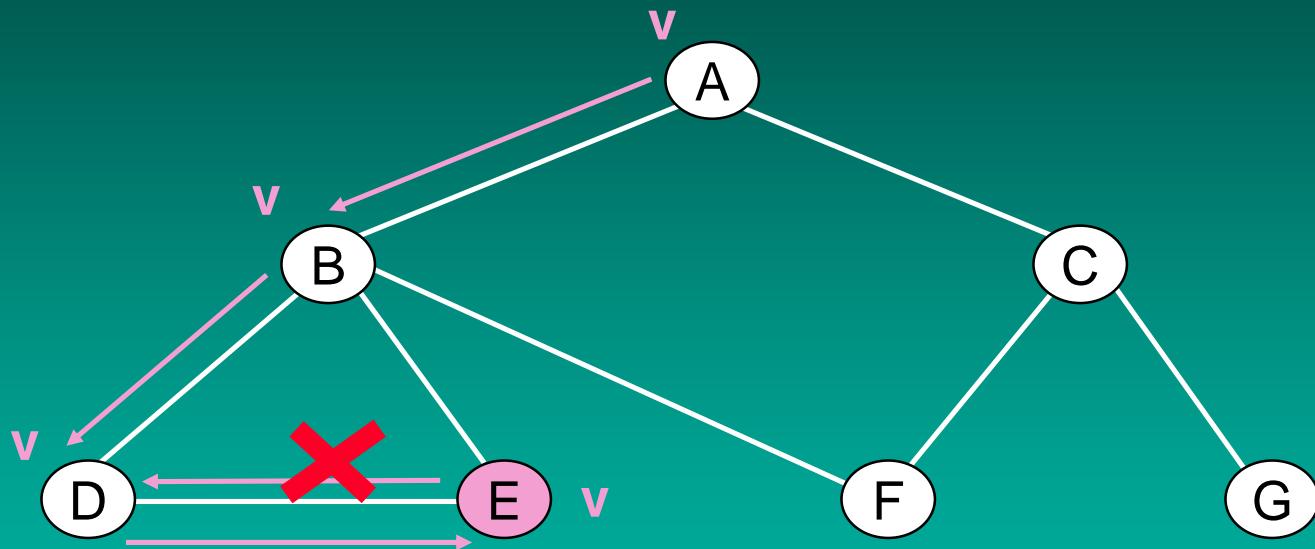
A B D E

Depth-First Search



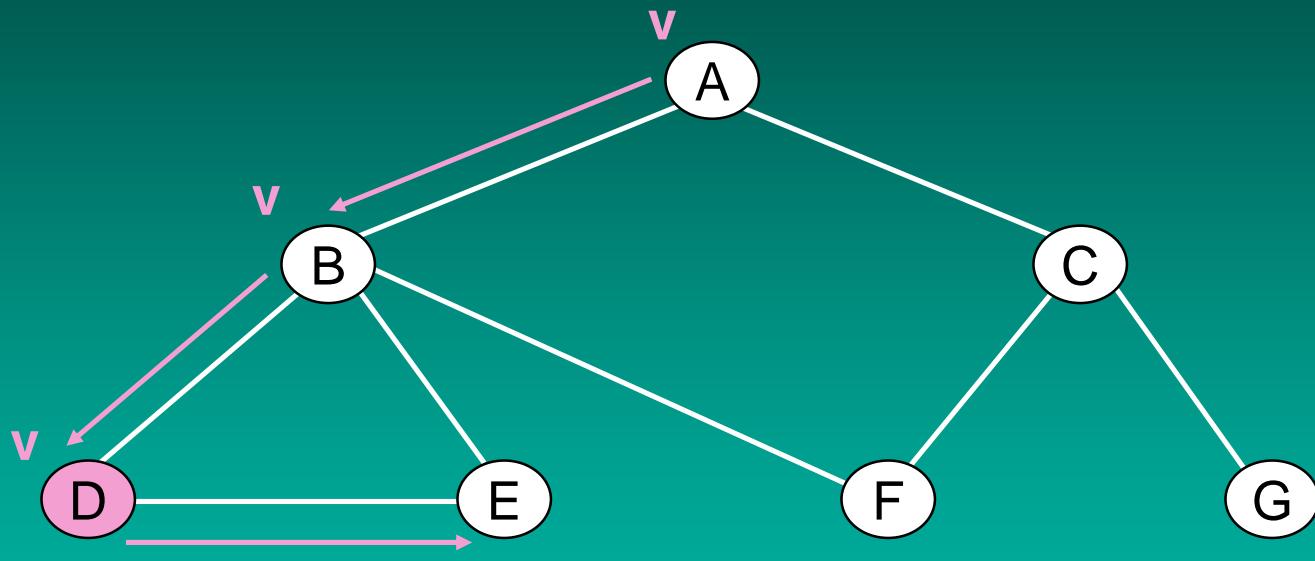
A B D E

Depth-First Search



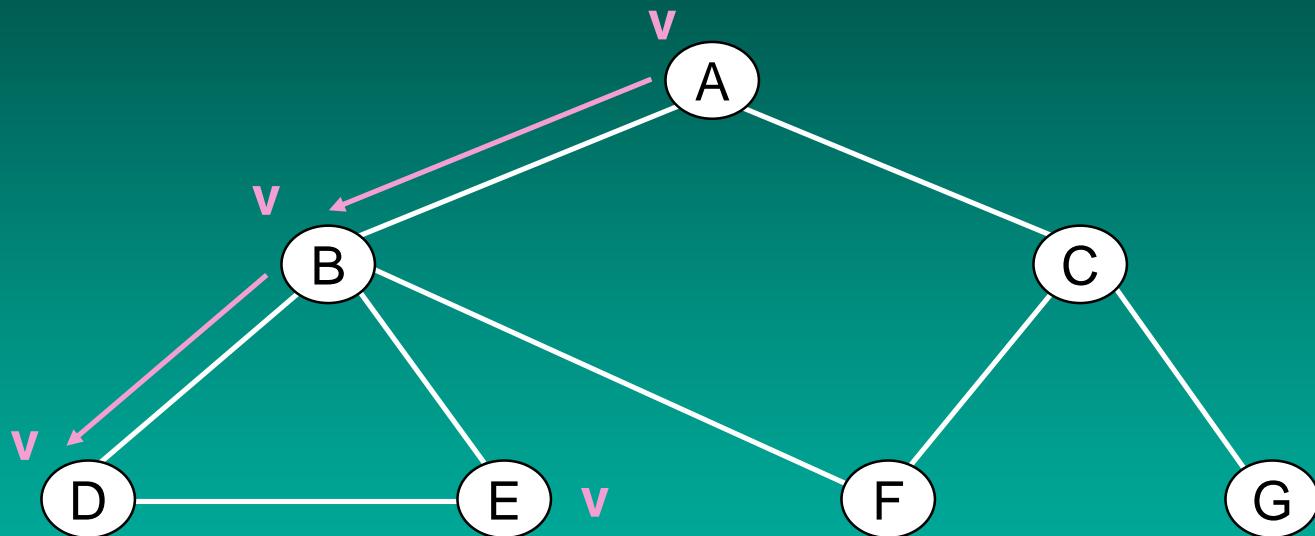
A B D E

Depth-First Search



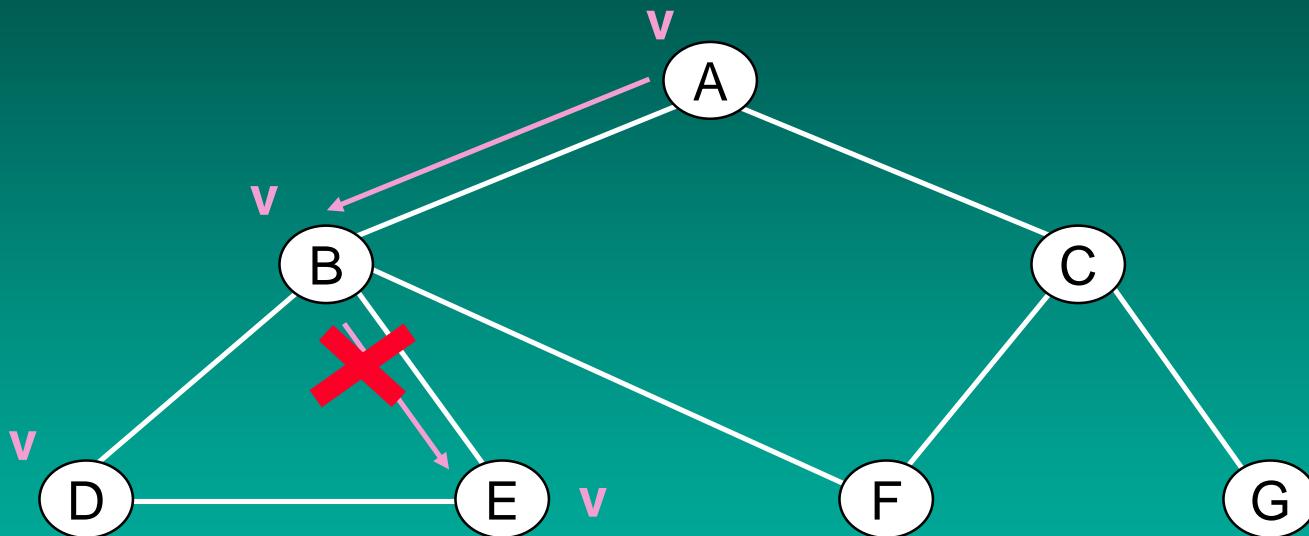
A B D E

Depth-First Search



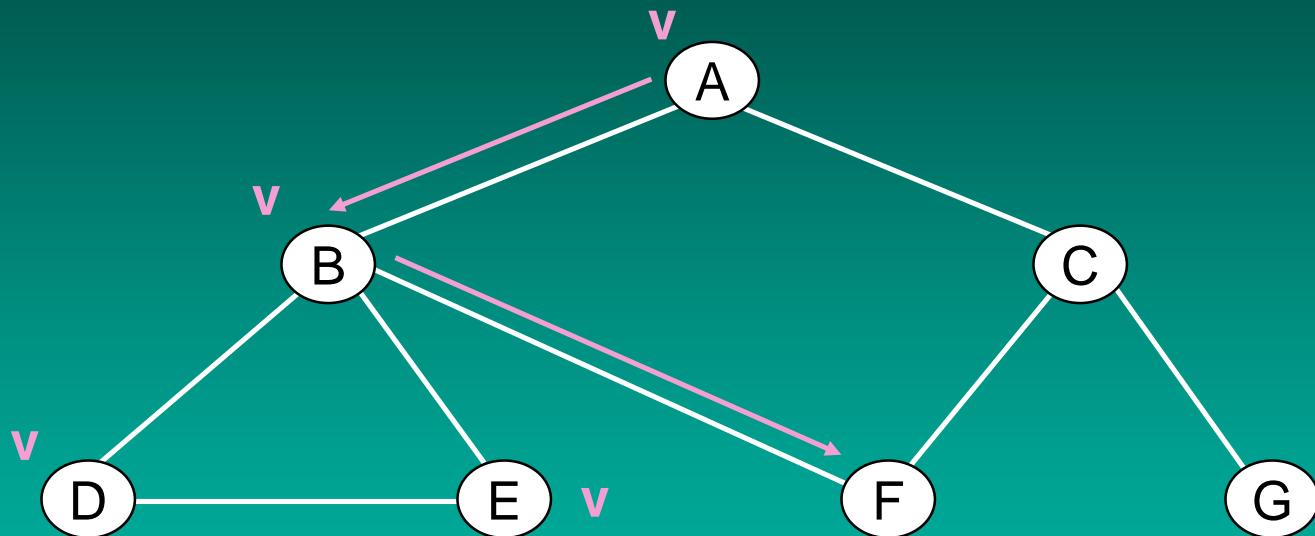
A B D E

Depth-First Search



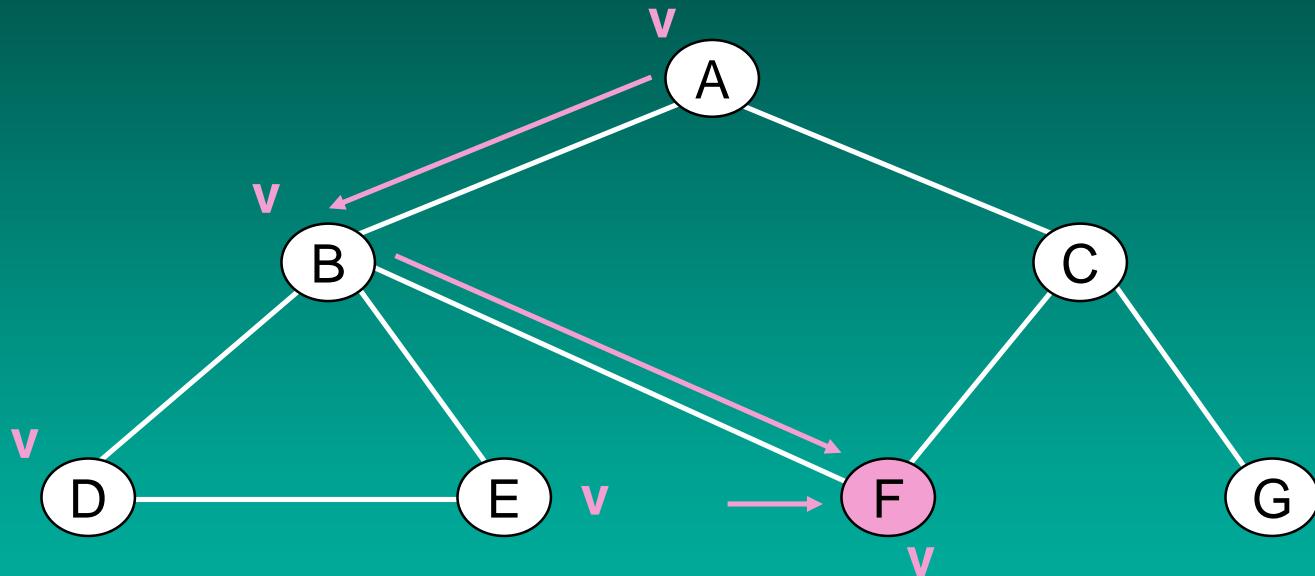
A B D E

Depth-First Search



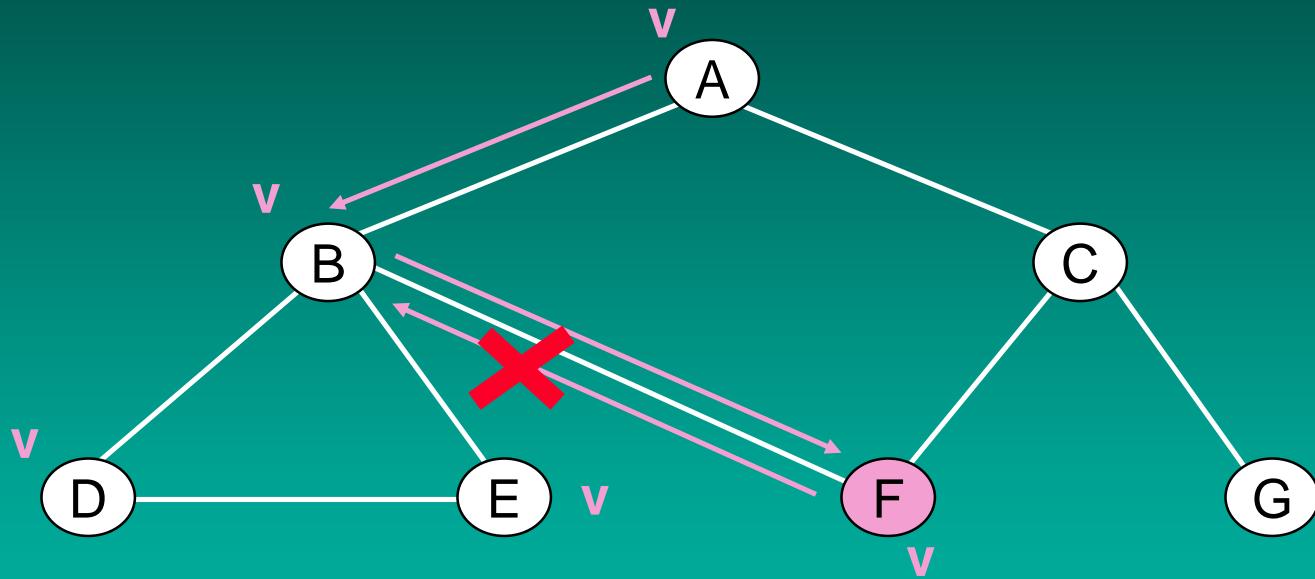
A B D E

Depth-First Search



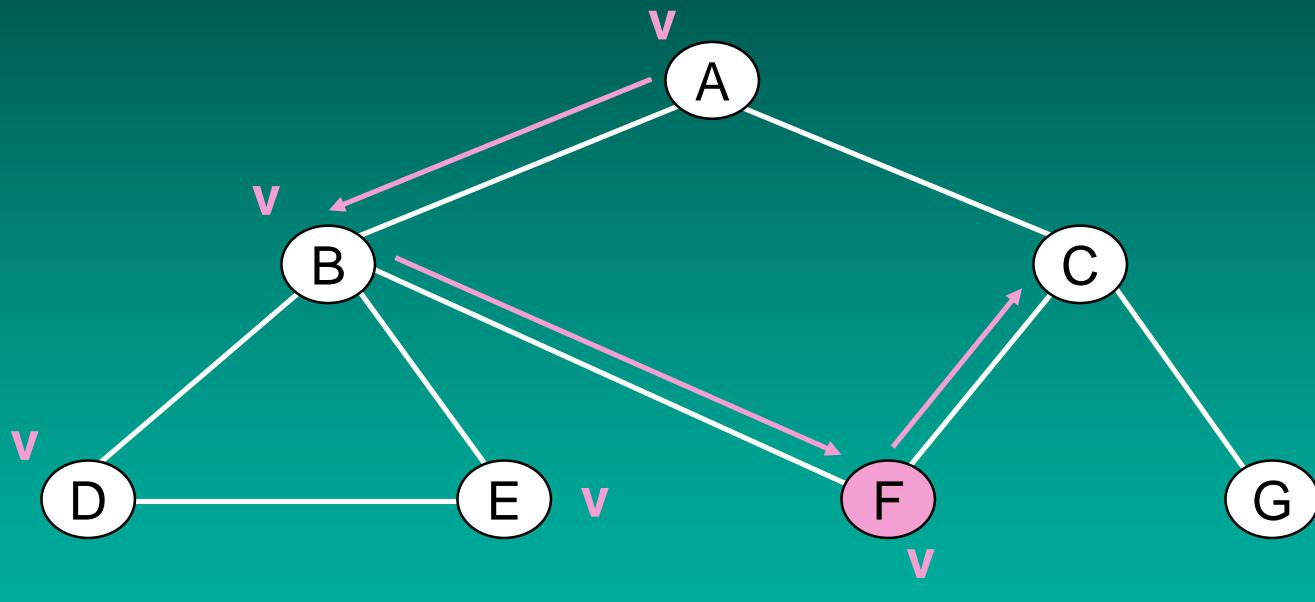
A B D E F

Depth-First Search



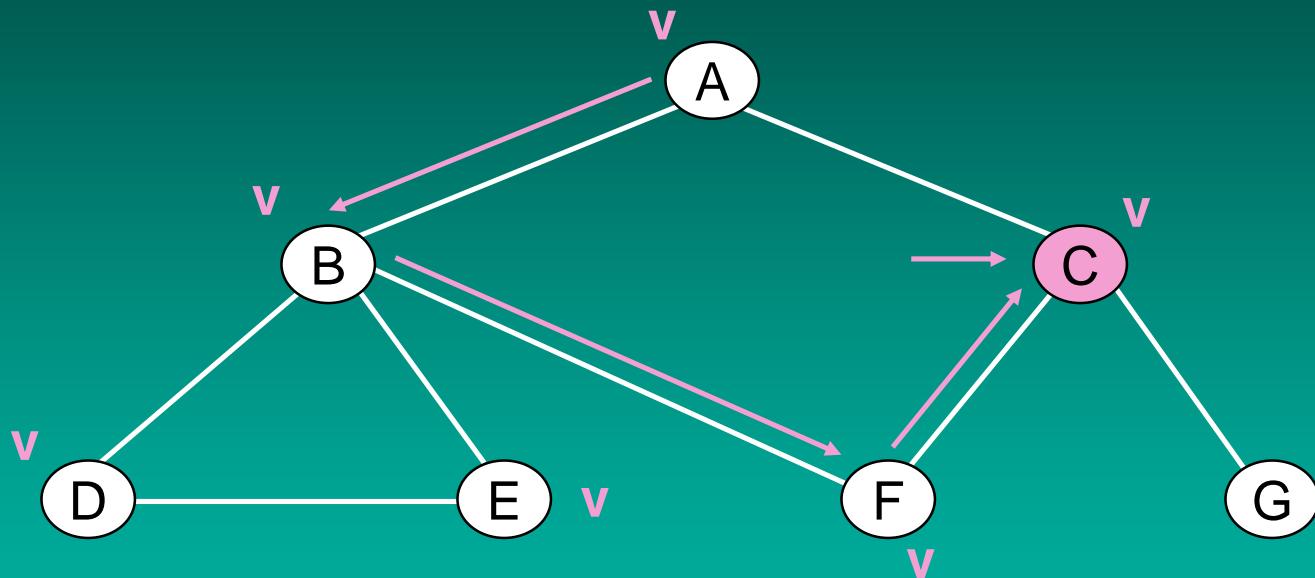
A B D E F

Depth-First Search



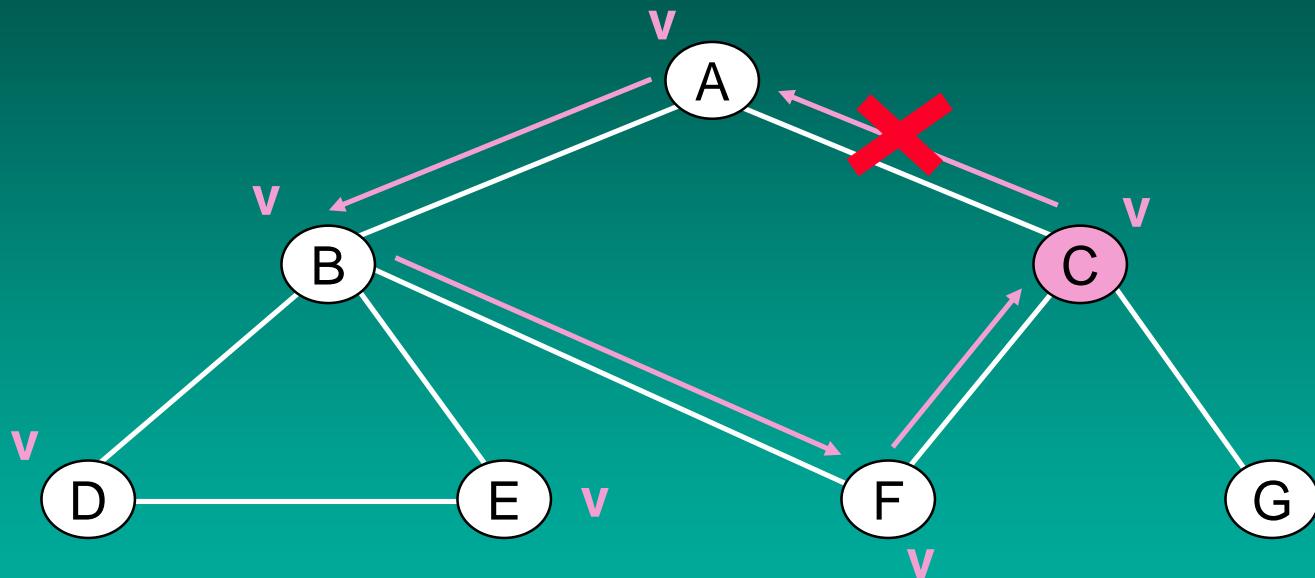
A B D E F

Depth-First Search



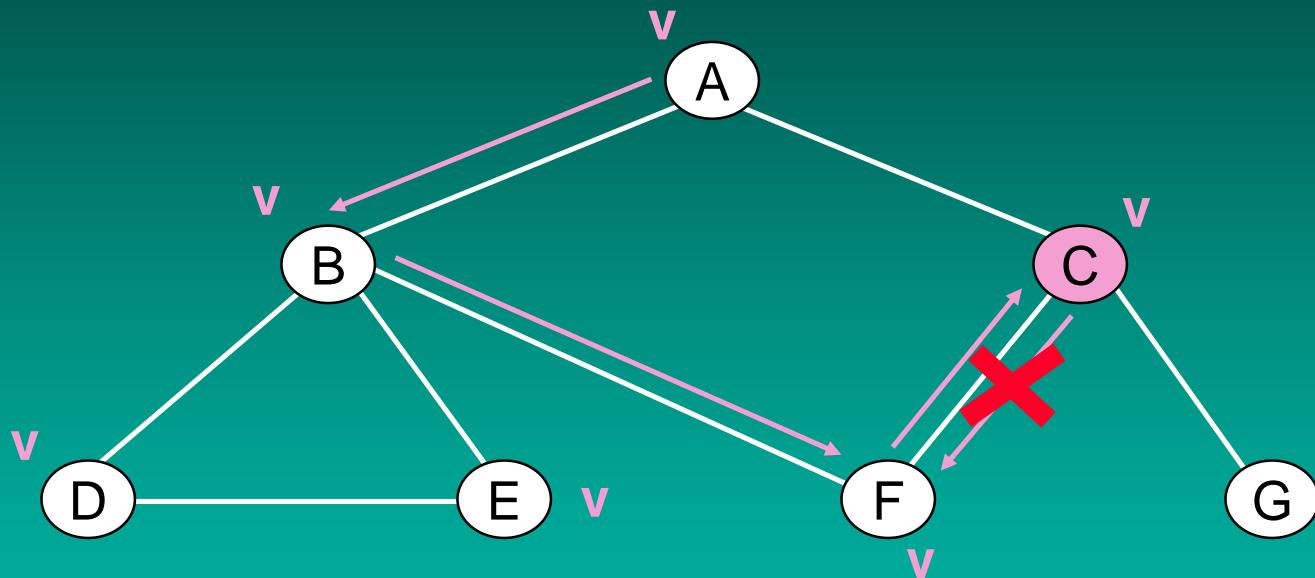
A B D E F C

Depth-First Search



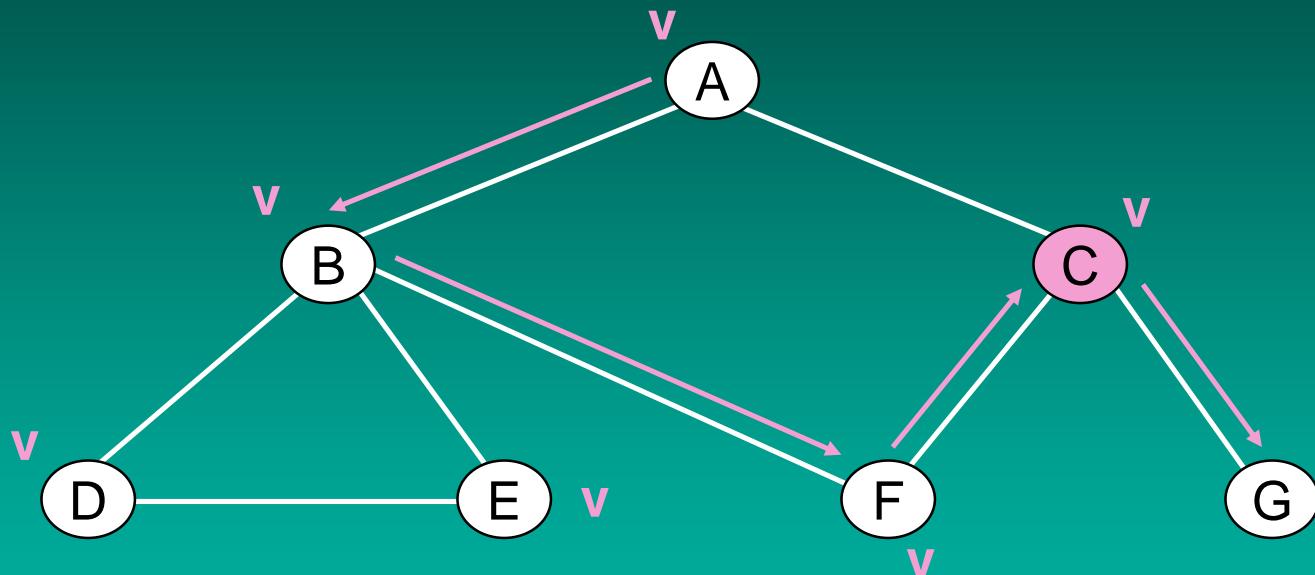
A B D E F C

Depth-First Search



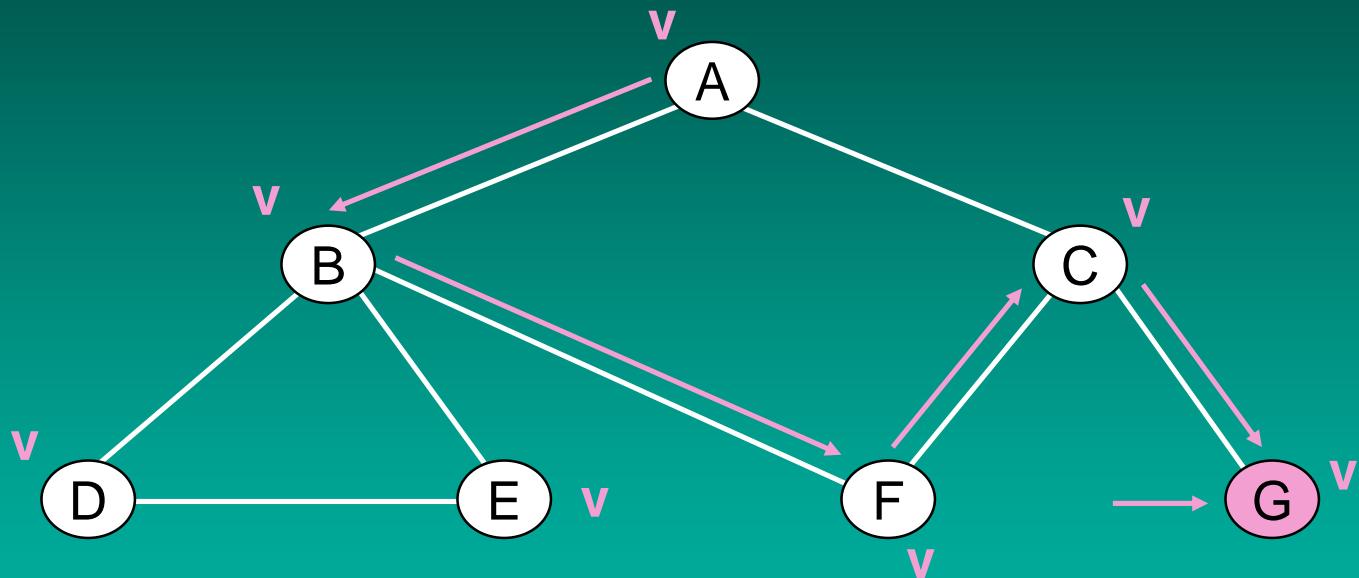
A B D E F C

Depth-First Search



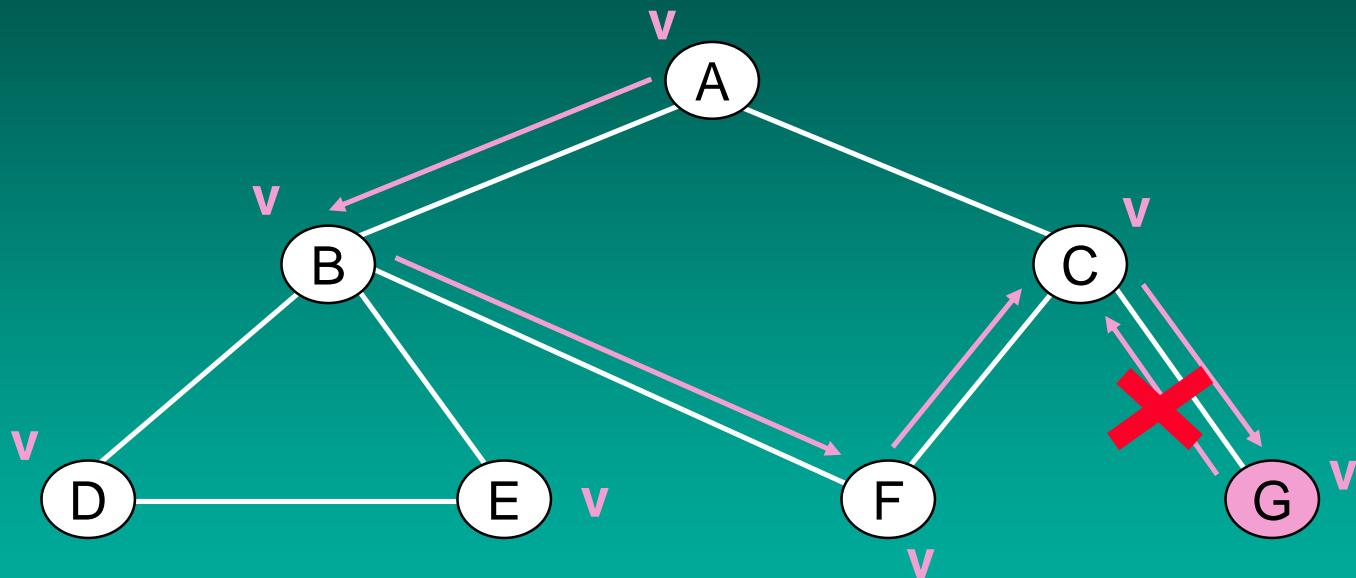
A B D E F C

Depth-First Search



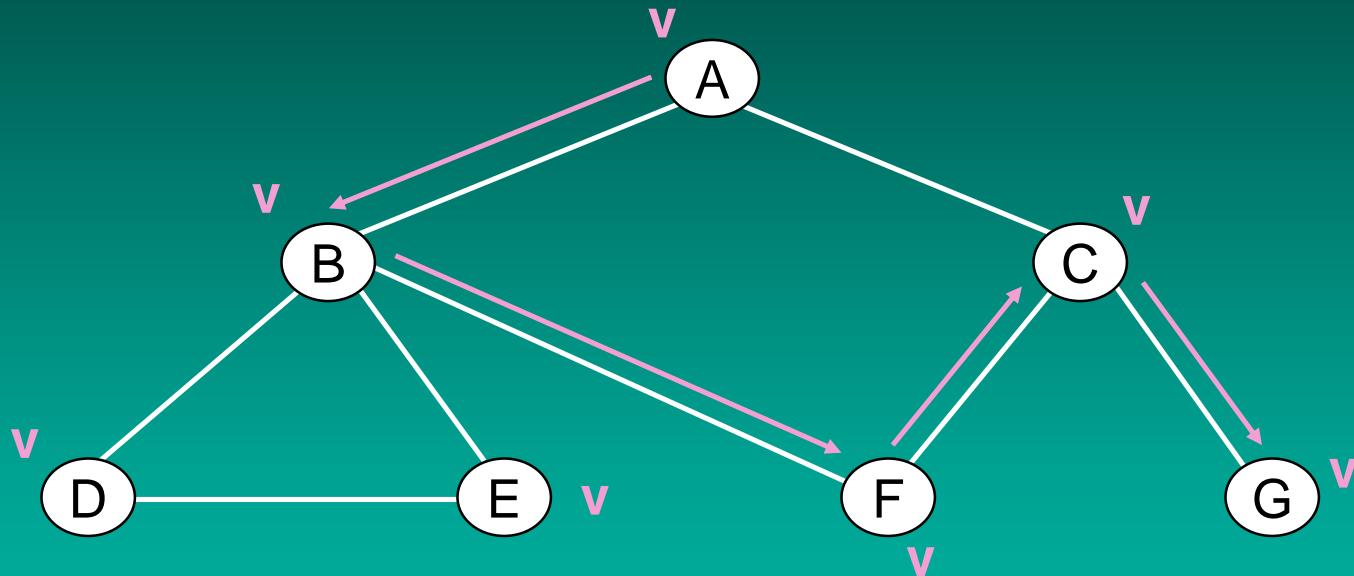
A B D E F C G

Depth-First Search



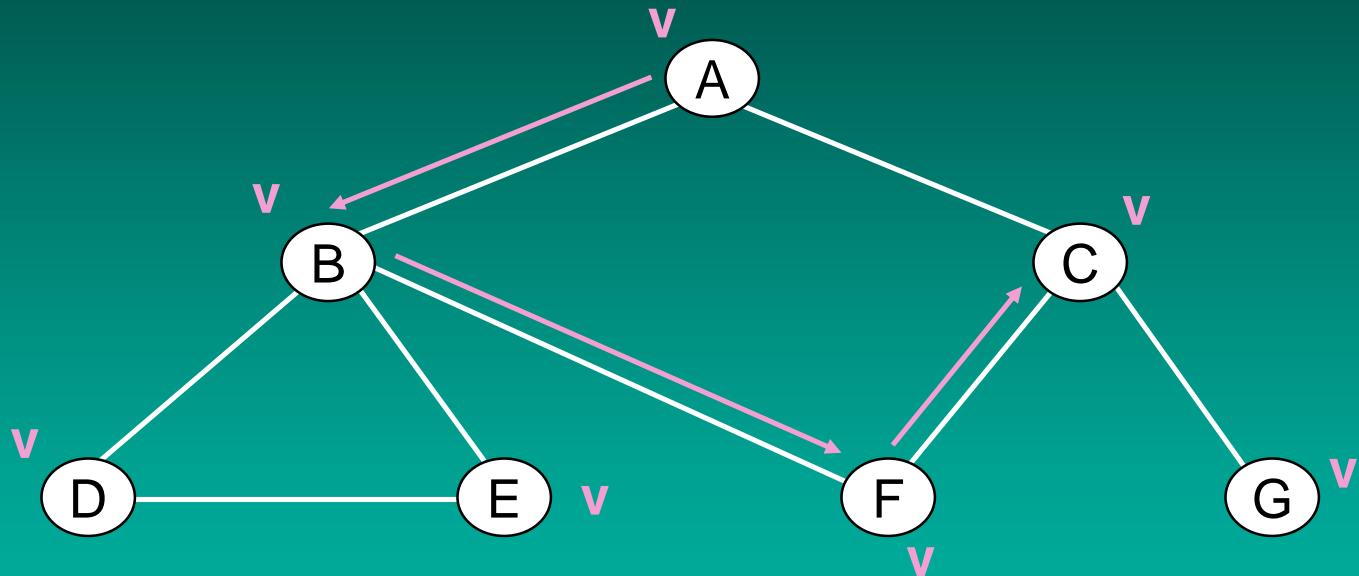
A B D E F C G

Depth-First Search



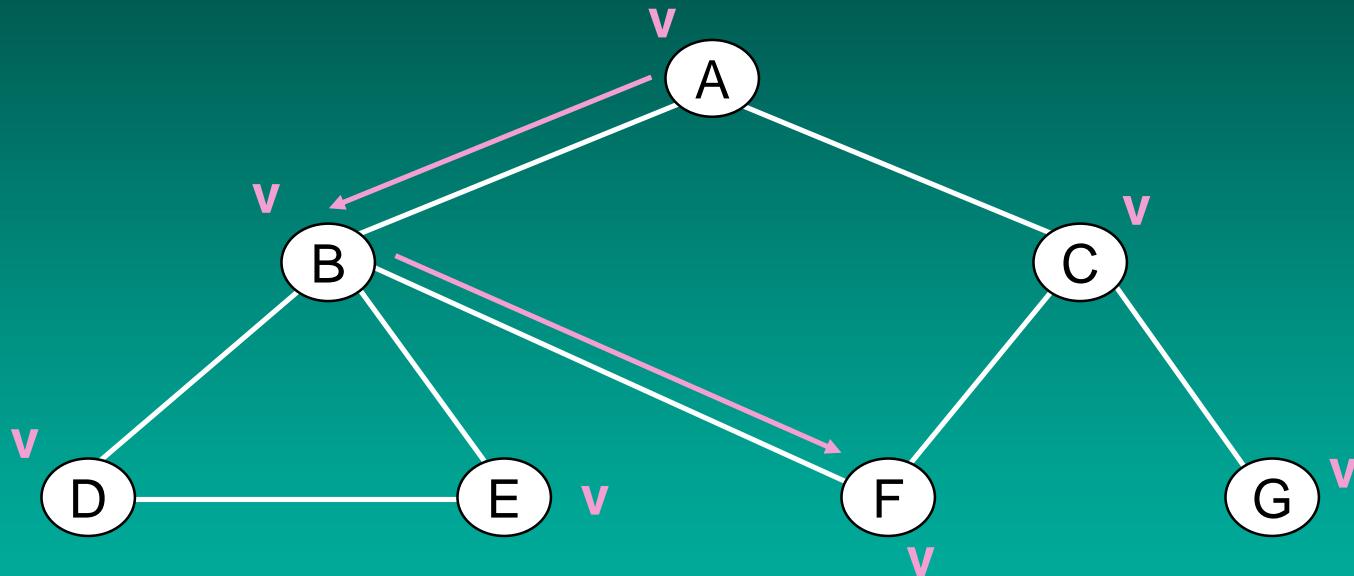
A B D E F C G

Depth-First Search



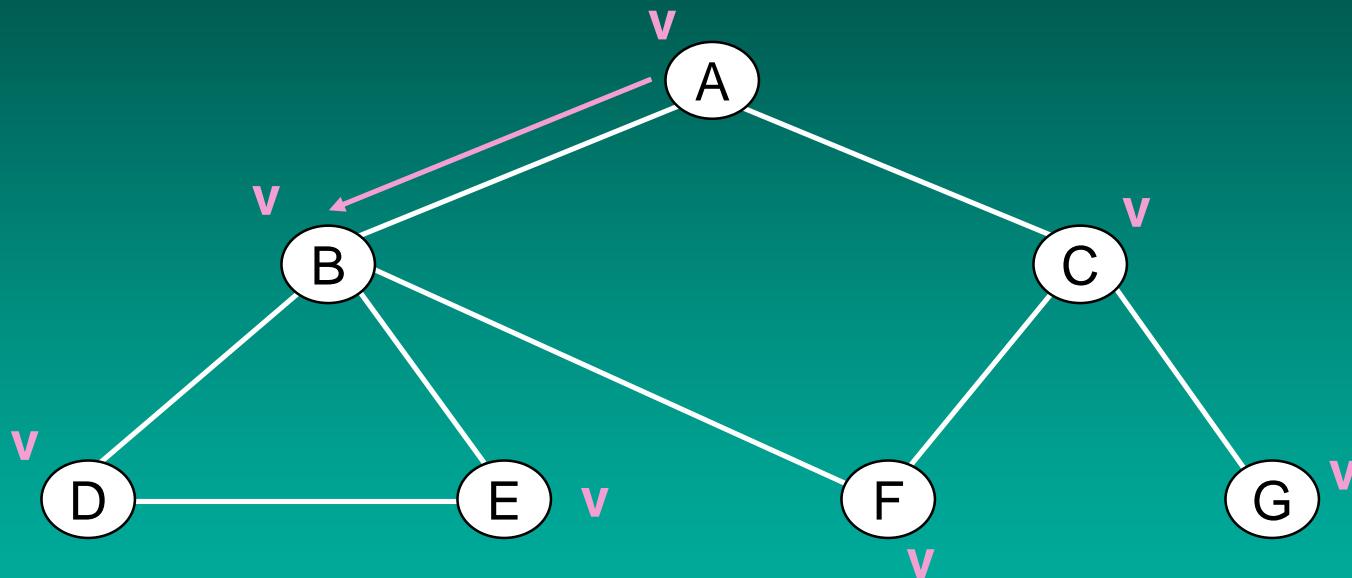
A B D E F C G

Depth-First Search



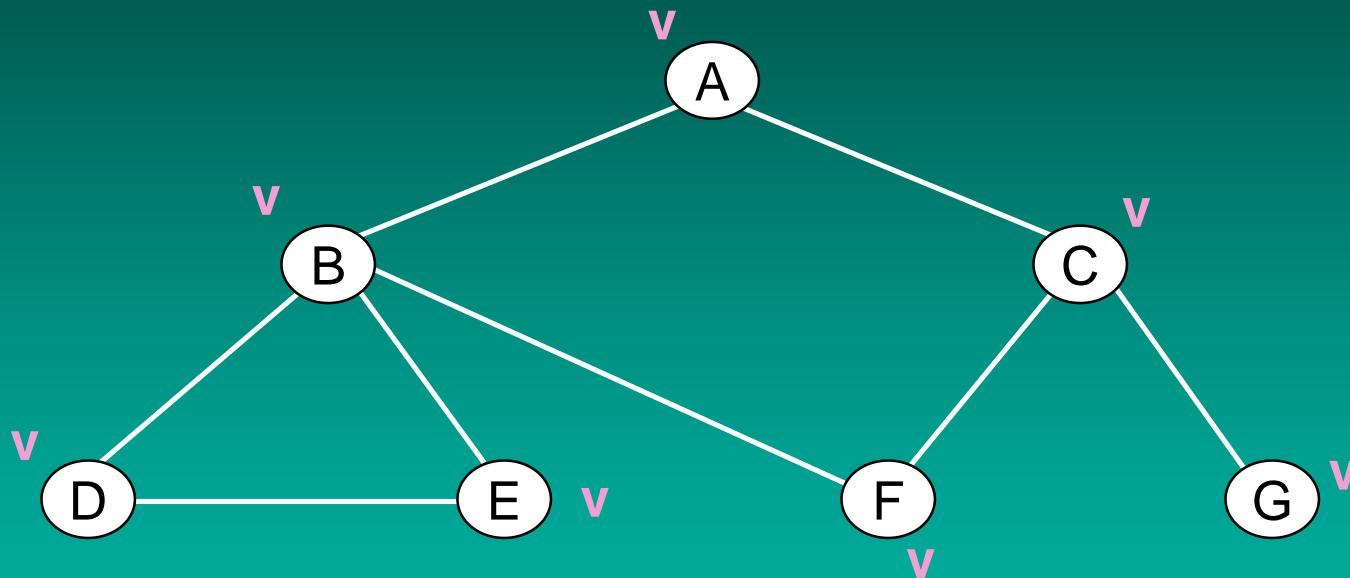
A B D E F C G

Depth-First Search



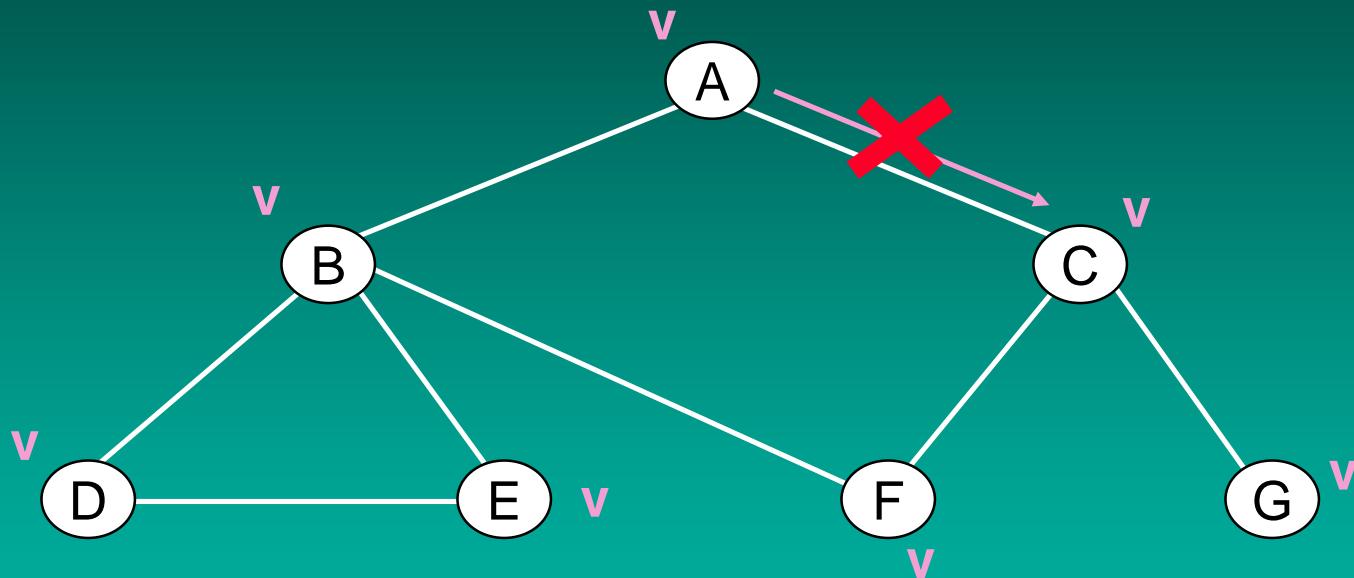
A B D E F C G

Depth-First Search



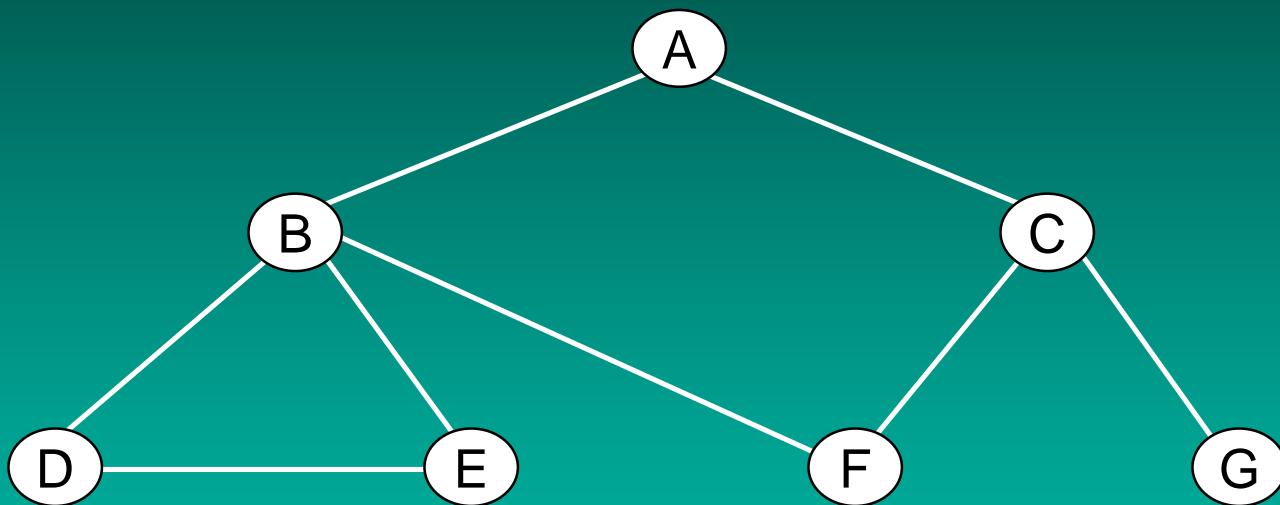
A B D E F C G

Depth-First Search



A B D E F C G

Depth-First Search

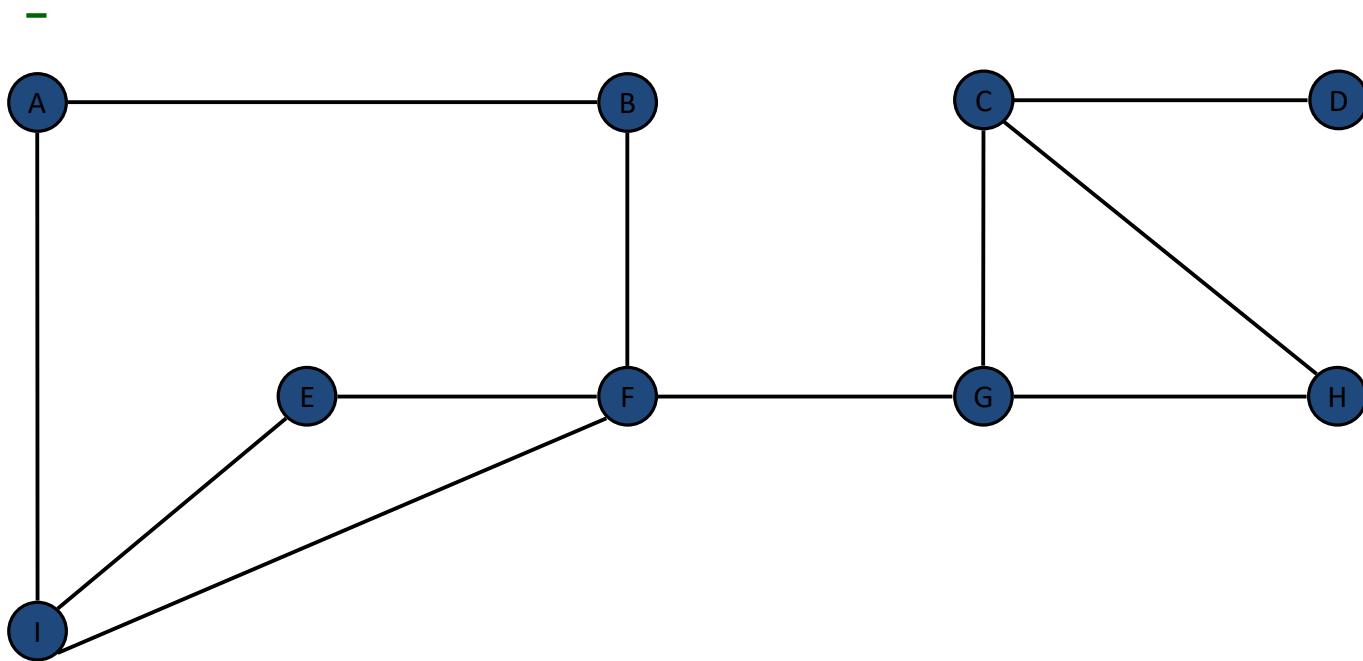


A B D E F C G

Graph Traversal Techniques (cont'd)

- **Breadth-First-Traversal/ Breadth-First-Search (BFS)**
 - **BFS visits the neighbour vertices before visiting the child vertices**
 - **A queue is used** in the implementation process.
 - **Used to find the shortest path from one vertex to another**

Breadth First Search



front

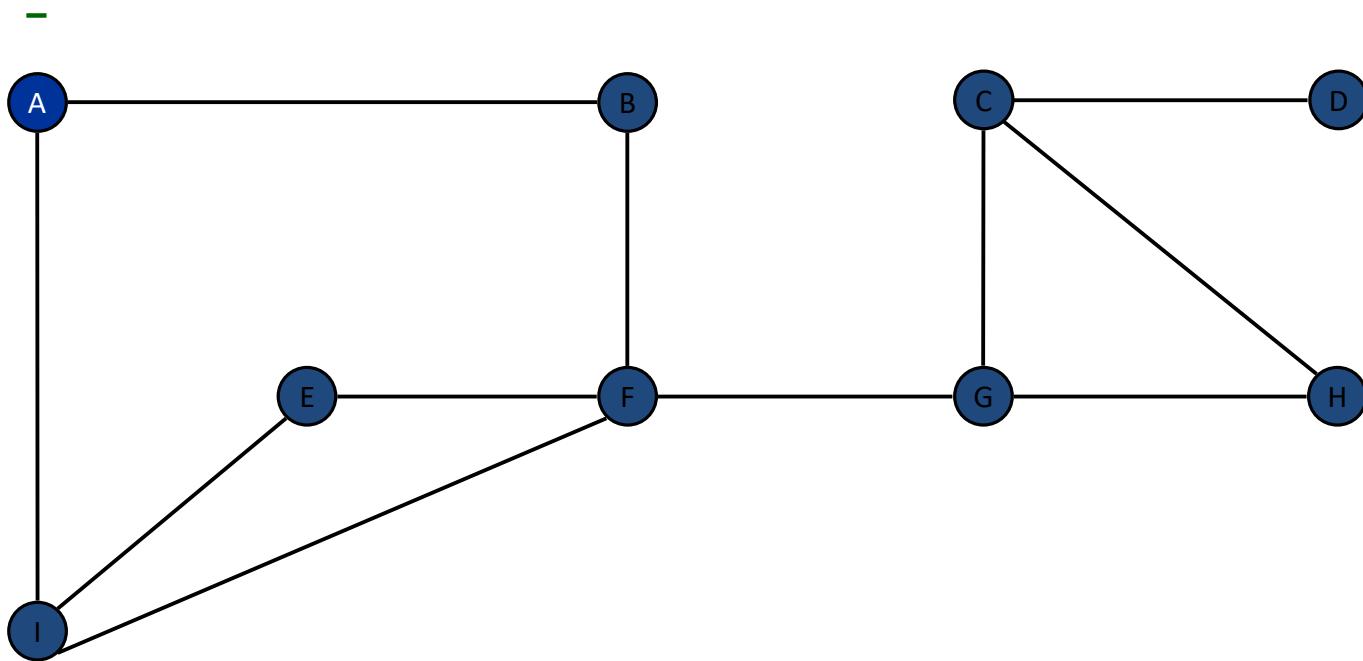
FIFO Queue

BFS ALGORITHM

1. Enqueue The starting Vertex
2. While (Q is not Empty)
 - i. P= Dequeue() and Print
 - ii. Enqueue all the adjacent vertices of P

Check the visited array

Breadth First Search



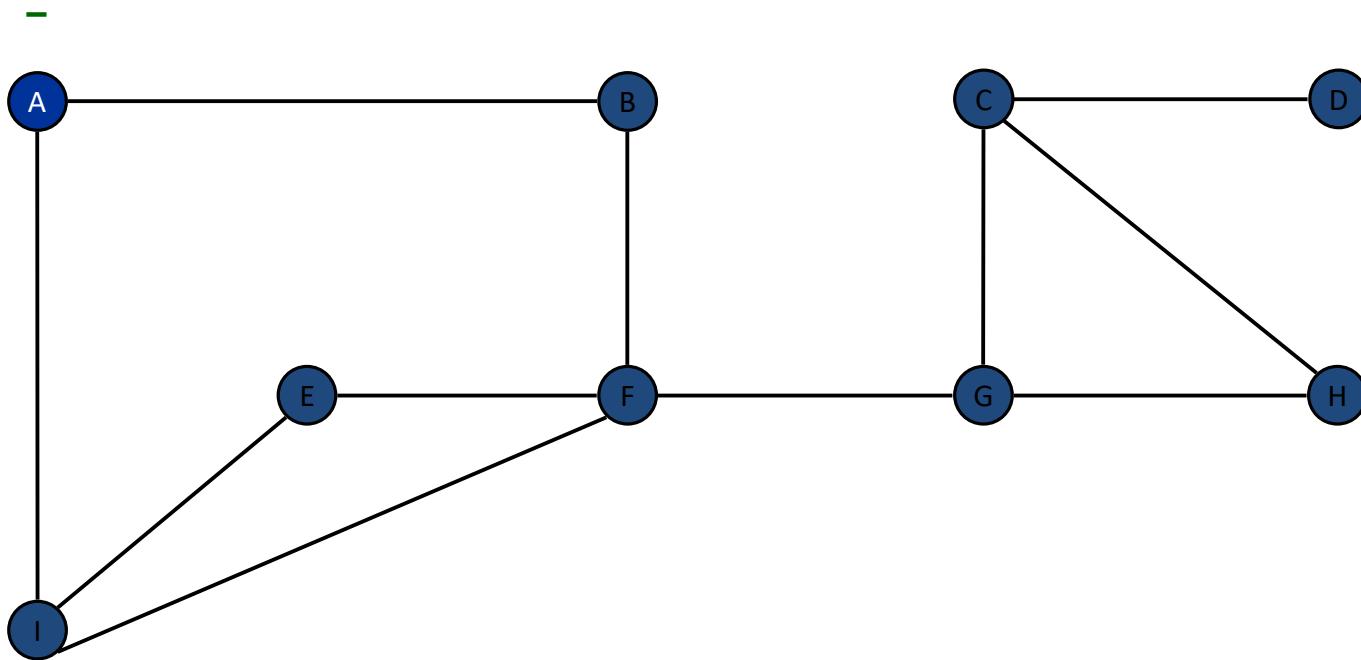
enqueue source node

front

A

FIFO Queue

Breadth First Search



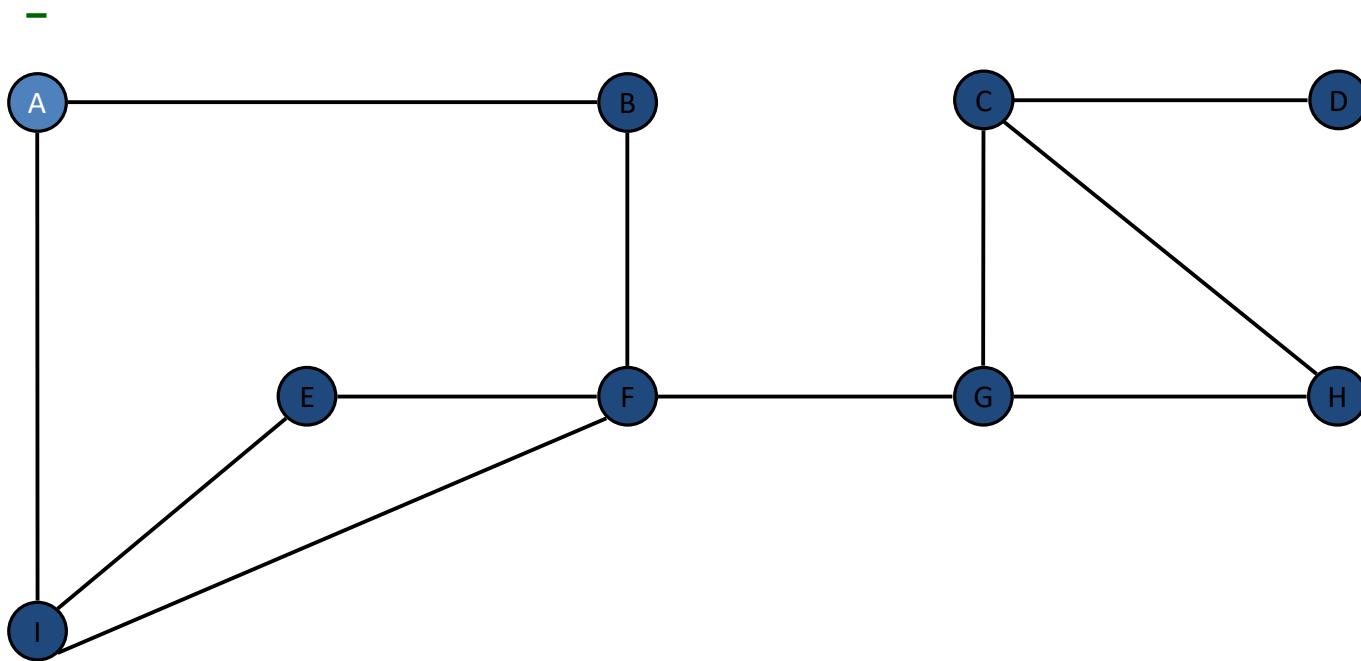
dequeue next vertex

front

A

FIFO Queue

Breadth First Search

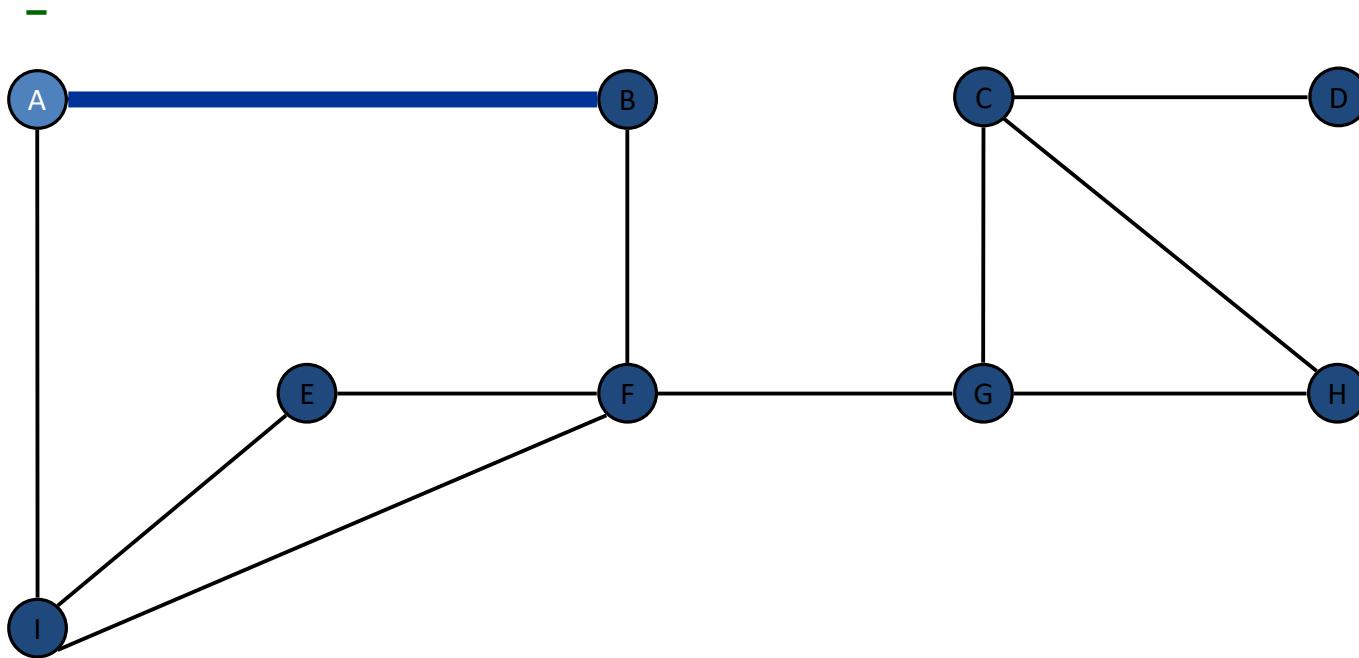


visit neighbors of A

front

FIFO Queue

Breadth First Search

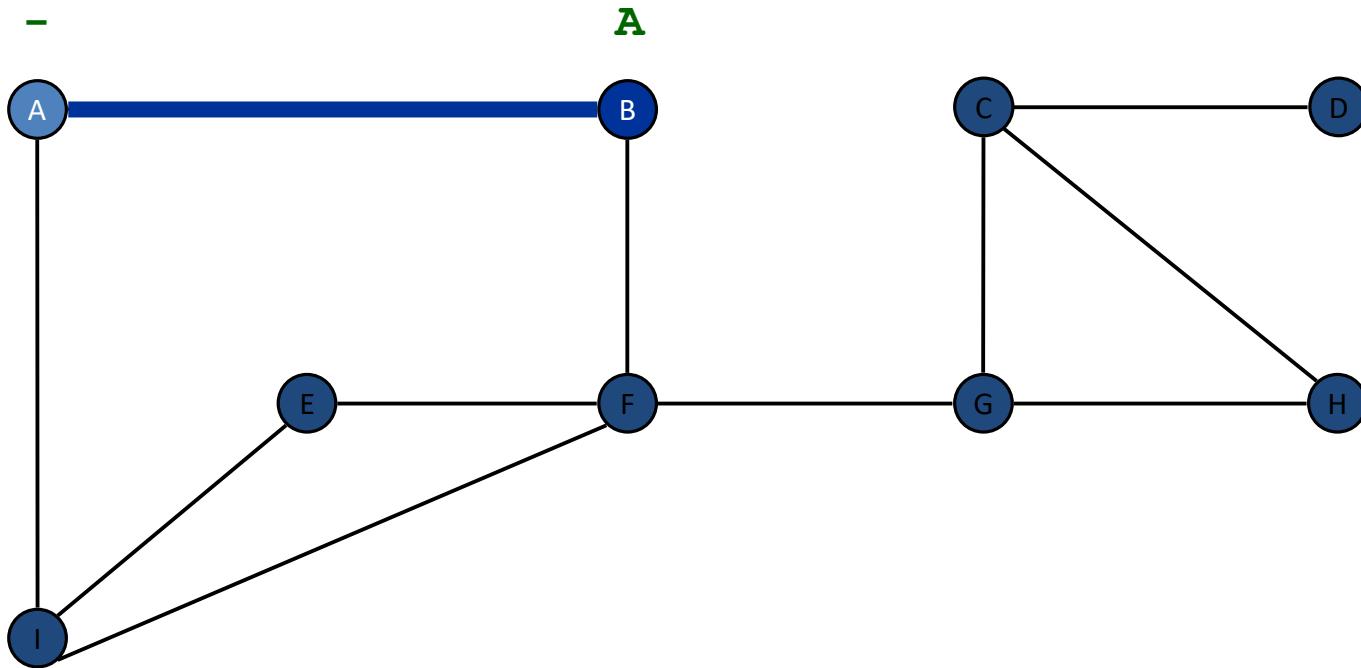


visit neighbors of A

front

FIFO Queue

Breadth First Search



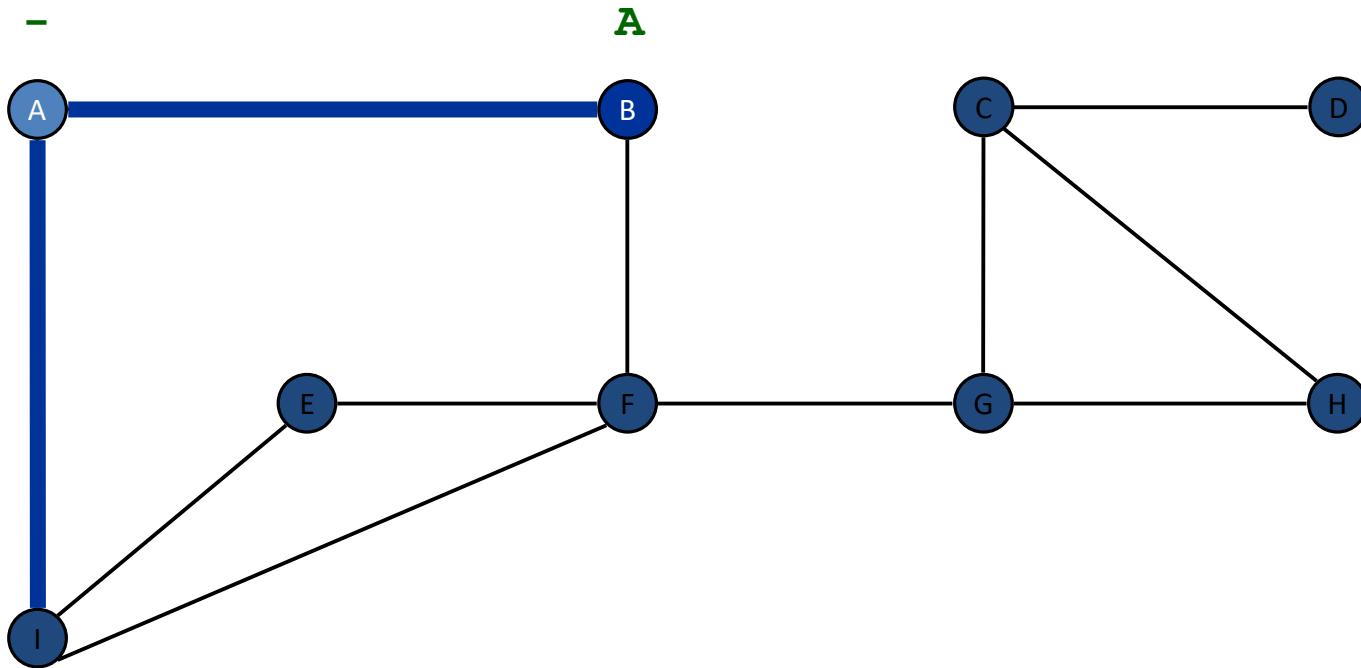
B discovered

front

B

FIFO Queue

Breadth First Search



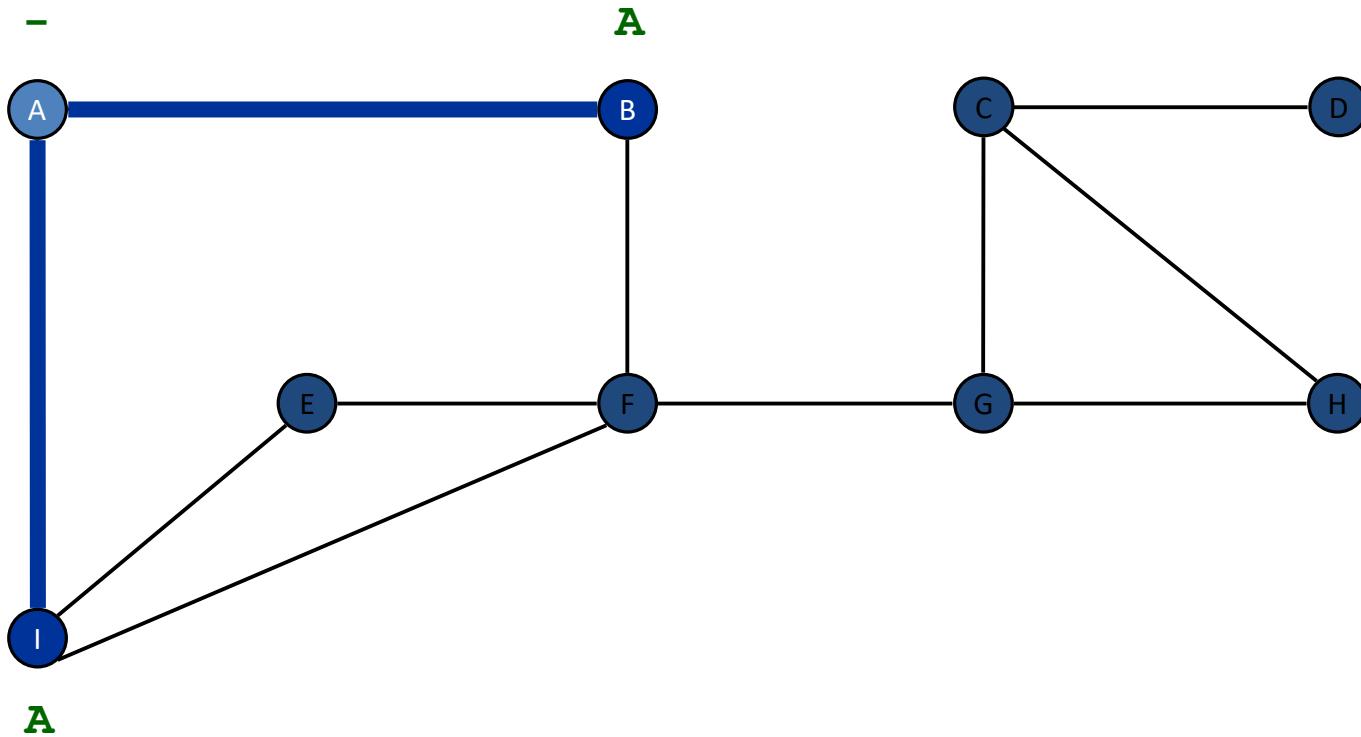
visit neighbors of A

front

B

FIFO Queue

Breadth First Search



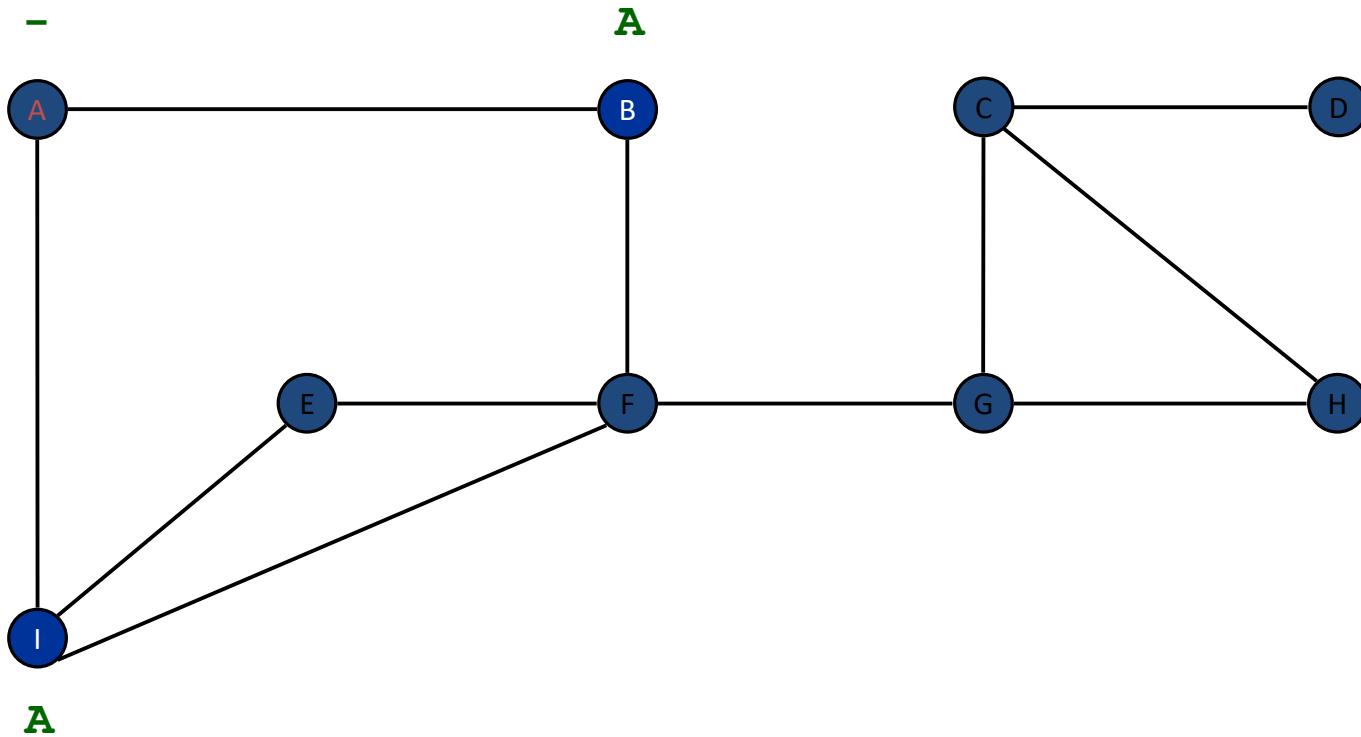
I discovered

front

B I

FIFO Queue

Breadth First Search



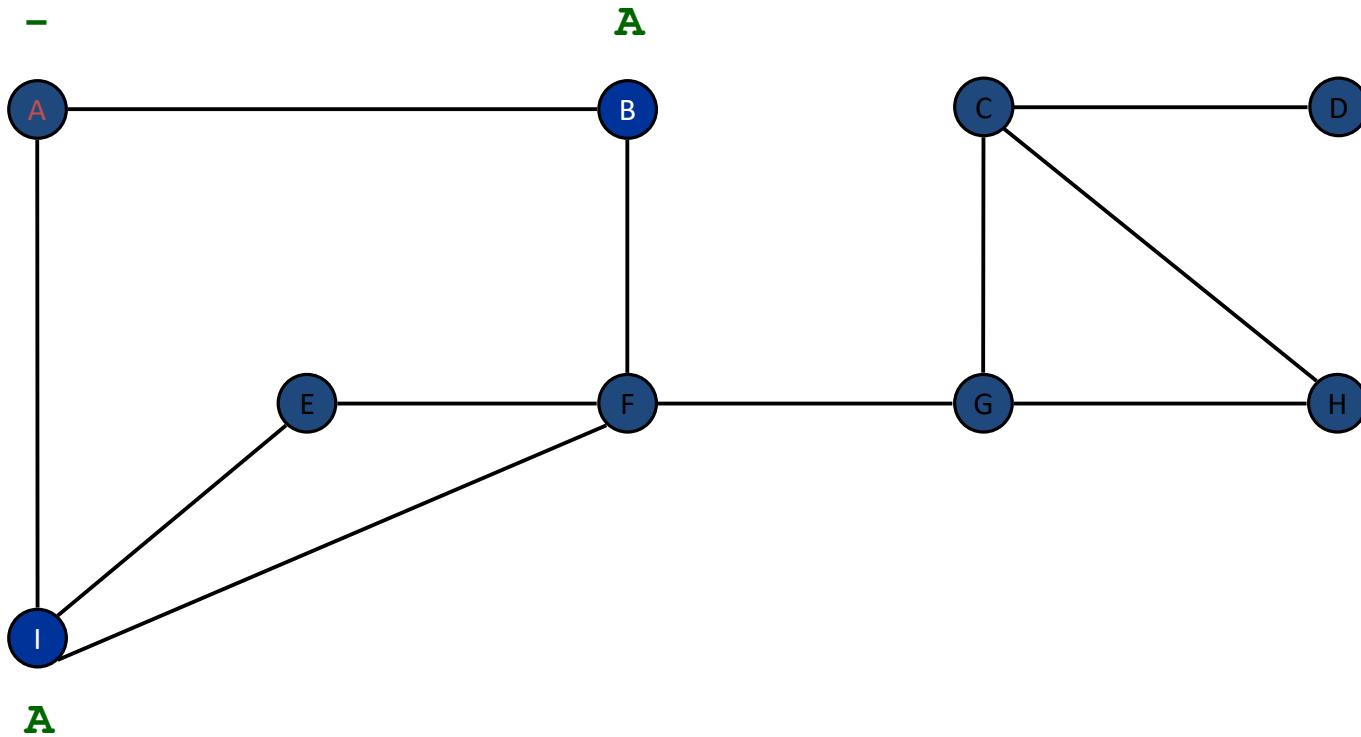
finished with A

front

B I

FIFO Queue

Breadth First Search



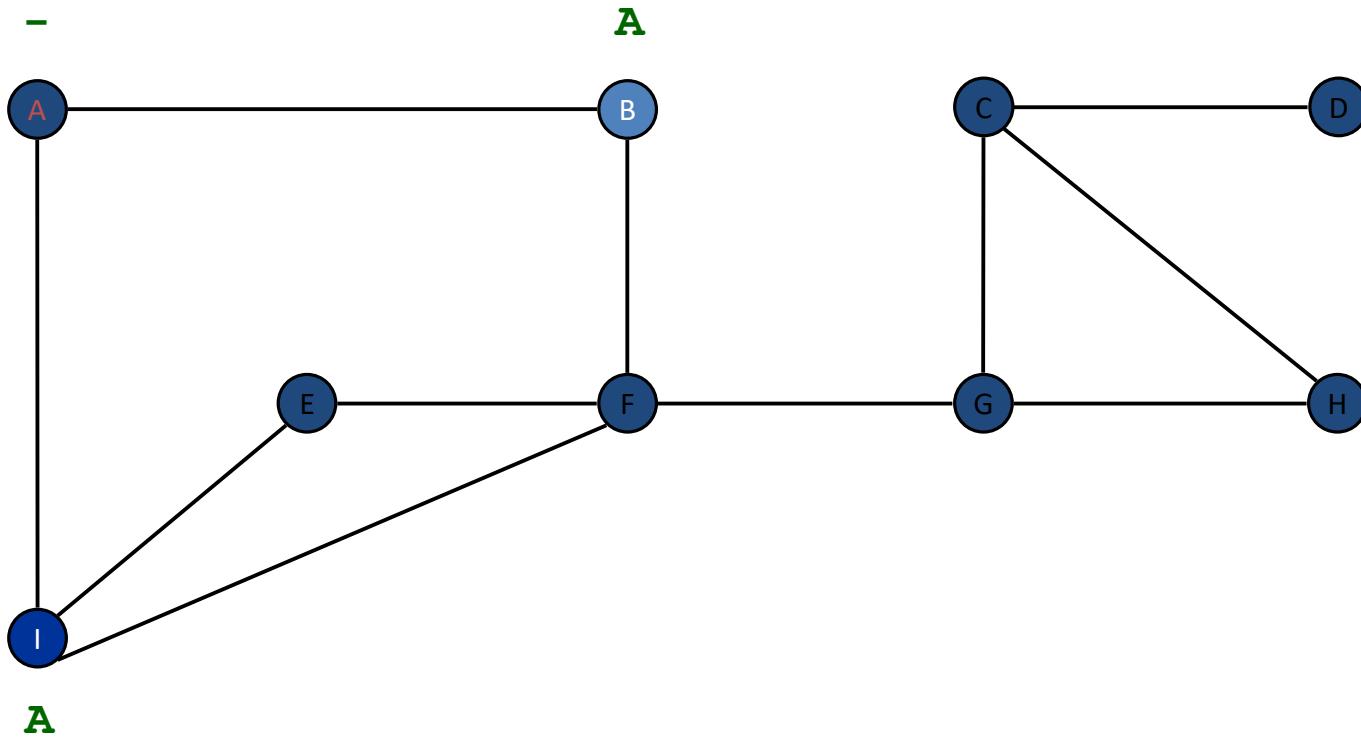
dequeue next vertex

front

B I

FIFO Queue

Breadth First Search



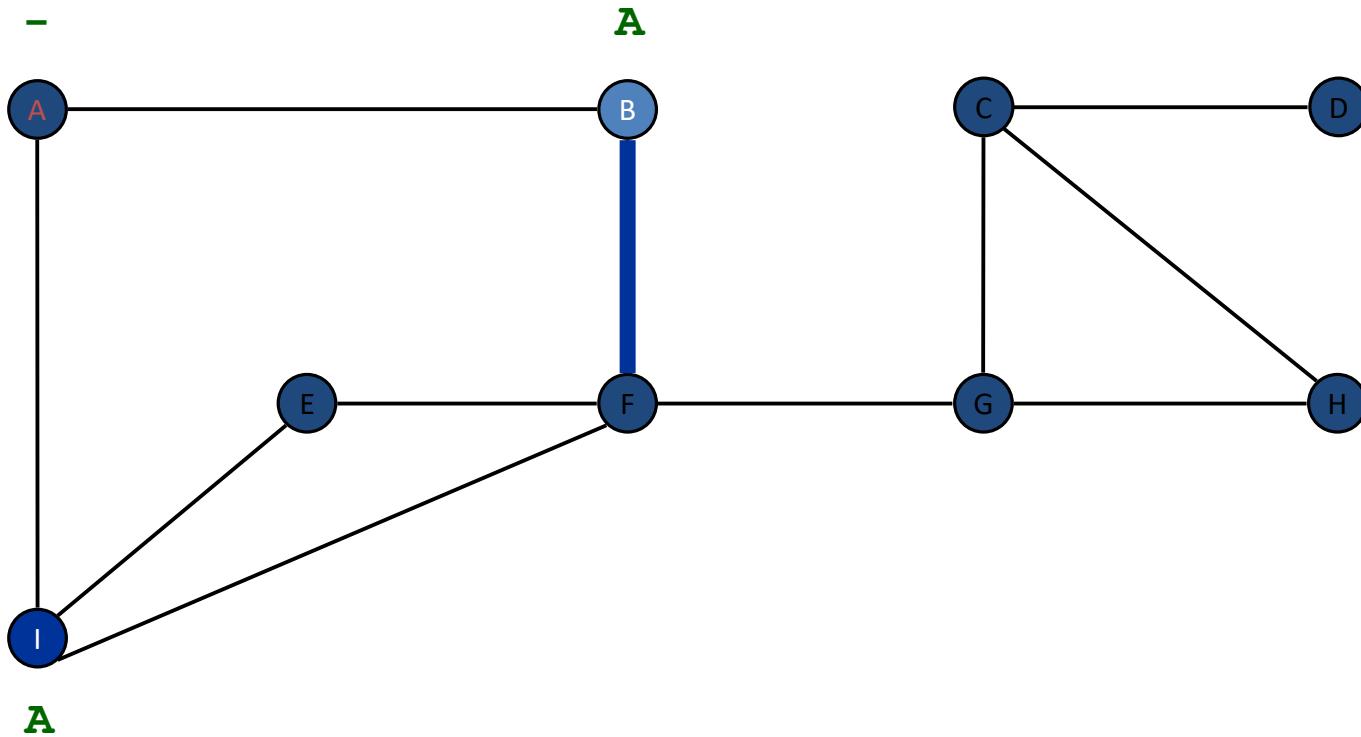
visit neighbors of B

front

I

FIFO Queue

Breadth First Search



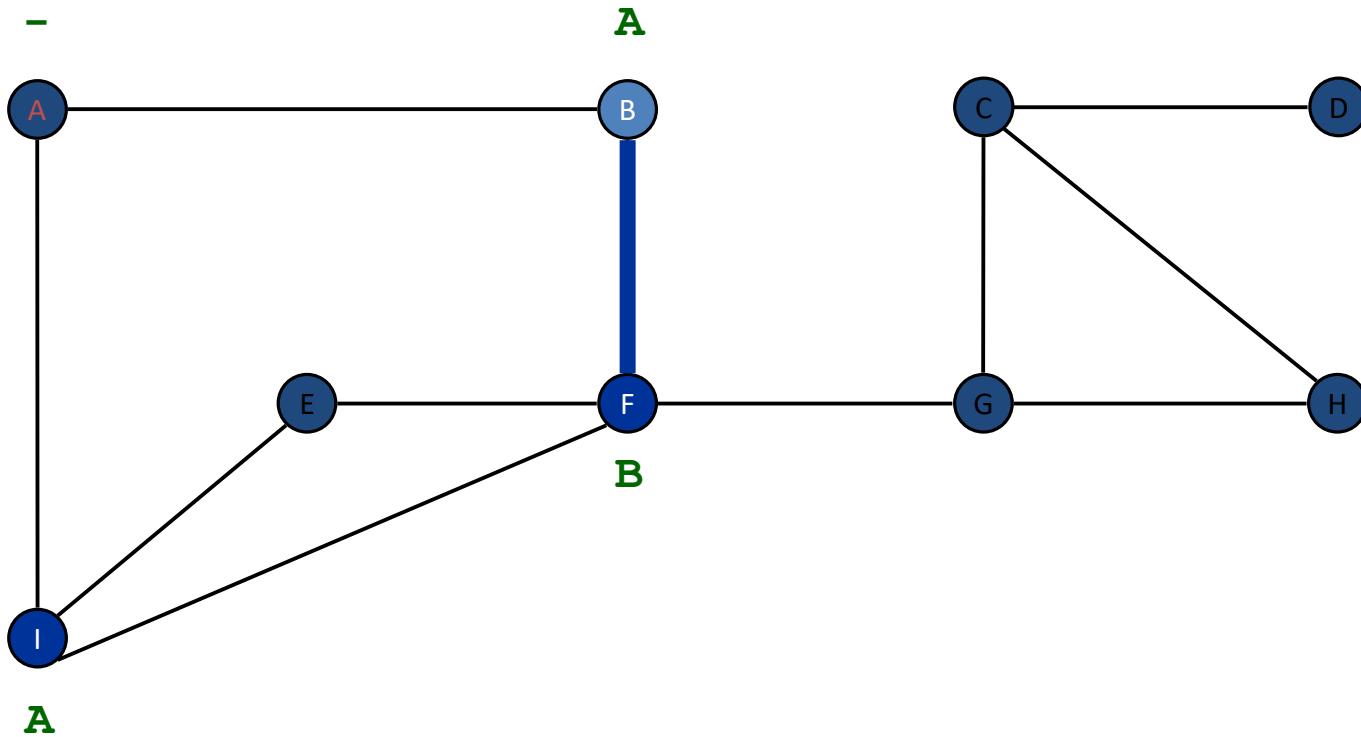
visit neighbors of B

front

I

FIFO Queue

Breadth First Search



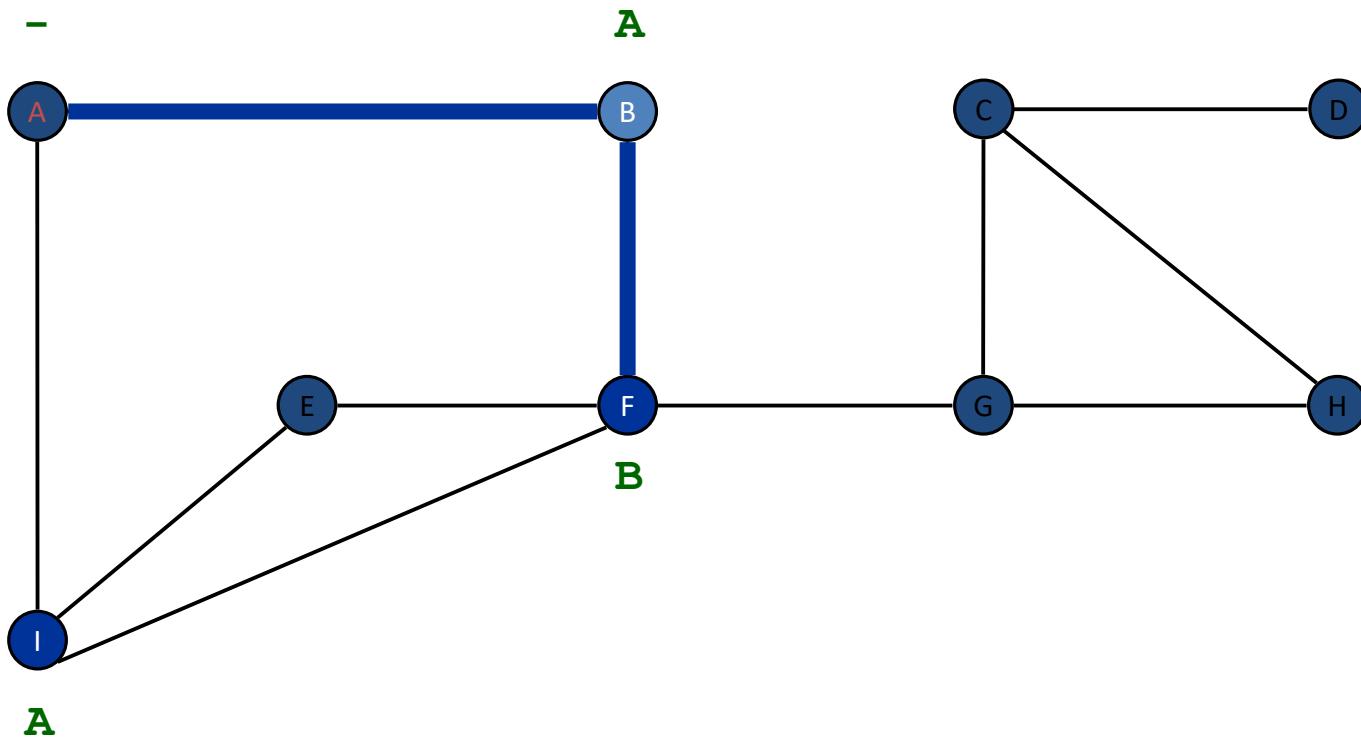
F discovered

front

I F

FIFO Queue

Breadth First Search



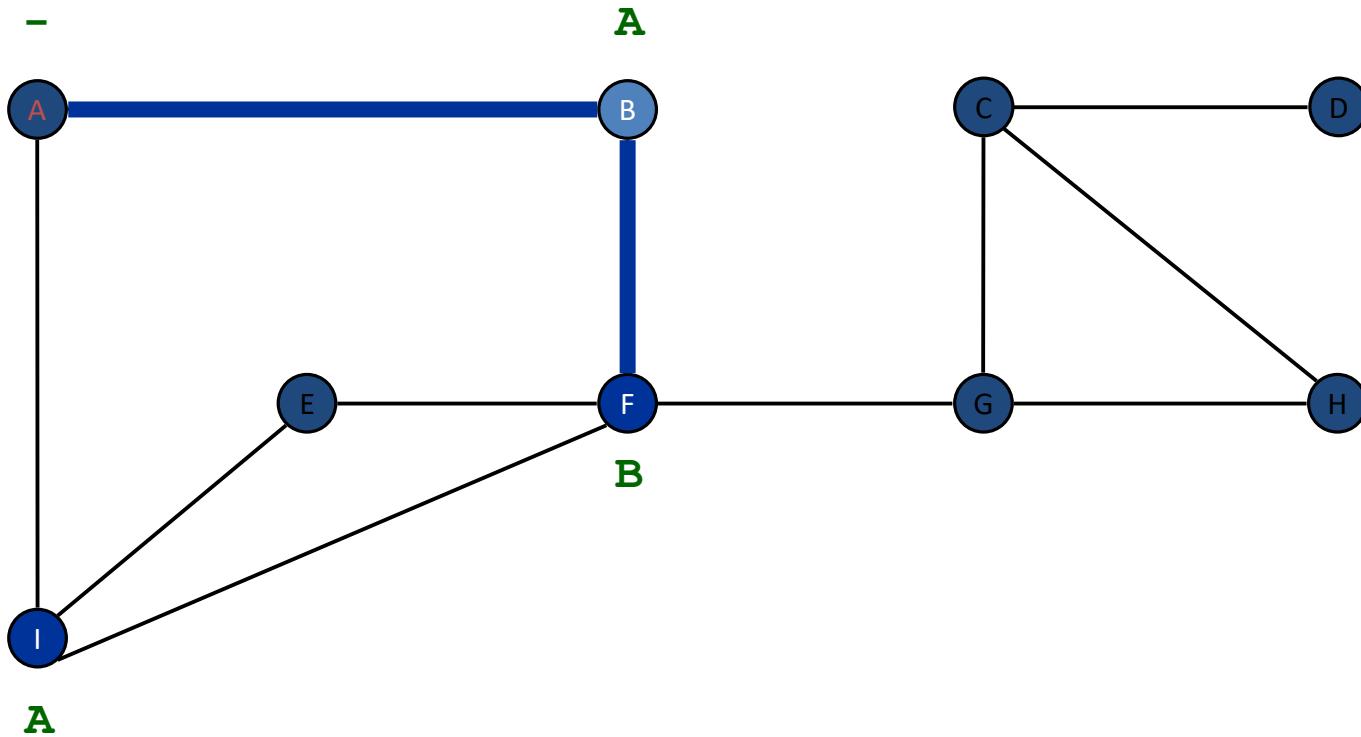
visit neighbors of B

front

I F

FIFO Queue

Breadth First Search



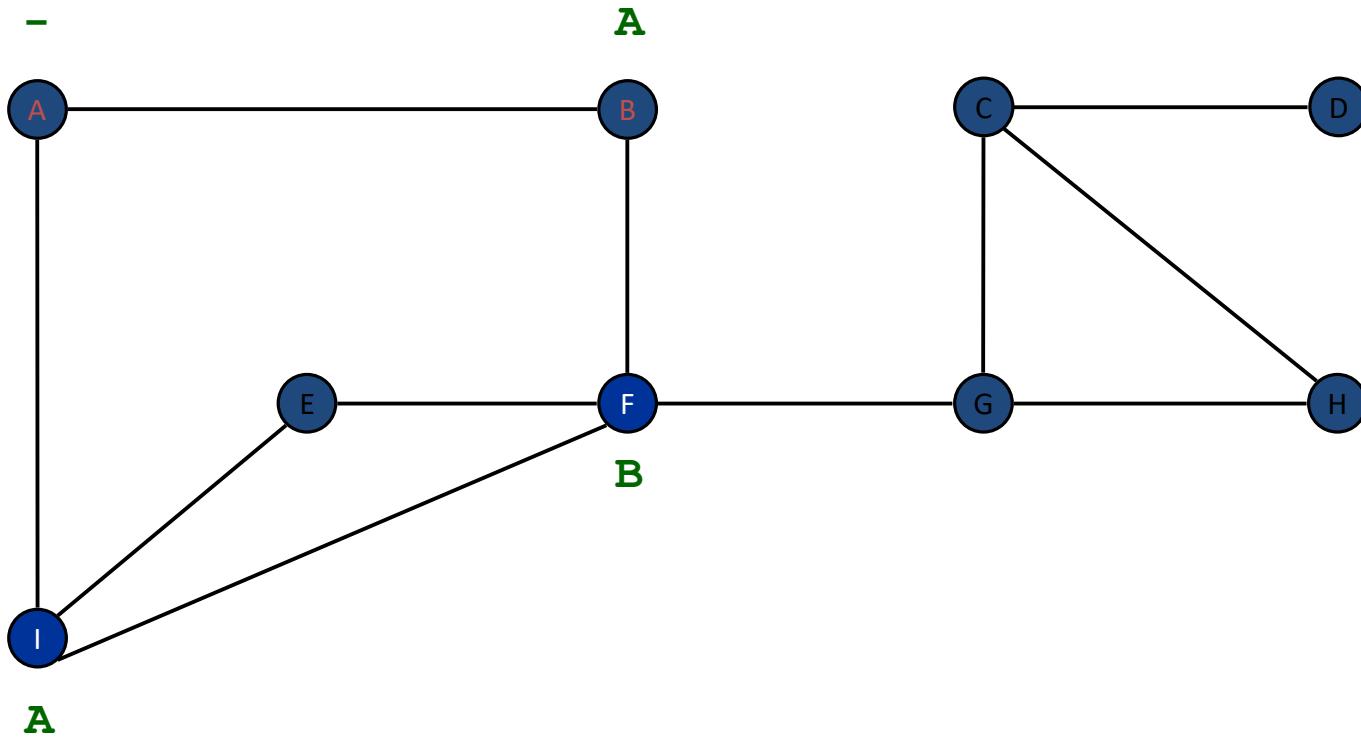
A already discovered

front

I F

FIFO Queue

Breadth First Search



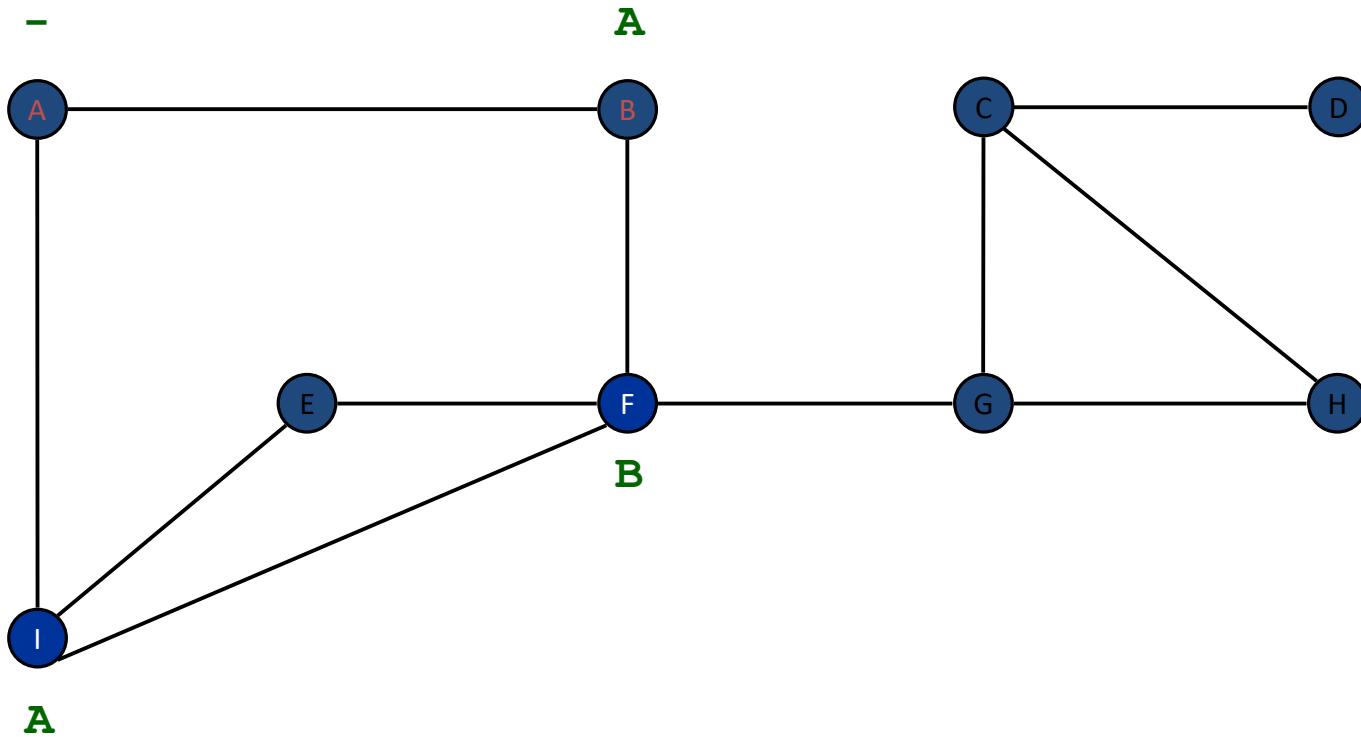
finished with B

front

I F

FIFO Queue

Breadth First Search



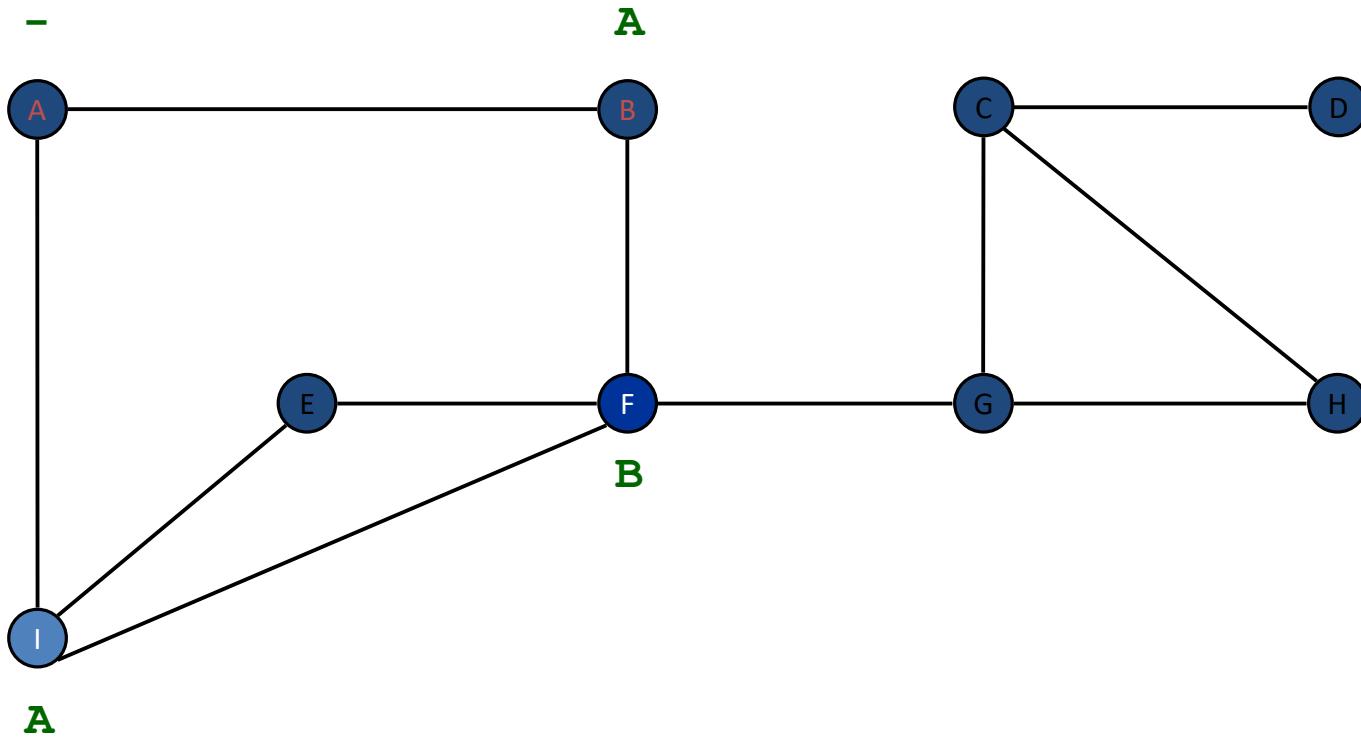
dequeue next vertex

front

I F

FIFO Queue

Breadth First Search



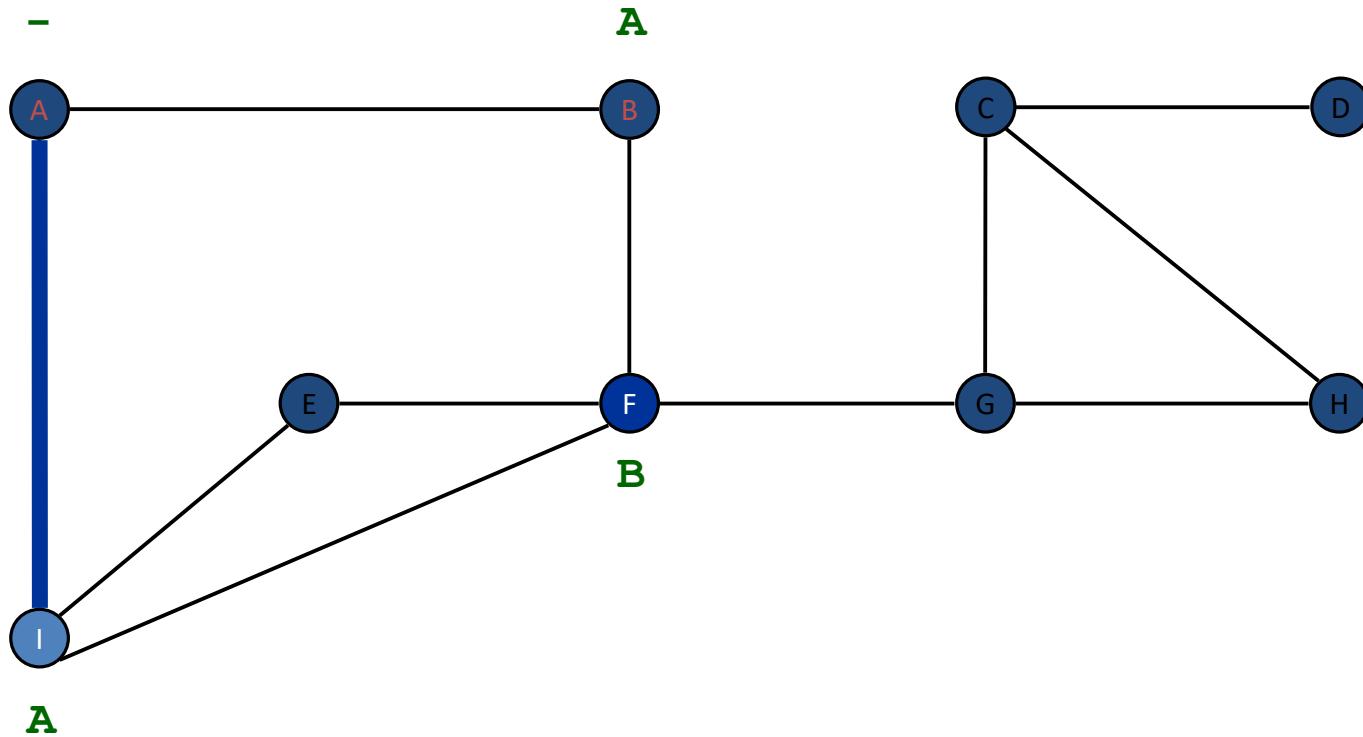
visit neighbors of I

front

F

FIFO Queue

Breadth First Search



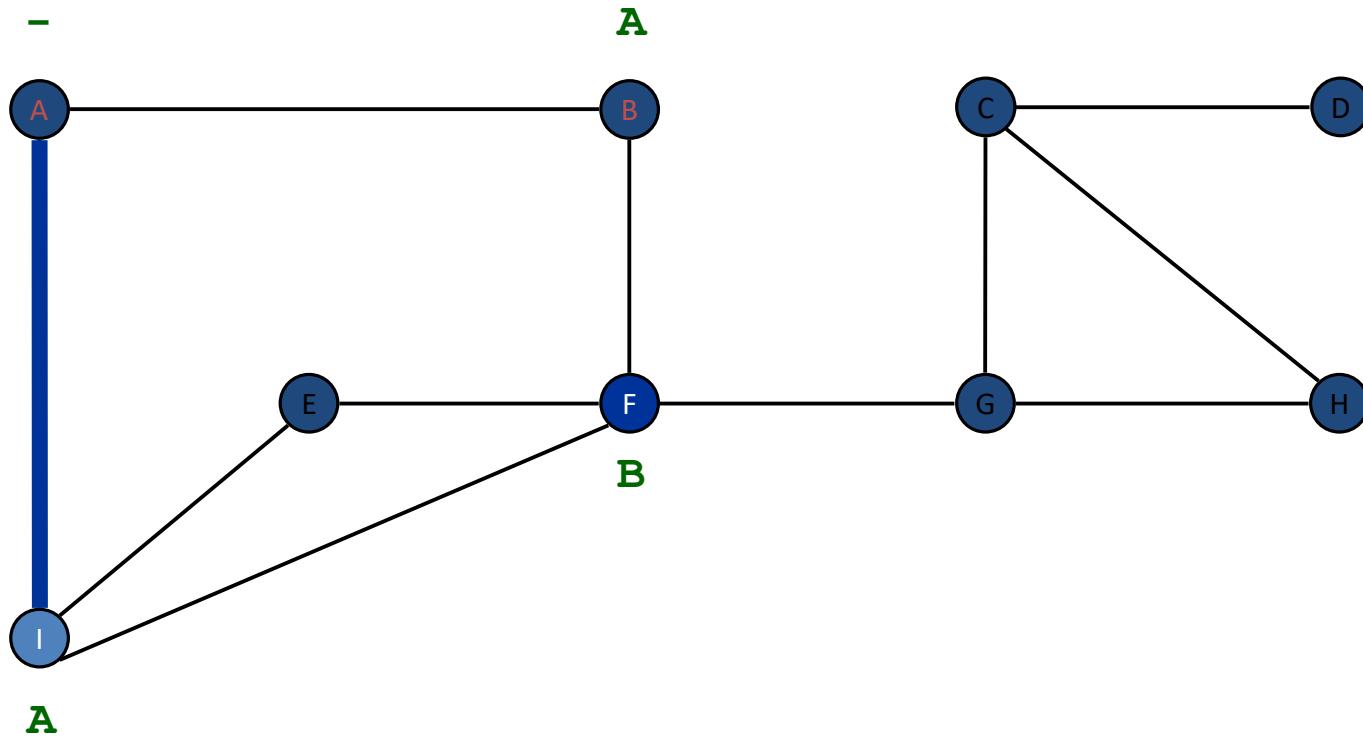
visit neighbors of I

front

F

FIFO Queue

Breadth First Search



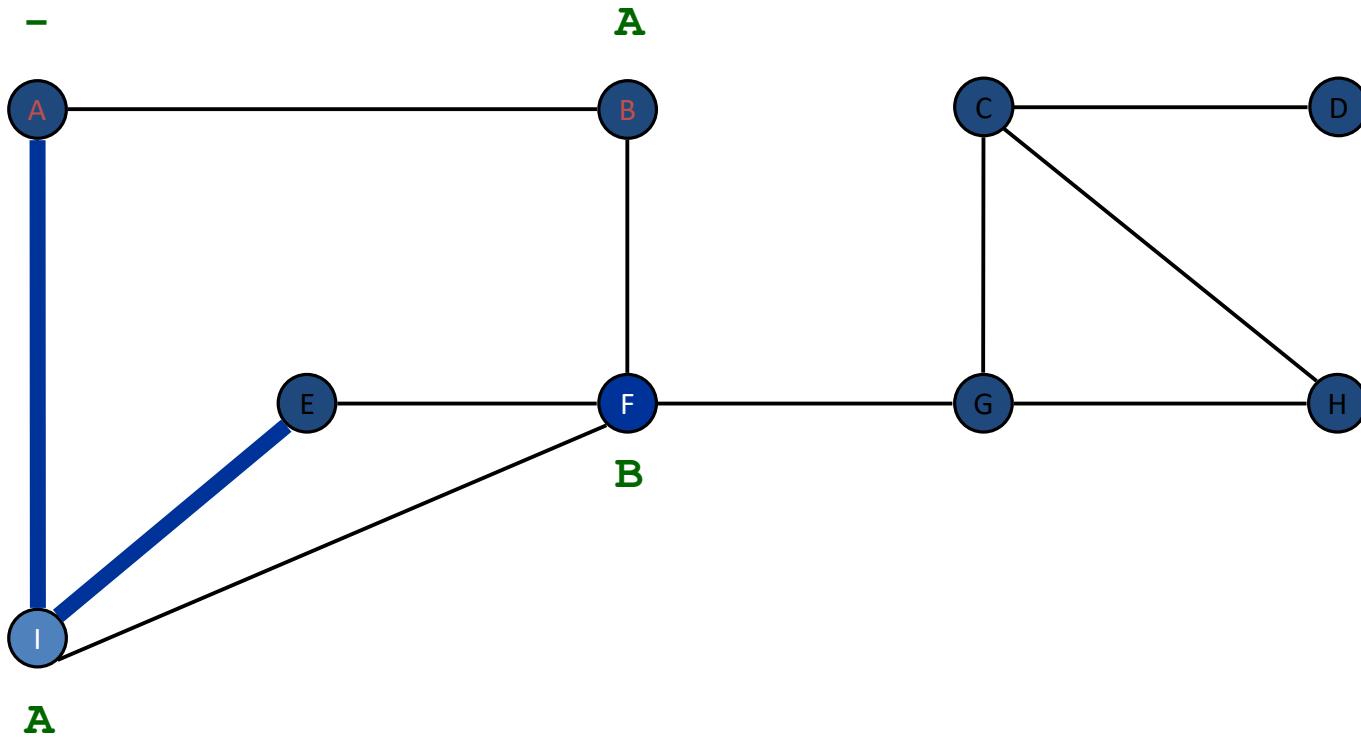
A already discovered

front

F

FIFO Queue

Breadth First Search



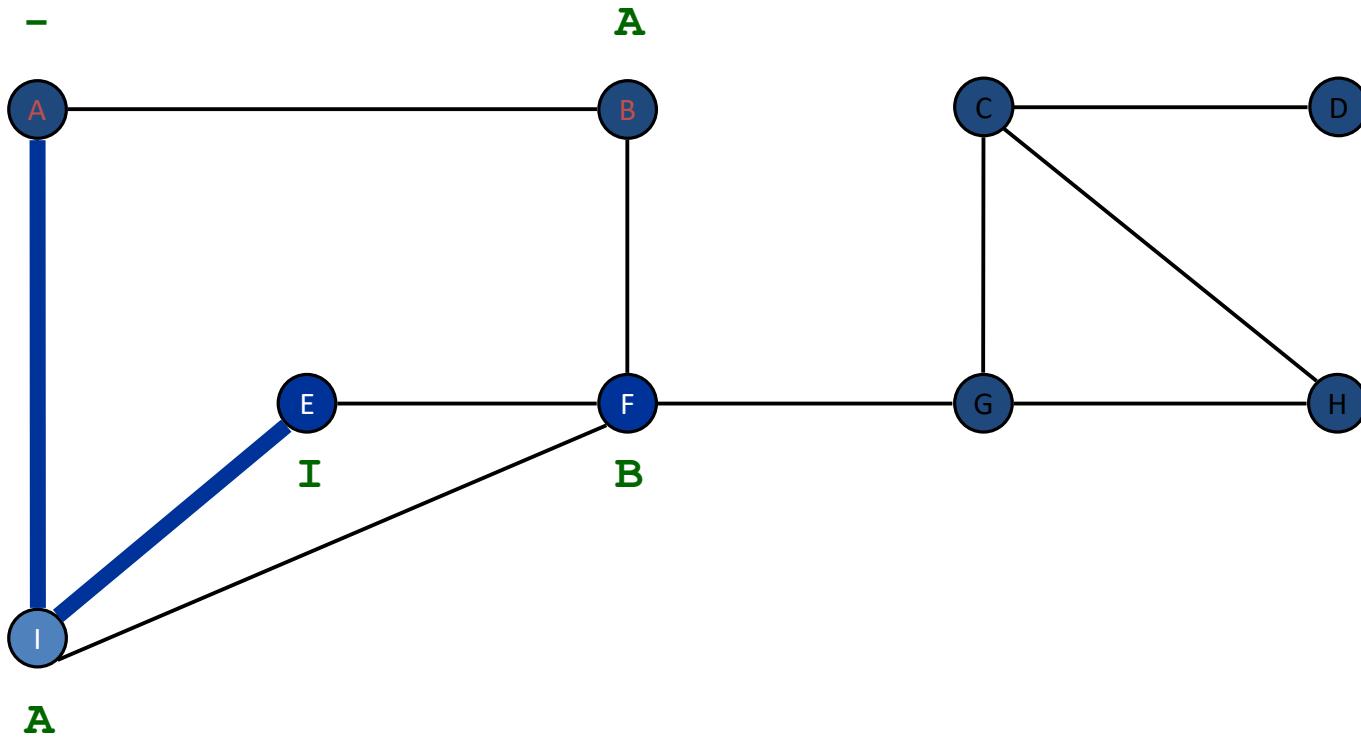
visit neighbors of I

front

F

FIFO Queue

Breadth First Search



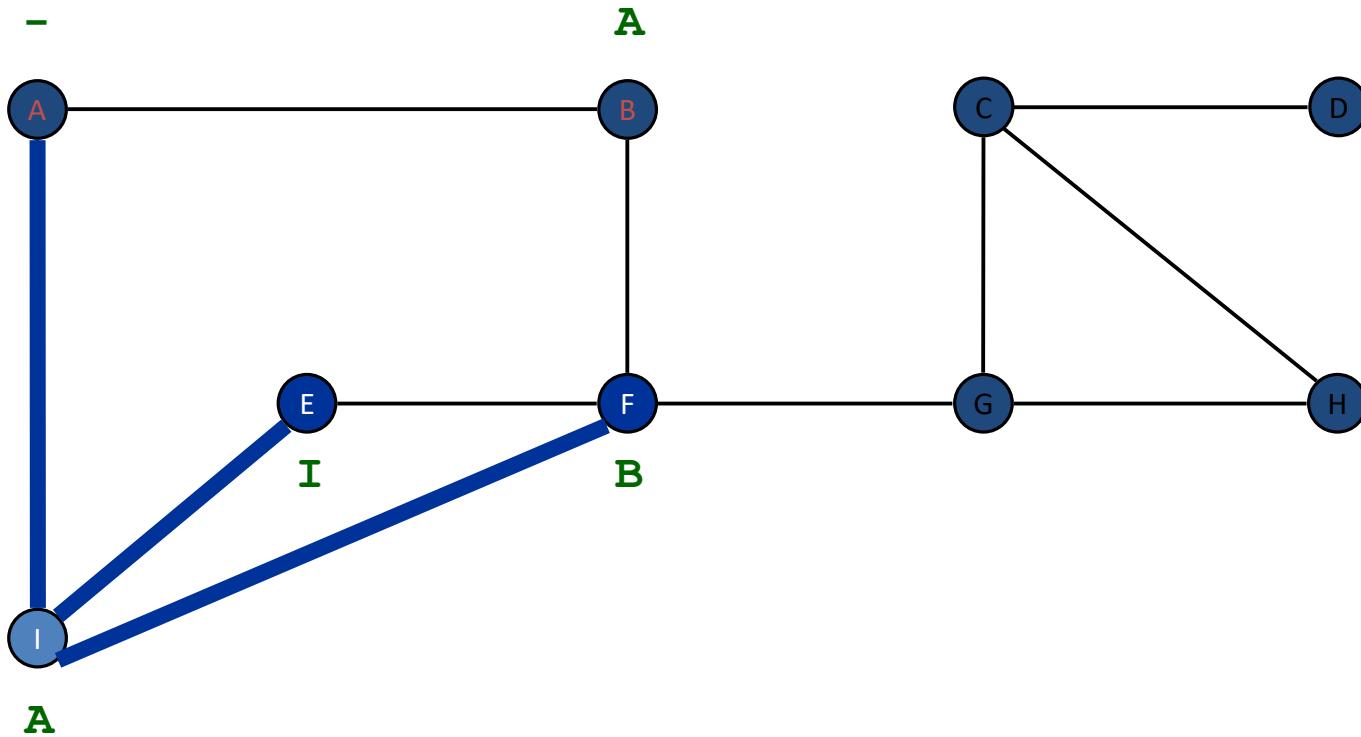
E discovered

front

F E

FIFO Queue

Breadth First Search



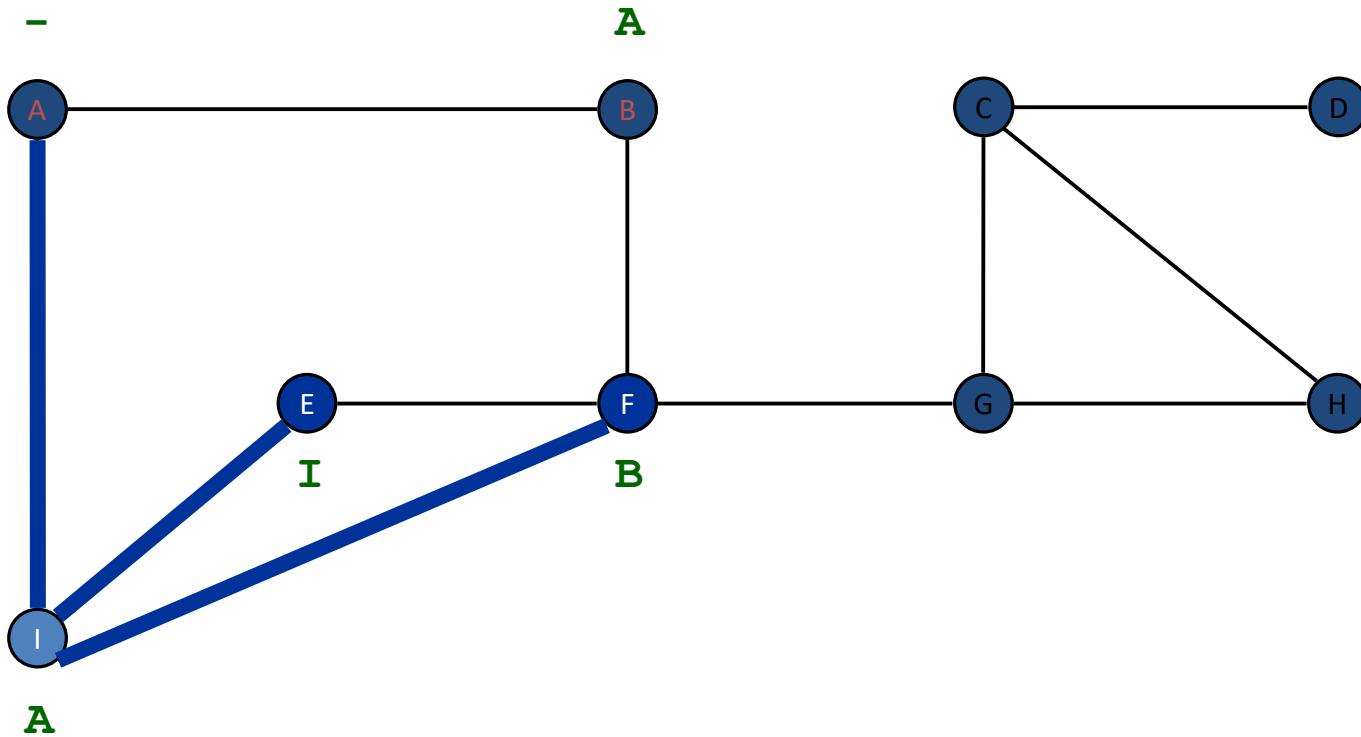
visit neighbors of I

front

F E

FIFO Queue

Breadth First Search



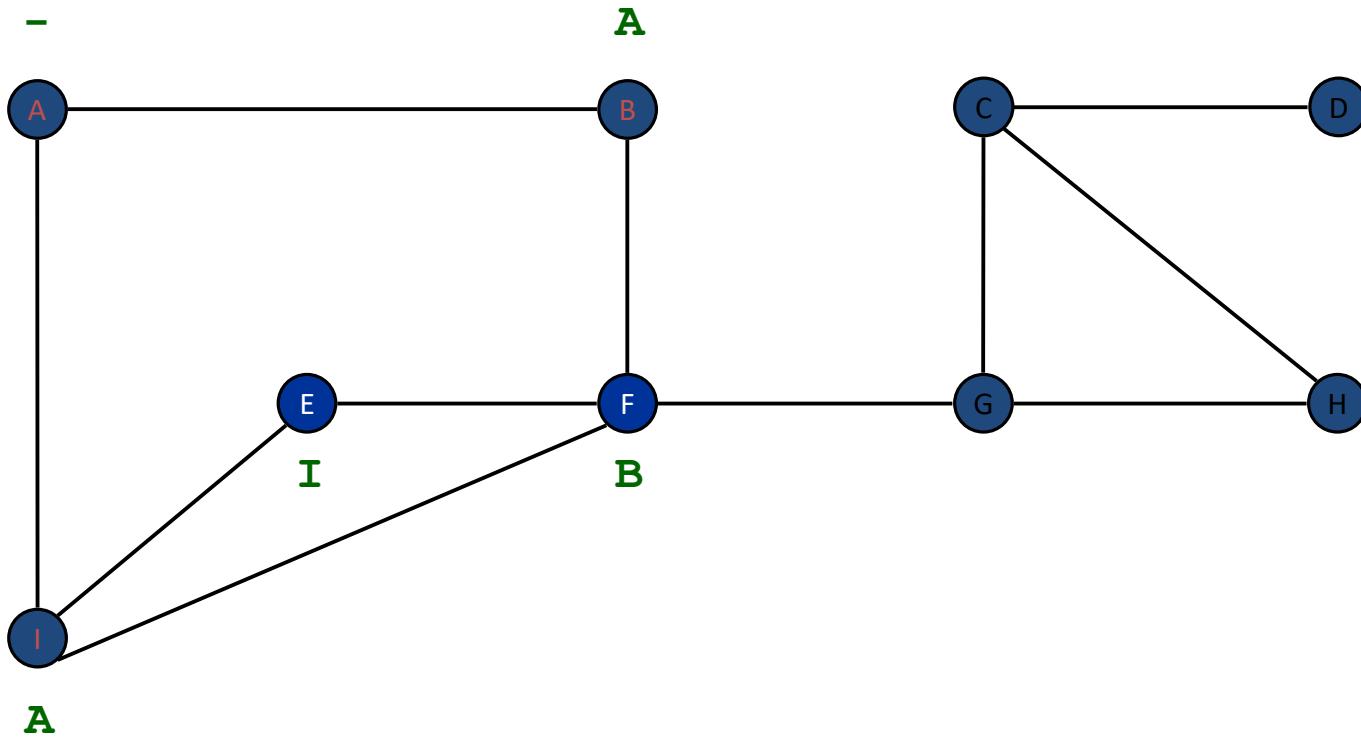
F already discovered

front

F E

FIFO Queue

Breadth First Search



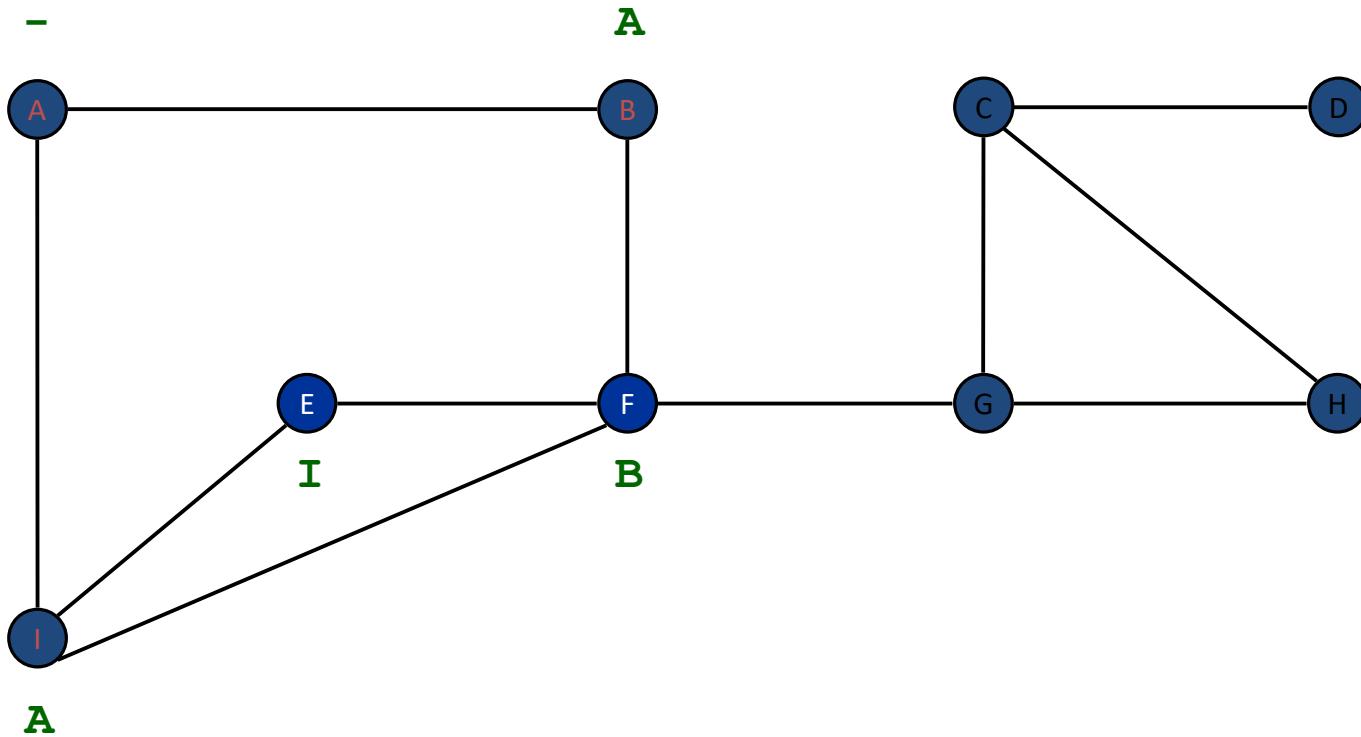
I finished

front

F E

FIFO Queue

Breadth First Search



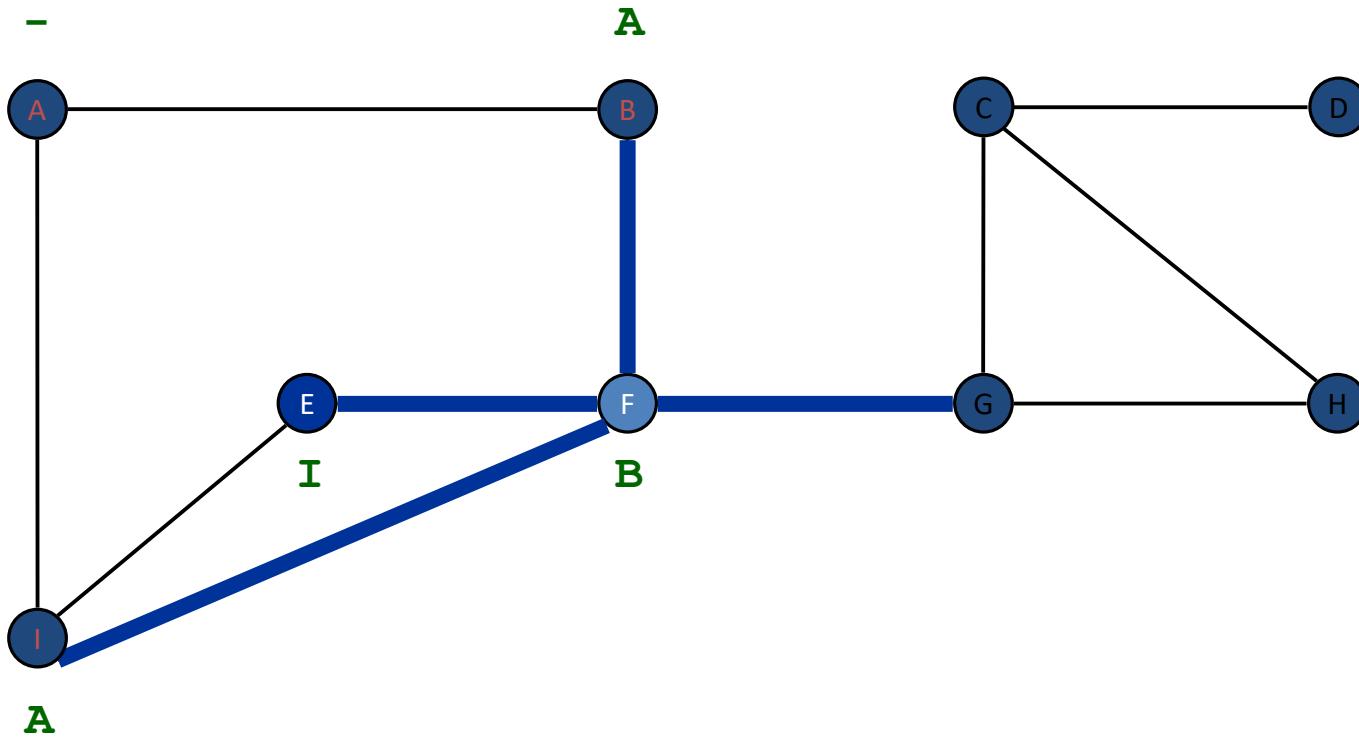
dequeue next vertex

front

F E

FIFO Queue

Breadth First Search



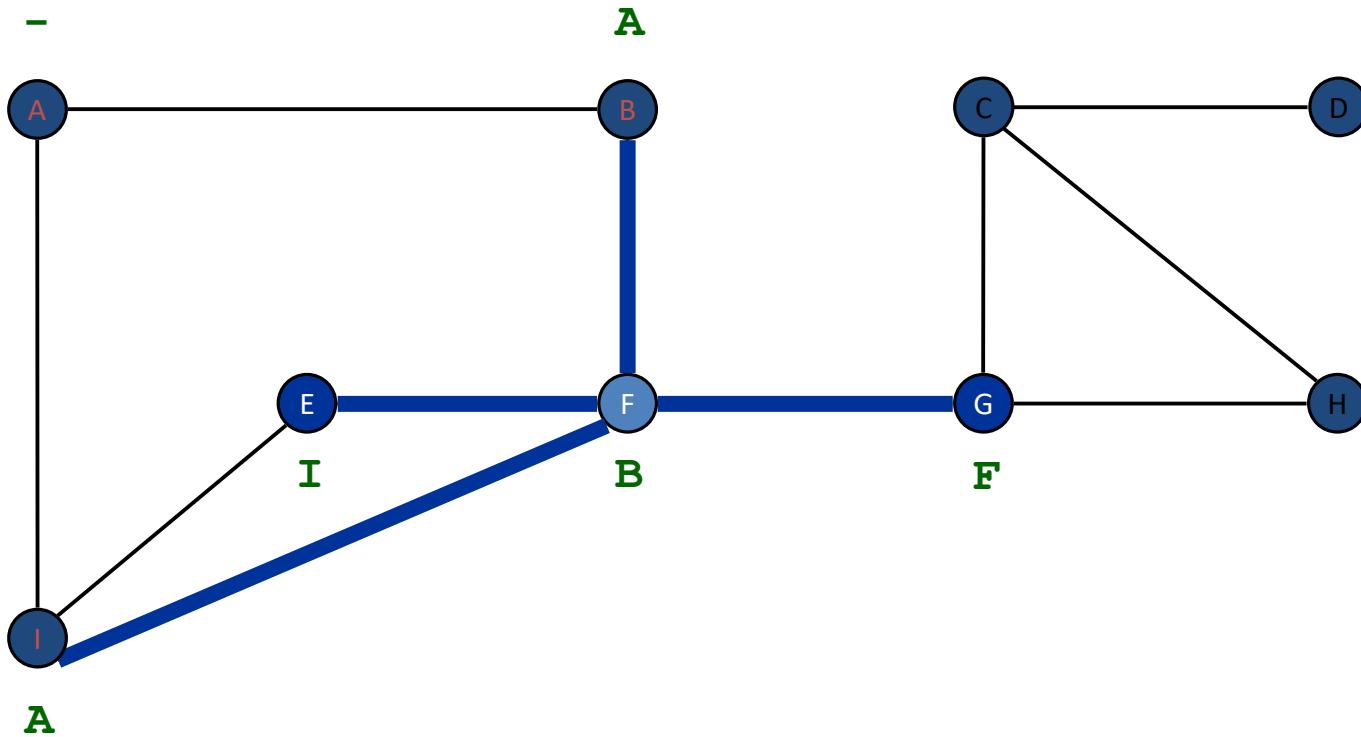
visit neighbors of F

front

E

FIFO Queue

Breadth First Search



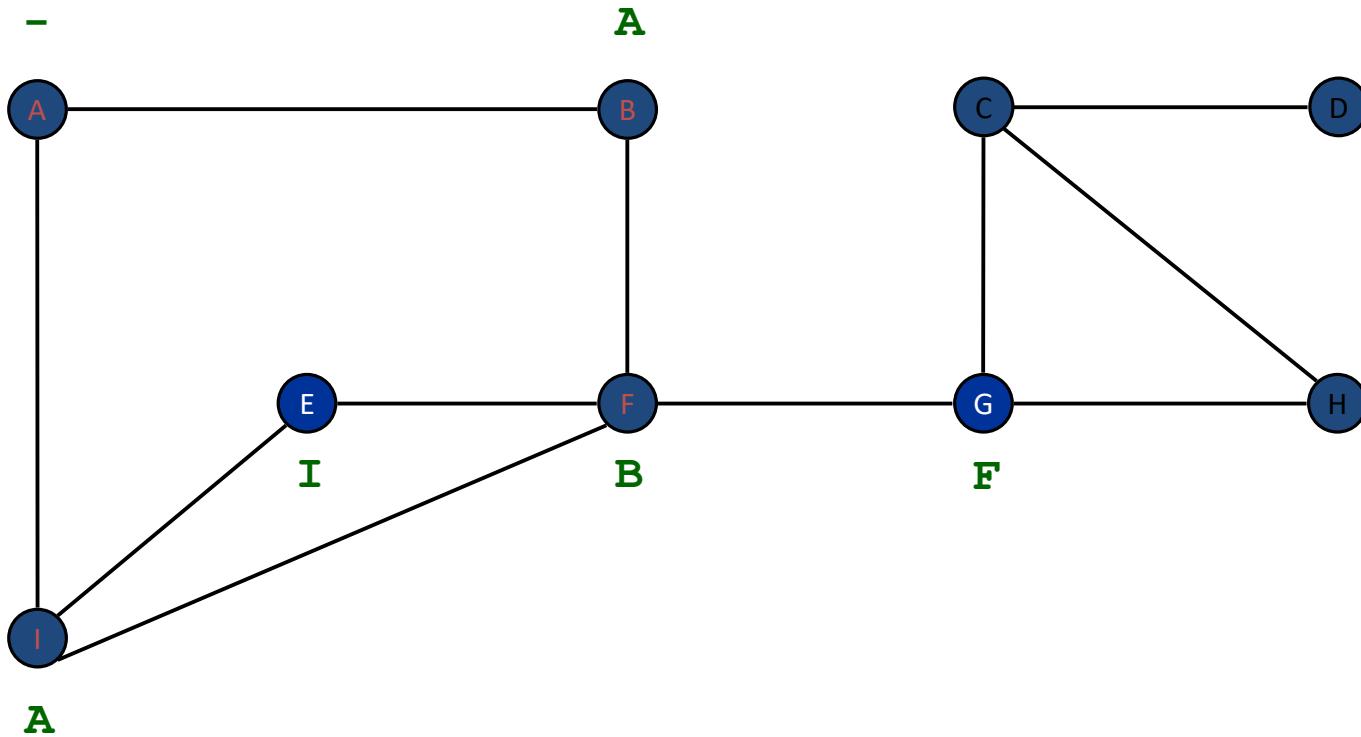
G discovered

front

E G

FIFO Queue

Breadth First Search



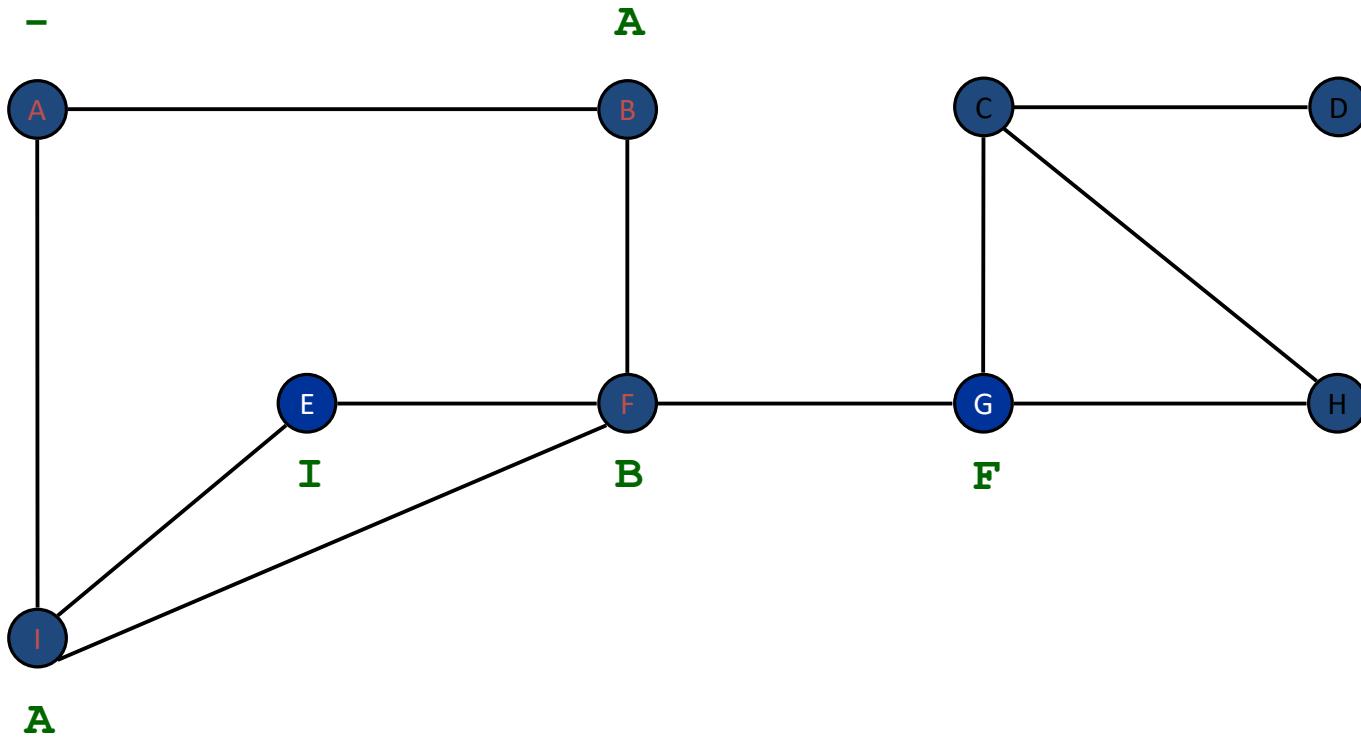
F finished

front

E G

FIFO Queue

Breadth First Search



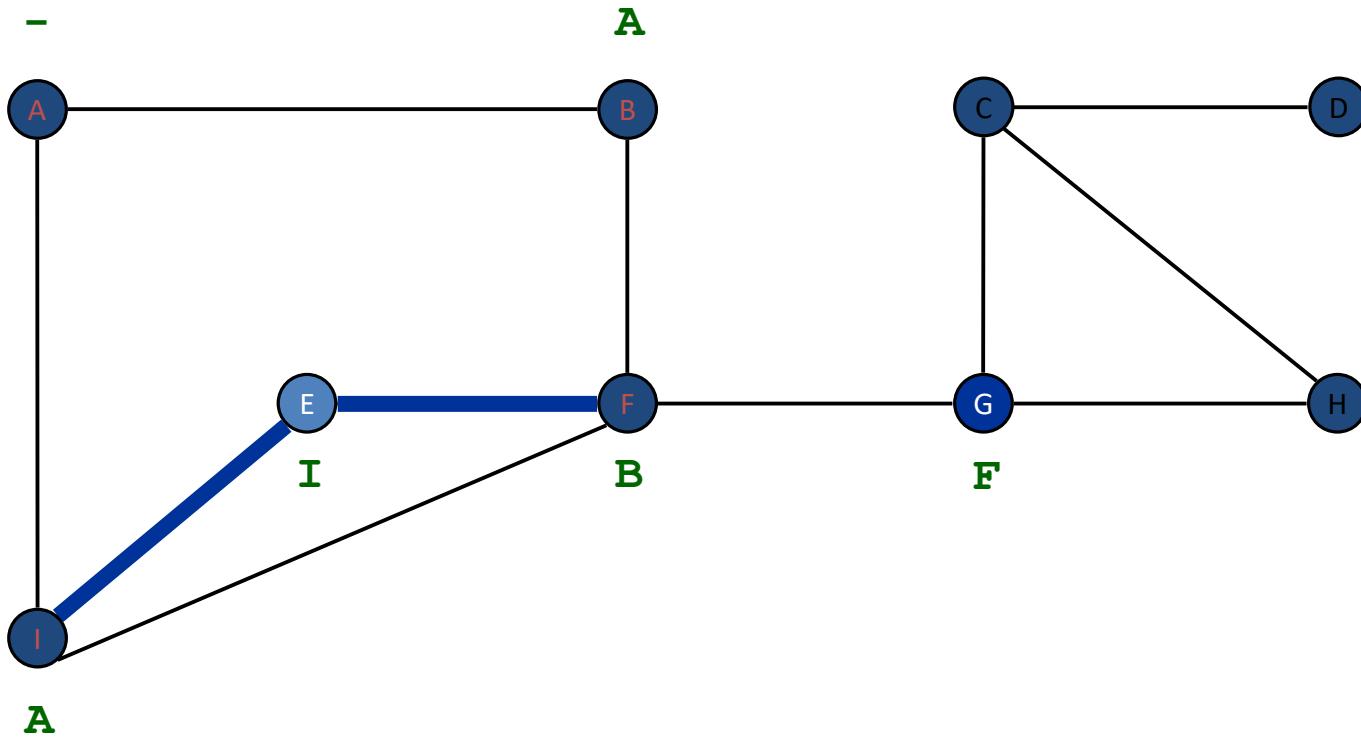
dequeue next vertex

front

E G

FIFO Queue

Breadth First Search



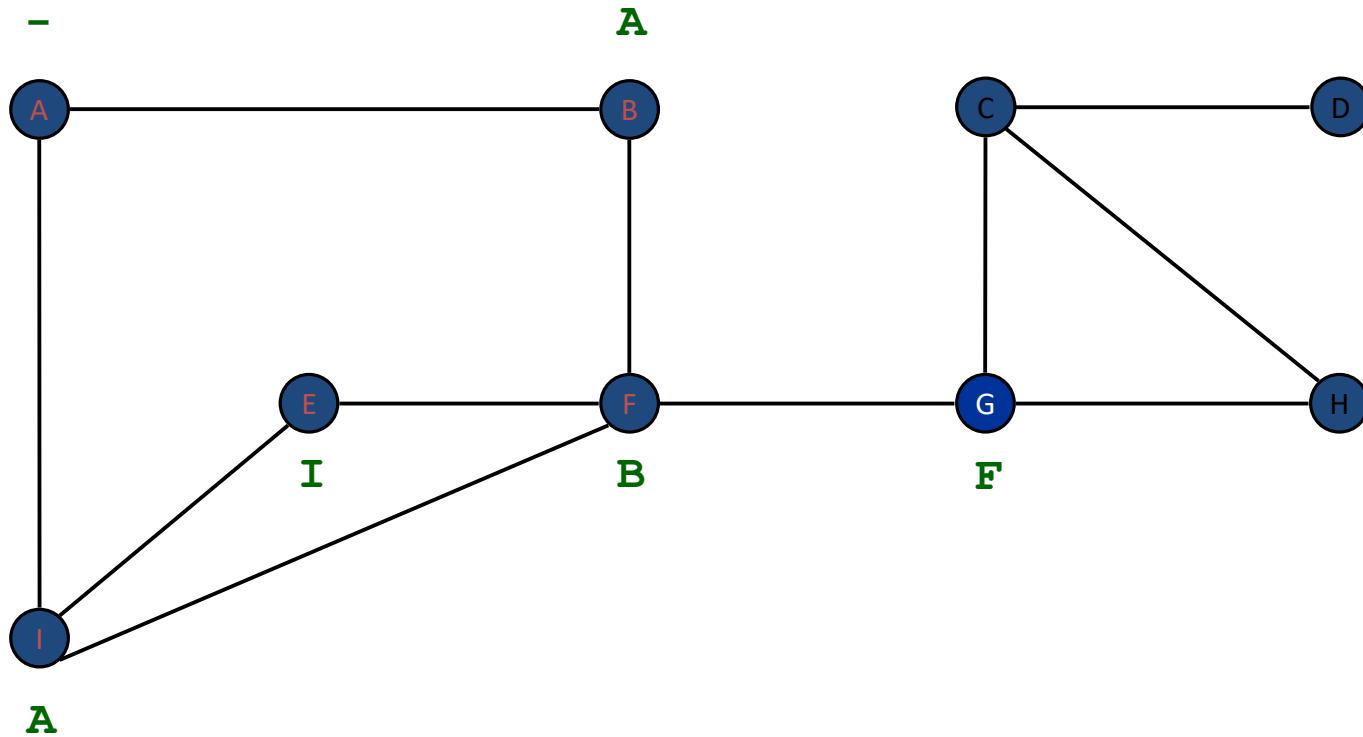
visit neighbors of E

front

G

FIFO Queue

Breadth First Search



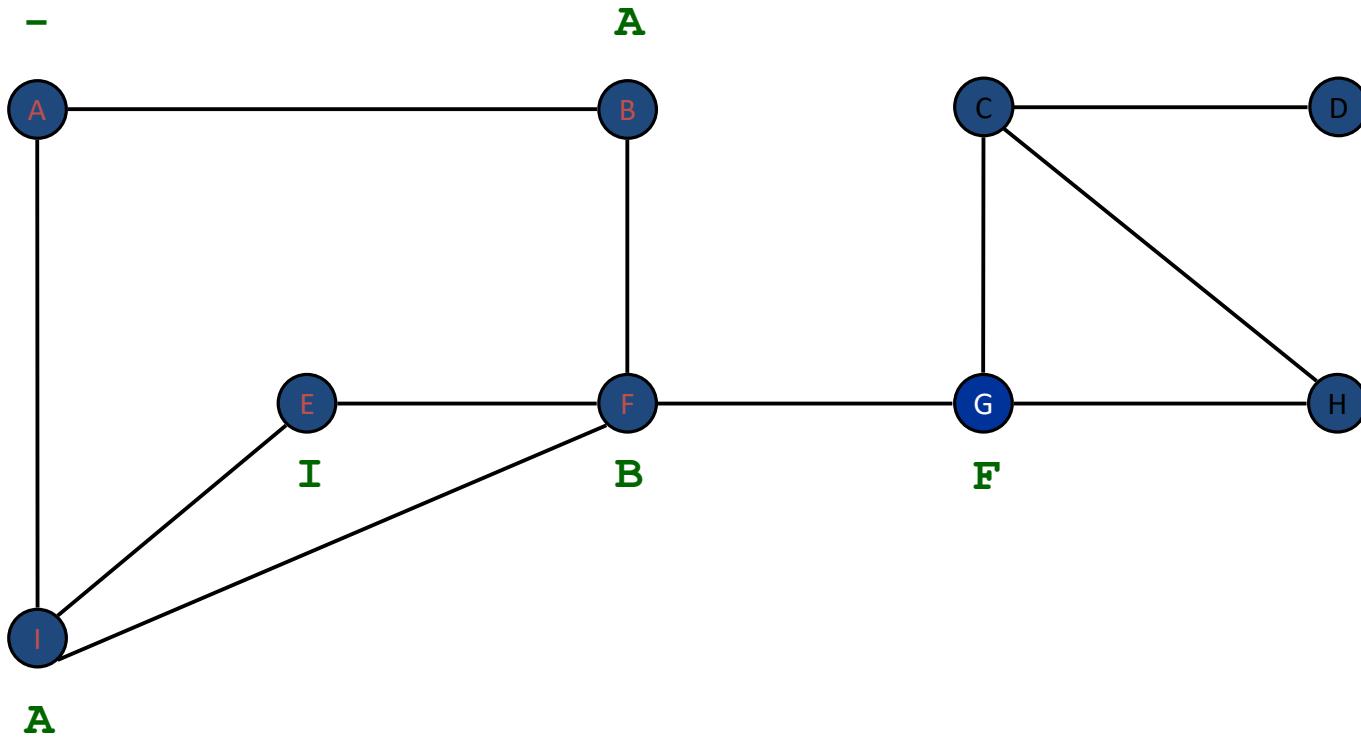
E finished

front

G

FIFO Queue

Breadth First Search



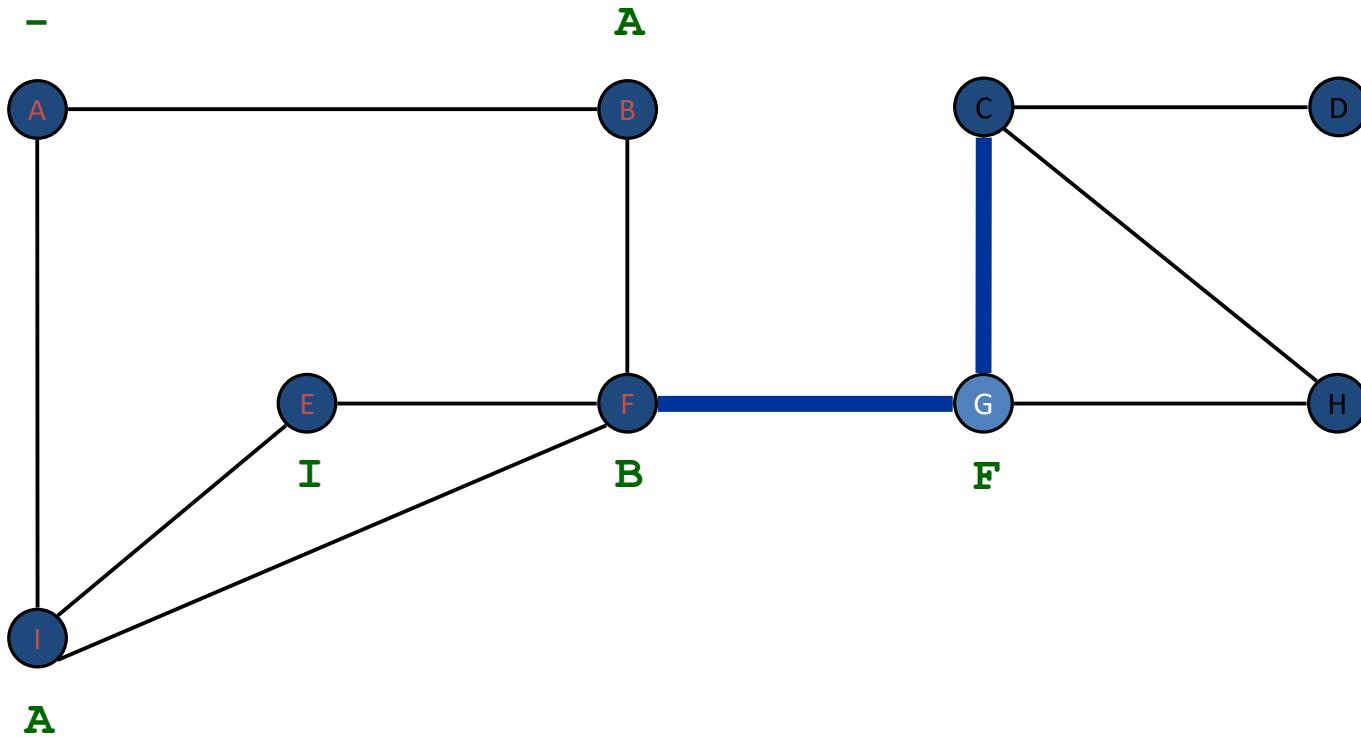
dequeue next vertex

front

G

FIFO Queue

Breadth First Search

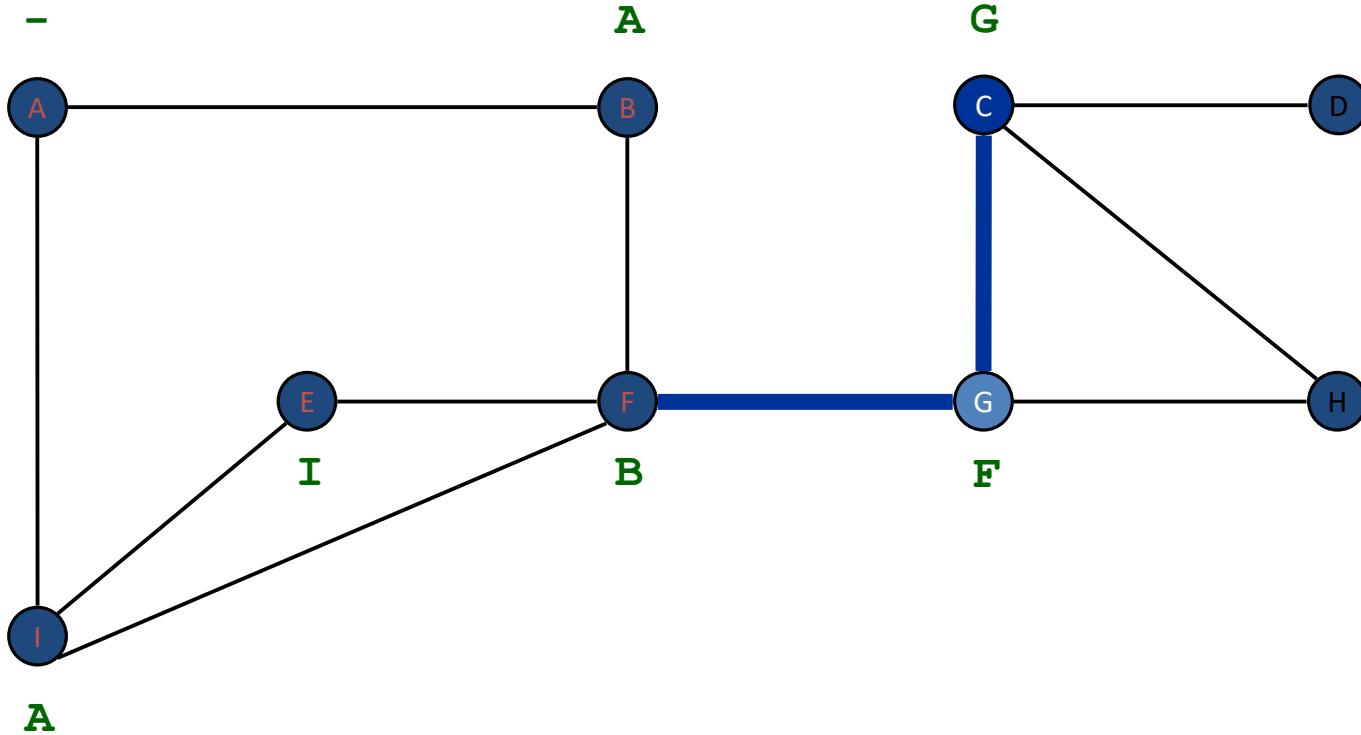


visit neighbors of G

front

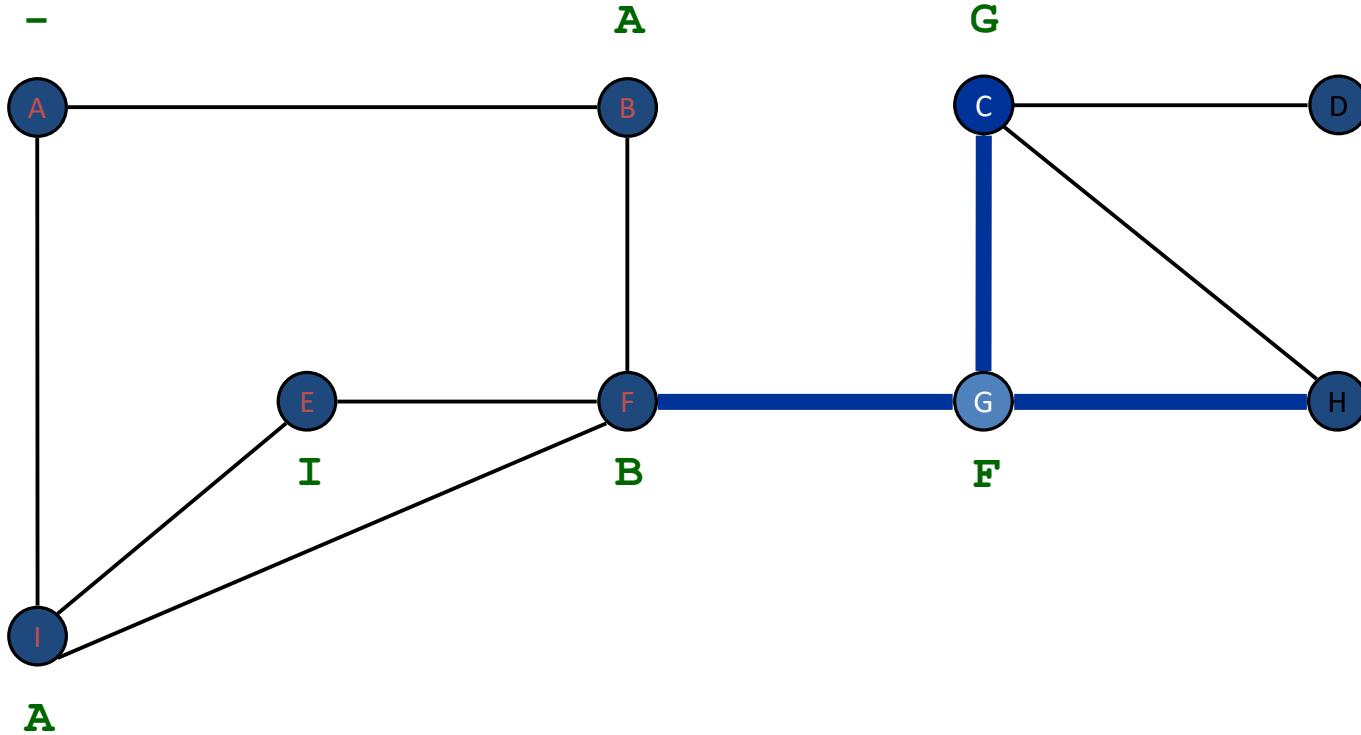
FIFO Queue

Breadth First Search



FIFO Queue

Breadth First Search



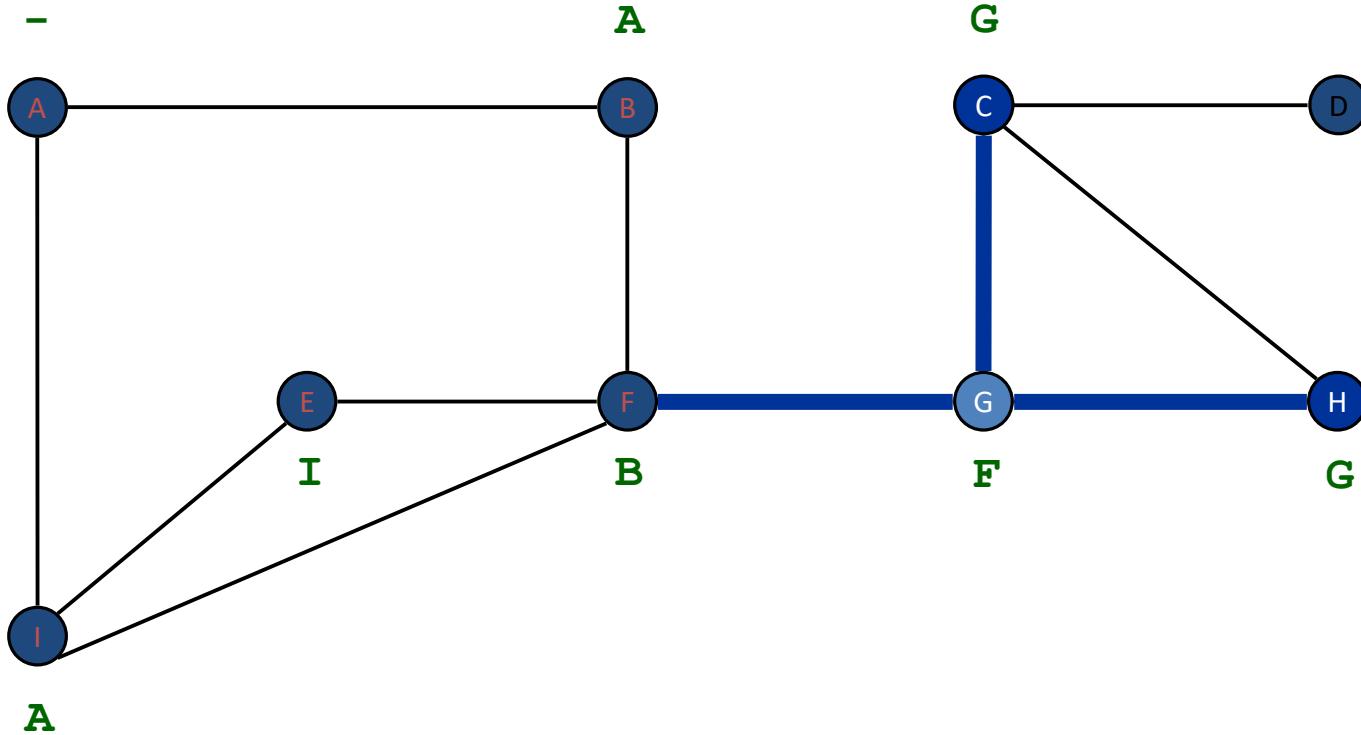
visit neighbors of G

front

C

FIFO Queue

Breadth First Search



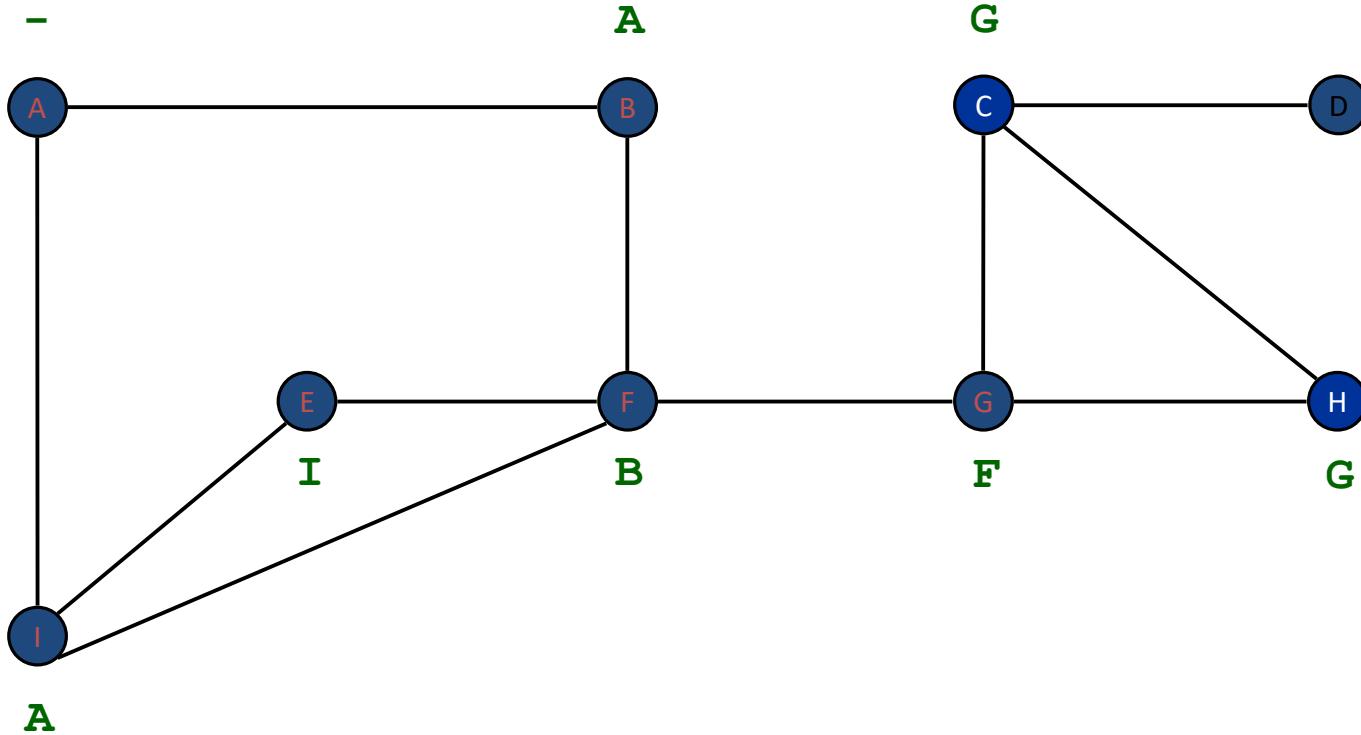
H discovered

front

C H

FIFO Queue

Breadth First Search



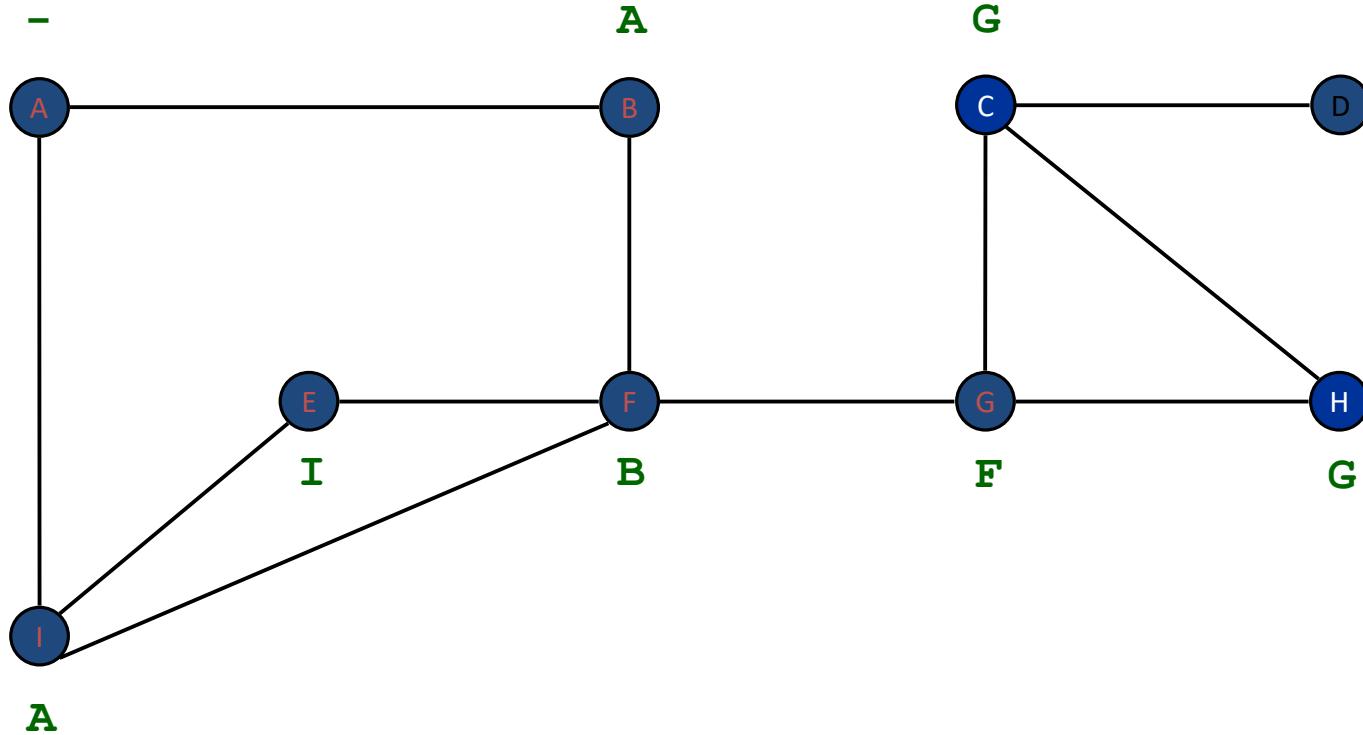
G finished

front

C H

FIFO Queue

Breadth First Search



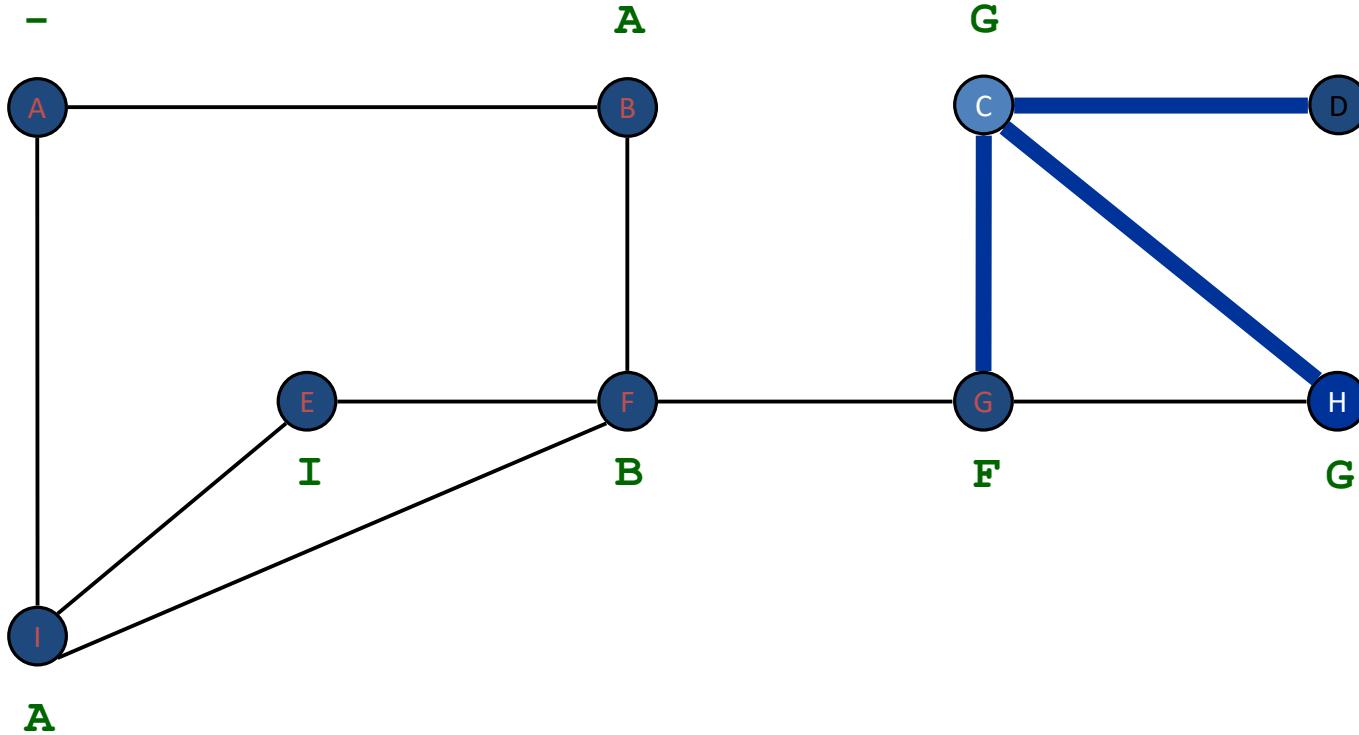
dequeue next vertex

front

C H

FIFO Queue

Breadth First Search



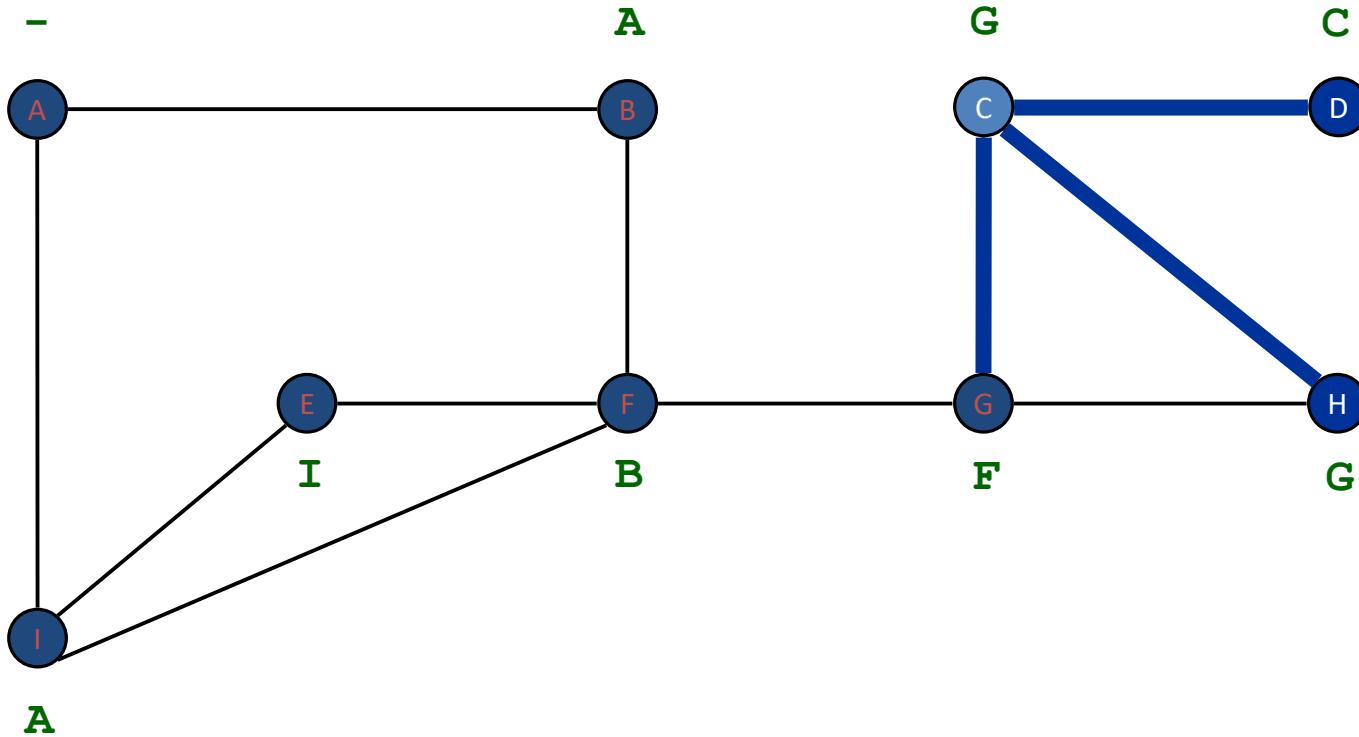
visit neighbors of C

front

H

FIFO Queue

Breadth First Search



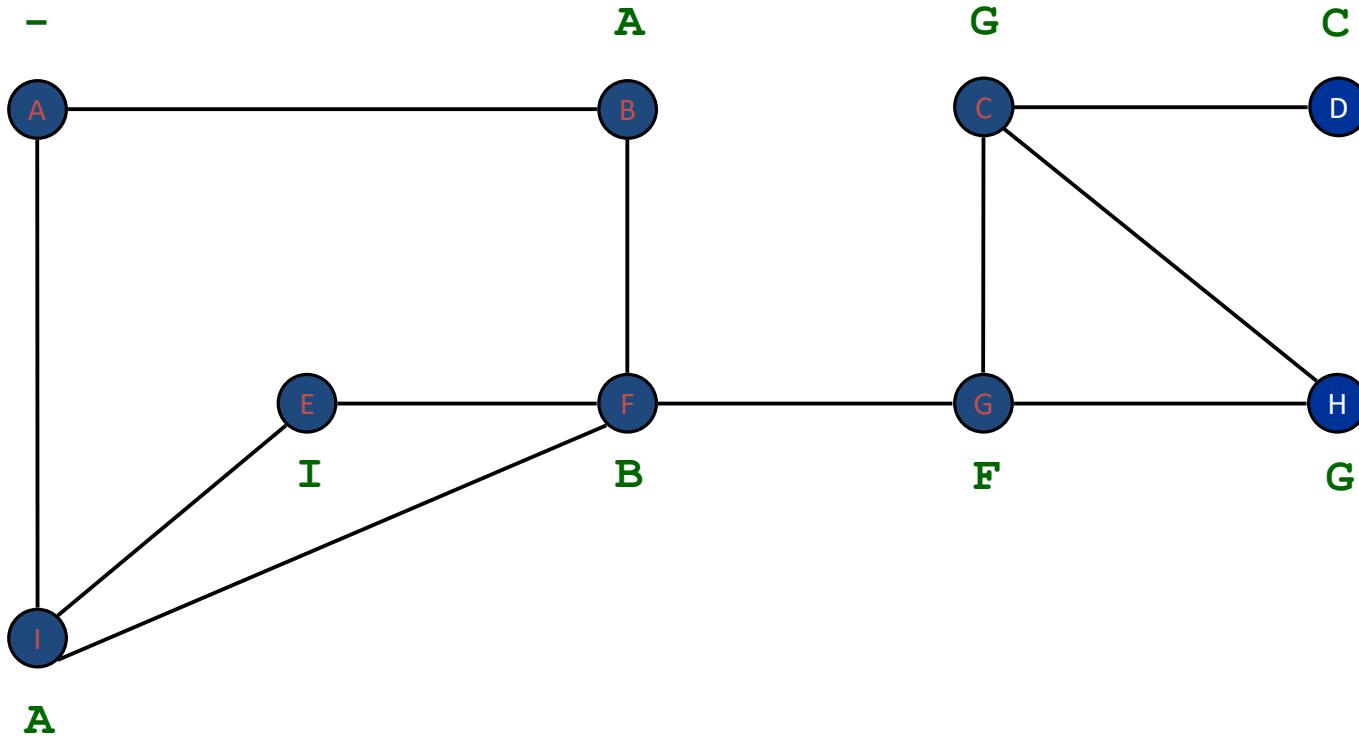
D discovered

front

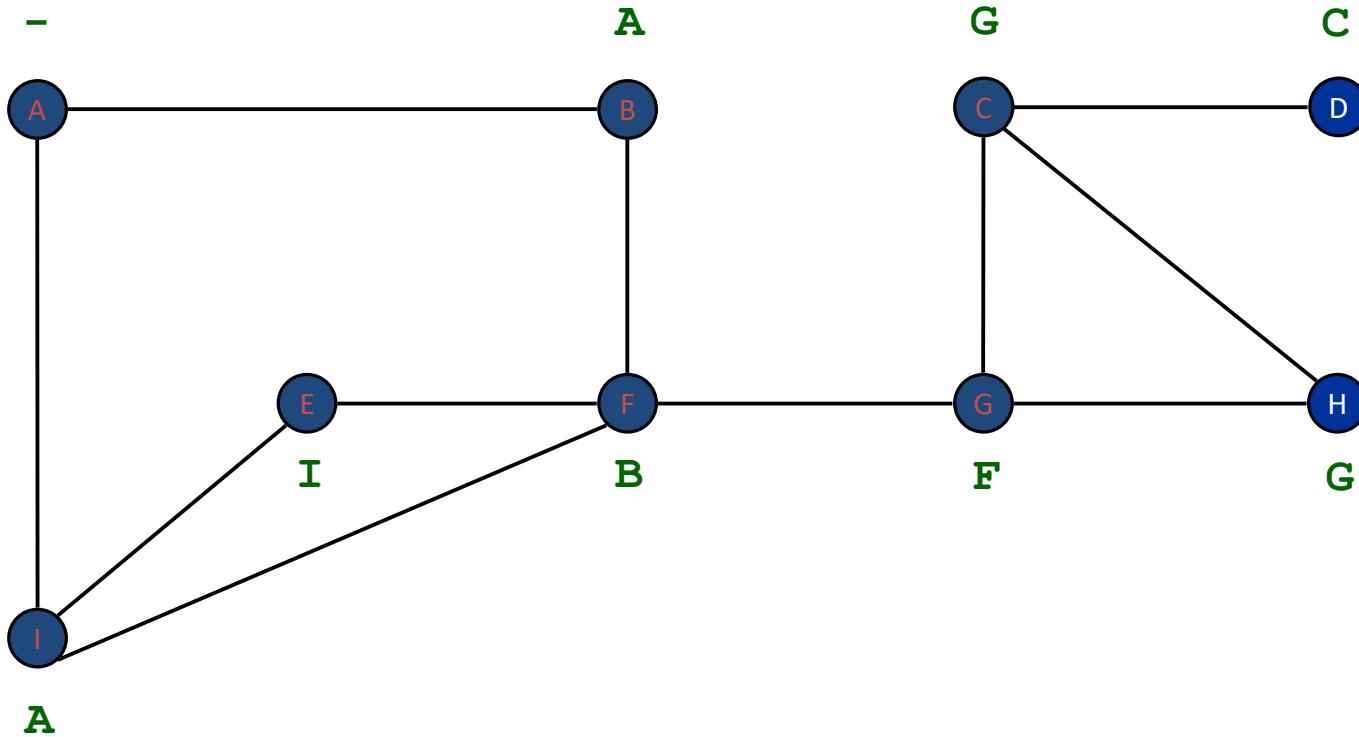
H D

FIFO Queue

Breadth First Search



Breadth First Search



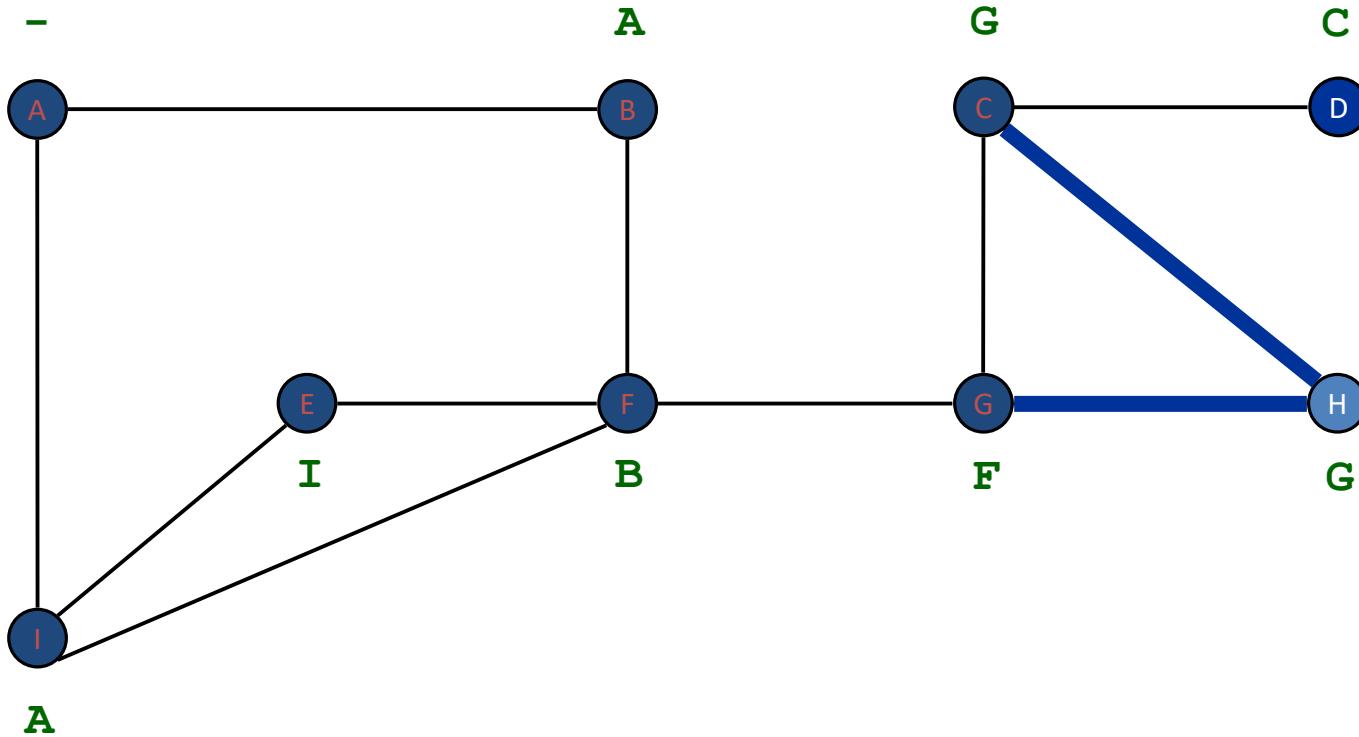
get next vertex

front

H D

FIFO Queue

Breadth First Search



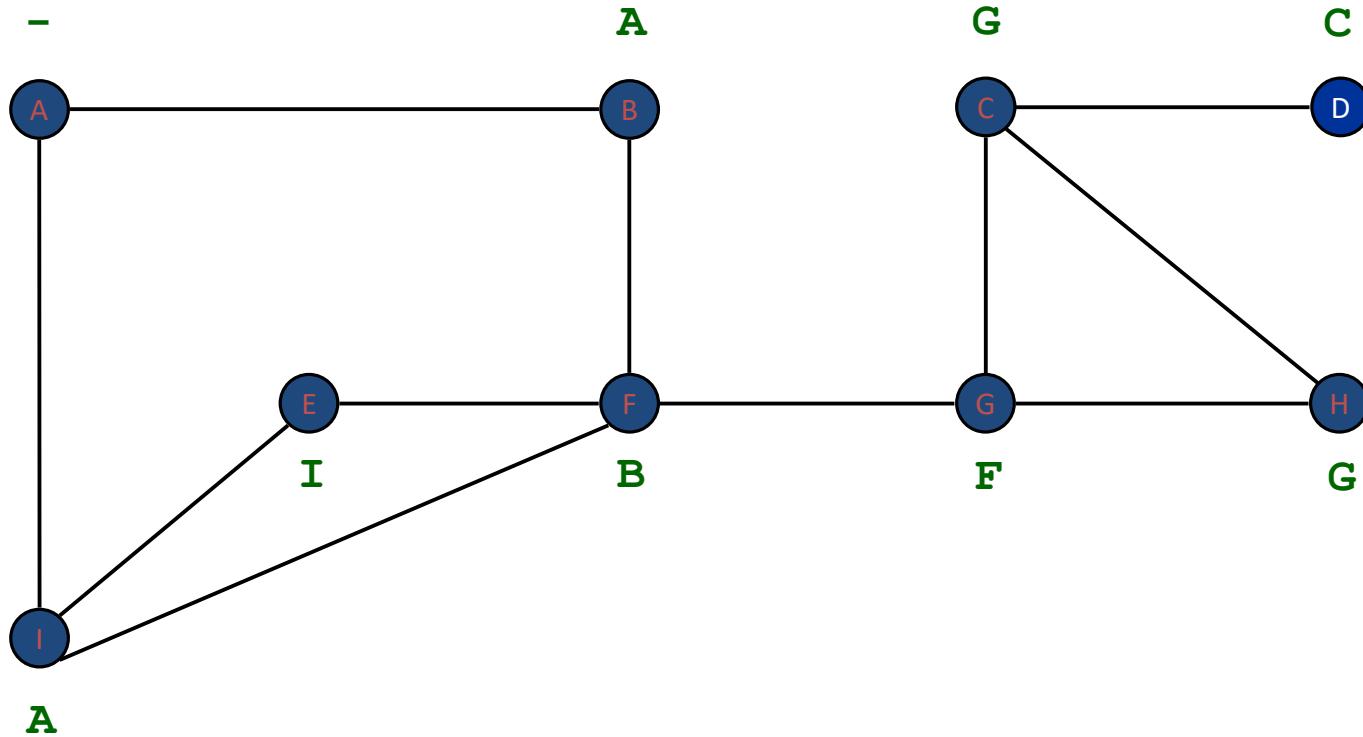
visit neighbors of H

front

D

FIFO Queue

Breadth First Search



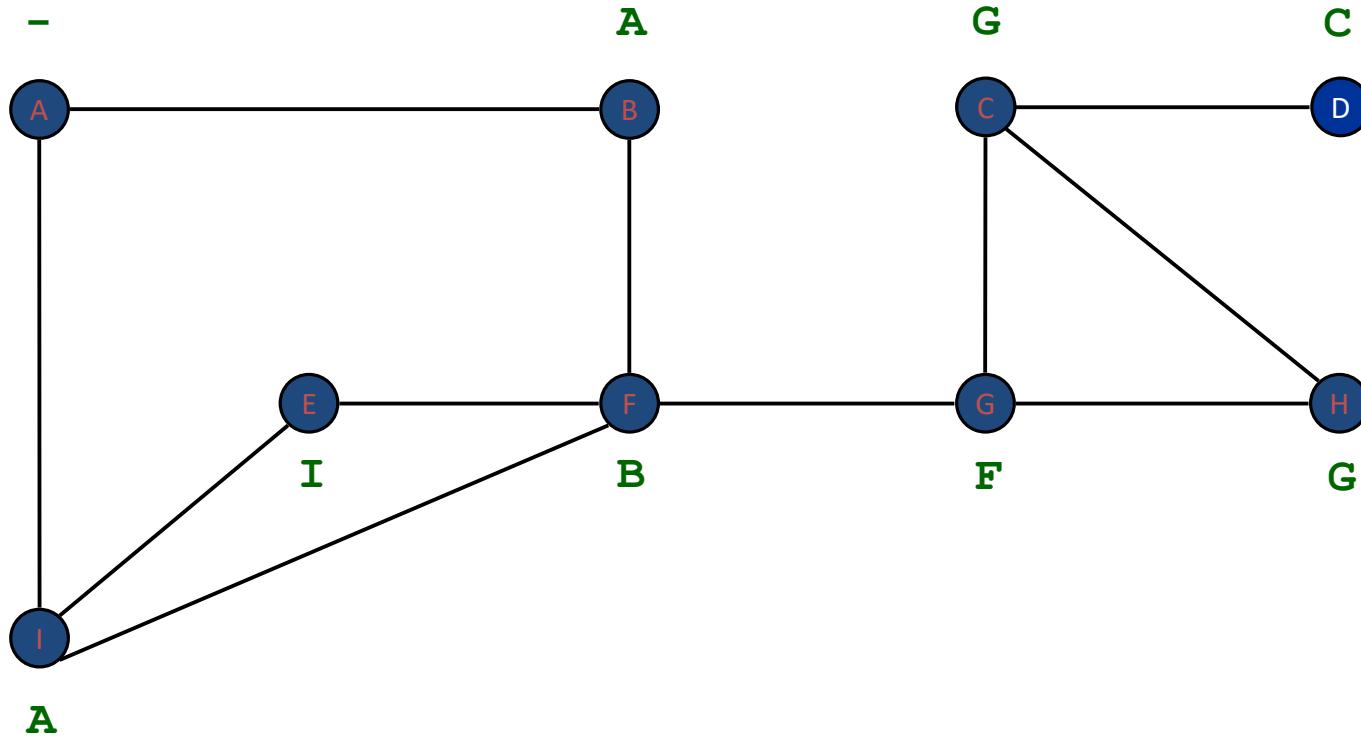
finished H

front

D

FIFO Queue

Breadth First Search



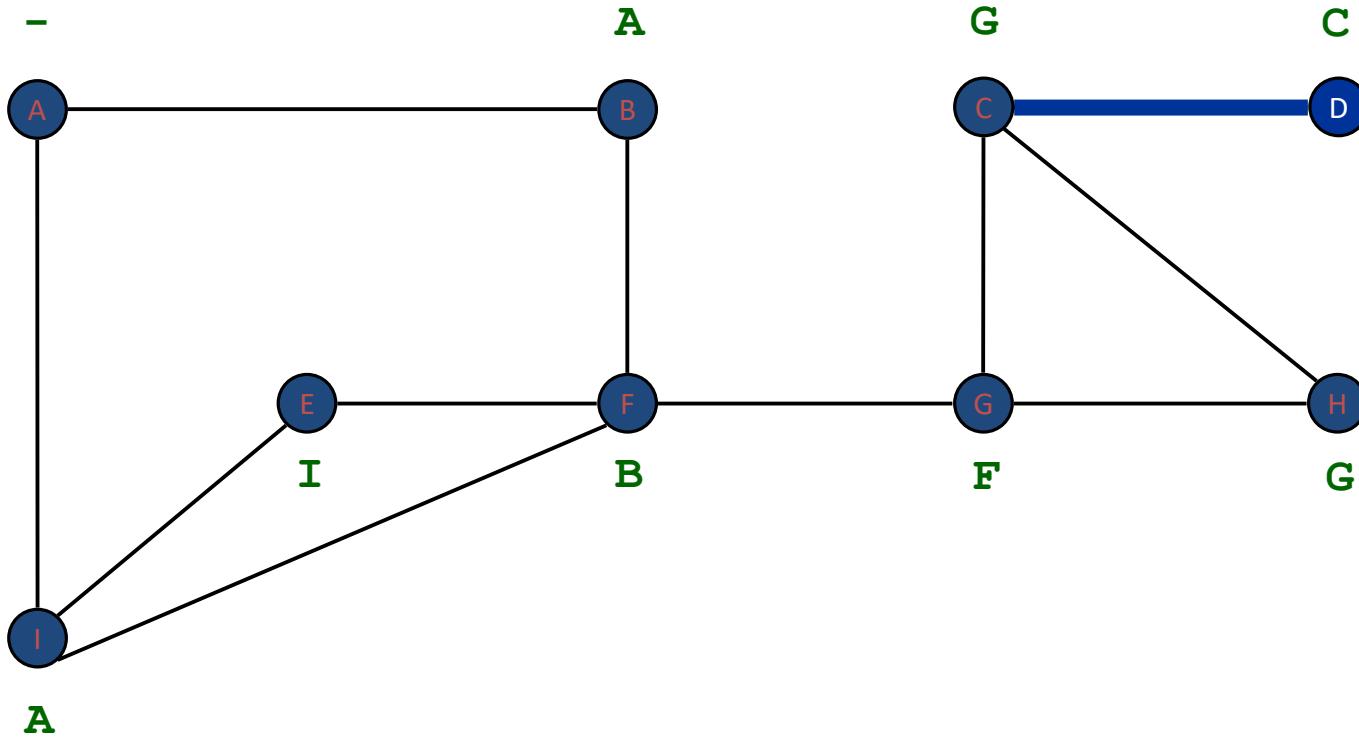
dequeue next vertex

front

D

FIFO Queue

Breadth First Search

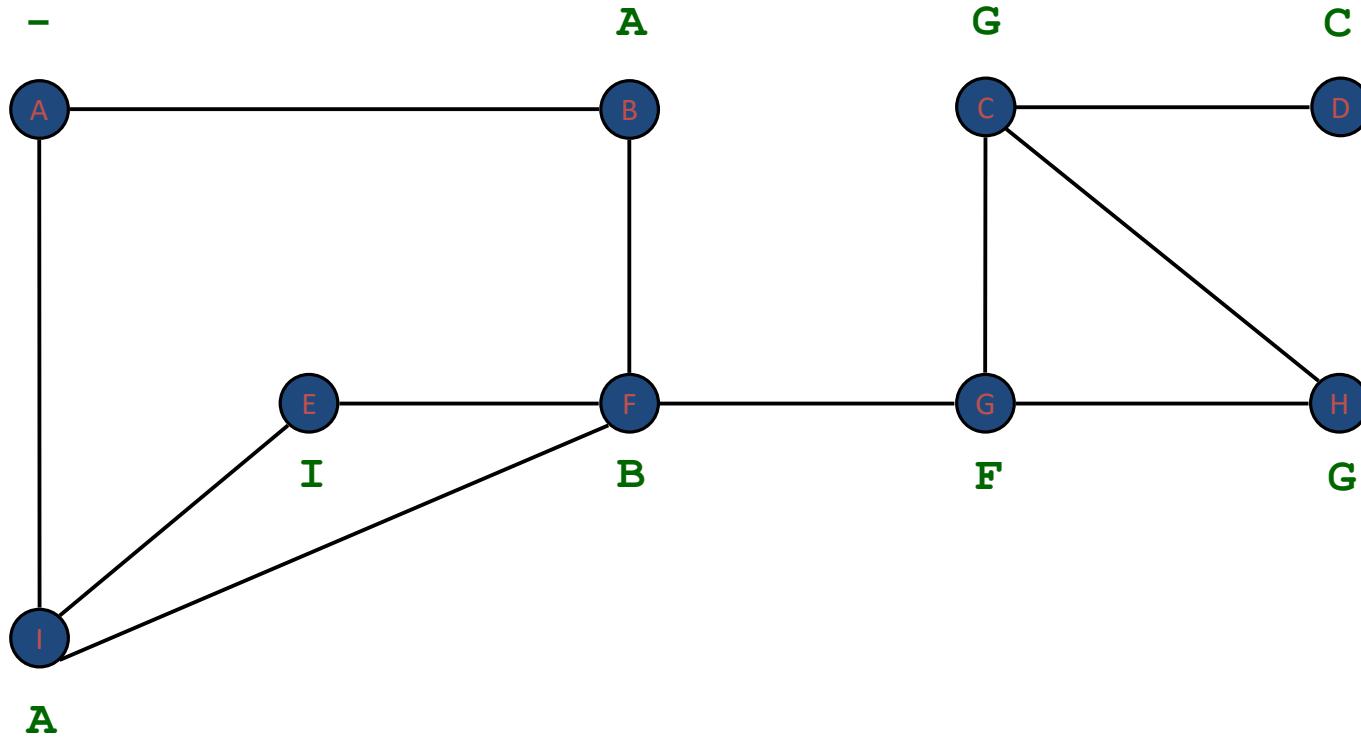


visit neighbors of D

front

FIFO Queue

Breadth First Search

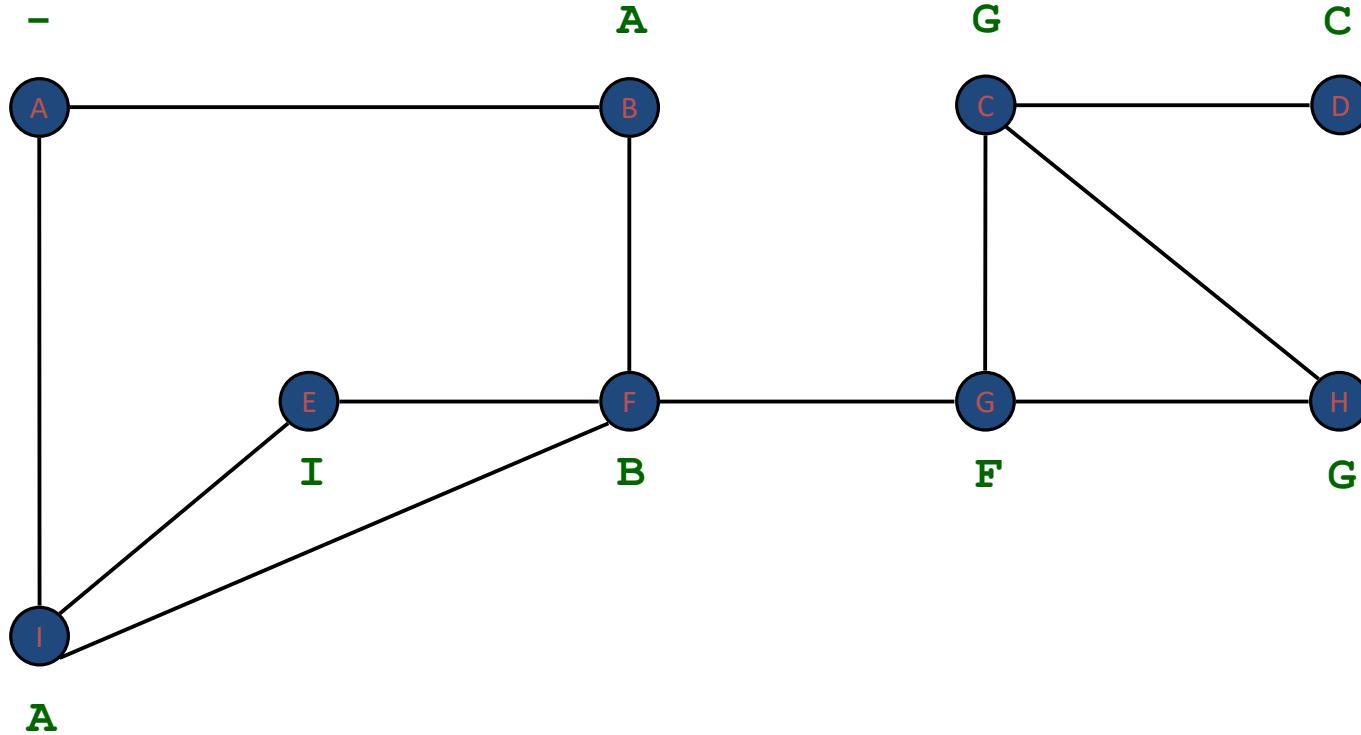


D finished

front

FIFO Queue

Breadth First Search

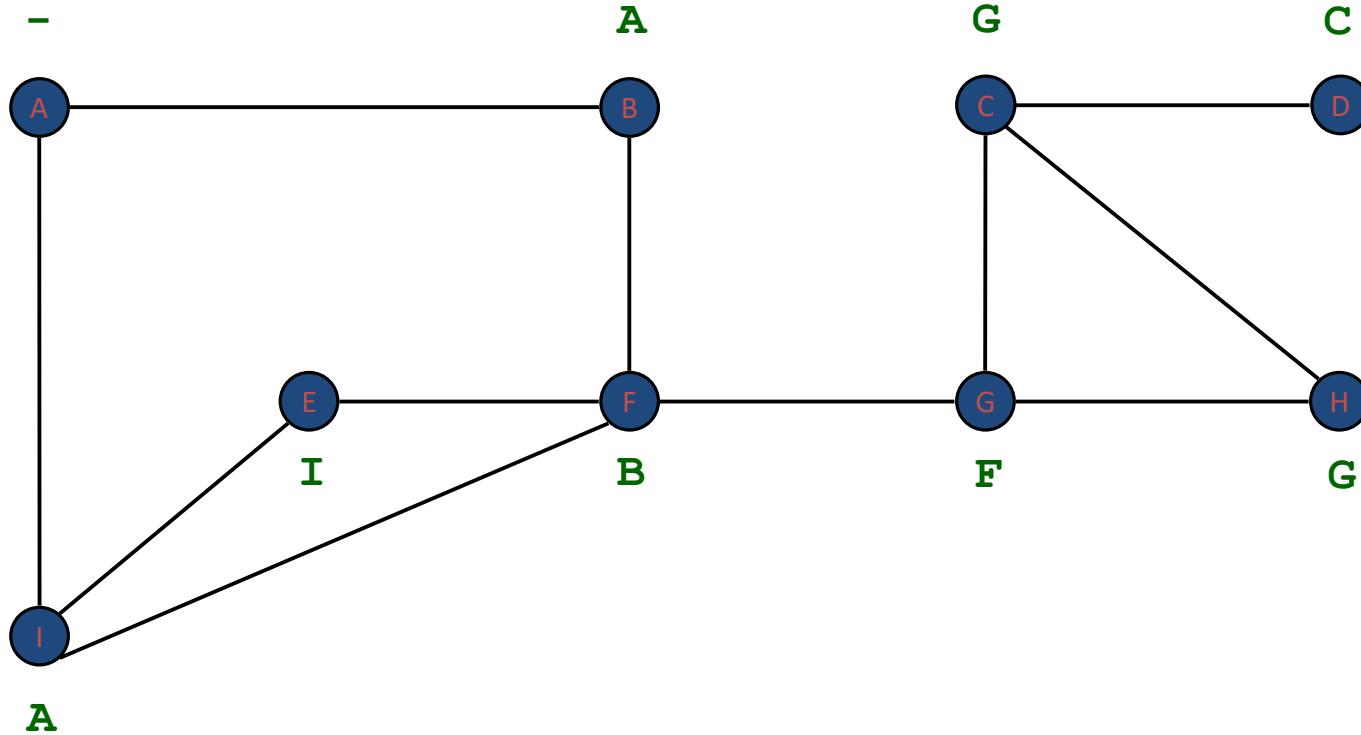


dequeue next vertex

front

FIFO Queue

Breadth First Search



STOP

front

FIFO Queue

Graph Traversal Techniques (cont'd)

BFS-Also(G,s)

for all v in $V[G]$ do

visited[v] := false

end for

$Q := \text{EmptyQueue}$

visited[s] := true //print s

Enqueue(Q,s)

while not **Empty**(Q) do

$u := \text{Dequeue}(Q)$

 for all w in $\text{Adj}[u]$ do

If not **visited**[w] **then**

visited[w] := true //print w

Enqueue(Q,w)

end if

end for

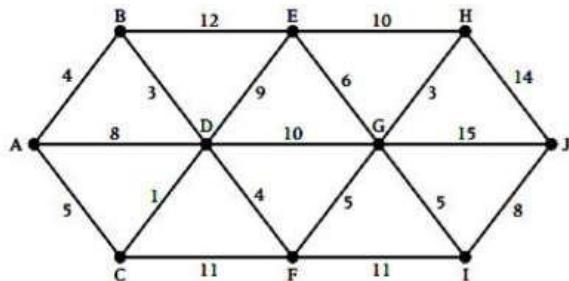
end while

- Assume an adjacency list representation, V is the number of vertices, E the number of edges.
- Each vertex is enqueued and dequeued at most once.
- Scanning for all adjacent vertices takes $O(|E|)$ time, since sum of lengths of adjacency lists is $|E|$.
- Hence The Time Complexity of BFS Gives a $O(|V|+|E|)$ time complexity.

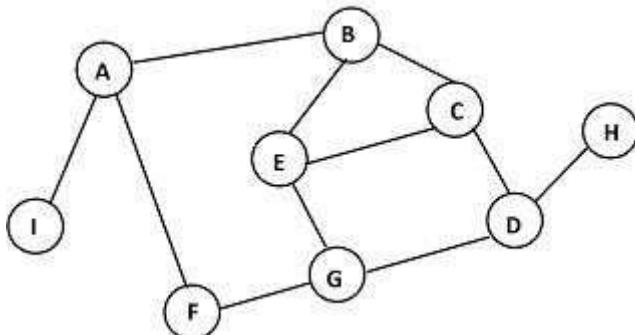
CSE2003 Data Structures and Algorithms

Practice questions given to Slow Learners

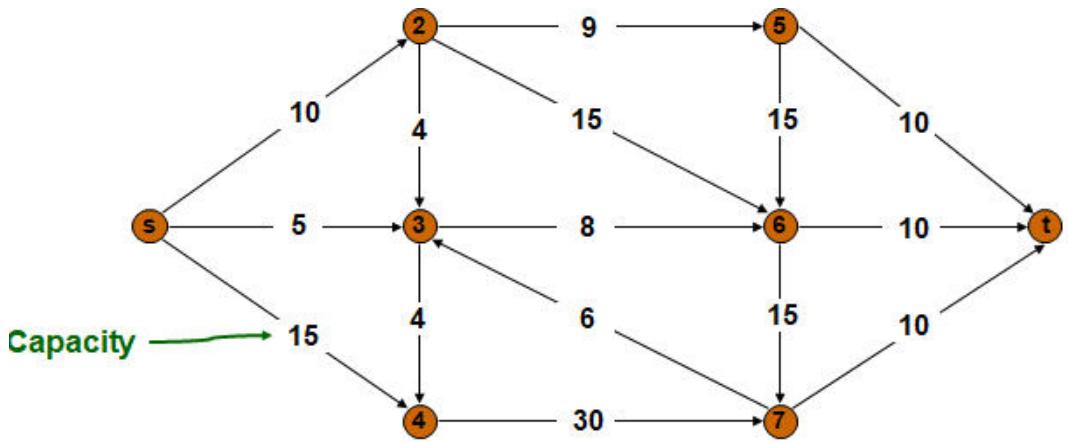
1. Assuming the table size as the smallest prime number greater than the input size, hash the following keys: [62, 56, 45, 14, 78, 44, 36, 29, 39]. To handle collision, use (i) linear probing and (ii) quadratic probing. Which of the two methods has less number of total probes?
2. Sort the following data using (MAX) heapsort: [20, 12, 35, 15, 10, 80, 30] and illustrate with appropriate figures for each iteration
3. Construct a Binary Search Tree for the following order of input (step by step construction is expected) [40, 29, 12, 34, 78, 54, 90, 57, 77, 44, 23, 11, 8, 19]
4. For the above Traverse the above BST through Inorder, Postorder and Preorder.
5. For the given graph $G=(V,E)$ find the shortest path from the source vertex A to all other vertices using Dijkstra's Algorithm



6. a) Perform the Depth First Search on the following graph starting from vertex A. Identify backward and cross edges (if any).



- b) Perform the Breadth First Search on the above graph starting from vertex G.
7. When a spanning tree is called a minimum spanning tree? Find the minimum spanning tree of this graph using Prim's algorithm



8.

Dynamic programming

Longest Common Subsequence

0-1 Knap Sack Problem

Divide and Conquer

- Divide and Conquer is an algorithmic paradigm. A typical Divide and Conquer algorithm solves a problem using following three steps.
- **1. *Divide*:** Break the given problem into sub problems of same type.
- **2. *Conquer*:** Recursively solve these sub problems
- **3. *Combine*:** Appropriately combine the answers

- Following are some standard algorithms that are Divide and Conquer algorithms.

1) Binary Search

2) Quicksort

3) Merge Sort

4) Closest Pair of Points

5) Strassen's Algorithm (Matrix Multiplication)

In divide and conquer approach,

- the problem in hand, is divided into smaller sub-problems and then **each problem is solved independently.**
- When we keep on dividing the sub problems into even smaller sub-problems, **we may eventually reach a stage where no more division is possible.**
- Those "atomic" smallest possible sub-problem (fractions) are solved.
- The **solution of all sub-problems is finally merged** in order **to obtain the solution of an original problem.**

Divide & Conquer

- They *call themselves recursively one or more times to deal with closely related sub problems.*
- D&C does more work on the sub-problems and hence has more time consumption.
- In D&C the sub problems are **independent of each other.**

Example: Merge Sort, Binary Search

Dynamic Programming (DP)

- Like divide-and-conquer, solve problem by combining the solutions to sub-problems.
- ***Differences between divide-and-conquer and DP:***

Divide&conquer -Independent sub-problems, solve sub-problems ***independently and recursively***, (so same sub(sub)problems solved ***repeatedly***)

DP- Sub-problems are ***dependent***, i.e., ***sub-problems share sub-sub-problems***, every ***sub(sub)problem solved just once, solutions to sub(sub)problems are stored in a table*** and used for solving higher level sub-problems.

- **Dynamic programming** is both a mathematical optimization method and a computer programming method.
- It refers to simplifying a complicated problem by breaking it down into simpler sub-problems in a recursive manner.
- While some decision problems cannot be taken apart this way, decisions that span several points in time do often break apart recursively.
- In computer science, if a problem can be solved optimally by breaking it into sub-problems and then recursively finding the optimal solutions to the sub-problems, then it is said to have optimal substructure.

- Wherever we see a recursive solution that has repeated calls for same inputs, we can optimize it using Dynamic Programming.
- The idea is to simply store the results of sub problems, so that we **do not have to re-compute them when needed later**. This simple **optimization reduces time complexities** from exponential to polynomial.

```
int fib(int n)
{
    if (n <= 1)
        return n;
    return fib(n-1) + fib(n-2);
}
```

Recursion : Exponential

```
f[0] = 0;
f[1] = 1;

for (i = 2; i <= n; i++)
{
    f[i] = f[i-1] + f[i-2];
}

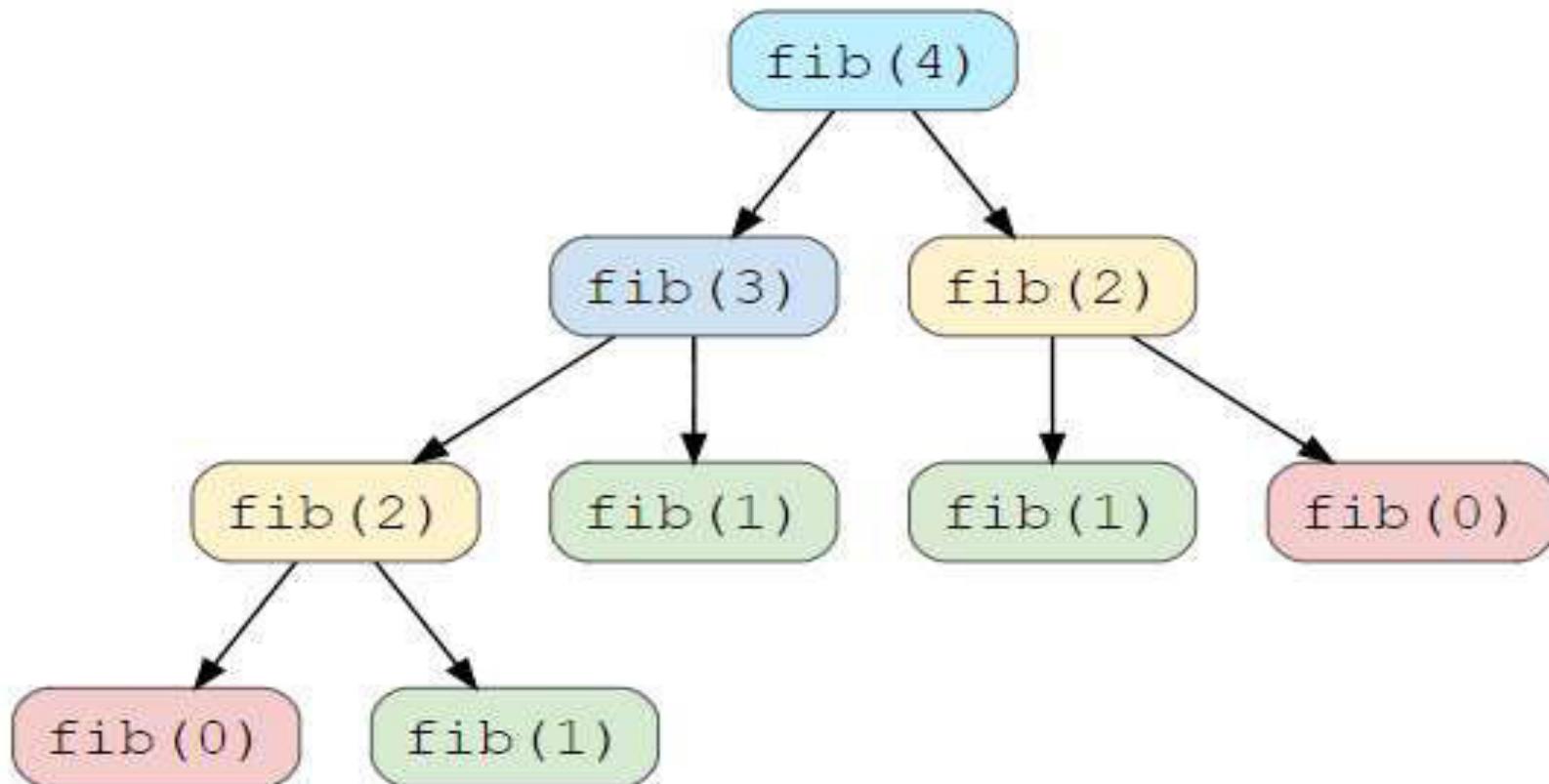
return f[n];
```

Dynamic Programming : Linear



- Characteristics of Dynamic Programming
 - 1. Overlapping Subproblems
 - 2. Optimal Substructure Property
- Dynamic Programming Methods
 - 1. Top-down with Memoization
 - 2. Bottom-up with Tabulation

- **Characteristics of Dynamic Programming**
- ## 1. Overlapping Sub problems
- Sub problems are smaller versions of the original problem. Any problem has overlapping sub-problems if finding its solution involves solving the same sub problem multiple times.
 - Take the example of the Fibonacci numbers; to find the $\text{fib}(4)$, we need to break it down into the following sub-problems:



Recursion tree for calculating Fibonacci numbers

the overlapping sub problem pattern **here**, as **fib(2)** has been called twice and **fib(1)** has been called three times.

- **2. Optimal Substructure Property #**
- Any problem has optimal substructure property if its overall optimal solution can be constructed from the optimal solutions of its sub problems.

For Fibonacci numbers, as we know,

- $\text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2)$

This clearly shows that a problem of size ‘n’ has been reduced to **sub problems of size ‘n-1’ and ‘n-2’**. Therefore, Fibonacci numbers **have optimal substructure property**.

- **Dynamic Programming Methods**

DP offers two methods to solve a problem.

1. Top-down with Memorization

- In this approach, we try to **solve the bigger problem by recursively finding the solution to smaller sub-problems**. Whenever we solve a sub-problem, we catch its result so that we don't end up solving it repeatedly if it's called multiple times. **Instead, we can just return the saved result**. This technique of storing the results of already solved sub problems is called **Memorization**.

- **2. Bottom-up with Tabulation #**
- Tabulation is the opposite of the top-down approach and avoids recursion. In this approach, we solve the problem “bottom-up” (i.e. by solving all the related sub-problems first). This is typically **done by filling up an n-dimensional table**. **Based on the results in the table, the solution to the top/original problem is then computed.**
- Tabulation is the opposite of **Memorization**, as in **Memorization** we **solve the problem and maintain a map of already solved sub-problems**. In other words, in ***memoization***, we **do it top-down** in the sense that we solve the top problem first (which typically recurses down to solve the sub-problems).

Dynamic programming

- It is used, when the solution can be recursively described in terms of solutions to subproblems (*optimal substructure*)
- Algorithm finds solutions to subproblems and stores them in memory for later use
- More efficient than “*brute-force methods*”, which solve the same subproblems over and over again

Application domain of DP

- Optimization problem: find a solution with optimal (maximum or minimum) value.
- *An* optimal solution, not *the* optimal solution, since there may more than one optimal solution, any one is OK.

Typical steps of DP

- Characterize the structure of an optimal solution.
- Recursively define the value of an optimal solution.
- Compute the value of an optimal solution in a bottom-up fashion.
- Compute an optimal solution from computed/stored information.

Longest Common Subsequence (LCS)

- DNA analysis, two DNA string comparison.
- DNA string: a sequence of symbols A,C,G,T.
 - $S=ACCGGTCGAGCTTCGAAT$
- Subsequence (of X): is X with some symbols left out.
 - $Z=CGTC$ is a subsequence of $X=ACGCTAC$.
- Common subsequence Z (of X and Y): a subsequence of X and also a subsequence of Y .
 - $Z=CGA$ is a common subsequence of both $X=ACGCTAC$ and $Y=CTGACA$.
- Longest Common Subsequence (LCS): the longest one of common subsequences.
 - $Z'=CGCA$ is the LCS of the above X and Y .
- LCS problem: given $X=\langle x_1, x_2, \dots, x_m \rangle$ and $Y=\langle y_1, y_2, \dots, y_n \rangle$, find their LCS.

LCS Intuitive Solution –brute force

- List all possible subsequences of X, check whether they are also subsequences of Y, keep the longer one each time.
- Each subsequence corresponds to a subset of the indices $\{1,2,\dots,m\}$, there are 2^m . So exponential.

Longest Common Subsequence (LCS)

Application: comparison of two DNA strings

Ex: $X = \{A B C B D A B\}$, $Y = \{B D C A B A\}$

Longest Common Subsequence:

$X = A \textcolor{green}{B} \quad \textcolor{green}{C} \quad \textcolor{green}{B} D \textcolor{green}{A} B$

$Y = \quad \textcolor{green}{B} D \textcolor{green}{C} A B \quad A$

Brute force algorithm would compare each subsequence of X with the symbols in Y

LCS Algorithm

- if $|X| = m, |Y| = n$, then there are 2^m subsequences of X ; we must compare each with Y (n comparisons)
- So the running time of the brute-force algorithm is $O(n 2^m)$
- Notice that the LCS problem has *optimal substructure*:
solutions of subproblems are parts of the final solution.
- **Subproblems:** “find LCS of pairs of *prefixes* of X and Y ”
- Define X_i, Y_j to be *the prefixes of X and Y of length i and j respectively*

LCS Algorithm

- First we'll find the length of LCS. Later we'll modify the algorithm to find LCS itself.
- Define X_i , Y_j to be the prefixes of X and Y of length i and j respectively
- Define $c[i,j]$ to be the length of LCS of X_i and Y_j
- Then the length of LCS of X and Y will be $c[m,n]$

$$c[i, j] = \begin{cases} c[i - 1, j - 1] + 1 & \text{if } x[i] = y[j], \\ \max(c[i, j - 1], c[i - 1, j]) & \text{otherwise} \end{cases}$$

Step 1: Characterizing a longest common subsequence

- Theorem 15.1 (Optimal substructure of an LCS)
 - Let $Z = \langle z_1, \dots, z_k \rangle$ be any LCS of X and Y .
 - 1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1} .
 - 2. If $x_m \neq y_n$, then $z_k \neq x_m \Rightarrow Z$ is an LCS of X_{m-1} and Y .
 - 3. If $x_m \neq y_n$, then $z_k \neq y_n \Rightarrow Z$ is an LCS of X and Y_{n-1} .

LCS recursive solution

$$c[i, j] = \begin{cases} c[i - 1, j - 1] + 1 & \text{if } x[i] = y[j], \\ \max(c[i, j - 1], c[i - 1, j]) & \text{otherwise} \end{cases}$$

- When we calculate $c[i, j]$, we consider two cases:
- **First case: $x[i]=y[j]$:** one more symbol in strings X and Y matches, so the length of LCS X_i and Y_j equals to the length of LCS of smaller strings X_{i-1} and Y_{j-1} , plus 1

LCS recursive solution

$$c[i, j] = \begin{cases} c[i - 1, j - 1] + 1 & \text{if } x[i] = y[j], \\ \max(c[i, j - 1], c[i - 1, j]) & \text{otherwise} \end{cases}$$

- We start with $i = j = 0$ (empty substrings of x and y)
- Since X_0 and Y_0 are empty strings, their LCS is always empty (i.e. $c[0,0] = 0$)
- LCS of empty string and any other string is empty, so for every i and j: $c[0, j] = c[i, 0] = 0$

LCS recursive solution

$$c[i, j] = \begin{cases} c[i - 1, j - 1] + 1 & \text{if } x[i] = y[j], \\ \max(c[i, j - 1], c[i - 1, j]) & \text{otherwise} \end{cases}$$

- **Second case:** $x[i] \neq y[j]$
- As symbols don't match, our solution is not improved, and the length of $\text{LCS}(X_i, Y_j)$ is the same as before (i.e. maximum of $\text{LCS}(X_i, Y_{j-1})$ and $\text{LCS}(X_{i-1}, Y_j)$)

Step 3: computing the length of an LCS

LCS-LENGTH(X, Y)

```
1   $m \leftarrow \text{length}[X]$ 
2   $n \leftarrow \text{length}[Y]$ 
3  for  $i \leftarrow 1$  to  $m$ 
4      do  $c[i, 0] \leftarrow 0$ 
5  for  $j \leftarrow 0$  to  $n$ 
6      do  $c[0, j] \leftarrow 0$ 
7  for  $i \leftarrow 1$  to  $m$ 
8      do for  $j \leftarrow 1$  to  $n$ 
9          do if  $x_i = y_j$ 
10             then  $c[i, j] \leftarrow c[i - 1, j - 1] + 1$ 
11              $b[i, j] \leftarrow \swarrow$ 
12         else if  $c[i - 1, j] \geq c[i, j - 1]$ 
13             then  $c[i, j] \leftarrow c[i - 1, j]$ 
14              $b[i, j] \leftarrow \uparrow$ 
15         else  $c[i, j] \leftarrow c[i, j - 1]$ 
16              $b[i, j] \leftarrow \leftarrow$ 
17 return  $c$  and  $b$ 
```

O(mn)

Table of c and b

j	0	1	2	3	4	5	6
i	y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	1	1	1	1	2	2
3	C	1	1	2	2	2	2
4	B	1	1	2	2	3	3
5	D	1	2	2	2	3	3
6	A	1	2	2	3	3	4
7	B	1	2	2	3	4	4

Solution :
BCBA

LCS Length Algorithm

LCS-Length(X, Y)

1. $m = \text{length}(X)$ // get the # of symbols in X
2. $n = \text{length}(Y)$ // get the # of symbols in Y
3. for $i = 1$ to m $c[i,0] = 0$ // special case: Y_0
4. for $j = 1$ to n $c[0,j] = 0$ // special case: X_0
5. for $i = 1$ to m // for all X_i
6. for $j = 1$ to n // for all Y_j
7. if ($X_i == Y_j$)
8. $c[i,j] = c[i-1,j-1] + 1$
9. else $c[i,j] = \max(c[i-1,j], c[i,j-1])$
10. return c

LCS Example

We'll see how LCS algorithm works on the following example:

- $X = ABCB$
- $Y = BDCAB$

What is the Longest Common Subsequence of X and Y?

$$\text{LCS}(X, Y) = BCB$$

$$X = A \ B \ C \ B$$

$$Y = \quad B \ D \ C \ A \ B$$

LCS Example (0)

j	0	1	2	3	4	5
i	Yj	B	D	C	A	B
0	Xi					
1	A					
2	B					
3	C					
4	B					

$$X = ABCB; \ m = |X| = 4$$

$$Y = BDCAB; \ n = |Y| = 5$$

Allocate array c[5,4]

LCS Example (1)

j	0	1	2	3	4	5
i	Yj	B	D	C	A	B
0	Xi	0	0	0	0	0
1	A	0				
2	B	0				
3	C	0				
4	B	0				

for $i = 1$ to m

$$c[i,0] = 0$$

for $j = 1$ to n

$$c[0,j] = 0$$

ABCB
BDCAB

LCS Example (2)

ABC
BDCAB

j	0	1	2	3	4	5
i	Yj	B	D	C	A	B
0	Xi	0	0	0	0	0
1	A	0	0			
2	B					
3	C					
4	B					

if ($X_i == Y_j$)
 $c[i,j] = c[i-1,j-1] + 1$
 else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (3)

ABC
BDCAB

j	0	1	2	3	4	5
i	Yj	B	D	C	A	B
0	Xi	0	0	0	0	0
1	A	0	0	0	0	
2	B	0				
3	C	0				
4	B	0				

```

if ( Xi == Yj )
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )

```

LCS Example (4)

ABC
BDCA
B

i	j	0	1	2	3	4	5
	Yj	B	D	C	A	B	
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	
2	B	0					
3	C	0					
4	B	0					

```

if (  $X_i == Y_j$  )
     $c[i,j] = c[i-1,j-1] + 1$ 
else  $c[i,j] = \max( c[i-1,j], c[i,j-1] )$ 

```

LCS Example (5)

ABC
BDCA B

j	0	1	2	3	4	5
i	Yj	B	D	C	A	B
	Xi	0	0	0	0	0
0	0	0	0	0	0	0
1	A	0	0	0	0	1 → 1
2	B	0				
3	C	0				
4	B	0				

```

if ( Xi == Yj )
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )

```

LCS Example (6)

ABC
BDCAB

j	0	1	2	3	4	5
i	Yj	B	D	C	A	B
Xi	0	0	0	0	0	0
A	0	0	0	0	1	1
2	B	0	1			
C	0					
4	B					

```

if (  $X_i == Y_j$  )
     $c[i,j] = c[i-1,j-1] + 1$ 
else  $c[i,j] = \max( c[i-1,j], c[i,j-1] )$ 

```

LCS Example (7)

ABC
BDCAB

i	j	0	1	2	3	4	5
	Yj	B	D	C	A	B	
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	
3	C	0					
4	B	0					

```

if (  $X_i == Y_j$  )
     $c[i,j] = c[i-1,j-1] + 1$ 
else  $c[i,j] = \max( c[i-1,j], c[i,j-1] )$ 

```

LCS Example (8)

ABC
BDCA B

j	0	1	2	3	4	5
i	Yj	B	D	C	A	B
0	Xi	0	0	0	0	0
1	A	0	0	0	0	1
2	B	0	0	0	1	1
3	C	0				
4	B	0				

$\text{if } (X_i == Y_j)$
 $c[i,j] = c[i-1,j-1] + 1$
 $\text{else } c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (10)

ABC
BD CAB

i	j	0	1	2	3	4	5
	Yj	B	D	C	A	B	
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1			
4	B	0					

```

if (  $X_i == Y_j$  )
     $c[i,j] = c[i-1,j-1] + 1$ 
else  $c[i,j] = \max( c[i-1,j], c[i,j-1] )$ 

```

LCS Example (11)

ABC
BD**C**A**B**

j	0	1	2	3	4	5
i	Yj	B	D	C	A	B
0	Xi	0	0	0	0	0
1	A	0	0	0	0	1
2	B	0	1	1	1	1
3	C	0	1	1	2	
4	B	0				

```

if ( Xi == Yj )
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )

```

LCS Example (12)

ABC
BDCAB

i	j	0	1	2	3	4	5
	Yj	Y	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0					

```

if (  $X_i == Y_j$  )
     $c[i,j] = c[i-1,j-1] + 1$ 
else  $c[i,j] = \max( c[i-1,j], c[i,j-1] )$ 

```

LCS Example (13)

ABCB
BDCAB

j	0	1	2	3	4	5
i	Yj	B	D	C	A	B
Xi	0	0	0	0	0	0
A	0	0	0	0	1	1
B	0	1	1	1	1	2
C	0	1	1	2	2	2
4	B	0	1			

```

if (  $X_i == Y_j$  )
     $c[i,j] = c[i-1,j-1] + 1$ 
else  $c[i,j] = \max( c[i-1,j], c[i,j-1] )$ 

```

LCS Example (14)

ABCB
BDCAB

i	j	0	1	2	3	4	5
	Yj	B	D	C	A	B	
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1	1	2	2	

```

if (  $X_i == Y_j$  )
     $c[i,j] = c[i-1,j-1] + 1$ 
else  $c[i,j] = \max( c[i-1,j], c[i,j-1] )$ 

```

LCS Example (15)

ABCB
BDCA_B

j	0	1	2	3	4	5
i	Y _j	B	D	C	A	B
0	X _i	0	0	0	0	0
1	A	0	0	0	0	1
2	B	0	1	1	1	2
3	C	0	1	1	2	2
4	B	0	1	1	2	3

if ($X_i == Y_j$)
 $c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Algorithm Running Time

- LCS algorithm calculates the values of each entry of the array $c[m,n]$
- So what is the running time?

$O(m*n)$

since each $c[i,j]$ is calculated in constant time, and there are $m*n$ elements in the array

How to find actual LCS

- So far, we have just found the *length* of LCS, but not LCS itself.
- We want to modify this algorithm to make it output Longest Common Subsequence of X and Y

Each $c[i,j]$ depends on $c[i-1,j]$ and $c[i,j-1]$
or $c[i-1, j-1]$

For each $c[i,j]$ we can say how it was acquired:

2	2
2	3

For example, here
 $c[i,j] = c[i-1,j-1] + 1 = 2+1=3$

How to find actual LCS - continued

- Remember that

$$c[i, j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

- So we can start from $c[m, n]$ and go backwards
- Whenever $c[i, j] = c[i-1, j-1] + 1$, remember $x[i]$ (because $x[i]$ is a part of LCS)
- When $i=0$ or $j=0$ (i.e. we reached the beginning), output remembered letters in reverse order

Finding LCS

i	j	0	1	2	3	4	5
	Yj	B	D	C	A	B	
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1 ←	1	1	1	2
3	C	0	1	1	2 ←	2	2
4	B	0	1	1	2	2	3

Finding LCS (2)

i	j	0	1	2	3	4	5
	Yj	B	D	C	A	B	
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1 ← 1	1	1	1	2
3	C	0	1	1	2 ← 2	2	2
4	B	0	1	1	2	2	3

LCS (reversed order): B C B

LCS (straight order): B C B
 (this string turned out to be a palindrome)

- Solve the problem of LCS , given
- $X= \langle A, B, C, B, D, A, B \rangle$
- $Y= \langle B, D, C, A, B, A \rangle$

Step 4: Constructing an LCS

PRINT-LCS(b, X, i, j)

- 1 **if** $i = 0$ or $j = 0$
- 2 **then return**
- 3 **if** $b[i, j] = “\nwarrow”$
- 4 **then** PRINT-LCS($b, X, i - 1, j - 1$)
- 5 print x_i
- 6 **elseif** $b[i, j] = “\uparrow”$
- 7 **then** PRINT-LCS($b, X, i - 1, j$)
- 8 **else** PRINT-LCS($b, X, i, j - 1$)

O(m+n)

Knapsack problem

Given some items, pack the knapsack to get the maximum total value.

Each item has some weight and some value.

Total weight that we can carry is no more than some fixed number W .

So we must consider weights of items as well as their value.

Item #	Weight	Value
1	1	8
2	3	6
3	5	5

Knapsack problem

There are two versions of the problem:

- (1) “0-1 knapsack problem” and
- (2) “Fractional knapsack problem”

- (1) **Items are indivisible**; you either take an item or not. Solved with *dynamic programming*
- (2) **Items are divisible**: you can take any fraction of an item. Solved with a *greedy algorithm*.

0-1 Knapsack problem

- Given a knapsack with maximum capacity W , and a set S consisting of n items
- Each item i has some **w_i** ; and benefit **b_i** (**all w_i and W are integer values**)
- Problem: How to pack the knapsack to achieve maximum total value of packed items?

0-1 Knapsack problem

- Problem, in other words, is **to find**

$$\max \sum_{i \in T} b_i \text{ subject to } \sum_{i \in T} w_i \leq W$$

- ◆ Each item i has some **weight w_i** and **benefit value b_i**
- ◆ The problem is called a “*0-1*” problem, because each item must be entirely accepted or rejected.

0-1 Knapsack problem: brute-force approach

Let's first solve this problem with a straightforward algorithm

- Since **there are n items, there are 2^n possible combinations of items.**
- We go through all combinations and find the one with maximum value and with total weight less or equal to W
- Running time will be $O(2^n)$

0-1 Knapsack problem: dynamic programming approach

- We can do better with an algorithm based on dynamic programming
- We need to carefully identify the subproblems

Defining a Subproblem

- Given a knapsack with maximum capacity W , and a set S consisting of n items
- Each item i has some weight w_i and benefit value b_i (**all w_i and W are integer values**)
- Problem: How to pack the knapsack to achieve maximum total value of packed items?

Defining a Subproblem

- We can do better with an algorithm based on dynamic programming
- We need to carefully identify the subproblems

Let's try this:

If items are labeled $1..n$, then a subproblem would be to find an optimal solution for

$$S_k = \{ \text{items labeled } 1, 2, \dots k \}$$

Defining a Subproblem

If items are labeled $1..n$, then a subproblem would be to find an optimal solution for S_k
 $= \{items labeled 1, 2, .. k\}$

- This is a reasonable sub problem definition.
- **The question is: can we describe the final solution (S_n) in terms of sub problems (S_k)?**
- Unfortunately, we can't do that.

Defining a Subproblem

$w_1 = 2$	$w_2 = 4$	$w_3 = 5$	$w_4 = 3$	$b_4 = 4$?
$b_1 = 3$	$b_2 = 5$	$b_3 = 8$			

Max weight: $W = 20$

For S_4 :

Total weight: 14

Maximum benefit: 20

$w_1 = 2$	$w_2 = 4$	$w_3 = 5$	$w_4 = 3$	$w_5 = 9$
$b_1 = 3$	$b_2 = 5$	$b_3 = 8$	$b_4 = 4$	$b_5 = 10$

For S_5 :

Total weight: 20

Maximum benefit: 26

Item #	Weight w_i	Benefit b_i
1	2	3
2	4	5
3	5	8
4	3	4
5	9	10

**Solution for S_4 is
not part of the
solution for S_5 !!!**

Defining a Subproblem

- As we have seen, the solution for S_4 is not part of the solution for S_5
- So our definition of a subproblem is flawed and we need another one!

Defining a Subproblem

- Given a knapsack with maximum capacity W , and a set S consisting of n items
- Each item i has some weight w_i and benefit value b_i (**all w_i and W are integer values**)
- Problem: How to pack the knapsack to achieve maximum total value of packed items?

Defining a Subproblem

- Let's add another parameter: V , which will represent the maximum weight for each subset of items
- The sub problem then will be to compute $V[k, w]$, i.e., to find an optimal solution for $S_k = \{items labeled 1, 2, .. k\}$ in a knapsack of size w

Recursive Formula for subproblems

- The subproblem will then be to compute $V[k, w]$, i.e., to find an optimal solution for $S_k = \{items labeled 1, 2, .. k\}$ in a knapsack of size w
- Assuming knowing $V[i, j]$, where $i=0, 1, 2, \dots k-1$, $j=0, 1, 2, \dots w$, how to derive $V[k, w]$?

Recursive Formula for subproblems (continued)

Recursive formula for subproblems:

$$V[k, w] = \begin{cases} V[k - 1, w] & \text{if } w_k > w \\ \max\{V[k - 1, w], V[k - 1, w - w_k] + b_k\} & \text{else} \end{cases}$$

It means, that the best subset of S_k that has total weight w is:

- 1) the best subset of S_{k-1} that has total weight $\leq w$,
or
- 2) the best subset of S_{k-1} that has total weight $\leq w - w_k$ plus the item k

Recursive Formula

$$V[k, w] = \begin{cases} V[k - 1, w] & \text{if } w_k > w \\ \max \{V[k - 1, w], V[k - 1, w - w_k] + b_k\} & \text{else} \end{cases}$$

- The best subset of S_k that has the total weight $\leq w$, either contains item k or not.
- ***First case:*** $w_k > w$. Item k can't be part of the solution, since if it was, the total weight would be $> w$, which is unacceptable.
- ***Second case:*** $w_k \leq w$. Then the item k can be in the solution, and we choose *the case with greater value*.

0-1 Knapsack Algorithm

for $w = 0$ to W

$$V[0,w] = 0$$

for $i = 1$ to n

$$V[i,0] = 0$$

for $i = 1$ to n

for $w = 0$ to W

if $w_i \leq w$ // item i can be part of the solution

if $b_i + V[i-1, w-w_i] > V[i-1, w]$

$$V[i,w] = b_i + V[i-1, w-w_i]$$

else

$$V[i,w] = V[i-1,w]$$

else $V[i,w] = V[i-1,w]$ // $w_i > w$

Running time

for $w = 0$ to W

$O(W)$

$V[0,w] = 0$

for $i = 1$ to n

$V[i,0] = 0$

for $i = 1$ to n

Repeat n times

for $w = 0$ to W

$O(W)$

< the rest of the code >

What is the running time of this algorithm?

$O(n * W)$

Remember that the brute-force algorithm
takes $O(2^n)$

Example

Let's run our algorithm on the following data:

$n = 4$ (# of elements)

$W = 5$ (max weight)

Elements (weight, benefit):

(2,3), (3,4), (4,5), (5,6)

Example (2)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1						
2						
3						
4						

for $w = 0$ to W

$$V[0,w] = 0$$

Example (3)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

for $i = 1$ to n
 $V[i,0] = 0$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=1$

$b_i=3$

$w_i=2$

$w=1$

$w-w_i = -1$

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0				
2	0					
3	0					
4	0					

if $w_i \leq w$ // item i can be part of the solution

 if $b_i + V[i-1, w-w_i] > V[i-1, w]$

$V[i, w] = b_i + V[i-1, w-w_i]$

 else

$V[i, w] = V[i-1, w]$

else $V[i, w] = V[i-1, w]$ // $w_i > w$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=1$

$b_i=3$

$w_i=2$

$w=2$

$w-w_i = 0$

Example (5)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3			
2	0					
3	0					
4	0					

if $w_i \leq w$ // item i can be part of the solution

if $b_i + V[i-1, w-w_i] > V[i-1, w]$

$V[i, w] = b_i + V[i-1, w - w_i]$

else

$V[i, w] = V[i-1, w]$

else $V[i, w] = V[i-1, w]$ // $w_i > w$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=1$

$b_i=3$

$w_i=2$

$w=3$

$w-w_i = 1$

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3		
2	0					
3	0					
4	0					

if $w_i \leq w$ // item i can be part of the solution

if $b_i + V[i-1, w-w_i] > V[i-1, w]$

$V[i, w] = b_i + V[i-1, w - w_i]$

else

$V[i, w] = V[i-1, w]$

else $V[i, w] = V[i-1, w]$ // $w_i > w$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=1$

$b_i=3$

$w_i=2$

$w=4$

$w - w_i = 2$

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	
2	0					
3	0					
4	0					

if $w_i \leq w$ // item i can be part of the solution

if $b_i + V[i-1, w-w_i] > V[i-1, w]$

$V[i, w] = b_i + V[i-1, w - w_i]$

else

$V[i, w] = V[i-1, w]$

else $V[i, w] = V[i-1, w]$ // $w_i > w$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=1$

$b_i=3$

$w_i=2$

$w=5$

$w - w_i = 3$

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0					
3	0					
4	0					

if $w_i \leq w$ // item i can be part of the solution

if $b_i + V[i-1, w-w_i] > V[i-1, w]$

$V[i, w] = b_i + V[i-1, w - w_i]$

else

$V[i, w] = V[i-1, w]$

else $V[i, w] = V[i-1, w]$ // $w_i > w$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=2$

$b_i=4$

$w_i=3$

$w=1$

$w - w_i = -2$

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0				
3	0					
4	0					

if $w_i \leq w$ // item i can be part of the solution

 if $b_i + V[i-1, w-w_i] > V[i-1, w]$

$V[i, w] = b_i + V[i-1, w - w_i]$

 else

$V[i, w] = V[i-1, w]$

else $V[i, w] = V[i-1, w]$ // $w_i > w$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=2$

$b_i=4$

$w_i=3$

$w=2$

$w-w_i = -1$

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3			
3	0					
4	0					

if $w_i \leq w$ // item i can be part of the solution

if $b_i + V[i-1, w-w_i] > V[i-1, w]$

$V[i, w] = b_i + V[i-1, w-w_i]$

else

$V[i, w] = V[i-1, w]$

else $V[i, w] = V[i-1, w]$ // $w_i > w$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=2$

$b_i=4$

$w_i=3$

$w=3$

$w - w_i = 0$

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4		
3	0					
4	0					

if $w_i \leq w$ // item i can be part of the solution

if $b_i + V[i-1, w-w_i] > V[i-1, w]$

$V[i, w] = b_i + V[i-1, w - w_i]$

else

$V[i, w] = V[i-1, w]$

else $V[i, w] = V[i-1, w]$ // $w_i > w$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=2$

$b_i=4$

$w_i=3$

$w=4$

$w - w_i = 1$

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	
3	0					
4	0					

if $w_i \leq w$ // item i can be part of the solution

if $b_i + V[i-1, w-w_i] > V[i-1, w]$

$V[i, w] = b_i + V[i-1, w - w_i]$

else

$V[i, w] = V[i-1, w]$

else $V[i, w] = V[i-1, w]$ // $w_i > w$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=2$

$b_i=4$

$w_i=3$

$w=5$

$w - w_i = 2$

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0					
4	0					

if $w_i \leq w$ // item i can be part of the solution

if $b_i + V[i-1, w-w_i] > V[i-1, w]$

$V[i, w] = b_i + V[i-1, w - w_i]$

else

$V[i, w] = V[i-1, w]$

else $V[i, w] = V[i-1, w]$ // $w_i > w$

Items:

- 1: (2,3)
- 2: (3,4)
- 3: (4,5)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4		
4	0					

$i=3$

$4: (5,6)$

$b_i=5$

$w_i=4$

$w= 1..3$

if $w_i \leq w$ // item i can be part of the solution

if $b_i + V[i-1, w-w_i] > V[i-1, w]$

$V[i, w] = b_i + V[i-1, w-w_i]$

else

$V[i, w] = V[i-1, w]$

else $V[i, w] = V[i-1, w]$ // $w_i > w$

Items:

- 1: (2,3)
- 2: (3,4)
- 3: (4,5)

Example (15)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	
4	0					

$i=3$ 4: (5,6)

$b_i=5$

$w_i=4$

$w= 4$

$w - w_i = 0$

if $w_i \leq w$ // item i can be part of the solution

if $b_i + V[i-1, w-w_i] > V[i-1, w]$

$V[i, w] = b_i + V[i-1, w - w_i]$

else

$V[i, w] = V[i-1, w]$

else $V[i, w] = V[i-1, w]$ // $w_i > w$

Items:

- 1: (2,3)
- 2: (3,4)
- 3: (4,5)

Example (16)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0					

$i=3$ 4: (5,6)

$b_i=5$

$w_i=4$

$w= 5$

$w - w_i = 1$

if $w_i \leq w$ // item i can be part of the solution

 if $b_i + V[i-1, w-w_i] > V[i-1, w]$

$V[i, w] = b_i + V[i-1, w - w_i]$

 else

$V[i, w] = V[i-1, w]$

else $V[i, w] = V[i-1, w]$ // $w_i > w$

Items:

- 1: (2,3)
- 2: (3,4)
- 3: (4,5)
- 4: (5,6)

$i=4$

$b_i=6$

$w_i=5$

$w= 1..4$

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	

if $w_i \leq w$ // item i can be part of the solution

if $b_i + V[i-1, w-w_i] > V[i-1, w]$

$V[i, w] = b_i + V[i-1, w-w_i]$

else

$V[i, w] = V[i-1, w]$

else $V[i, w] = V[i-1, w]$ // $w_i > w$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=4$

$b_i=6$

$w_i=5$

$w= 5$

$w - w_i = 0$

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

if $w_i \leq w$ // item i can be part of the solution

if $b_i + V[i-1, w-w_i] > V[i-1, w]$

$V[i, w] = b_i + V[i-1, w - w_i]$

else

$V[i, w] = V[i-1, w]$

else $V[i, w] = V[i-1, w]$ // $w_i > w$

Exercise

- P303 8.2.1 (a).
1. a. Apply the bottom-up dynamic programming algorithm to the following instance of the knapsack problem:

item	weight	value
1	3	\$25
2	2	\$20
3	1	\$15
4	4	\$40
5	5	\$50

, capacity $W = 6$.

- How to find out which items are in the optimal subset?

Comments

- This algorithm only finds the max possible value that can be carried in the knapsack
 - i.e., the value in $V[n,W]$
- To know the items that make this maximum value, an addition to this algorithm is necessary

How to find actual Knapsack Items

- All of the information we need is in the table.
- $V[n, W]$ is the maximal value of items that can be placed in the Knapsack.
- Let $i=n$ and $k=W$
 - if $V[i, k] \neq V[i-1, k]$ then
 - mark the i^{th} item as in the knapsack
 - $i = i-1, k = k-w_i$
 - else
 - $i = i-1$ // Assume the i^{th} item is not in the knapsack
 - // Could it be in the optimally packed knapsack?

Items:

- | |
|----------|
| 1: (2,3) |
| 2: (3,4) |
| 3: (4,5) |
| 4: (5,6) |

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

i=4

k=5

b_i=6

w_i=5

V[i,k] = 7

V[i-1,k] = 7

i=n, k=W

while i,k > 0

if V[i,k] ≠ V[i-1,k] then

mark the *i*th item as in the knapsack

i = *i*-1, *k* = *k*-*w_i*

else

i = *i*-1

Items:

- | |
|----------|
| 1: (2,3) |
| 2: (3,4) |
| 3: (4,5) |
| 4: (5,6) |

i\W	0	1	2	3	4	5	i=4
0	0	0	0	0	0	0	
1	0	0	3	3	3	3	
2	0	0	3	4	4	7	
3	0	0	3	4	5	7	$V[i,k] = 7$
4	0	0	3	4	5	7	$V[i-1,k] = 7$

i=n, k=W

while i,k > 0

if $V[i,k] \neq V[i-1,k]$ then

mark the i^{th} item as in the knapsack

$i = i-1, k = k-w_i$

else

$i = i-1$

Items:

- | |
|----------|
| 1: (2,3) |
| 2: (3,4) |
| 3: (4,5) |
| 4: (5,6) |

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

i=3

k=5

b_i=5

w_i=4

V[i,k] = 7

V[i-1,k] = 7

i=n, k=W

while i,k > 0

if V[i,k] ≠ V[i-1,k] then

mark the *i*th item as in the knapsack

i = *i*-1, *k* = *k*-*w_i*

else

i = *i*-1

Items:

- | |
|----------|
| 1: (2,3) |
| 2: (3,4) |
| 3: (4,5) |
| 4: (5,6) |

Finding the Items (4)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

$i=2$

$k=5$

$b_i=4$

$w_i=3$

$V[i,k] = 7$

$V[i-1,k] = 3$

$k - w_i = 2$

$i=n, k=W$

while $i, k > 0$

if $V[i,k] \neq V[i-1,k]$ then

mark the i^{th} item as in the knapsack

$i = i-1, k = k-w_i$

else

$i = i-1$

Items:

- 1: (2,3)
- 2: (3,4)
- 3: (4,5)
- 4: (5,6)

Finding the Items (5)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

i=1

k=2

$b_i = 3$

$w_i = 2$

$V[i,k] = 3$

$V[i-1,k] = 0$

$k - w_i = 0$

$i=n, k=W$

while $i, k > 0$

if $V[i,k] \neq V[i-1,k]$ then

mark the i^{th} item as in the knapsack

$i = i-1, k = k-w_i$

else

$i = i-1$

Items:

- | |
|----------|
| 1: (2,3) |
| 2: (3,4) |
| 3: (4,5) |
| 4: (5,6) |

Finding the Items (6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

i=0

k=0

The optimal knapsack should contain {1, 2}

i=n, k=W

while i,k > 0

if $V[i,k] \neq V[i-1,k]$ then

mark the n^{th} item as in the knapsack

$i = i-1, k = k-w_i$

else

$i = i-1$

Items:

- | |
|----------|
| 1: (2,3) |
| 2: (3,4) |
| 3: (4,5) |
| 4: (5,6) |

Finding the Items (7)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

$i=n, k=W$

while $i, k > 0$

if $V[i, k] \neq V[i-1, k]$ then

mark the n^{th} item as in the knapsack

$i = i-1, k = k - w_i$

else

$i = i-1$

The optimal
knapsack
should contain
 $\{1, 2\}$

Memorization (Memory Function Method)

- *Goal:*
 - *Solve only subproblems that are necessary and solve it only once*
- **Memorization** is another way to deal with overlapping subproblems in dynamic programming
- With memorization, we implement the algorithm ***recursively***:
 - If we encounter a new subproblem, we compute and store the solution.
 - If we encounter a subproblem we have seen, we look up the answer
- Most useful when the algorithm is easiest to implement recursively
 - Especially if we do not need solutions to all subproblems.

0-1 Knapsack Memory Function Algorithm

Conclusion

- Dynamic programming is a useful technique of solving certain kind of problems
- When the solution can be *recursively* described in terms of partial solutions, we can store these partial solutions and re-use them as necessary (memorization)
- Running time of dynamic programming algorithm vs. naïve algorithm:
 - 0-1 Knapsack problem: $O(W*n)$ vs. $O(2^n)$

Fractional Knapsack

- In this case, items can be broken into smaller pieces, hence the thief can select fractions of items.
- According to the problem statement,
 - There are n items in the store
 - Weight of i^{th} item $w_i > 0$
 - Profit for i^{th} item $p_i, i > 0$ and
 - Capacity of the Knapsack is W
- In this version of Knapsack problem, items can be broken into smaller pieces. So, the thief may take only a fraction x_i of i^{th} item.
$$0 \leq x_i \leq 1$$

The i^{th} item contributes the weight $x_i \cdot w_i$ to the total weight in the knapsack and profit $x_i \cdot p_i$ to the total profit.

- The i^{th} item contributes the weight $x_i \cdot w_i$ to the total weight in the knapsack and profit $x_i \cdot p_i$ to the total profit.
- Hence, the objective of this algorithm is to

$$\text{maximize } \sum(x_i \cdot p_i)$$

subject to constraint,

$$\sum (x_i \cdot w_i) \leq W$$

- It is clear that an optimal solution must fill the knapsack exactly, otherwise we could add a fraction of one of the remaining items and increase the overall profit.
- Thus, an optimal solution can be obtained by

$$\sum(x_i \cdot w_i) = W$$

- In this context, first we need to sort those items according to the value of p_i/w_i , so that $(p_{i+1}/w_{i+1}) \leq (p_i/w_i)$. Here, x is an array to store the fraction of items.

Problem with weight of knapsack

$$W = 60$$

Item	A	B	C	D
Profit	280	100	120	120
Weight	40	10	20	24
Ratio (π_i/w_i) ($\pi_i w_i$)	7	10	6	5

As the provided items are not sorted based on π/w_i . After sorting, the items are as shown in the following table.

Item	B	A	C	D
Profit	100	280	120	120
Weight	10	40	20	24
Ratio (π/w_i)	10	7	6	5

- After sorting all the items according to p_i/w_i . First all of **B** is chosen as weight of **B** is less than the capacity of the knapsack. Next, item **A** is chosen, as the available capacity of the knapsack is greater than the weight of **A**. Now, **C** is chosen as the next item. However, the whole item cannot be chosen as the remaining capacity of the knapsack is less than the weight of **C**.
- ***Hence, fraction of C (i.e. $(60 - 50)/20$) is chosen.***
- Now, the capacity of the Knapsack is equal to the selected items. Hence, no more item can be selected.
- ***The total weight of the selected items is $10 + 40 + 20 * (10/20) = 60$ (A + B+ fractional part of C)***
- ***And the total profit is $100 + 280 + 120 * (10/20) = 380 + 60 = 440$ (Corresponding profit of A,B, fractional part of C)***

Algorithm: Greedy-Fractional-Knapsack ($w[1..n]$,
 $p[1..n]$, W)

for $i = 1$ to n

 do $x[i] = 0$

 weight = 0

 for $i = 1$ to n

 if $weight + w[i] \leq W$ then

$x[i] = 1$

 weight = weight + $w[i]$

 else

$x[i] = (W - weight) / w[i]$

 weight = W

 break

 return x

If the provided items are already sorted into a decreasing order of p_i/w_i , then the while loop takes a time in **$O(n)$** ; Therefore, the total time including the sort is in **$O(n \log n)$**

Dynamic Programming

0-1 Knapsack problem

Module – 4 Algorithm Design Paradigms

- **Divide and Conquer** - Quick sort and Merge Sort
- **Brute Force** –Bubble sort and Selection sort
- **Greedy** – Activity Selection, Fractional Knapsack, Prims and Kruskal's
- **Backtracking**- Queens Problem, Subset-sum
- **Dynamic programming** – 0-1 Knapsack, **Longest Common Subsequence (LCS)**

Module 6 - Computational Complexity classes

- Definition with Example -Tractable and Intractable Problems, Decidable and Undecidable problems, Computational complexity Classes: P, NP, and NP-complete - Cooks Theorem(Definition), 3-CNF-SAT Problem, Reduction of 3-CNF-SAT to Clique Problem, Reduction of 3-CNF-SAT to Subset sum problem.

Knapsack problem

Given some items, pack the knapsack to get the maximum total value.

Each item has some weight and some value.

Total weight that we can carry is no more than some fixed number W .

So we must consider weights of items as well as their value.

Item #	Weight	Value
1	1	8
2	3	6
3	5	5

Knapsack problem

There are two versions of the problem:

- (1) “0-1 knapsack problem” and
- (2) “Fractional knapsack problem”

- (1) **Items are indivisible**; you either take an item or not. Solved with *dynamic programming*
- (2) **Items are divisible**: you can take any fraction of an item. Solved with a *greedy algorithm*.

0-1 Knapsack problem

- Given a knapsack with maximum capacity W , and a set S consisting of n items
- Each item i has some **weight w_i** , and **benefit value b_i** (all w_i and W are integer values)
- Problem: How to pack the knapsack to achieve maximum total value of packed items?

0-1 Knapsack problem

- Problem, in other words, is **to find**

$$\max \sum_{i \in T} b_i \text{ subject to } \sum_{i \in T} w_i \leq W$$

- ◆ Each item i has some **weight w_i** , and **benefit value b_i** ,
- ◆ The problem is called a “0-1” problem, because each item must be entirely accepted or rejected.

0-1 Knapsack problem: brute-force approach

Let's first solve this problem with a straightforward algorithm

- Since **there are n items, there are 2^n possible combinations of items.**
- We go through all combinations and find the one with maximum value and with total weight less or equal to W
- Running time will be $O(2^n)$

0-1 Knapsack problem: dynamic programming approach

- We can do better with an algorithm based on dynamic programming
- We need to carefully identify the subproblems

Defining a Subproblem

- Given a knapsack with maximum capacity W , and a set S consisting of n items
- Each item i has some weight w_i and benefit value b_i (**all w_i and W are integer values**)
- Problem: How to pack the knapsack to achieve maximum total value of packed items?

Defining a Subproblem

- We can do better with an algorithm based on dynamic programming
- We need to carefully identify the subproblems

Let's try this:

If items are labeled $1..n$, then a subproblem would be to find an optimal solution for

$$S_k = \{ \text{items labeled } 1, 2, \dots k \}$$

Defining a Subproblem

If items are labeled $1..n$, then a subproblem would be to find an optimal solution for S_k
 $= \{items labeled 1, 2, .. k\}$

- This is a reasonable sub problem definition.
- **The question is: can we describe the final solution (S_n) in terms of sub problems (S_k)?**
- Unfortunately, we can't do that.

Defining a Subproblem

$w_1 = 2$	$w_2 = 4$	$w_3 = 5$	$w_4 = 3$	$b_4 = 4$?
$b_1 = 3$	$b_2 = 5$	$b_3 = 8$			

Max weight: $W = 20$

For S_4 :

Total weight: 14

Maximum benefit: 20

$w_1 = 2$	$w_2 = 4$	$w_3 = 5$	$w_4 = 3$	$w_5 = 9$
$b_1 = 3$	$b_2 = 5$	$b_3 = 8$	$b_4 = 4$	$b_5 = 10$

For S_5 :

Total weight: 20

Maximum benefit: 26

Item #	Weight w_i	Benefit b_i
1	2	3
2	4	5
3	5	8
4	3	4
5	9	10

**Solution for S_4 is
not part of the
solution for S_5 !!!¹²**

Defining a Subproblem

- As we have seen, the solution for S_4 is not part of the solution for S_5
- So our definition of a subproblem is flawed and we need another one!

Defining a Subproblem

- Given a knapsack with maximum capacity W , and a set S consisting of n items
- Each item i has some weight w_i and benefit value b_i (**all w_i and W are integer values**)
- Problem: How to pack the knapsack to achieve maximum total value of packed items?

Defining a Subproblem

- Let's add another parameter: V , which will represent the maximum weight for each subset of items
- The sub problem then will be to compute $V[k, w]$, i.e., to find an optimal solution for $S_k = \{items labeled 1, 2, .. k\}$ in a knapsack of size w

Recursive Formula for subproblems

- The subproblem will then be to compute $V[k, w]$, i.e., to find an optimal solution for $S_k = \{items labeled 1, 2, .. k\}$ in a knapsack of size w
- Assuming knowing $V[i, j]$, where $i=0, 1, 2, \dots k-1$, $j=0, 1, 2, \dots w$, how to derive $V[k, w]$?

Recursive Formula for subproblems (continued)

Recursive formula for subproblems:

$$V[k, w] = \begin{cases} V[k-1, w] & \text{if } w_k > w \\ \max\{V[k-1, w], V[k-1, w-w_k] + b_k\} & \text{else} \end{cases}$$

It means, that the best subset of S_k that has total weight w is:

- 1) the best subset of S_{k-1} that has total weight $\leq w$,
or
- 2) the best subset of S_{k-1} that has total weight $\leq w - w_k$ plus the item k

Recursive Formula

$$V[k, w] = \begin{cases} V[k - 1, w] & \text{if } w_k > w \\ \max \{V[k - 1, w], V[k - 1, w - w_k] + b_k\} & \text{else} \end{cases}$$

- The best subset of S_k that has the total weight $\leq w$, either contains item k or not.
- ***First case:*** $w_k > w$. Item k can't be part of the solution, since if it was, the total weight would be $> w$, which is unacceptable.
- ***Second case:*** $w_k \leq w$. Then the item k can be in the solution, and we choose *the case with greater value*.

0-1 Knapsack Algorithm

for $w = 0$ to W

$V[0,w] = 0$

for $i = 1$ to n

$V[i,0] = 0$

for $i = 1$ to n

 for $w = 0$ to W

 if $w_i \leq w$ // item i can be part of the solution

 if $b_i + V[i-1, w-w_i] > V[i-1, w]$

$V[i,w] = b_i + V[i-1, w-w_i]$

 else

$V[i,w] = V[i-1,w]$

 else $V[i,w] = V[i-1,w]$ // $w_i > w$

Running time

for $w = 0$ to W

$O(W)$

$V[0,w] = 0$

for $i = 1$ to n

$V[i,0] = 0$

for $i = 1$ to n

Repeat n times

for $w = 0$ to W

$O(W)$

< the rest of the code >

What is the running time of this algorithm?

$O(n * W)$

Remember that the brute-force algorithm
takes $O(2^n)$

Example

Let's run our algorithm on the following data:

$n = 4$ (# of elements)

$W = 5$ (max weight)

Elements (**weight w_i , benefit b_i**):
 $(2,3), (3,4), (4,5), (5,6)$

Example (2)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1						
2						
3						
4						

for $w = 0$ to W
 $V[0,w] = 0$

Example (3)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

for $i = 1$ to n
 $V[i,0] = 0$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=1$

$b_i=3$

$w_i=2$

$w=1$

$w-w_i = -1$

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0				
2	0					
3	0					
4	0					

if $w_i \leq w$ // item i can be part of the solution

if $b_i + V[i-1, w-w_i] > V[i-1, w]$

$V[i, w] = b_i + V[i-1, w-w_i]$

else

$V[i, w] = V[i-1, w]$

else $V[i, w] = V[i-1, w]$ // $w_i > w$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=1$

$b_i=3$

$w_i=2$

$w=2$

$w-w_i = 0$

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3			
2	0					
3	0					
4	0					

if $w_i \leq w$ // item i can be part of the solution

if $b_i + V[i-1, w-w_i] > V[i-1, w]$

$V[i, w] = b_i + V[i-1, w - w_i]$

else

$V[i, w] = V[i-1, w]$

else $V[i, w] = V[i-1, w]$ // $w_i > w$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=1$

$b_i=3$

$w_i=2$

$w=3$

$w-w_i = 1$

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3		
2	0					
3	0					
4	0					

if $w_i \leq w$ // item i can be part of the solution

if $b_i + V[i-1, w-w_i] > V[i-1, w]$

$V[i, w] = b_i + V[i-1, w - w_i]$

else

$V[i, w] = V[i-1, w]$

else $V[i, w] = V[i-1, w]$ // $w_i > w$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=1$

$b_i=3$

$w_i=2$

$w=4$

$w - w_i = 2$

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	
2	0					
3	0					
4	0					

if $w_i \leq w$ // item i can be part of the solution

if $b_i + V[i-1, w-w_i] > V[i-1, w]$

$V[i, w] = b_i + V[i-1, w - w_i]$

else

$V[i, w] = V[i-1, w]$

else $V[i, w] = V[i-1, w]$ // $w_i > w$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=1$

$b_i=3$

$w_i=2$

$w=5$

$w - w_i = 3$

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0					
3	0					
4	0					

if $w_i \leq w$ // item i can be part of the solution

if $b_i + V[i-1, w-w_i] > V[i-1, w]$

$V[i, w] = b_i + V[i-1, w - w_i]$

else

$V[i, w] = V[i-1, w]$

else $V[i, w] = V[i-1, w]$ // $w_i > w$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=2$

$b_i=4$

$w_i=3$

$w=1$

$w-w_i = -2$

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0				
3	0					
4	0					

if $w_i \leq w$ // item i can be part of the solution

if $b_i + V[i-1, w-w_i] > V[i-1, w]$

$V[i, w] = b_i + V[i-1, w-w_i]$

else

$V[i, w] = V[i-1, w]$

else $V[i, w] = V[i-1, w]$ // $w_i > w$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=2$

$b_i=4$

$w_i=3$

$w=2$

$w-w_i = -1$

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3			
3	0					
4	0					

if $w_i \leq w$ // item i can be part of the solution

 if $b_i + V[i-1, w-w_i] > V[i-1, w]$

$V[i, w] = b_i + V[i-1, w-w_i]$

 else

$V[i, w] = V[i-1, w]$

else $V[i, w] = V[i-1, w]$ // $w_i > w$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=2$

$b_i=4$

$w_i=3$

$w=3$

$w - w_i = 0$

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4		
3	0					
4	0					

if $w_i \leq w$ // item i can be part of the solution

if $b_i + V[i-1, w-w_i] > V[i-1, w]$

$V[i, w] = b_i + V[i-1, w - w_i]$

else

$V[i, w] = V[i-1, w]$

else $V[i, w] = V[i-1, w]$ // $w_i > w$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=2$

$b_i=4$

$w_i=3$

$w=4$

$w - w_i = 1$

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	
3	0					
4	0					

if $w_i \leq w$ // item i can be part of the solution

if $b_i + V[i-1, w-w_i] > V[i-1, w]$

$V[i, w] = b_i + V[i-1, w - w_i]$

else

$V[i, w] = V[i-1, w]$

else $V[i, w] = V[i-1, w]$ // $w_i > w$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=2$

$b_i=4$

$w_i=3$

$w=5$

$w - w_i = 2$

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0					
4	0					

if $w_i \leq w$ // item i can be part of the solution

if $b_i + V[i-1, w-w_i] > V[i-1, w]$

$V[i, w] = b_i + V[i-1, w - w_i]$

else

$V[i, w] = V[i-1, w]$

else $V[i, w] = V[i-1, w]$ // $w_i > w$

Items:

- 1: (2,3)
- 2: (3,4)
- 3: (4,5)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4		
4	0					

$i=3$

$4: (5,6)$

$b_i=5$

$w_i=4$

$w= 1..3$

if $w_i \leq w$ // item i can be part of the solution

if $b_i + V[i-1, w-w_i] > V[i-1, w]$

$V[i, w] = b_i + V[i-1, w-w_i]$

else

$V[i, w] = V[i-1, w]$

else $V[i, w] = V[i-1, w]$ // $w_i > w$

Items:

- 1: (2,3)
- 2: (3,4)
- 3: (4,5)

Example (15)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	
4	0					

$i=3$ 4: (5,6)

$b_i=5$

$w_i=4$

$w= 4$

$w - w_i = 0$

if $w_i \leq w$ // item i can be part of the solution

if $b_i + V[i-1, w-w_i] > V[i-1, w]$

$V[i, w] = b_i + V[i-1, w - w_i]$

else

$V[i, w] = V[i-1, w]$

else $V[i, w] = V[i-1, w]$ // $w_i > w$

Items:

- 1: (2,3)
- 2: (3,4)
- 3: (4,5)

Example (16)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0					

$i=3$ 4: (5,6)

$b_i=5$

$w_i=4$

$w= 5$

$w - w_i = 1$

if $w_i \leq w$ // item i can be part of the solution

 if $b_i + V[i-1, w-w_i] > V[i-1, w]$

$V[i, w] = b_i + V[i-1, w - w_i]$

 else

$V[i, w] = V[i-1, w]$

else $V[i, w] = V[i-1, w]$ // $w_i > w$

Items:

- 1: (2,3)
- 2: (3,4)
- 3: (4,5)
- 4: (5,6)

$i=4$

$b_i=6$

$w_i=5$

$w= 1..4$

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	

if $w_i \leq w$ // item i can be part of the solution

if $b_i + V[i-1, w-w_i] > V[i-1, w]$

$V[i, w] = b_i + V[i-1, w-w_i]$

else

$V[i, w] = V[i-1, w]$

else $V[i, w] = V[i-1, w]$ // $w_i > w$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=4$

$b_i=6$

$w_i=5$

$w= 5$

$w - w_i = 0$

$i \setminus W$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

if $w_i \leq w$ // item i can be part of the solution

if $b_i + V[i-1, w-w_i] > V[i-1, w]$

$V[i, w] = b_i + V[i-1, w - w_i]$

else

$V[i, w] = V[i-1, w]$

else $V[i, w] = V[i-1, w]$ // $w_i > w$

Exercise

- P303 8.2.1 (a).
1. a. Apply the bottom-up dynamic programming algorithm to the following instance of the knapsack problem:

item	weight	value
1	3	\$25
2	2	\$20
3	1	\$15
4	4	\$40
5	5	\$50

, capacity $W = 6$.

- How to find out which items are in the optimal subset?

Comments

- This algorithm only finds the max possible value that can be carried in the knapsack
 - i.e., the value in $V[n,W]$
- To know the items that make this maximum value, an addition to this algorithm is necessary

How to find actual Knapsack Items

- All of the information we need is in the table.
- $V[n, W]$ is the maximal value of items that can be placed in the Knapsack.
- ***Let $i=n$ and $k=W$***
if $V[i,k] \neq V[i-1,k]$ then
mark the i^{th} item as in the knapsack
$i = i-1$, $k = k-w_i$,
else
$i = i-1$ // Assume the i^{th} item is not in the knapsack
// Could it be in the optimally packed
knapsack?

Items:

- | |
|----------|
| 1: (2,3) |
| 2: (3,4) |
| 3: (4,5) |
| 4: (5,6) |

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

i=4

k=5

$b_i = 6$

$w_i = 5$

$V[i,k] = 7$

$V[i-1,k] = 7$

i=n, k=W

while i,k > 0

if $V[i,k] \neq V[i-1,k]$ then

mark the i^{th} item as in the knapsack

$i = i-1, k = k-w_i$

else

$i = i-1$

Items:

Finding the Items (2)

i\W	0	1	2	3	4	5	i=4
0	0	0	0	0	0	0	
1	0	0	3	3	3	3	
2	0	0	3	4	4	7	
3	0	0	3	4	5	7	
4	0	0	3	4	5	7	

k=5

b_i=6

w_i=5

V[i,k] = 7

V[i-1,k] = 7

i=n, k=W

while i,k > 0

if V[i,k] ≠ V[i-1,k] then

mark the *i*th item as in the knapsack

i = *i*-1, *k* = *k*-*w_i*

else

i = *i*-1

Items:

- | |
|----------|
| 1: (2,3) |
| 2: (3,4) |
| 3: (4,5) |
| 4: (5,6) |

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

i=3

k=5

b_i=5

w_i=4

V[i,k] = 7

V[i-1,k] = 7

i=n, k=W

while i,k > 0

if V[i,k] ≠ V[i-1,k] then

mark the *i*th item as in the knapsack

i = *i*-1, *k* = *k*-*w_i*

else

i = *i*-1

Items:

- | |
|----------|
| 1: (2,3) |
| 2: (3,4) |
| 3: (4,5) |
| 4: (5,6) |

Finding the Items (4)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

$i=2$

$k=5$

$b_i=4$

$w_i=3$

$V[i,k] = 7$

$V[i-1,k] = 3$

$k - w_i = 2$

$i=n, k=W$

while $i, k > 0$

if $V[i,k] \neq V[i-1,k]$ then

mark the i^{th} item as in the knapsack

$i = i-1, k = k-w_i$

else

$i = i-1$

Items:

- | |
|----------|
| 1: (2,3) |
| 2: (3,4) |
| 3: (4,5) |
| 4: (5,6) |

Finding the Items (5)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

$$i=1$$

$$k=2$$

$$b_i=3$$

$$w_i=2$$

$$V[i,k] = 3$$

$$V[i-1,k] = 0$$

$$k - w_i = 0$$

$$i=n, k=W$$

$$\text{while } i, k > 0$$

if $V[i,k] \neq V[i-1,k]$ then

mark the i^{th} item as in the knapsack

$i = i-1, k = k-w_i$

else

$i = i-1$

Items:

- 1: (2,3)
- 2: (3,4)
- 3: (4,5)
- 4: (5,6)

Finding the Items (6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

i=0

k=0

The optimal knapsack should contain {1, 2}

i=n, k=W

while i,k > 0

if $V[i,k] \neq V[i-1,k]$ then

mark the n^{th} item as in the knapsack

$i = i-1, k = k-w_i$

else

$i = i-1$

Items:

- 1: (2,3)
- 2: (3,4)
- 3: (4,5)
- 4: (5,6)

Finding the Items (7)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

$i=n, k=W$

while $i, k > 0$

if $V[i, k] \neq V[i-1, k]$ then

mark the n^{th} item as in the knapsack

$i = i-1, k = k - w_i$

else

$i = i-1$

The optimal knapsack should contain {1, 2}

Memorization (Memory Function Method)

- *Goal:*
 - *Solve only subproblems that are necessary and solve it only once*
- **Memorization** is another way to deal with overlapping subproblems in dynamic programming
- With memorization, we implement the algorithm ***recursively***:
 - If we encounter a new subproblem, we compute and store the solution.
 - If we encounter a subproblem we have seen, we look up the answer
- Most useful when the algorithm is easiest to implement recursively
 - Especially if we do not need solutions to all subproblems.

0-1 Knapsack Memory Function Algorithm

```
for i = 1 to n          MFKnapsack(i, w)
    for w = 1 to W
        V[i,w] = -1
    for w = 0 to W
        V[0,w] = 0
    for i = 1 to n
        V[i,0] = 0
            if V[i,w] < 0
                if w < wi
                    value = MFKnapsack(i-1, w)
                else
                    value = max(MFKnapsack(i-1, w),
                                bi + MFKnapsack(i-1, w-wi))
                V[i,w] = value
            return V[i,w]
```

Conclusion

- Dynamic programming is a useful technique of solving certain kind of problems
- When the solution can be *recursively* described in terms of partial solutions, we can store these partial solutions and re-use them as necessary (memorization)
- Running time of dynamic programming algorithm vs. naïve algorithm:
 - 0-1 Knapsack problem: $O(W*n)$ vs. $O(2^n)$

DIVIDE AND CONQUER

- SORTING ALGORITHM

Divide and Conquer

- Divide and Conquer is an algorithmic paradigm. A typical Divide and Conquer algorithm solves a problem using following three steps.
- **1. *Divide*:** Break the given problem into sub problems of same type.
- **2. *Conquer*:** Recursively solve these sub problems
- **3. *Combine*:** Appropriately combine the answers

- Following are some standard algorithms that are Divide and Conquer algorithms.

1) Binary Search

2) Quicksort

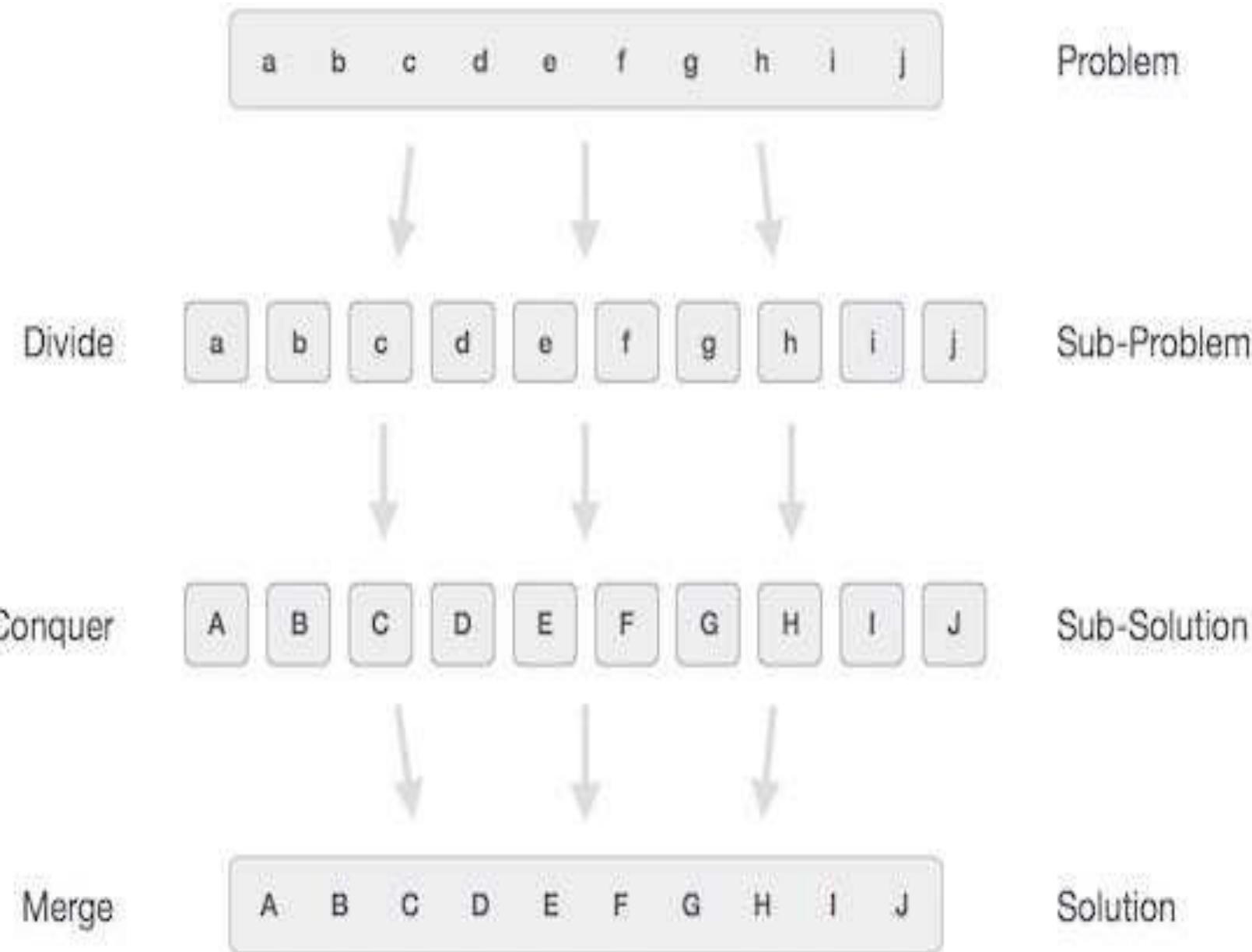
3) Merge Sort

4) Closest Pair of Points

5) Strassen's Algorithm (Matrix Multiplication)

In divide and conquer approach,

- the problem in hand, is divided into smaller sub-problems and then **each problem is solved independently.**
- When we keep on dividing the sub problems into even smaller sub-problems, **we may eventually reach a stage where no more division is possible.**
- Those "atomic" smallest possible sub-problem (fractions) are solved.
- The **solution of all sub-problems is finally merged** in order **to obtain the solution of an original problem.**



Sorting

What?

- Process of re-arranging a given set of objects in a specific order {either ascending or descending}

Why?

- Efficient to search in a ordered dataset than an unordered data set
- Relevant and Essential activity in data processing

TYPES

- Insertion Sort
- Quick Sort
- Merge Sort
- Shell Sort
- Bucket Sort
- Selection sort
- Heap Sort

Divide & Conquer

The divide-and-conquer paradigm involves three steps at each level of the recursion

- **Divide** - It first divides the problem into small chunks or sub-problems.
- **Conquer** - It then solve those sub-problems recursively so as to obtain a separate result for each sub-problem.
- **Combine** - It then combine the results of those sub-problems to arrive at a final result of the main problem.
- Some Divide and Conquer algorithms are Merge Sort, Quick Sort, Binary Search, etc.

Divide & Conquer

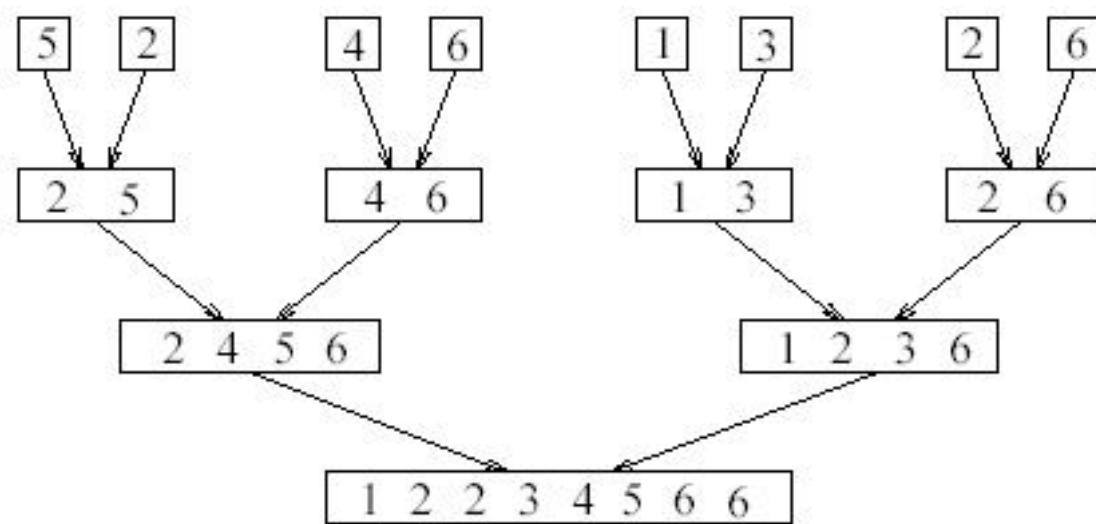
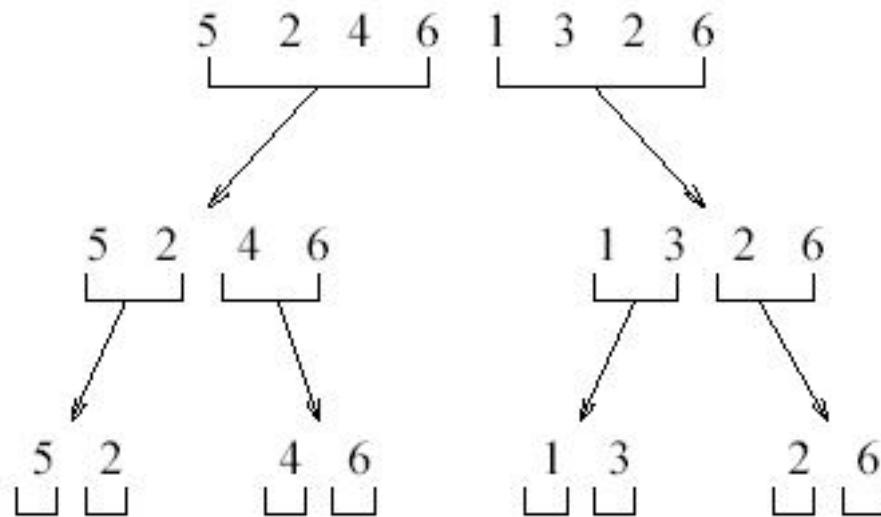
- They *call themselves recursively one or more times to deal with closely related sub problems.*
- D&C does more work on the sub-problems and hence has more time consumption.
- In D&C the sub problems are **independent of each other.**

Example: Merge Sort, Binary Search

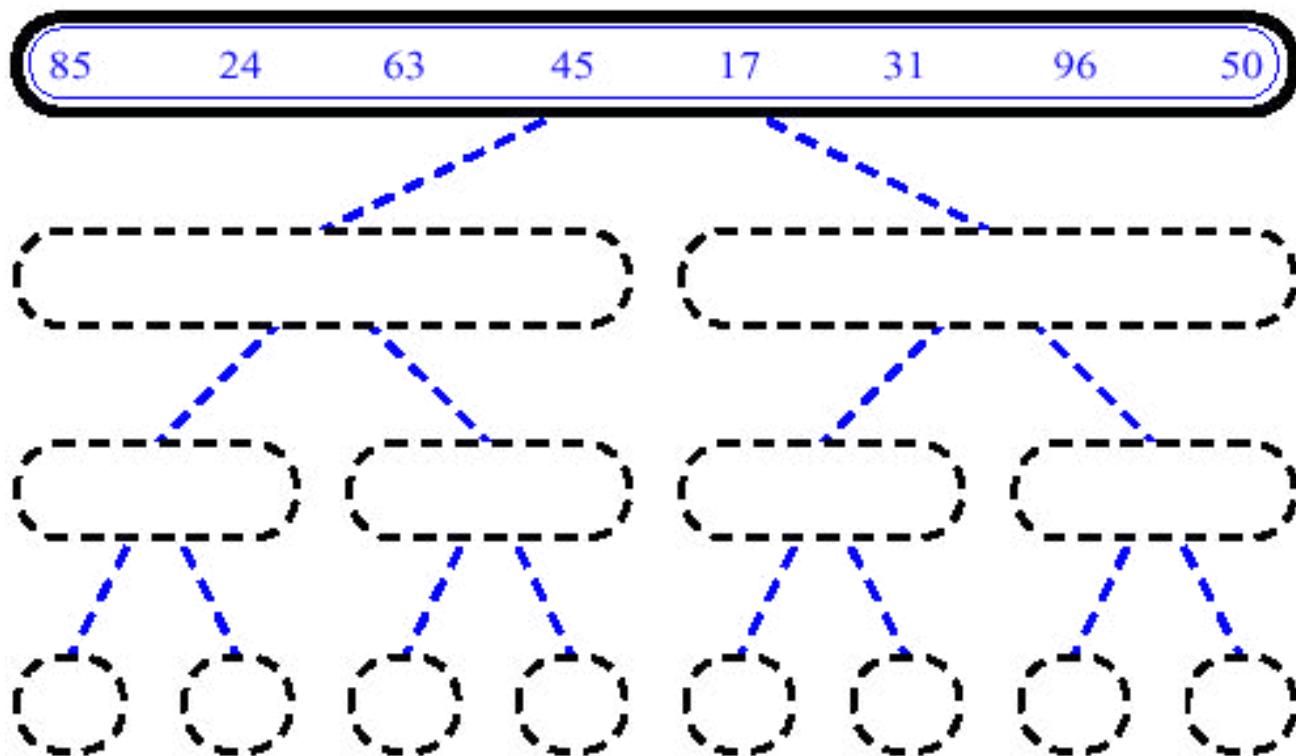
Mergesort

- Based on divide-and-conquer strategy
- Divide the list into two smaller lists of about equal sizes
- Sort each smaller list *recursively*
- Merge the two sorted lists to get one sorted list

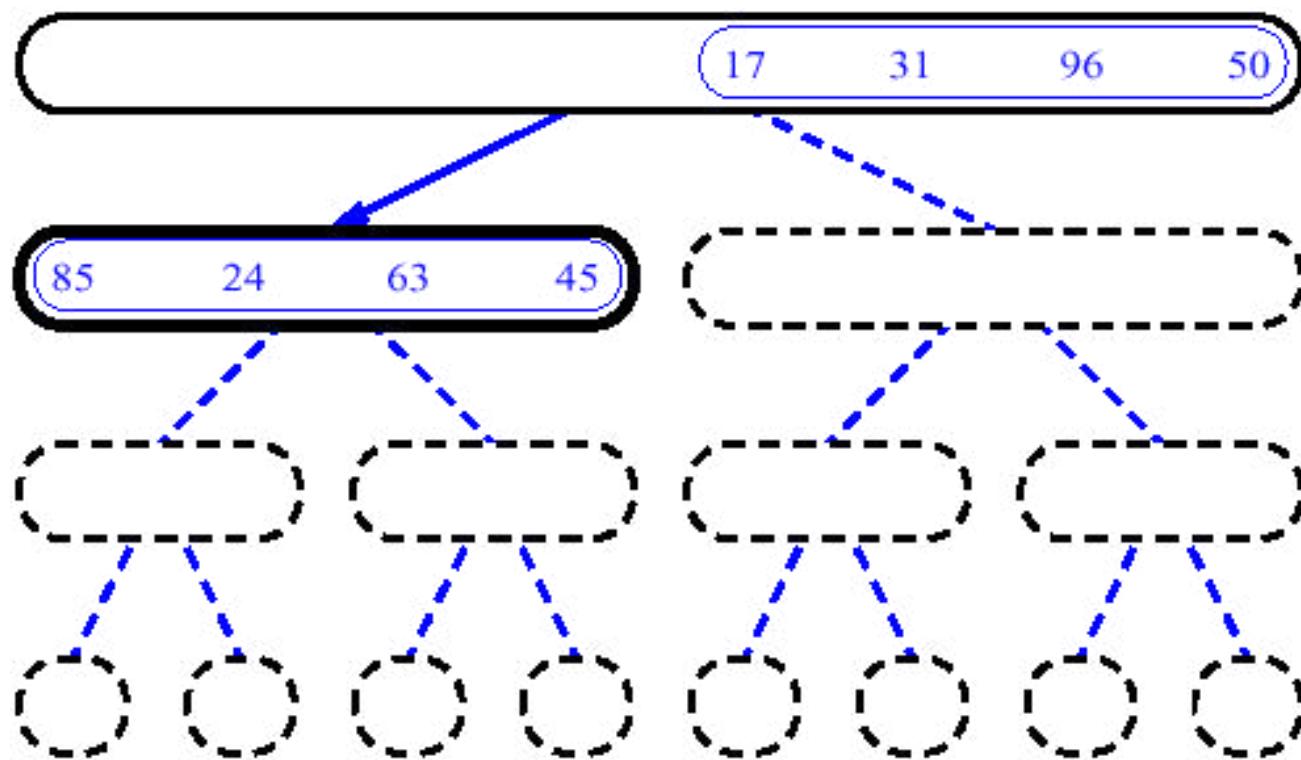
```
void mergesort(vector<int> & A, int left, int right)
{
    if (left < right) {
        int center = (left + right)/2;
        mergesort(A, left, center);
        mergesort(A, center+1, right);
        merge(A, left, center+1, right);
    }
}
```



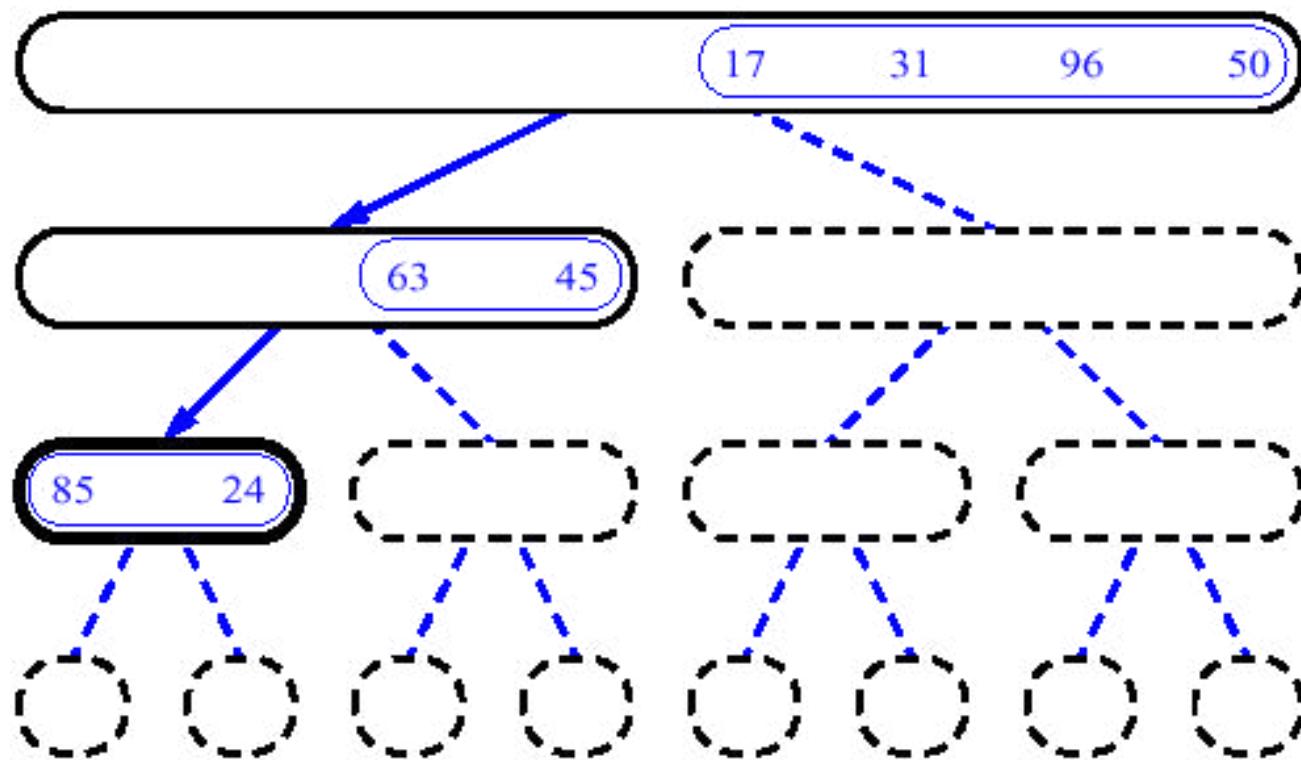
MergeSort (Example) - 1



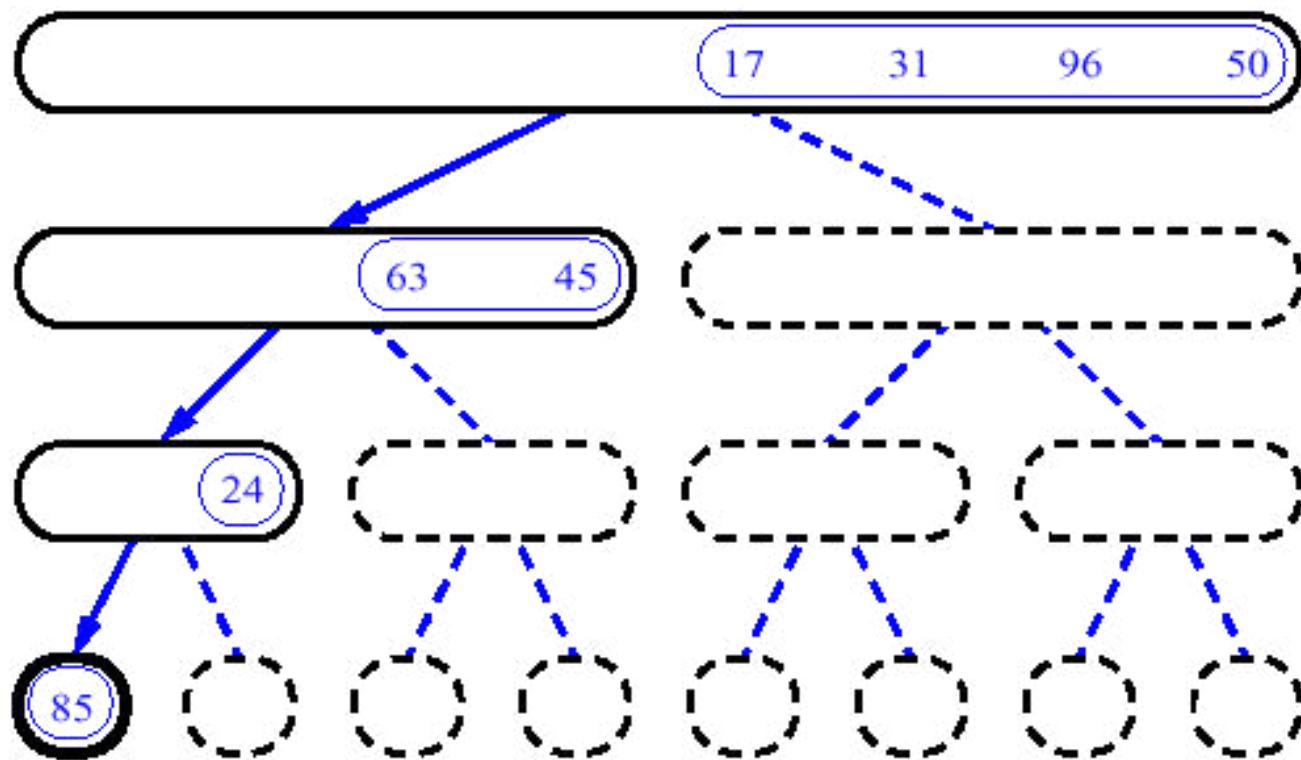
MergeSort (Example) - 2



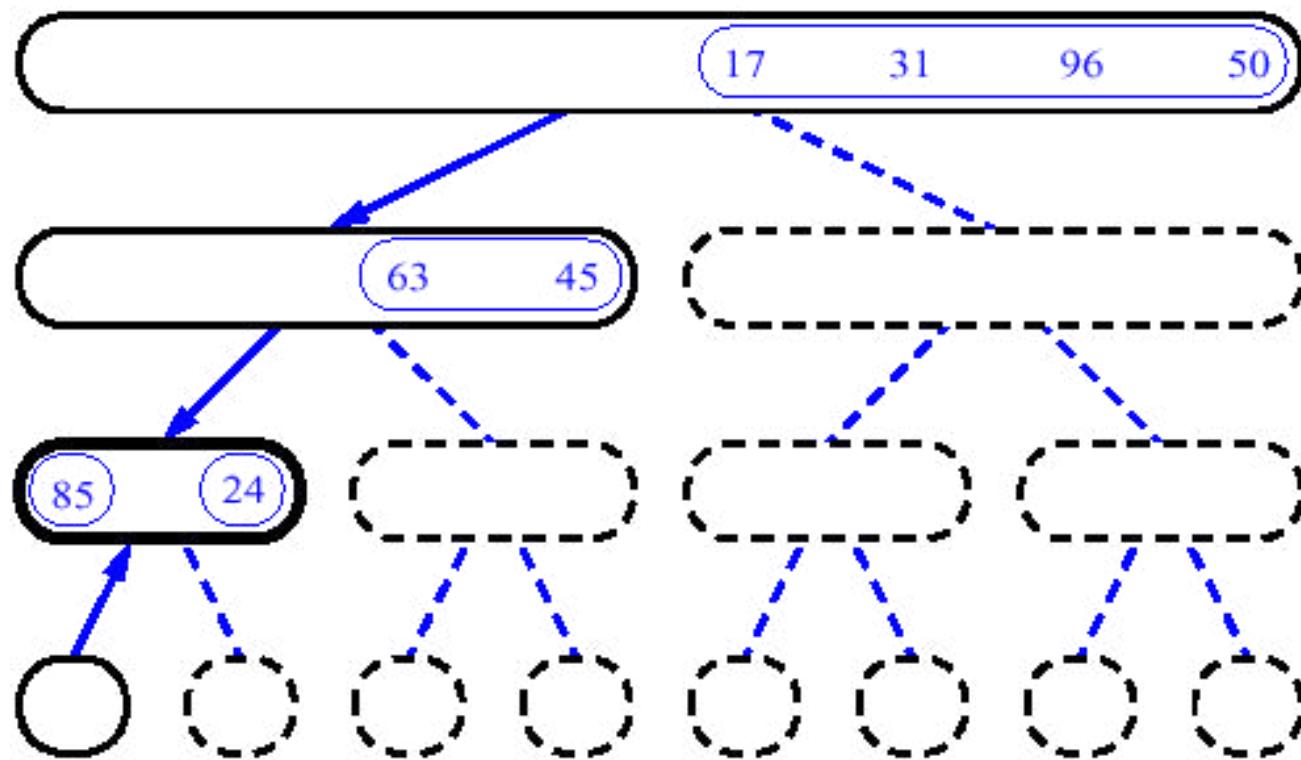
MergeSort (Example) - 3



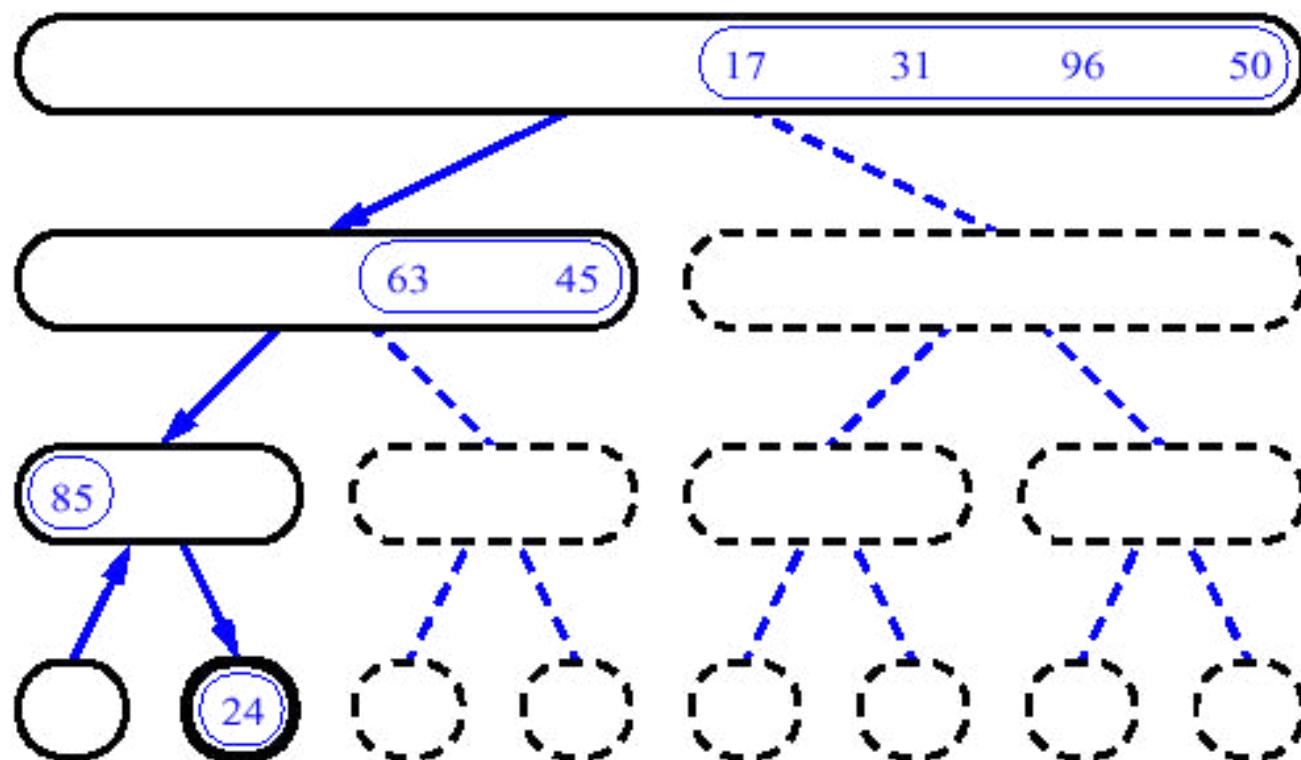
MergeSort (Example) - 4



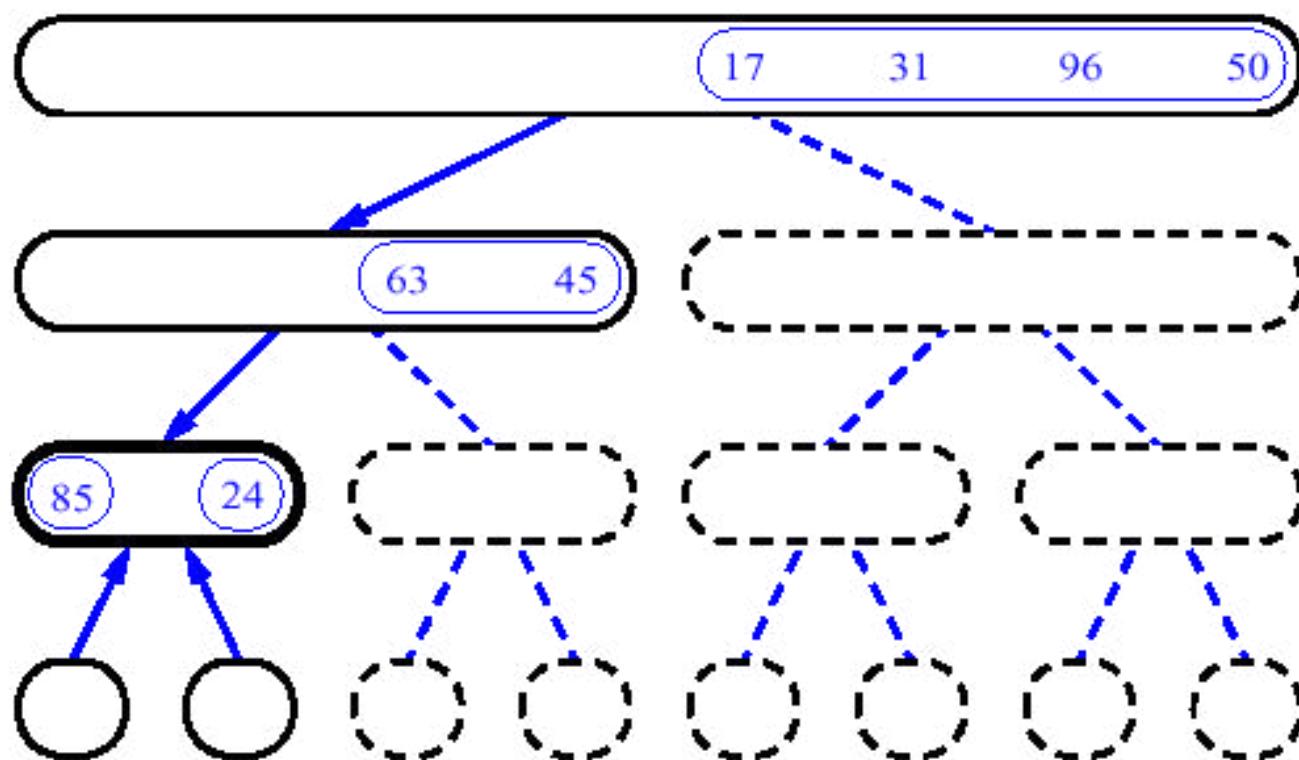
MergeSort (Example) - 5



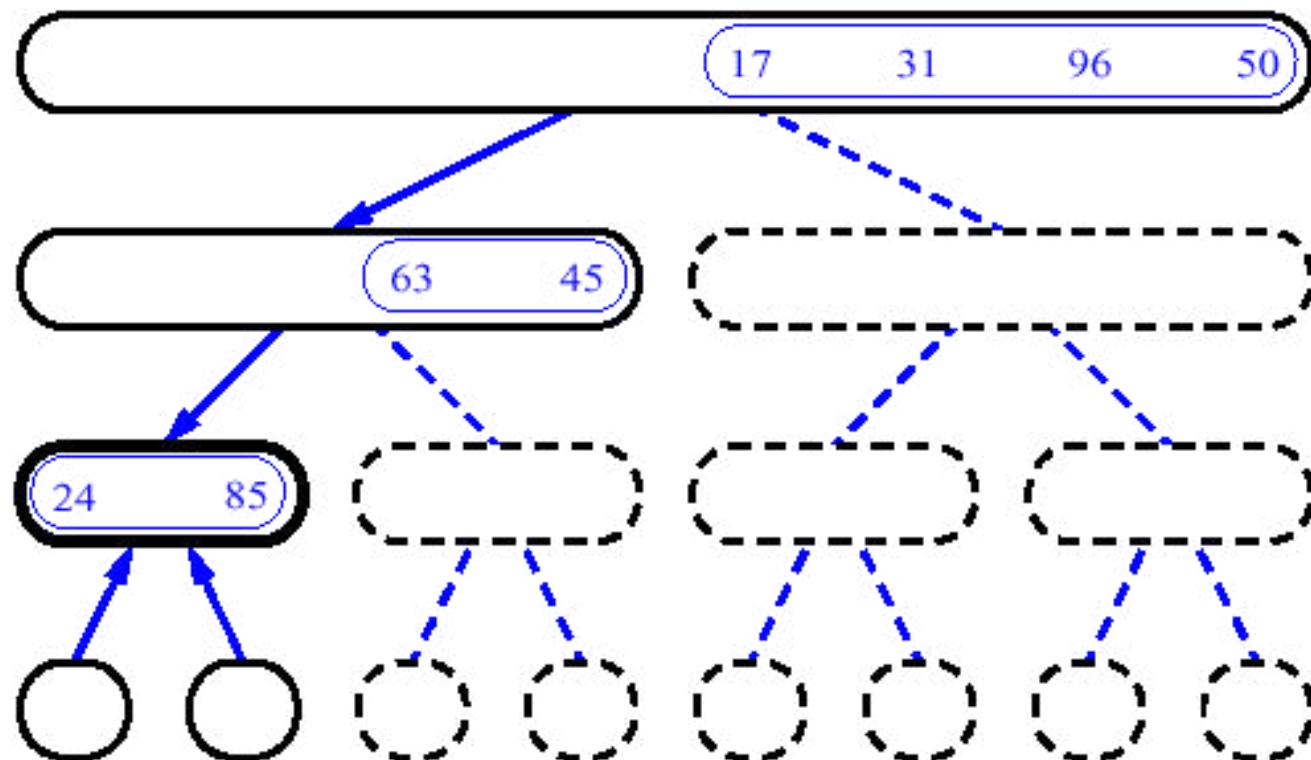
MergeSort (Example) - 6



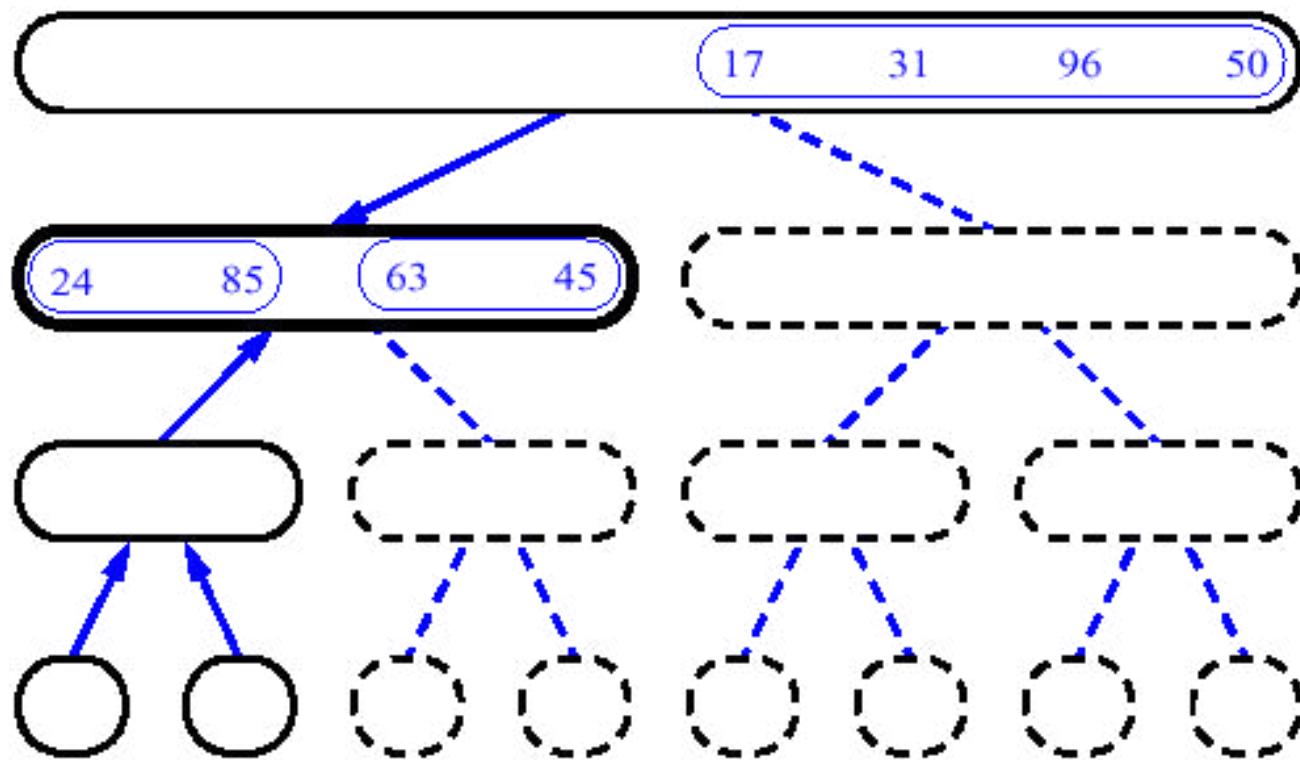
MergeSort (Example) - 7



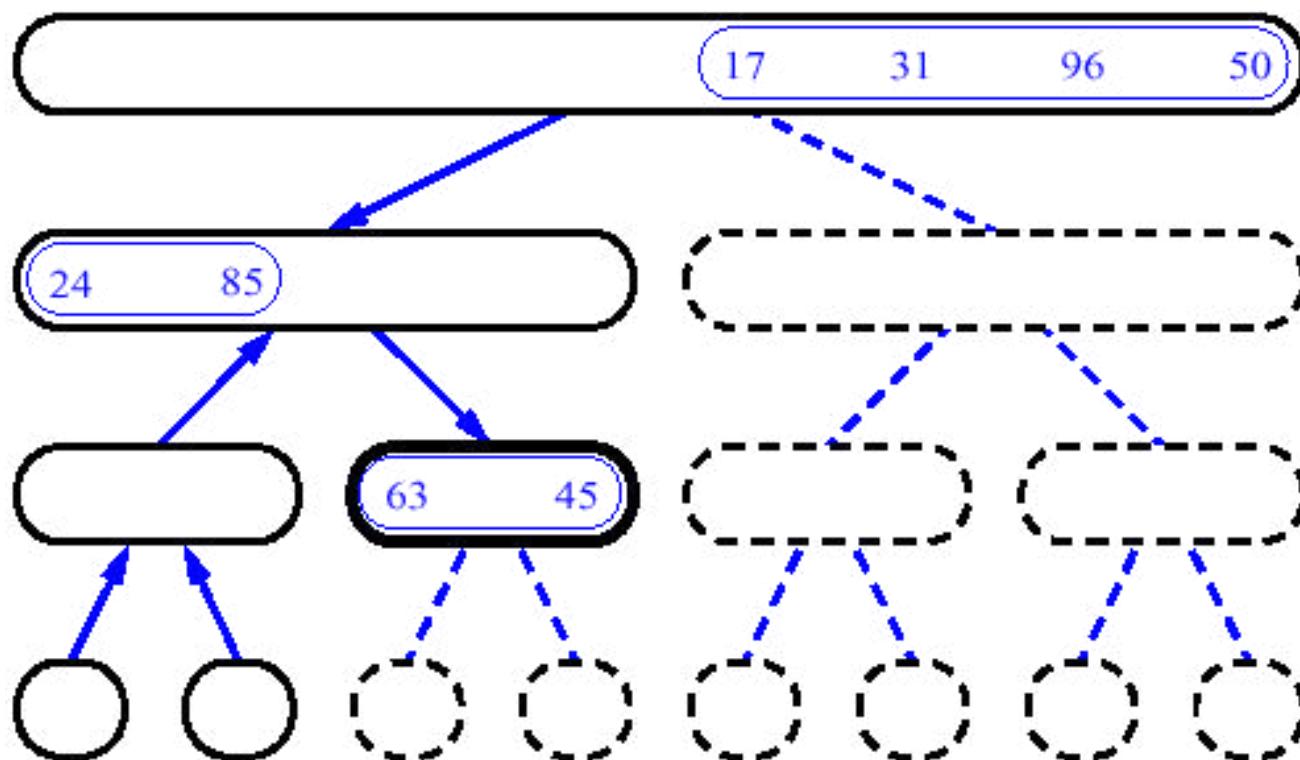
MergeSort (Example) - 8



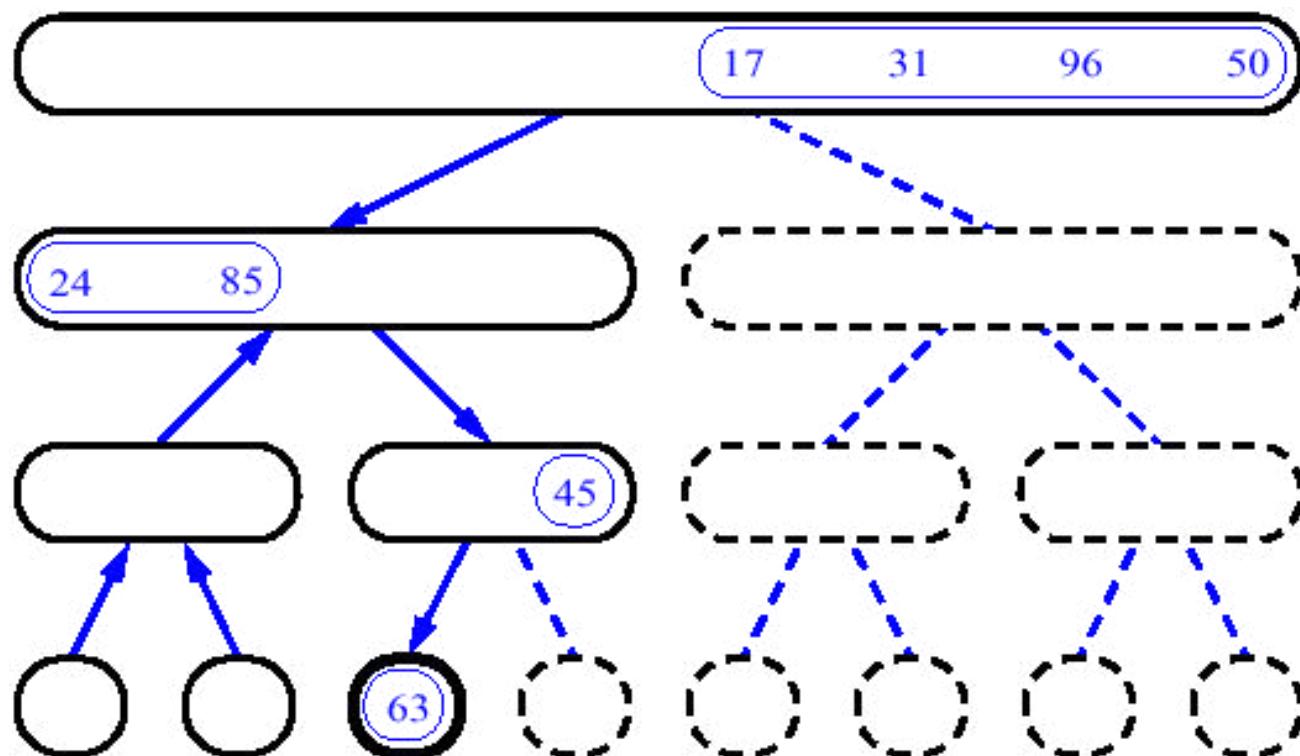
MergeSort (Example) - 9



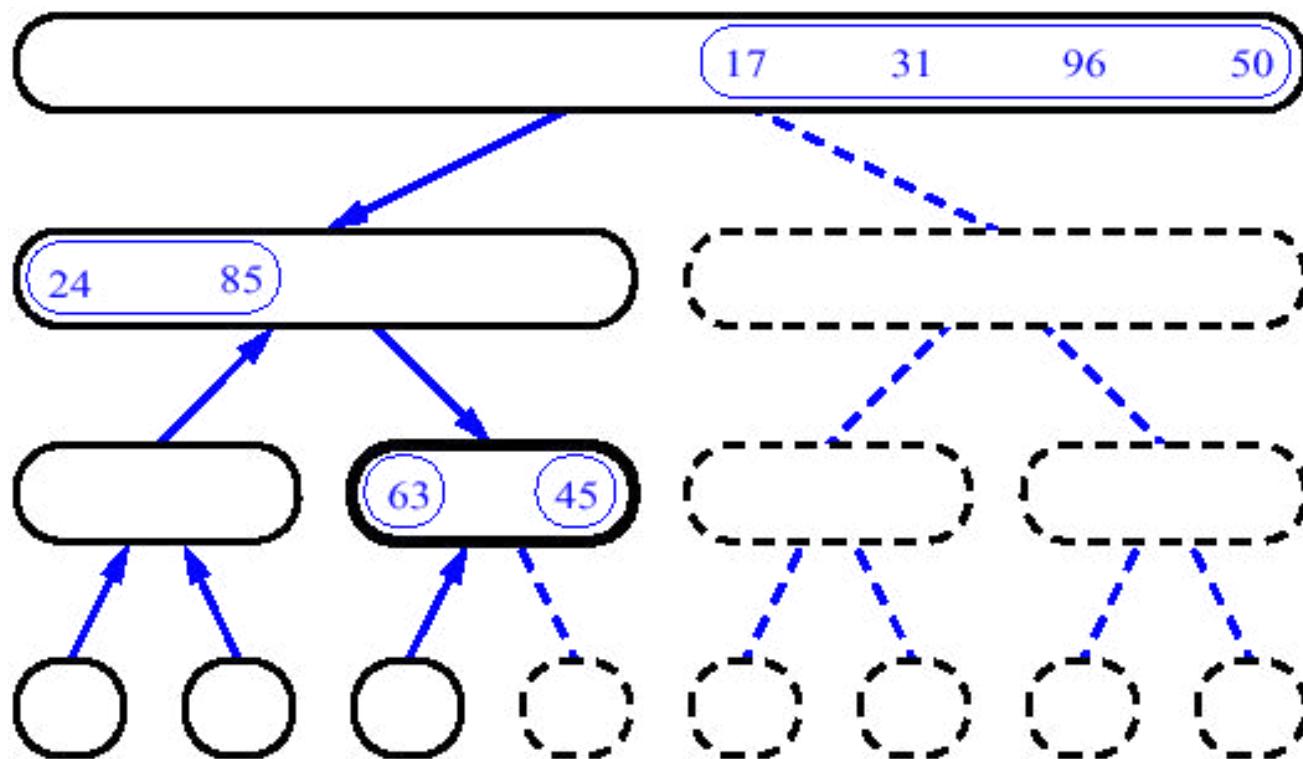
MergeSort (Example) - 10



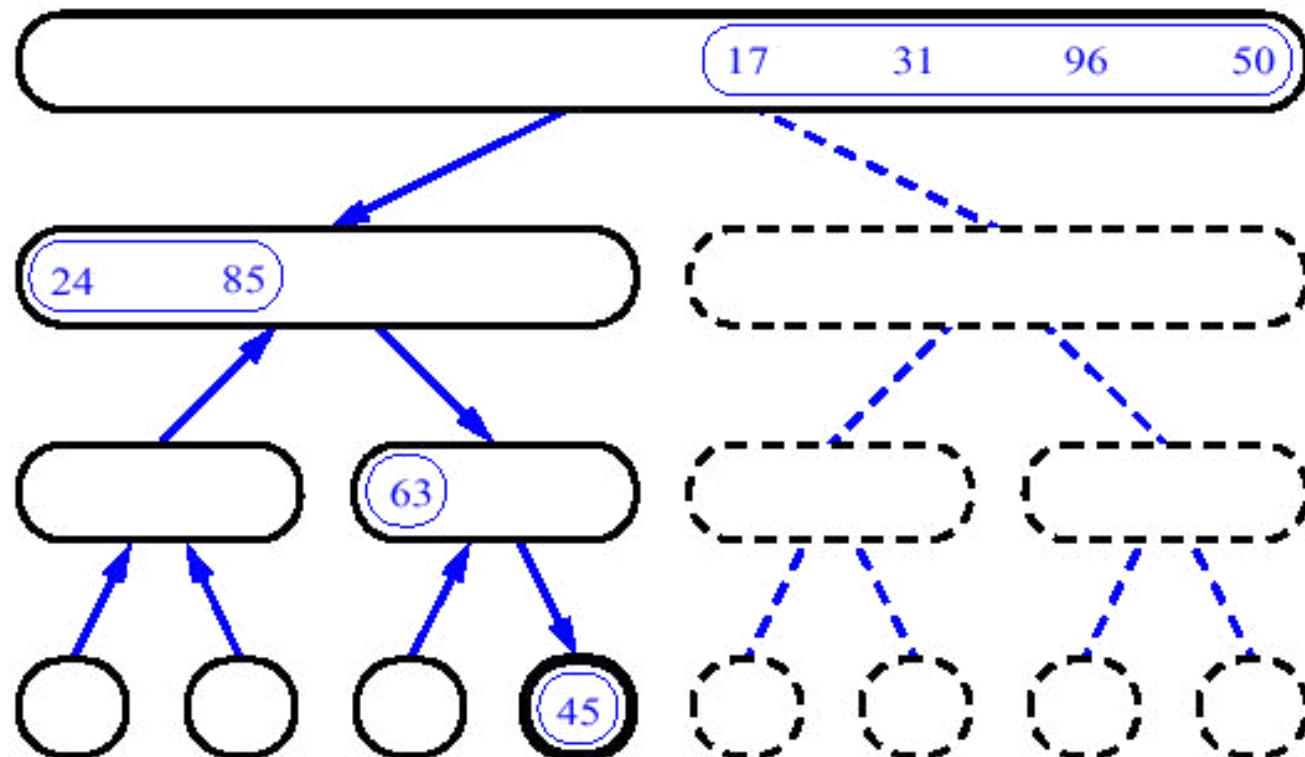
MergeSort (Example) - 11



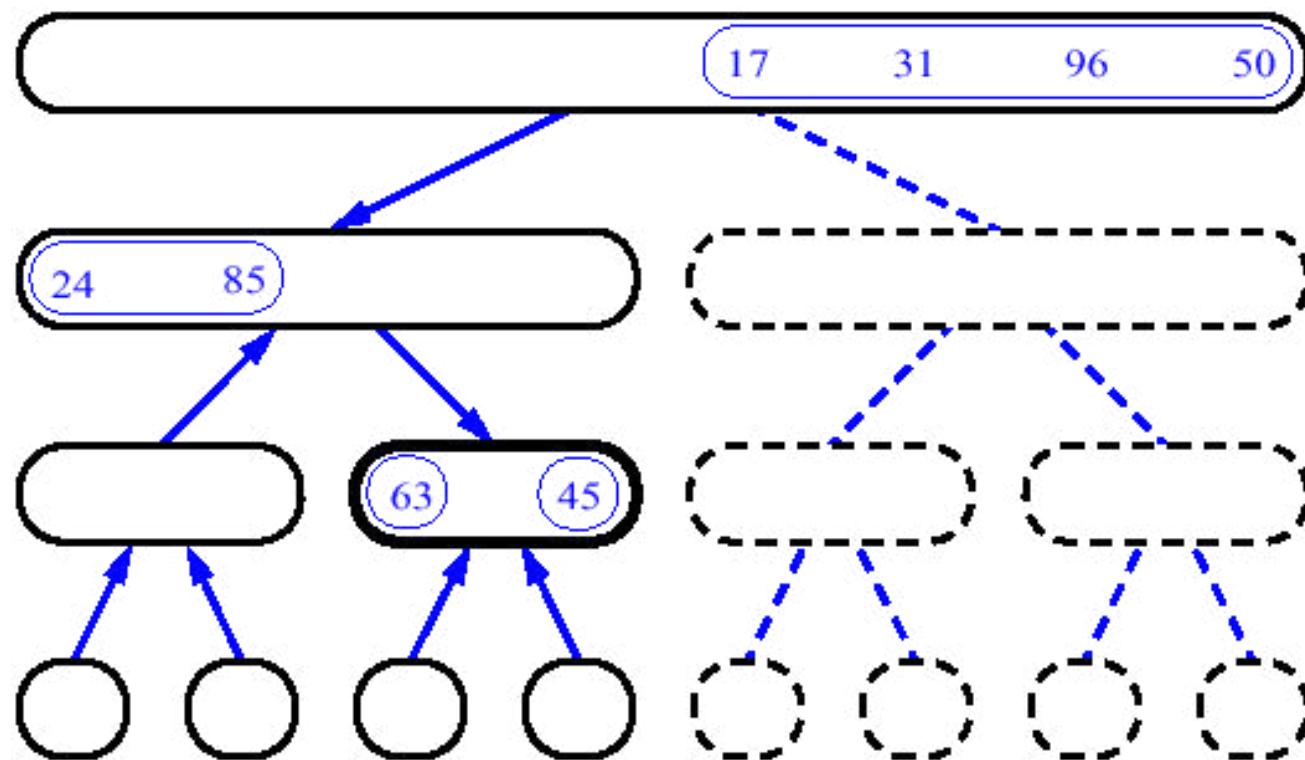
MergeSort (Example) - 12



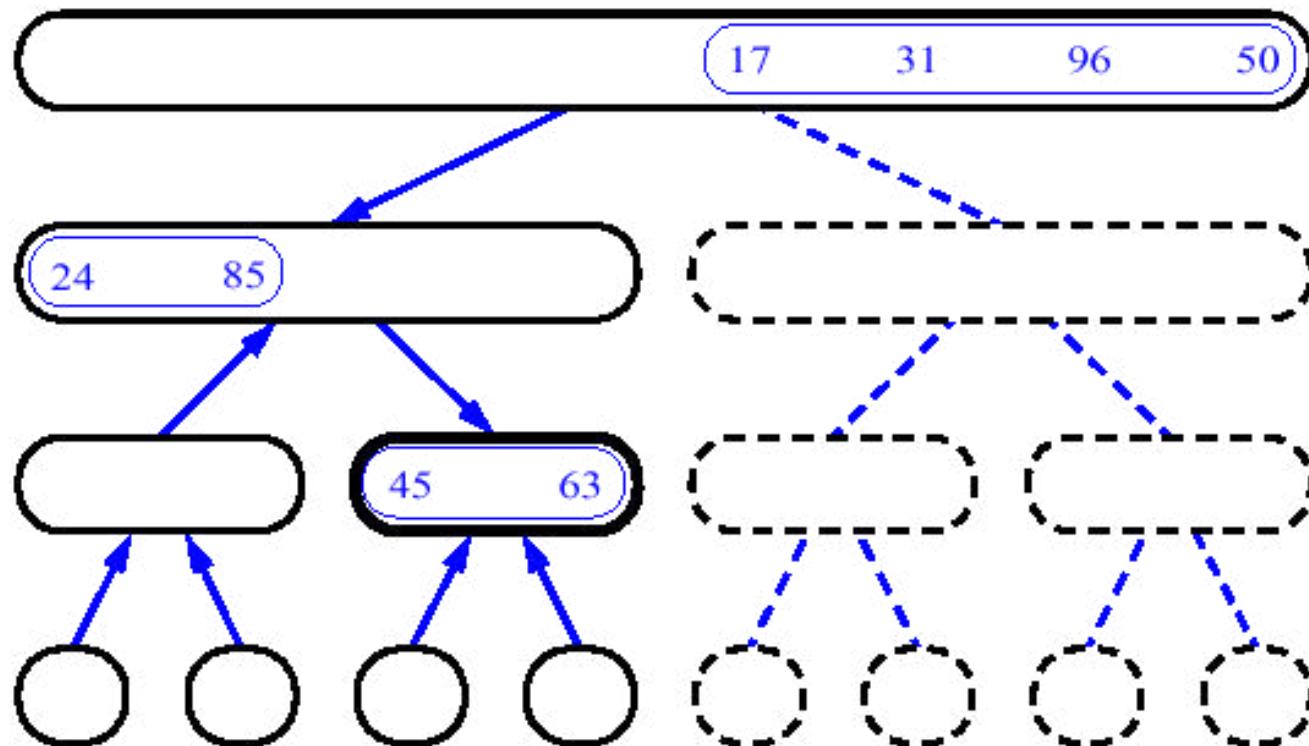
MergeSort (Example) - 13



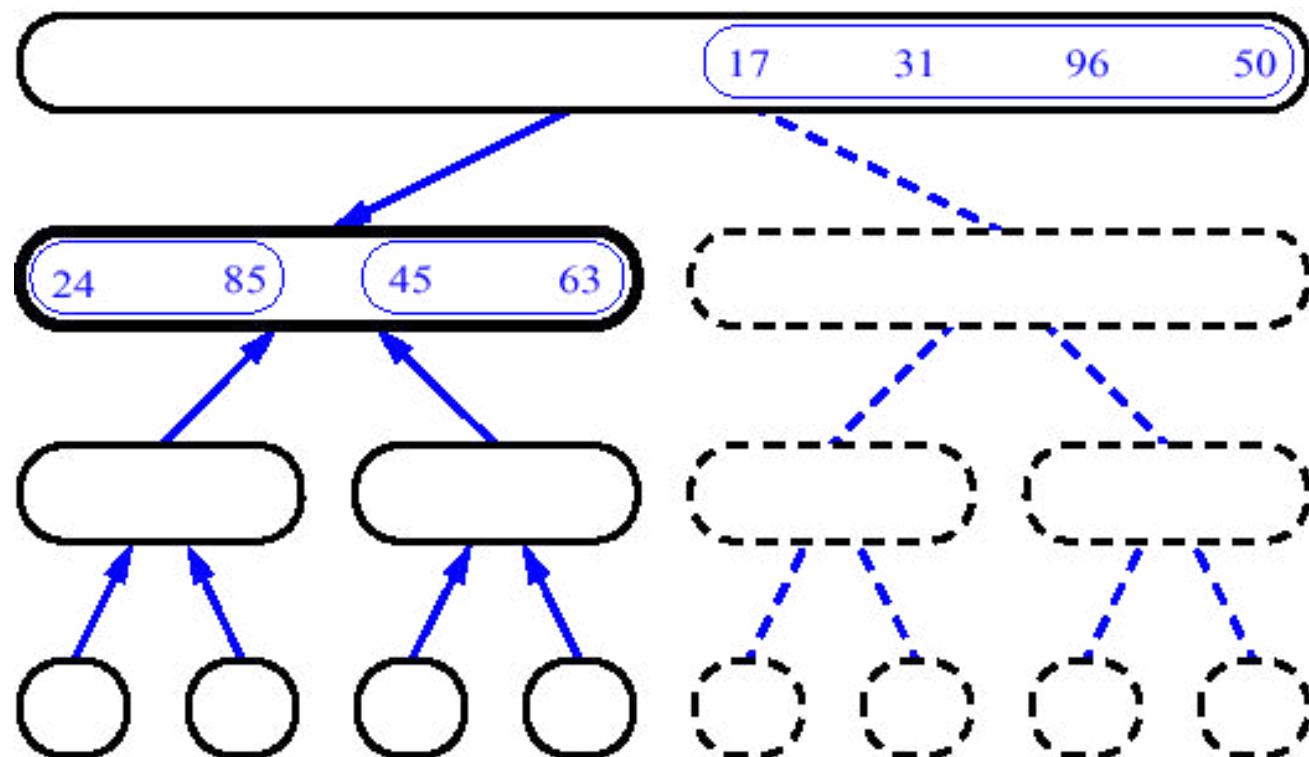
MergeSort (Example) - 14



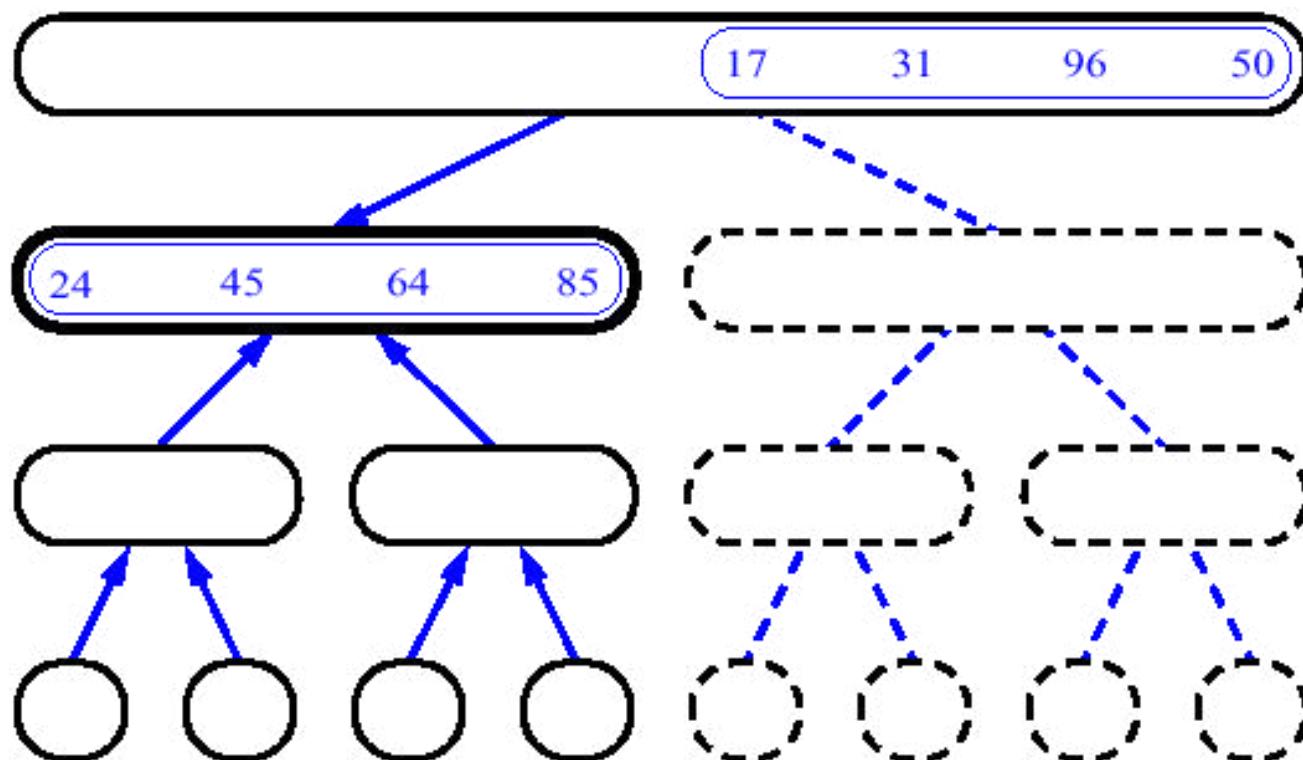
MergeSort (Example) - 15



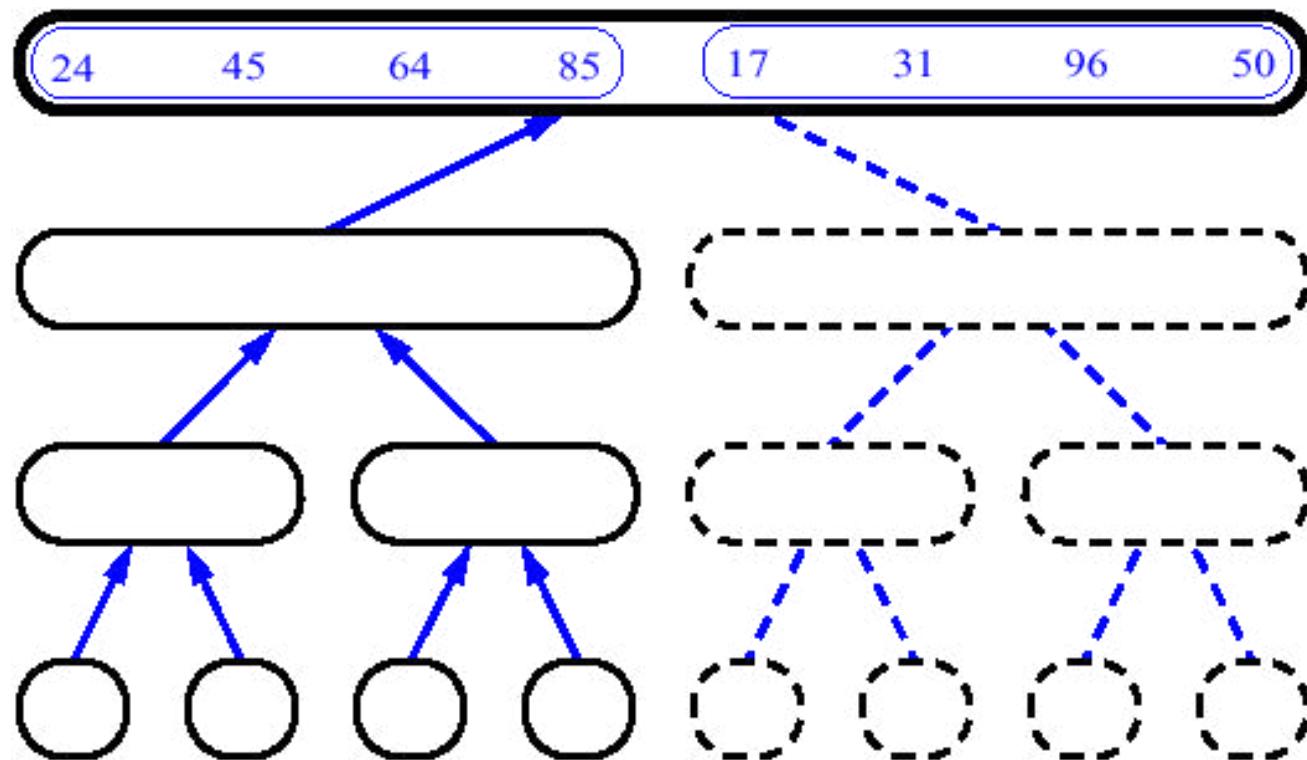
MergeSort (Example) - 16



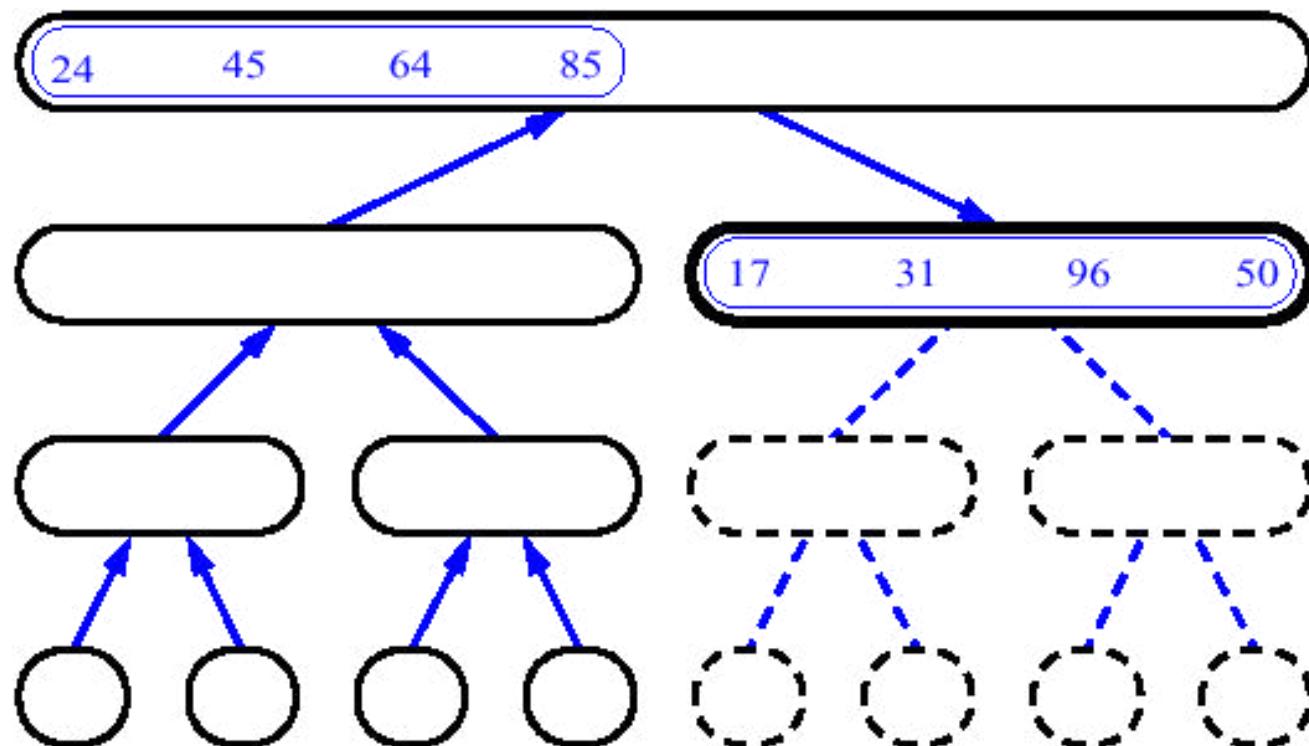
MergeSort (Example) - 17



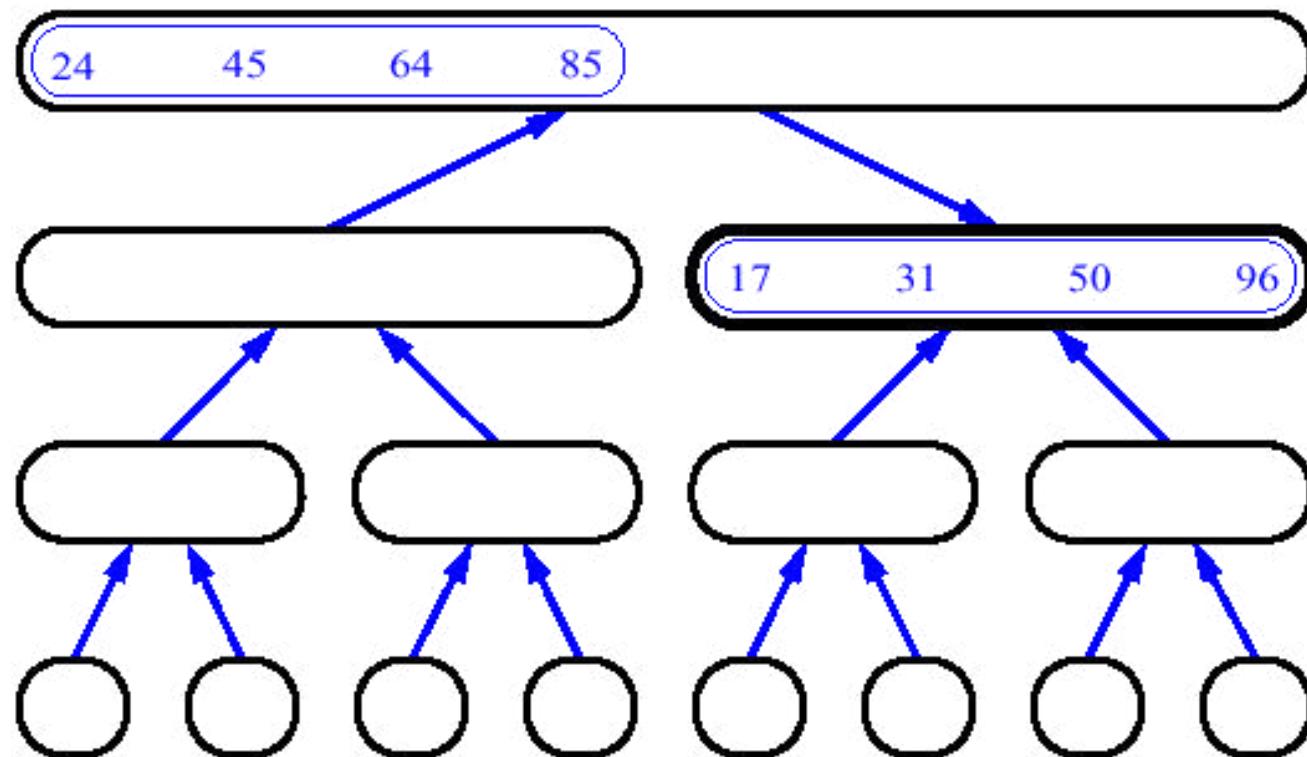
MergeSort (Example) - 18



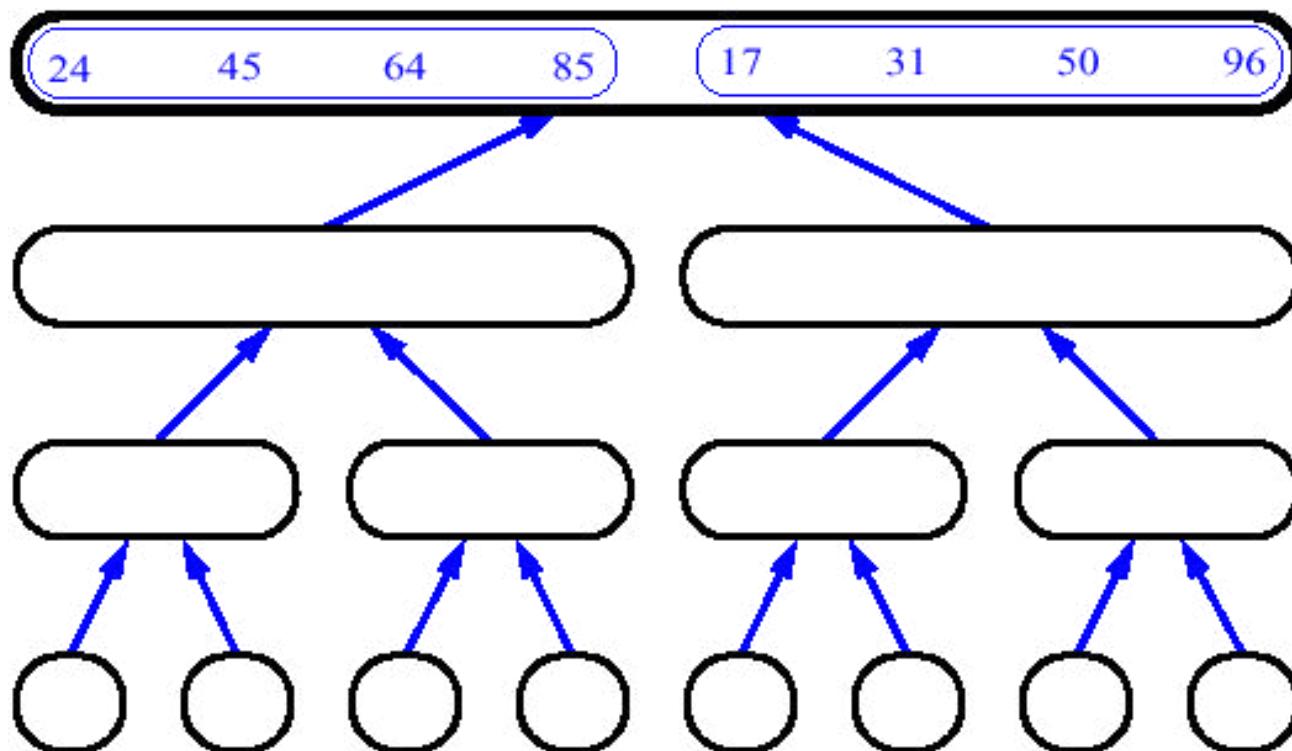
MergeSort (Example) - 19



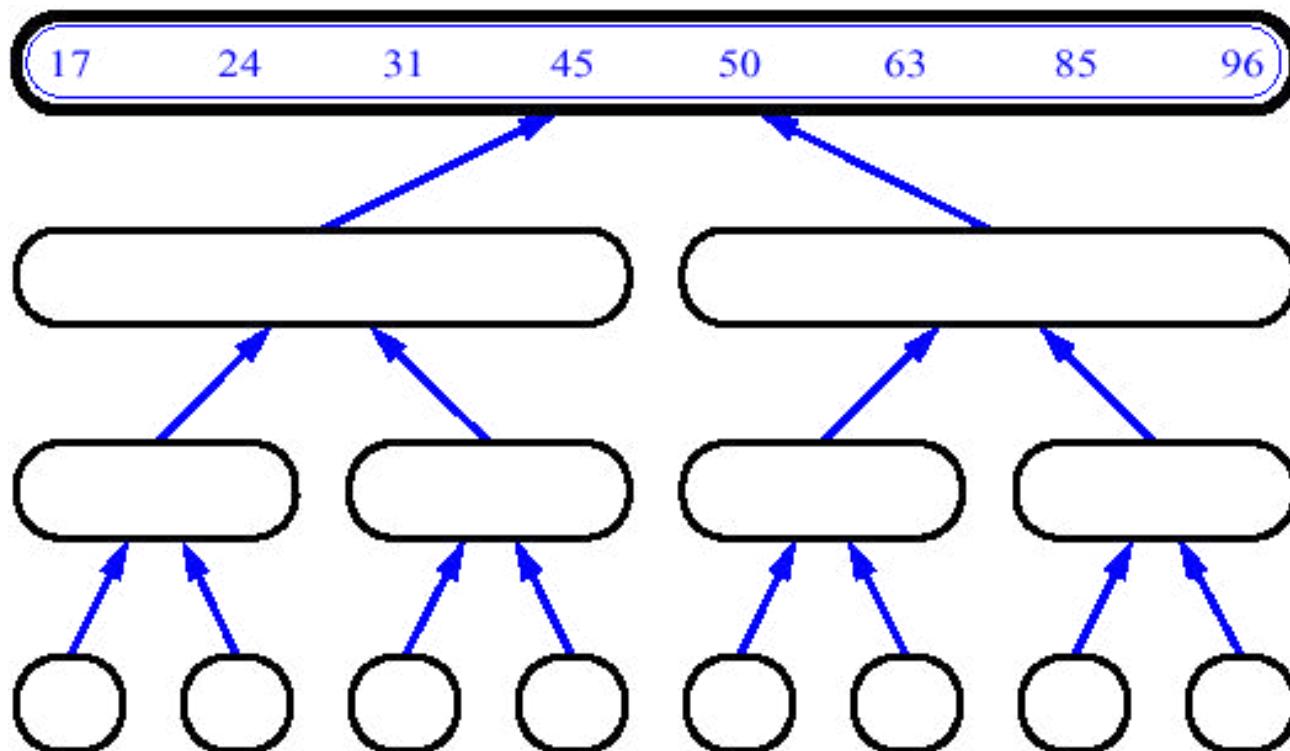
MergeSort (Example) - 20



MergeSort (Example) - 21



MergeSort (Example) - 22



14	23	45	98
----	----	----	----

6	33	42	67
---	----	----	----

14	23	45	98
----	----	----	----

6	33	42	67
---	----	----	----

Merge

14	23	45	98
----	----	----	----

6	33	42	67
---	----	----	----

6

Merge

14	23	45	98
----	----	----	----

6	33	42	67
---	----	----	----

6	14
---	----

Merge

14	23	45	98
----	----	----	----

6	33	42	67
---	----	----	----

6	14	23
---	----	----



14	23	45	98
----	----	----	----

6	33	42	67
---	----	----	----

6	14	23	33
---	----	----	----

Merge

14	23	45	98
----	----	----	----

6	33	42	67
---	----	----	----

6	14	23	33	42
---	----	----	----	----

Merge

14	23	45	98
----	----	----	----

6	33	42	67
---	----	----	----

6	14	23	33	42	45
---	----	----	----	----	----

Merge

14	23	45	98
----	----	----	----

6	33	42	67
---	----	----	----

6	14	23	33	42	45	67
---	----	----	----	----	----	----

Merge

14	23	45	98
----	----	----	----

6	33	42	67
---	----	----	----

6	14	23	33	42	45	67	98
---	----	----	----	----	----	----	----



```
// Merges two subarrays of arr[].
// First subarray is arr[l..m]
// Second subarray is arr[m+1..r]
void merge(int arr[], int l, int m, int r)
{
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;

    /* create temp arrays */
    int L[n1], R[n2];

    /* Copy data to temp arrays L[] and R[]
     * */
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1+ j];
```

```
/* Merge the temp arrays back into arr[l..r]*/
i = 0; // Initial index of first subarray
j = 0; // Initial index of second subarray
k = l; // Initial index of merged subarray
while (i < n1 && j < n2)
{
    if (L[i] <= R[j])
    {
        arr[k] = L[i];
        i++;
    }
    else
    {
        arr[k] = R[j];
        j++;
    }
    k++;
}
```

```
/* Copy the remaining elements of L[], if there are any */
while (i < n1)
{
    arr[k] = L[i];
    i++;
    k++;
}
/* Copy the remaining elements of R[], if there
are any */
while (j < n2)
{
    arr[k] = R[j];
    j++;
    k++; }
}
```

```
/* 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)  
    {  
        // Same as (l+r)/2, but avoids overflow for large l and h  
  
        int m = l+(r-l)/2;  
  
        // Sort first and second halves  
        mergeSort(arr, l, m);  
        mergeSort(arr, m+1, r);  
  
        merge(arr, l, m, r);  
    }  
}
```

11	26	14	2	32	18	7	21
----	----	----	---	----	----	---	----



11	26	14	2
----	----	----	---

32	18	7	21
----	----	---	----



11	26
----	----

14	2
----	---

32	18
----	----

7	21
---	----

11	26	14	2	32	18	7	21
----	----	----	---	----	----	---	----



11	26	14	2
----	----	----	---

32	18	7	21
----	----	---	----

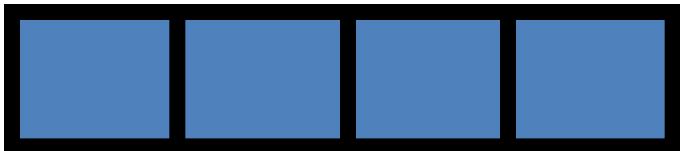
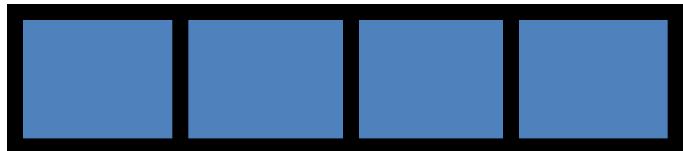
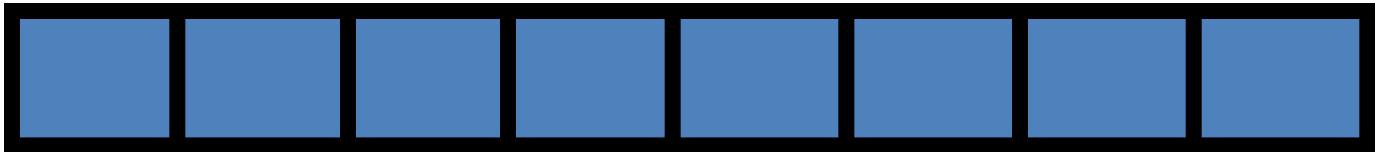


11	26
----	----

14	2
----	---

32	18
----	----

7	21
---	----



11 26

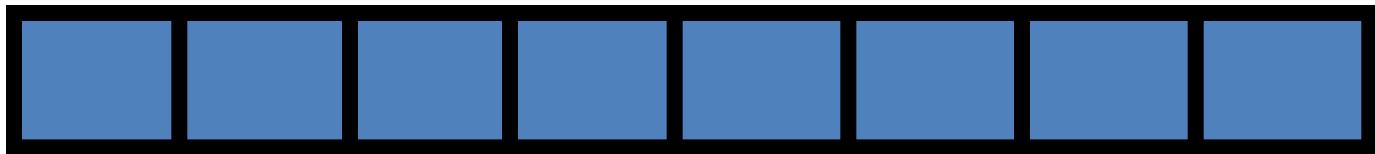


2 14

18 32



7 21



11	26	2	14
----	----	---	----

18	32	7	21
----	----	---	----

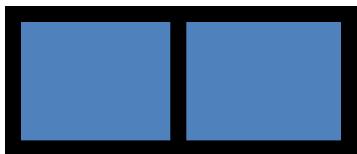
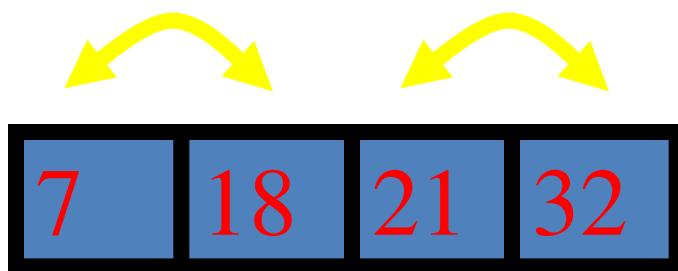
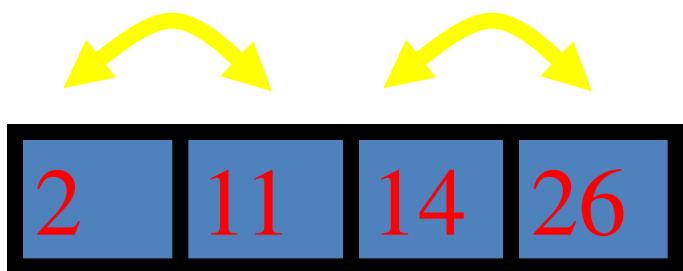
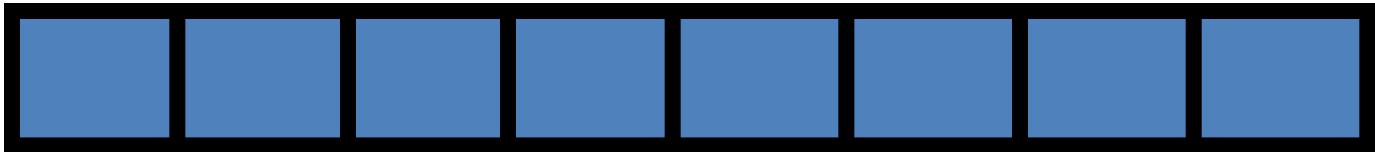
11	26
----	----

2	14
---	----

18	32
----	----

7	21
---	----





2	11	14	26	7	18	21	32
---	----	----	----	---	----	----	----



2	11	14	26
---	----	----	----

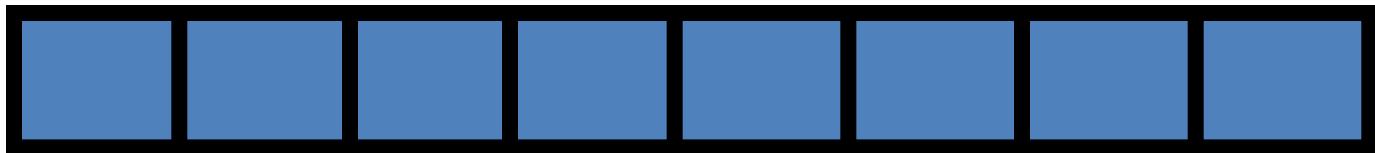
7	18	21	32
---	----	----	----

--	--

--	--

--	--

--	--



11	26	2	14
----	----	---	----

18	32	7	21
----	----	---	----

11	26
----	----

2	14
---	----

18	32
----	----

7	21
---	----



QUICK SORT

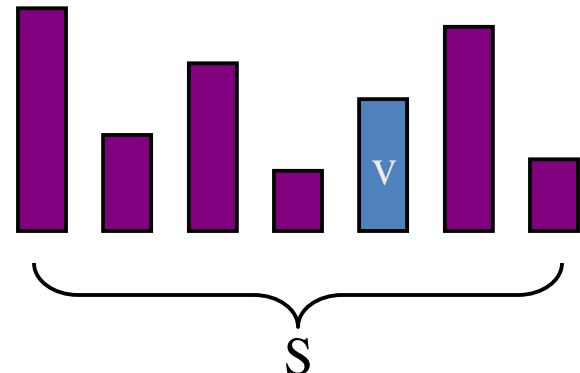
Quicksort

- **Divide step:**

- Pick any element (**pivot**) v in S
- Partition $S - \{v\}$ into two disjoint groups

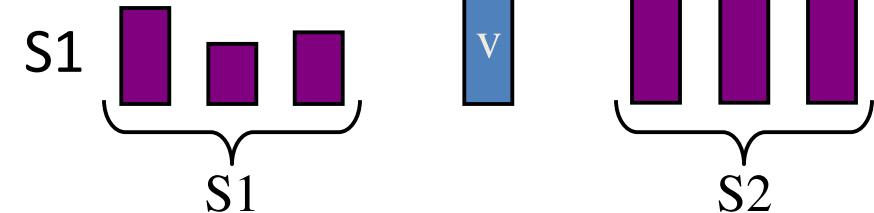
$$S_1 = \{x \in S - \{v\} \mid x \leq v\}$$

$$S_2 = \{x \in S - \{v\} \mid x \geq v\}$$



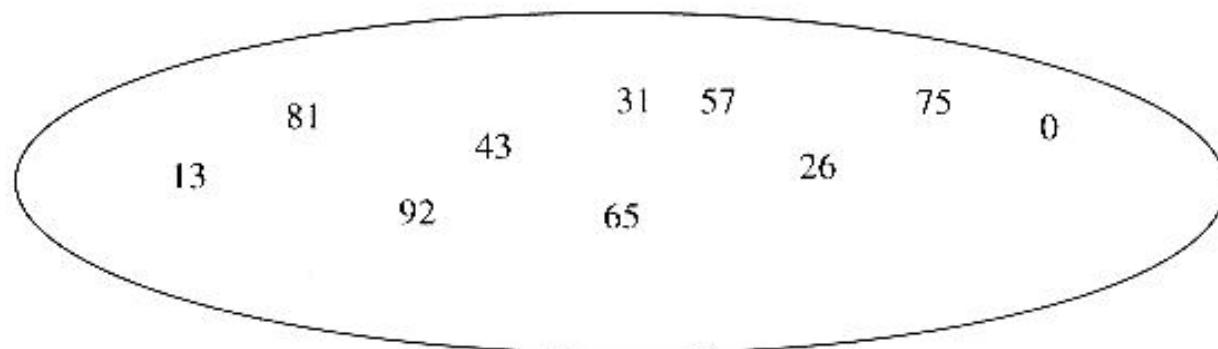
- **Conquer step:** recursively sort

and S_2

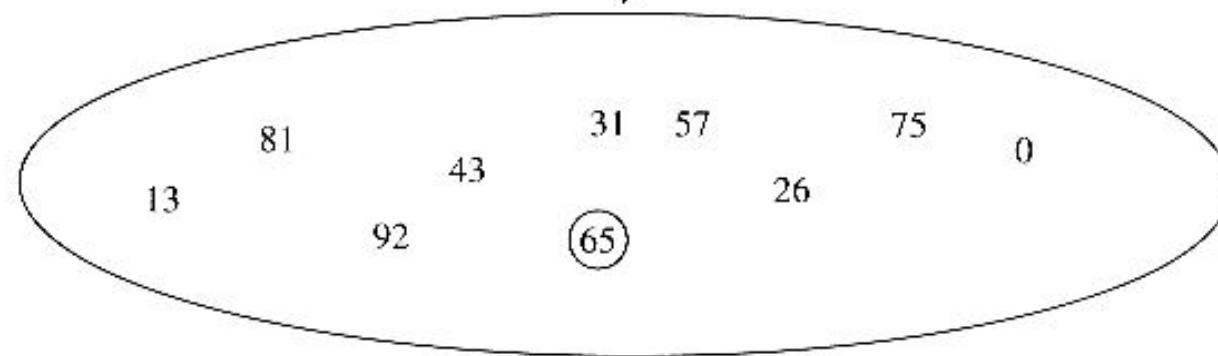


- **Combine step:** combine the sorted S_1 , followed by v , followed by the sorted S_2

Example: Quicksort

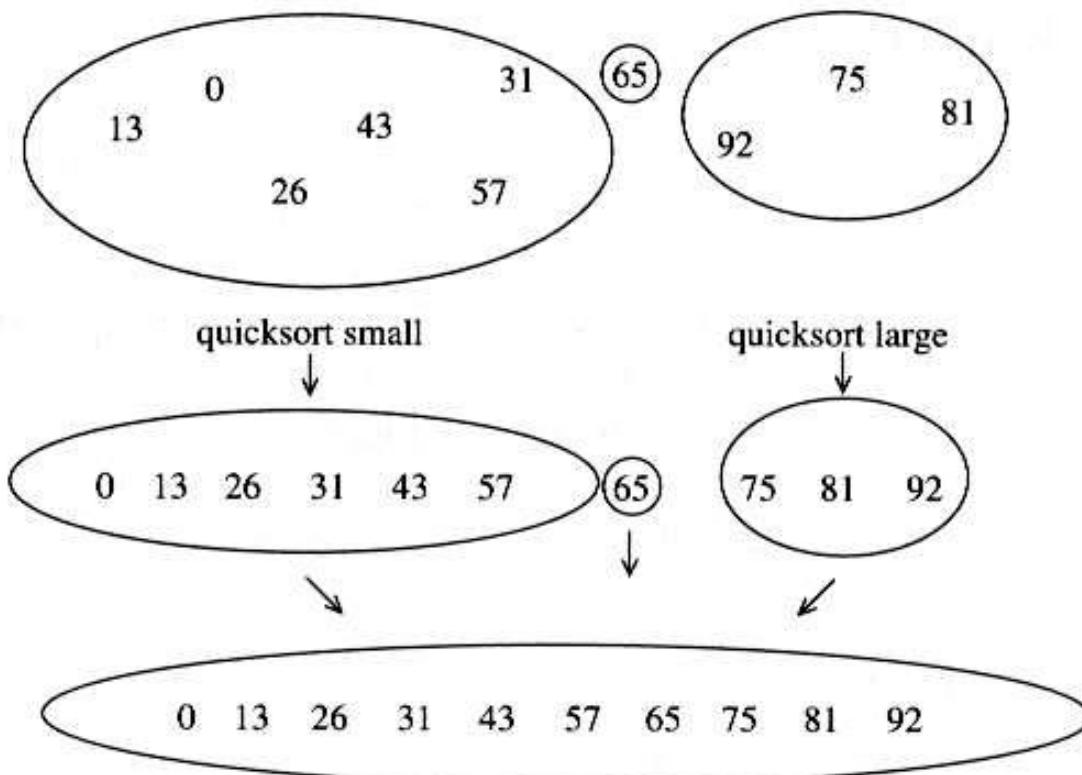


select pivot
↓



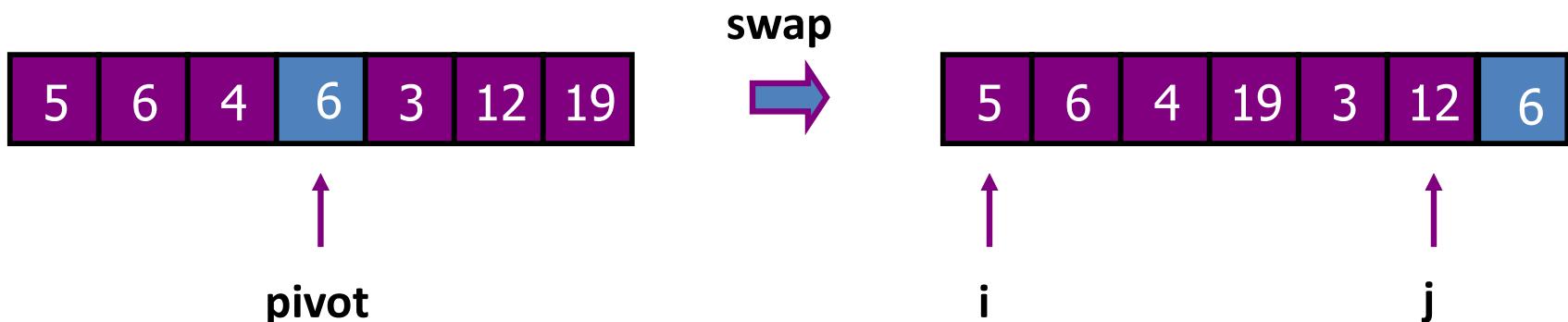
partition
↓

Example: Quicksort...



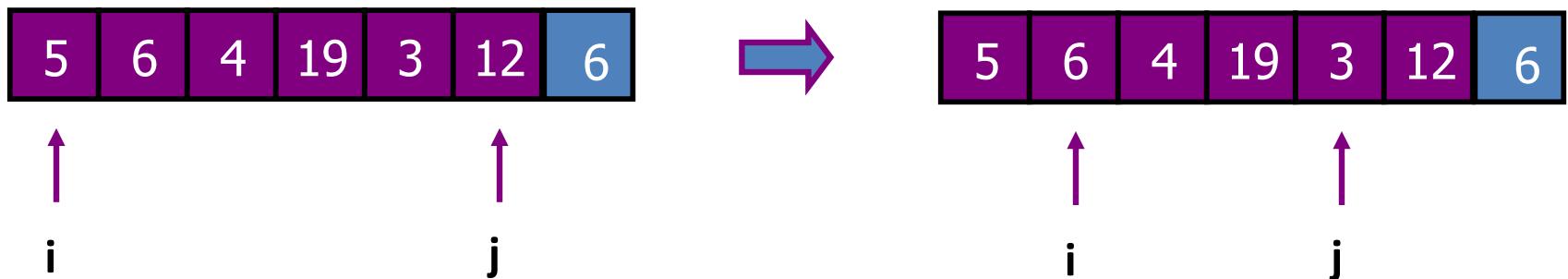
Partitioning Strategy

- To partition an array $A[\text{left} .. \text{right}]$
- First, get the pivot element out of the way by swapping it with the last element. (**Swap pivot and $A[\text{right}]$**)
- Let i start at the first element and j start at the next-to-last element ($i = \text{left}$, $j = \text{right} - 1$)



Partitioning Strategy

- (1) Value of 'i' must be **less than pivot** : if True then move 'i' towards '**Right**'
 - (2) Value of 'j' must be **greater than pivot** : if True then move 'j' towards '**Left**'
 - (3) If both 'i' and 'j' are **stopped** at any iteration & if $i < j$, then **swap i & j**
 - (4) If both i & j are **stopped** at any iteration & if $j < i$, **Swap jth element with Pivot**



Picking the Pivot

- Use the first element as pivot
- Choose the pivot randomly
- Use the median of the array

ALGORITHM

Quick _ Sort (K , LB , UB)

FLAG \leftarrow True

If LB < UB Then

I \leftarrow LB

J \leftarrow UB + 1

Key \leftarrow K [I]

Repeat While FLAG

$I \leftarrow I + 1$

Repeat While $K[I] < Key$

$I \leftarrow I + 1$

$J \leftarrow J - 1$

Repeat While $K[J] > Key$

$J \leftarrow J - 1$

if $I < J$

then

$K[I] \leftrightarrow K[J]$

else

FLAG \leftarrow FALSE

$K [LB] \longleftrightarrow [J]$

Call Quick_Sort ($K , LB , J - 1$)

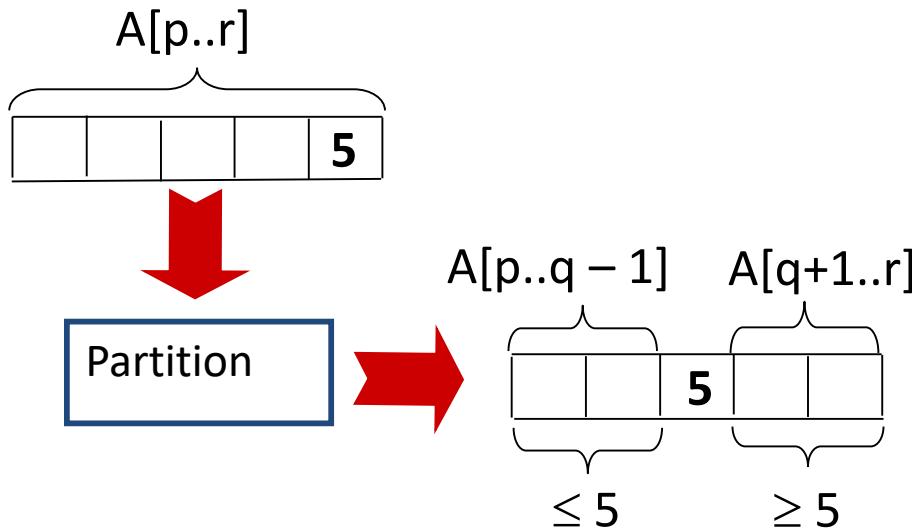
Call Quick_Sort ($K , J + 1 , UB$)

Return

Pseudocode

Quicksort(A, p, r)

```
if p < r then
    q := Partition(A, p, r);
    Quicksort(A, p, q - 1);
    Quicksort(A, q + 1, r)
```

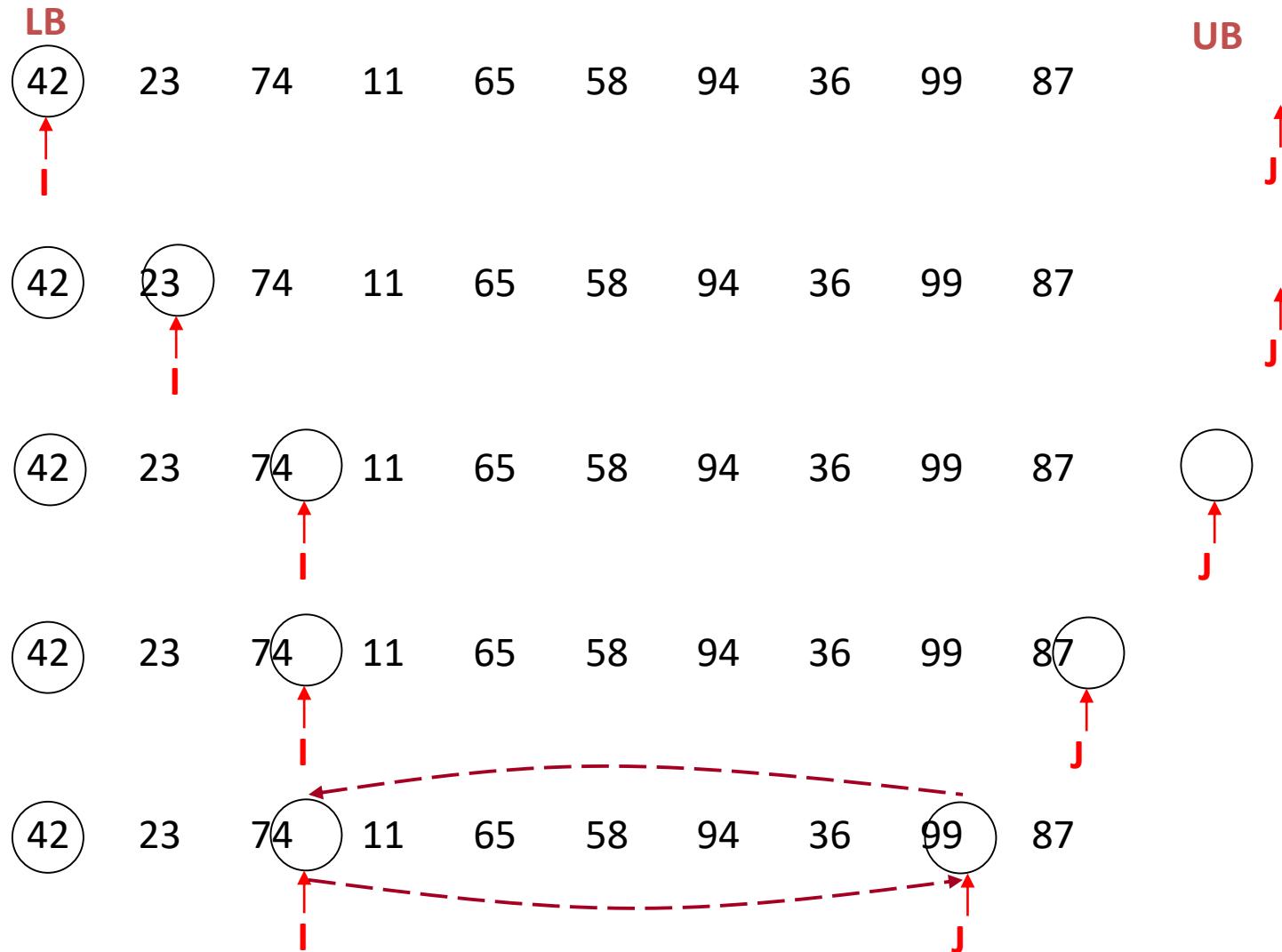


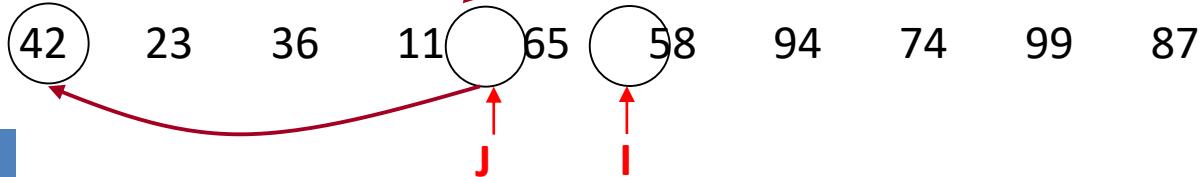
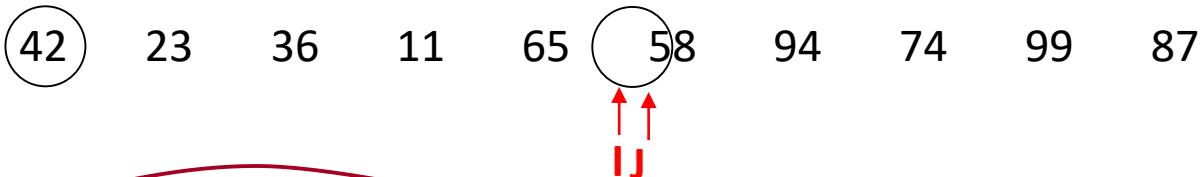
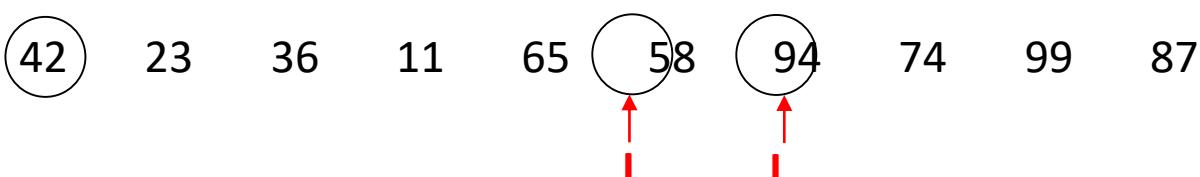
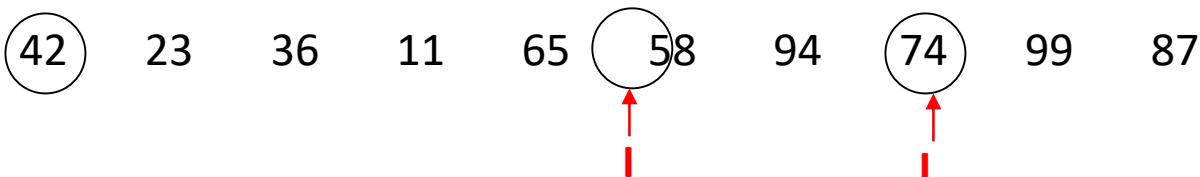
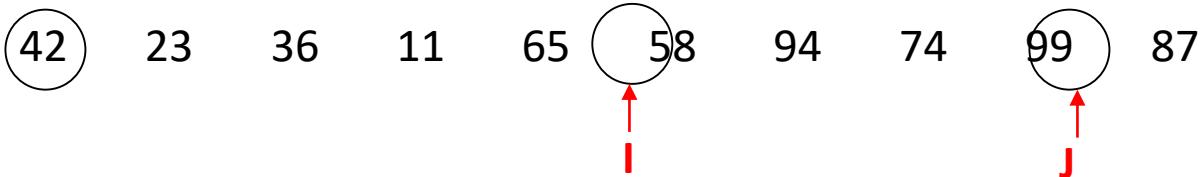
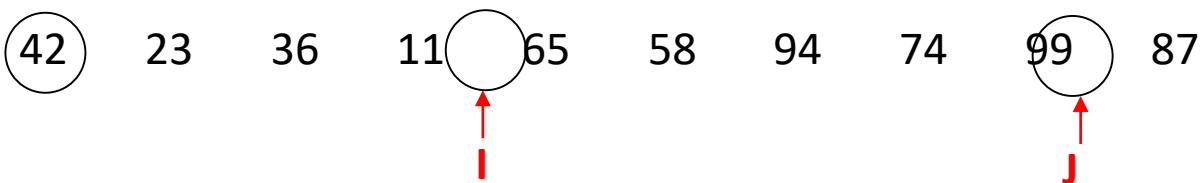
Partition(A, p, r)

{

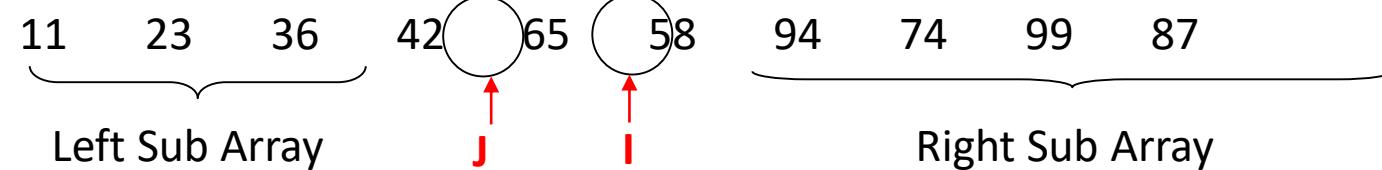
```
x, i := A[r], p - 1;
for j := p to r - 1 do
    { if A[j] ≤ x
then
    { i := i + 1;
      Swap A[i] ↔
A[j]
      } }
      Swap A[i + 1] ↔
A[r];
return i + 1
}
```

QS (K , 1 , 10)





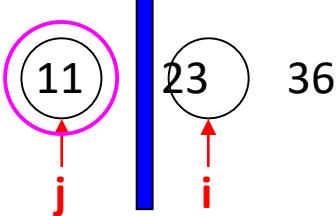
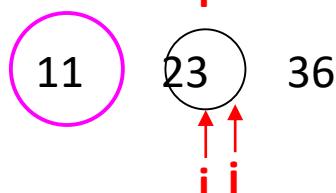
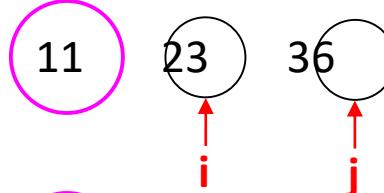
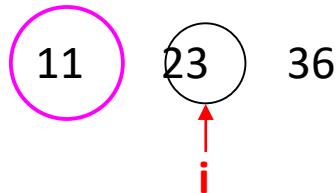
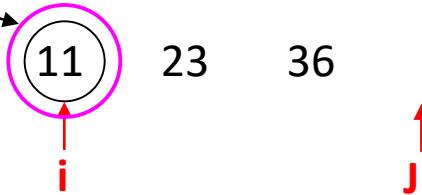
J < I



Left Sub Array [L1]

QS (K , 1 , 3)

Pivot



J = 4
LB = 1
UB = 3
UB=(J-1)

K [LB] = 11
K [UB] = 36

i > j
FLAG \leftarrow False
K[LB] \leftrightarrow K[j]

Left Sub Array [L1 – L1]

LB = 1

UB = 0

K [LB] = 11

K [UB] = Wrong Value

Condition violates
 $A[0] < B < UB$

Right Sub Array [L1 – R1]

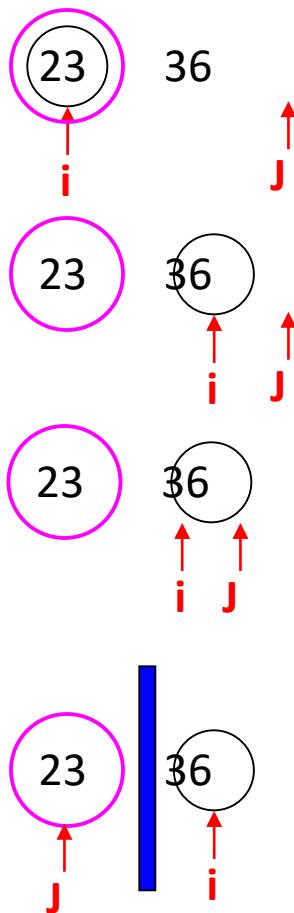
LB = 2

UB = 3

K [LB] = 23

K [UB] = 36

QS (K , 2 , 3)



$i > j$

FLAG \leftarrow False

K [LB] \leftrightarrow x [j]

Left Sub Array [L1- R1- L1]

LB = 2

UB = 1

J = 2

K [LB] = 23

K [UB] = 11

QS (K , 2 , 1)



23

Right Sub Array [L1- R1- R1]

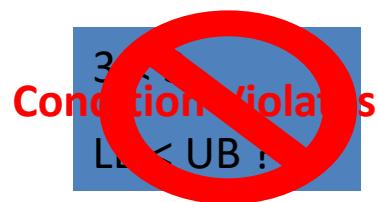
LB = 3

UB = 3

K [LB] = 36

K [UB] = 36

QS (K , 3 , 3)



36

LB = 5

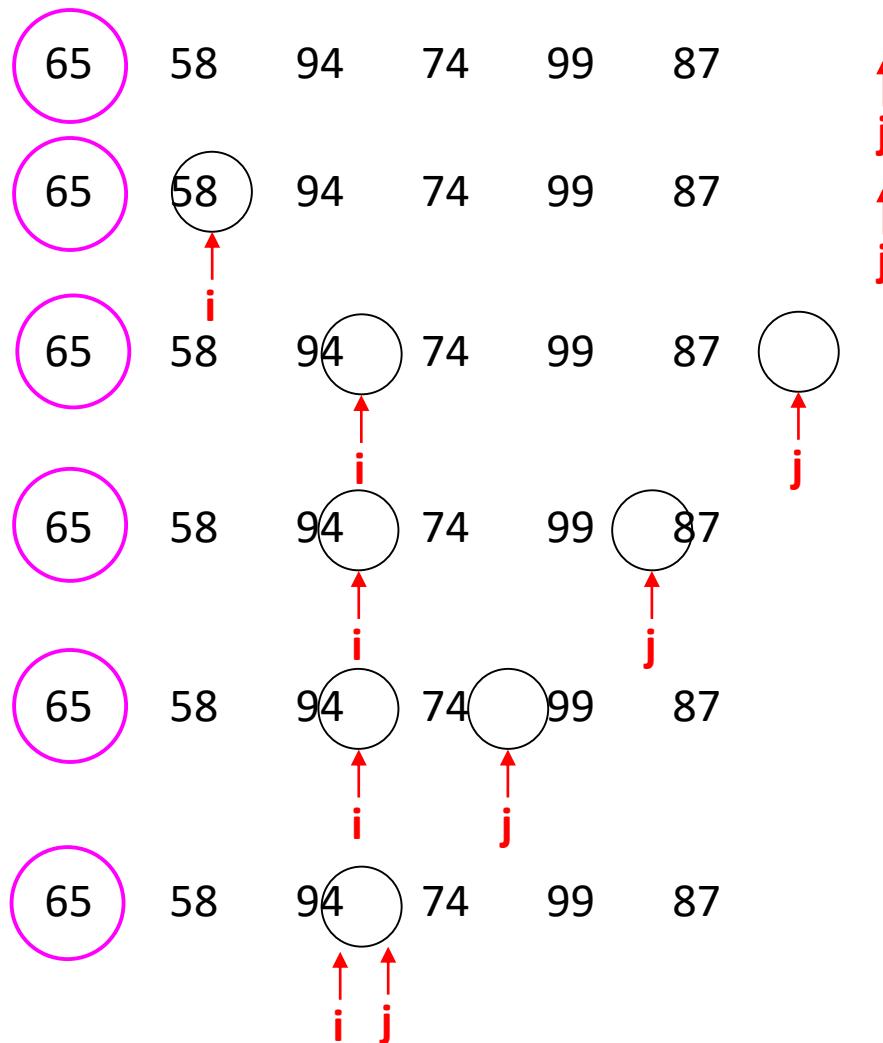
UB = 10

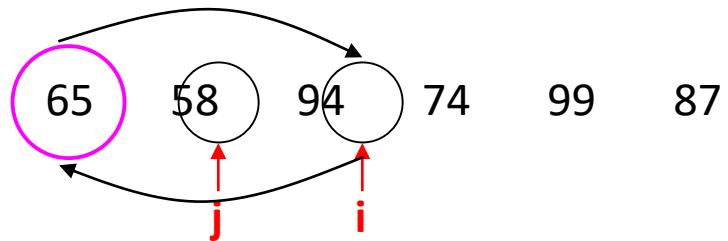
Right Sub Array [R1]

K [LB] = 65

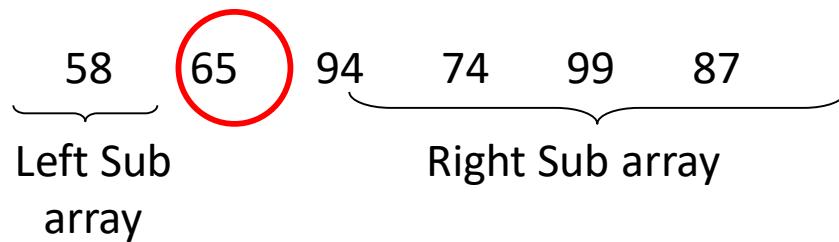
K [UB] = 87

QS (K , 5 , 10)





$i < j$
Condition Violates
FLAG \leftarrow False



Time Complexity

- **WORST CASE:** If we consider above partition strategy where last element is always picked as pivot.
- The worst case would occur when the array is already sorted in increasing or decreasing order.
- The solution of above recurrence is $O(n^2)$.

Time Complexity

AVERAGE/BEST CASE COMPLEXITY:

- The partition is exactly equally divided , then the quick sort algorithm will take $O(\log n)$ time as it calls partition and quick sort recursively for various values of p,r.
- The call will happen for n times , hence the time taken will be $O(n\log n)$

- Although the worst case time complexity of Quick Sort is $O(n^2)$ which is more than many other sorting algorithms like Merge Sort and Heap Sort
- Quick Sort is faster in practice, because its inner loop can be efficiently implemented on most architectures, and in most real-world data.
- Quick Sort can be implemented in different ways by changing the choice of pivot, so that the worst case rarely occurs for a given type of data.
- However, merge sort is generally considered better when data is huge and stored in external storage.

Review: Dynamic Programming

- ◆ *Dynamic programming* is another strategy for designing algorithms
- ◆ Use when problem breaks down into recurring small subproblems

Review: Optimal Substructure of LCS

$$c[i, j] = \begin{cases} c[i - 1, j - 1] + 1 & \text{if } x[i] = y[j], \\ \max(c[i, j - 1], c[i - 1, j]) & \text{otherwise} \end{cases}$$

- ◆ Observation 1: Optimal substructure
 - » A simple recursive algorithm will suffice
- ◆ Observation 2: Overlapping subproblems
 - » *Find some places where we solve the same subproblem more than once*

Review: Structure of Subproblems

- ◆ For the LCS problem:
 - » There are few subproblems in total
 - » And many recurring instances of each
 - (unlike divide & conquer, where subproblems unique)
- ◆ *How many distinct problems exist for the LCS of $x[1..m]$ and $y[1..n]$?*
- ◆ A: mn

Memoization

- ◆ *Memoization* is another way to deal with overlapping subproblems
 - » After computing the solution to a subproblem, store in a table
 - » Subsequent calls just do a table lookup
- ◆ Can modify recursive alg to use memoization:
 - » There are mn subproblems
 - » *How many times is each subproblem wanted?*
 - » *What will be the running time for this algorithm? The running space?*

Review: Dynamic Programming

- ◆ *Dynamic programming*: build table bottom-up
 - » Same table as memoization, but instead of starting at (m,n) and recursing down, start at $(1,1)$
- ◆ *Least Common Subsequence*: LCS easy to calculate from LCS of prefixes
 - As your homework shows, can actually reduce space to $O(\min(m,n))$

Review: Dynamic Programming

- ◆ Summary of the basic idea:
 - » Optimal substructure: optimal solution to problem consists of optimal solutions to subproblems
 - » Overlapping subproblems: few subproblems in total, many recurring instances of each
 - » Solve bottom-up, building a table of solved subproblems that are used to solve larger ones
- ◆ Variations:
 - » “Table” could be 3-dimensional, triangular, a tree, etc.

Greedy Algorithms

Overview

- ◆ Like dynamic programming, used to solve optimization problems.
- ◆ Dynamic programming can be overkill; greedy algorithms tend to be easier to code
- ◆ Problems exhibit optimal substructure (like DP).
- ◆ Problems also exhibit the **greedy-choice** property.
 - » When we have a choice to make, make the one that looks best *right now*.
 - » Make a **locally optimal choice** in hope of getting a **globally optimal solution**.

Greedy Strategy

- ◆ The choice that seems best at the moment is the one we go with.
 - » Prove that when there is a choice to make, one of the optimal choices is the greedy choice. Therefore, it's always safe to make the greedy choice.
 - » Show that all but one of the subproblems resulting from the greedy choice are empty.

Greedy algorithms

- ◆ A *greedy algorithm* always makes the choice that looks best at the moment
 - » My everyday examples:
 - Driving in Los Angeles, NY, or Boston for that matter
 - Playing cards
 - Invest on stocks
 - Choose a university
 - » The hope: a locally optimal choice will lead to a globally optimal solution
 - » For some problems, it works
- ◆ greedy algorithms tend to be easier to code

An Activity Selection Problem

(Conference Scheduling Problem)

- ◆ **Input:** A set of activities $S = \{a_1, \dots, a_n\}$
- ◆ Each activity has start time and a finish time
 - » $a_i = (s_i, f_i)$
- ◆ Two activities are compatible if and only if their interval does not overlap
- ◆ **Output:** a maximum-size subset of mutually compatible activities

Activity-Selection Problem

For a set of proposed activities that wish to use a lecture hall, select a maximum-size subset of “compatible activities”.



- Set of activities: $S = \{a_1, a_2, \dots, a_n\}$
- Duration of activity a_i : $[start_time_i, finish_time_i]$
- Activities *sorted in increasing order of finish time*:

i	1	2	3	4	5	6	7	8	9	10	11
start_time _i	1	3	0	5	3	5	6	8	8	2	12
finish_time _i	5	6	7	8	9	10	11	12	13	14	4

The Activity Selection Problem

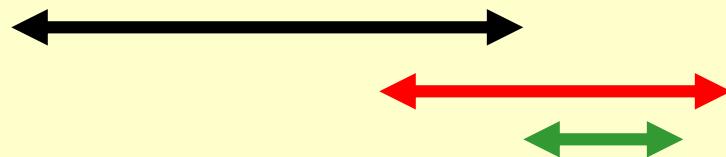
- ◆ Here are a set of start and finish times

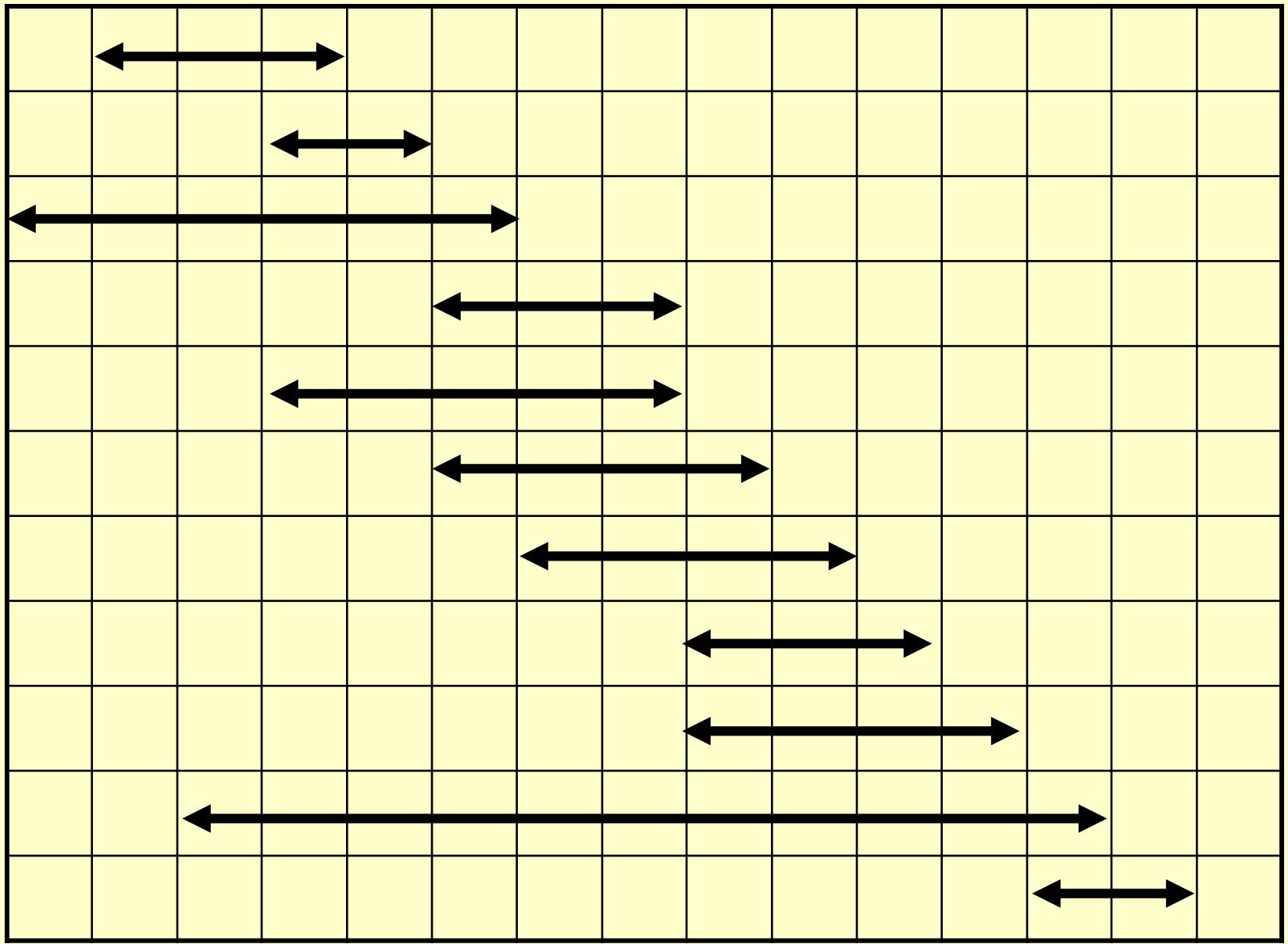
i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

- What is the maximum number of activities that can be completed?
 - $\{a_3, a_9, a_{11}\}$ can be completed
 - But so can $\{a_1, a_4, a_8, a_{11}\}$ which is a larger set
 - But it is not unique, consider $\{a_2, a_4, a_9, a_{11}\}$

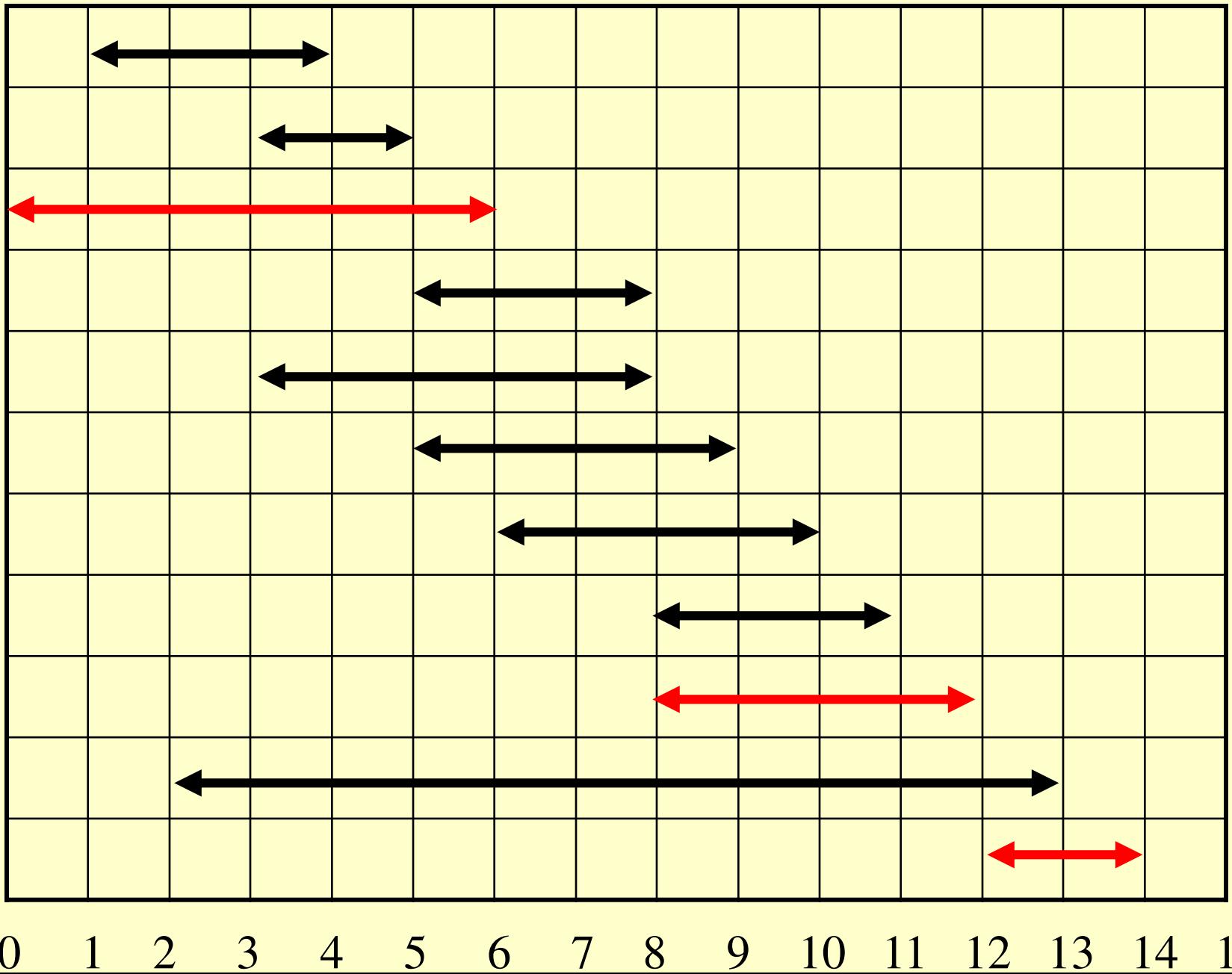
Interval Representation

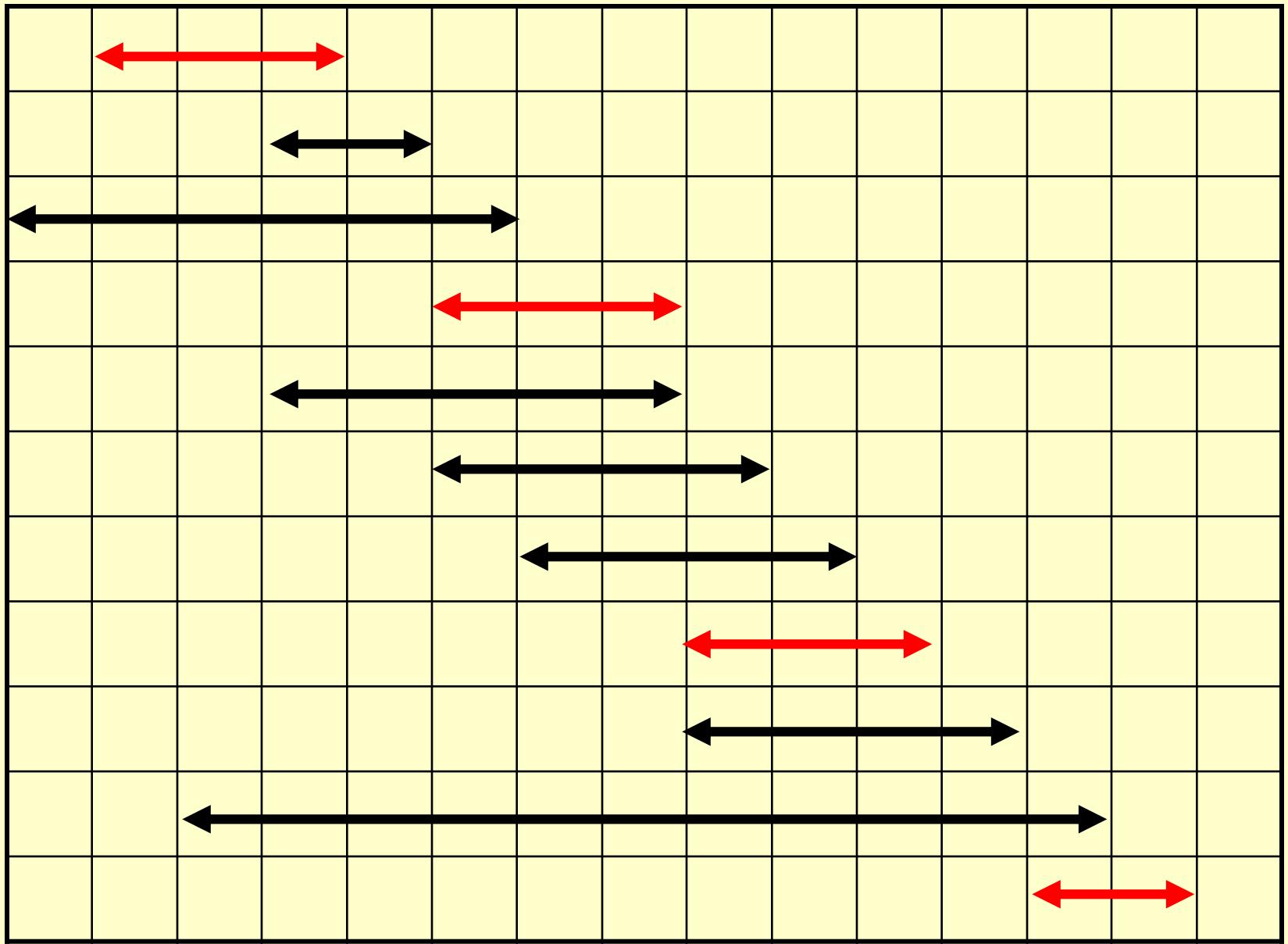
i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14



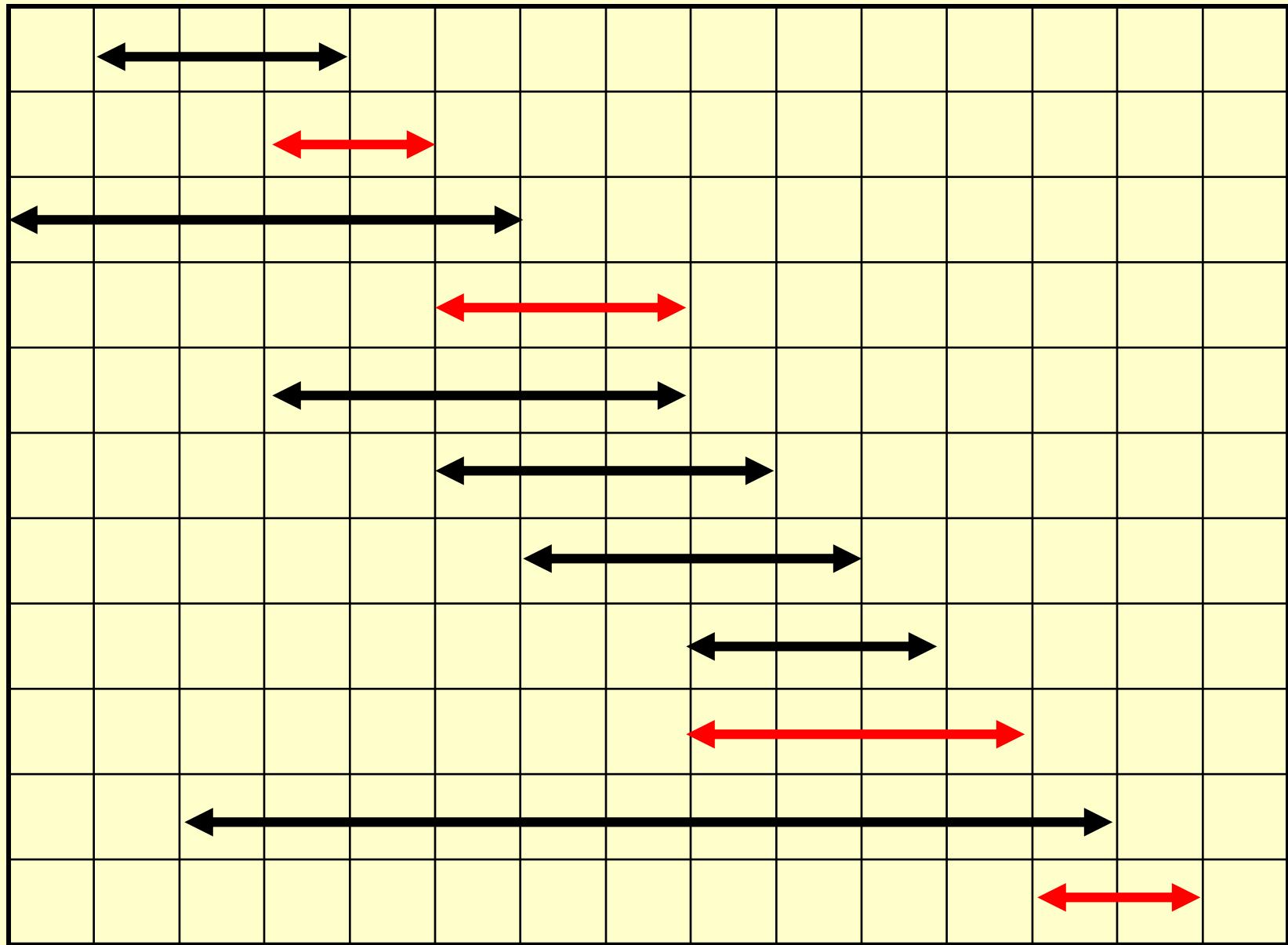


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15





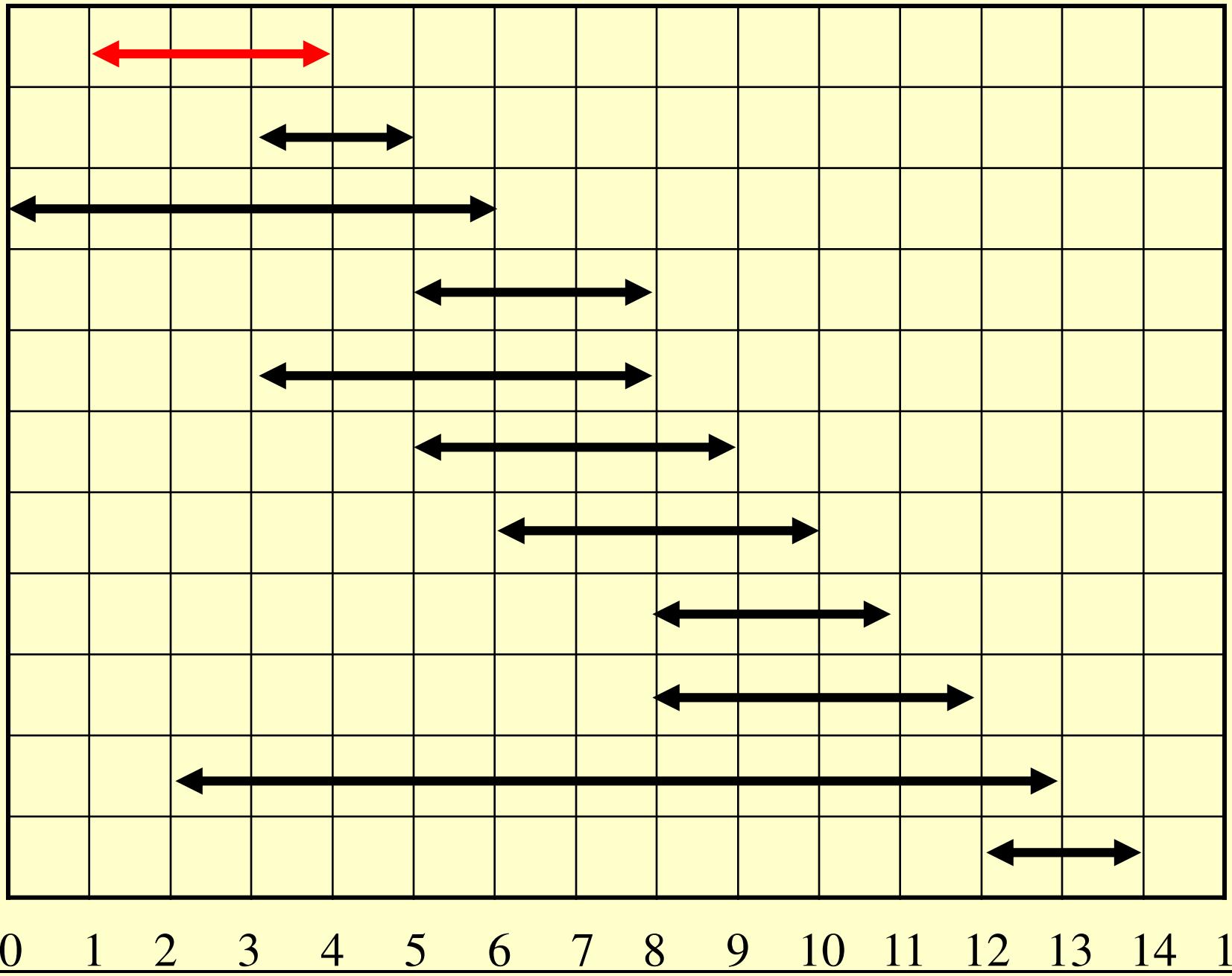
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

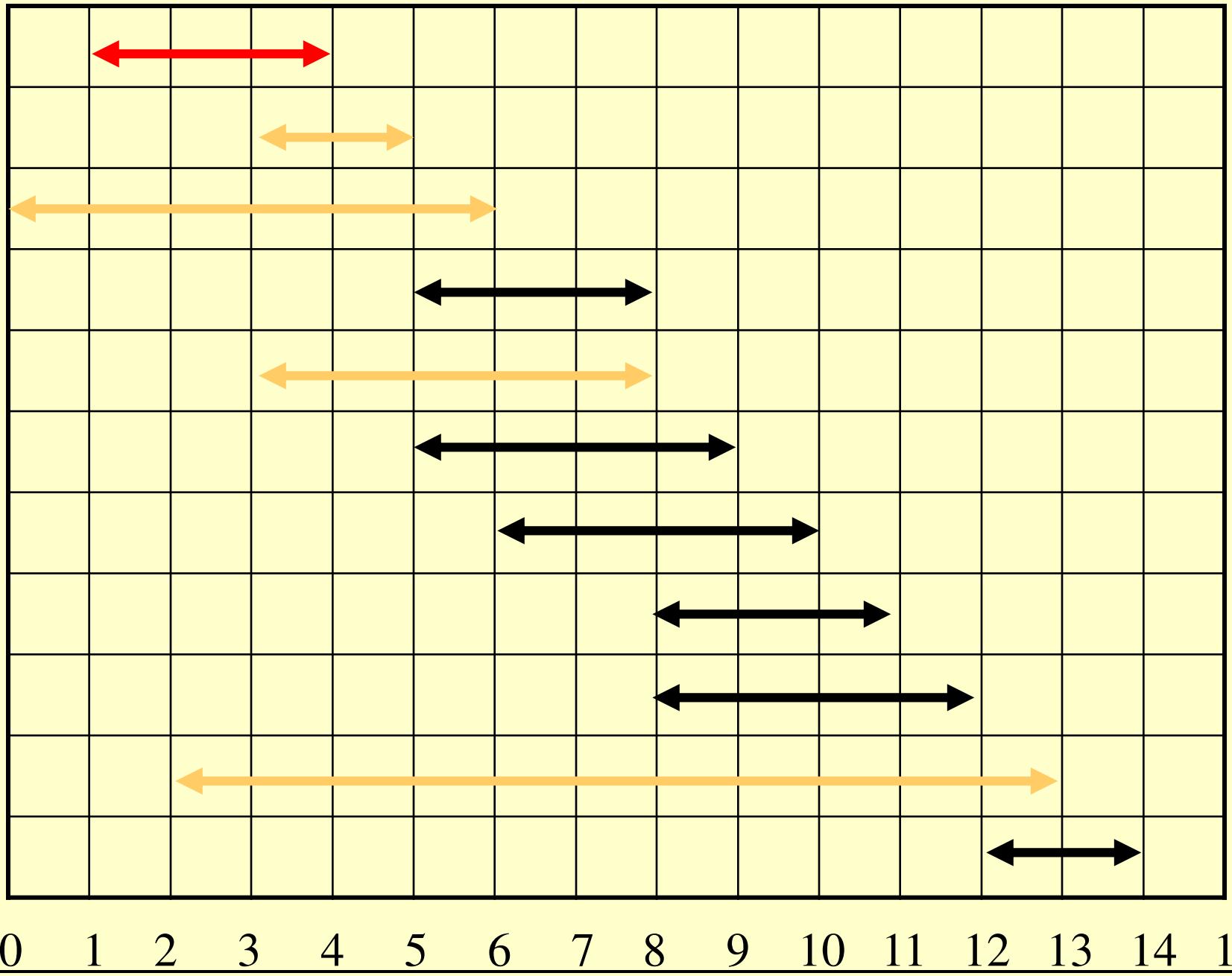


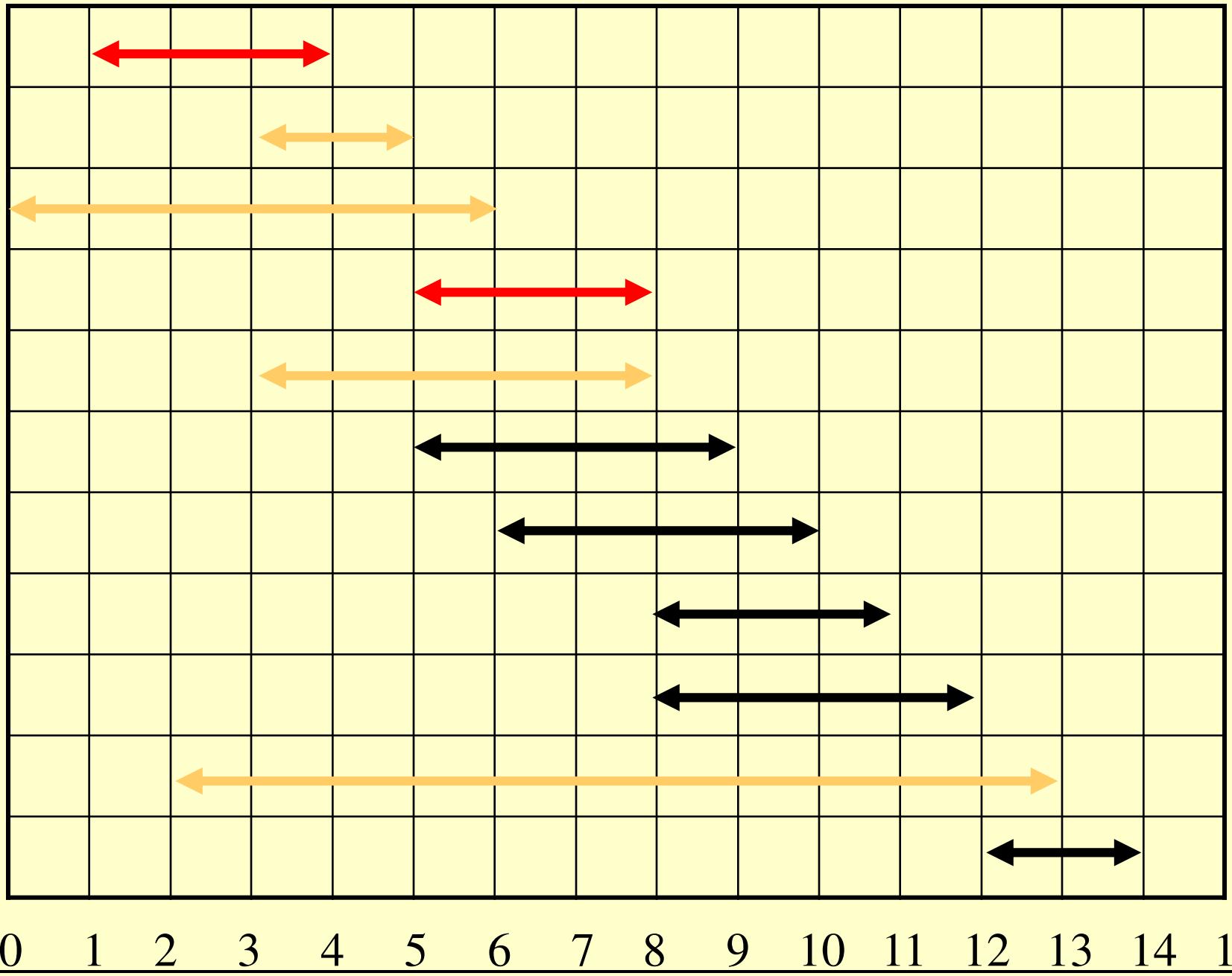
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

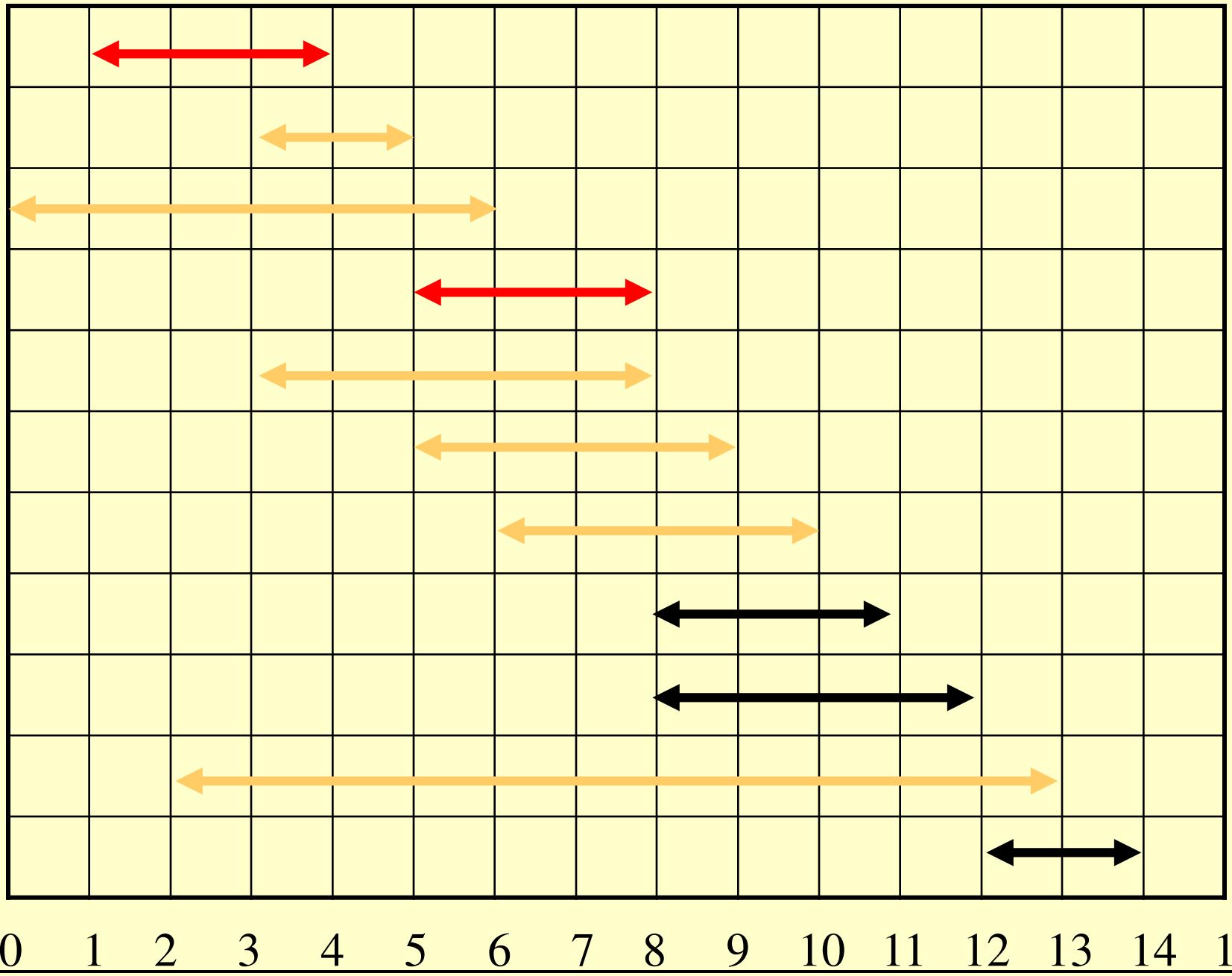
Early Finish Greedy

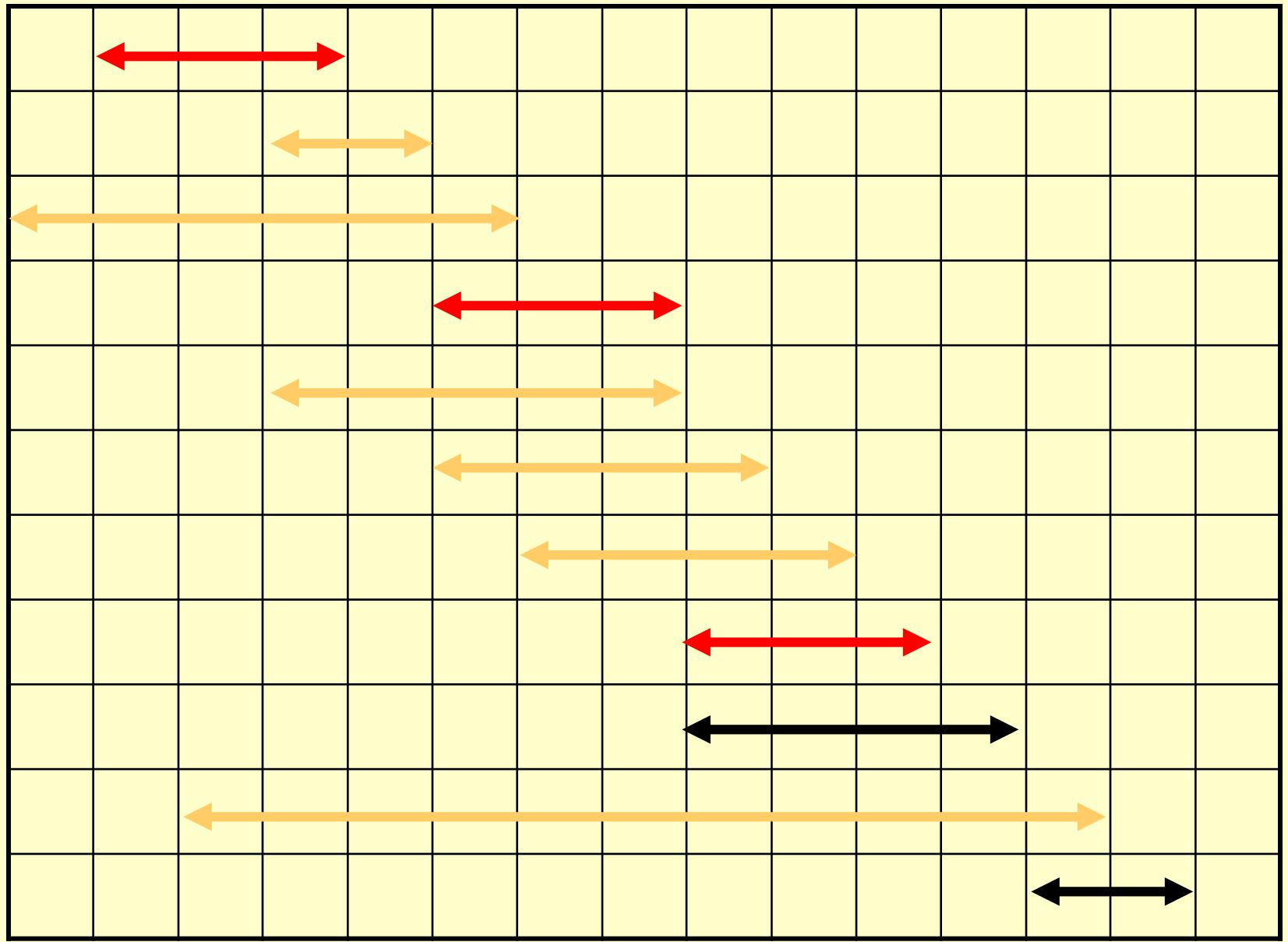
- ◆ Select the activity with the earliest finish
- ◆ Eliminate the activities that could not be scheduled
- ◆ Repeat!



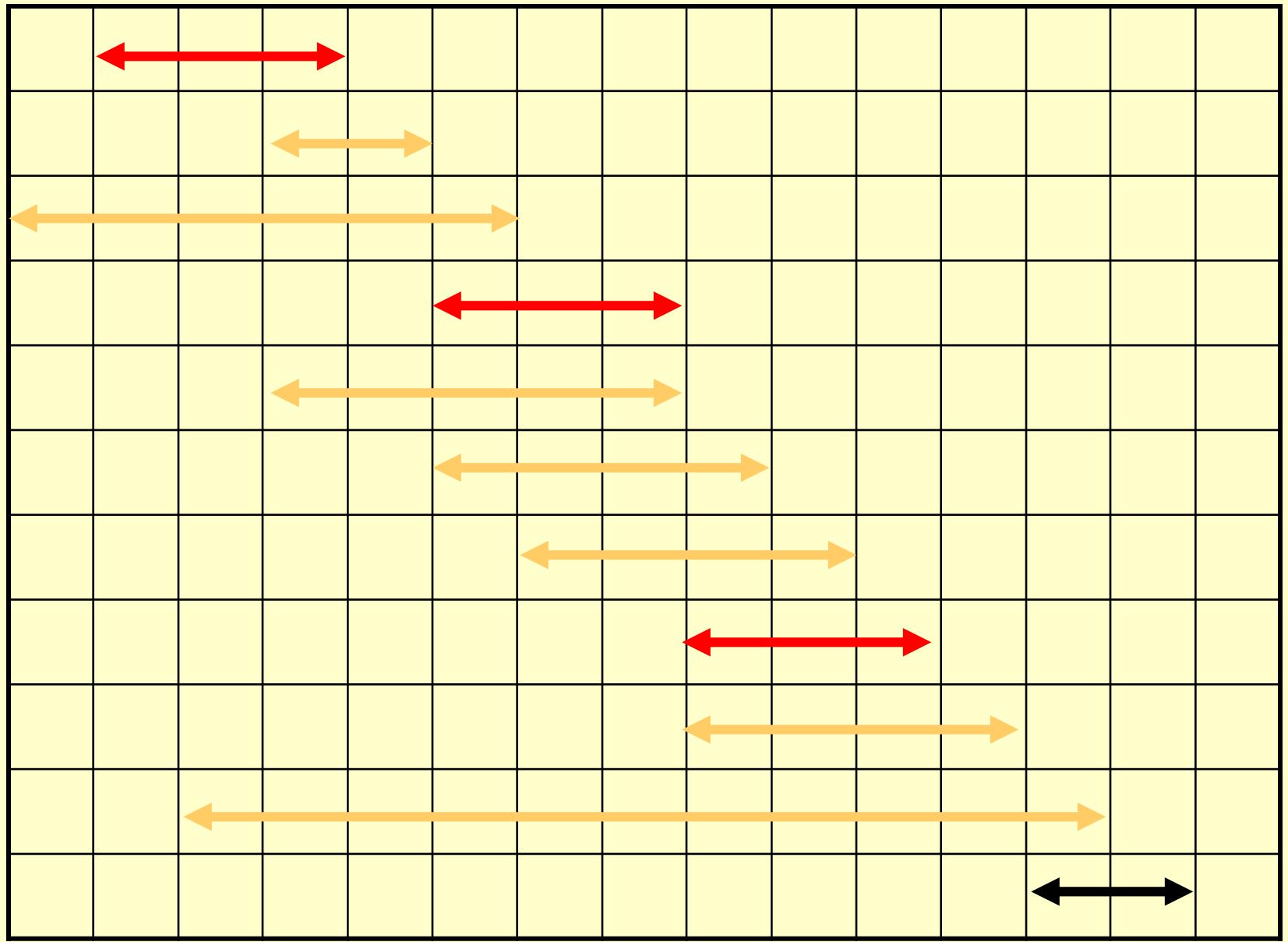




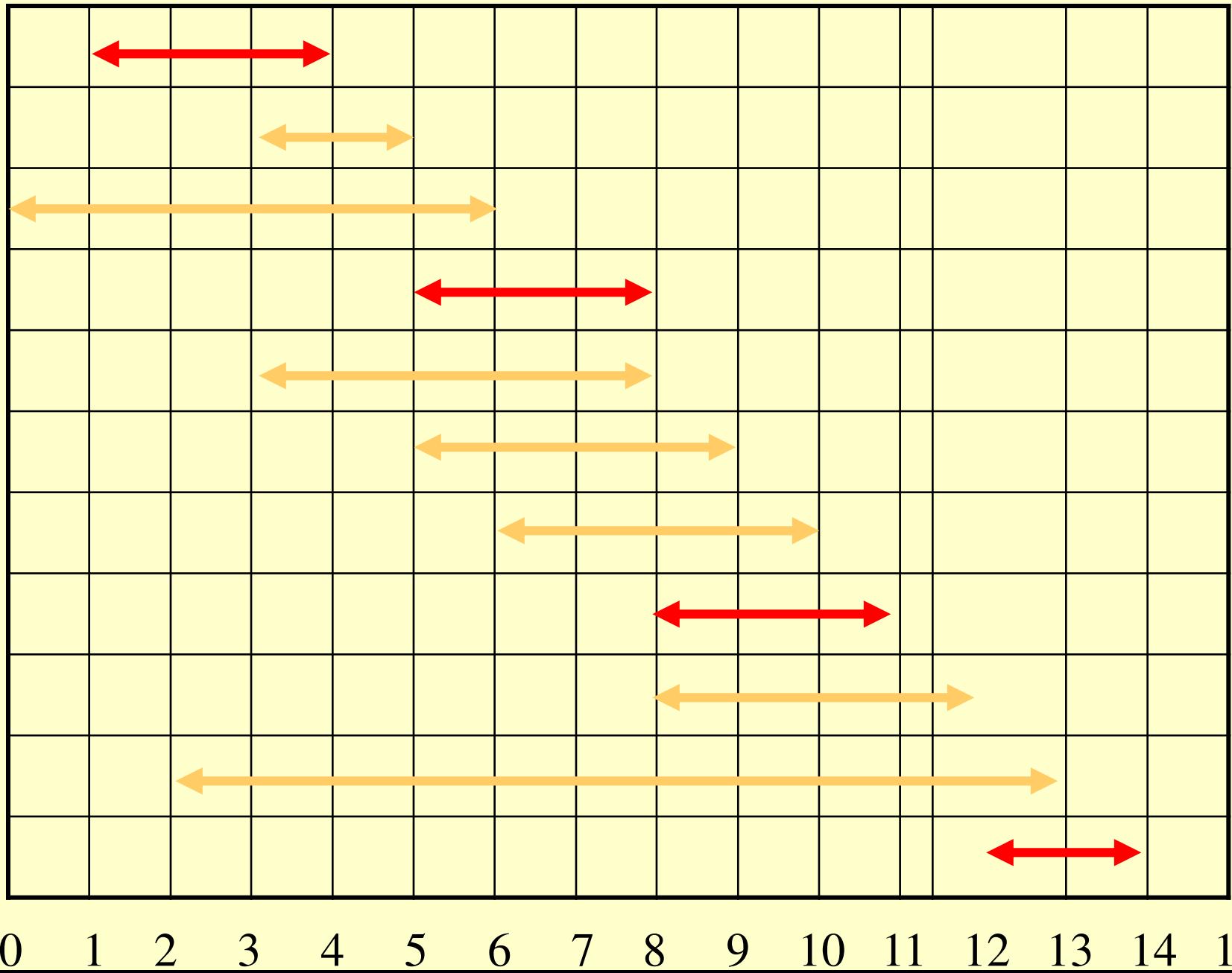




0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15



0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15



Assuming activities are sorted by finish time

GREEDY-ACTIVITY-SELECTOR(s, f)

```
1   $n \leftarrow \text{length}[s]$ 
2   $A \leftarrow \{a_1\}$ 
3   $i \leftarrow 1$ 
4  for  $m \leftarrow 2$  to  $n$ 
5      do if  $s_m \geq f_i$ 
6          then  $A \leftarrow A \cup \{a_m\}$ 
7               $i \leftarrow m$ 
8  return  $A$ 
```

Why it is Greedy?

- ◆ Greedy in the sense that it leaves as much opportunity as possible for the remaining activities to be scheduled
- ◆ The greedy choice is the one that maximizes the amount of unscheduled time remaining

Elements of Greedy Strategy

- ◆ An greedy algorithm makes a sequence of choices, each of the choices that seems best at the moment is chosen
 - » NOT always produce an optimal solution
- ◆ Two ingredients that are exhibited by most problems that lend themselves to a greedy strategy
 - » Greedy-choice property
 - » Optimal substructure

Greedy-Choice Property

- ◆ A globally optimal solution can be arrived at by making a locally optimal (greedy) choice
 - » Make whatever choice seems best at the moment and then solve the sub-problem arising after the choice is made
 - » The choice made by a greedy algorithm may depend on choices so far, but it cannot depend on any future choices or on the solutions to sub-problems
- ◆ Of course, we must prove that a greedy choice at each step yields a globally optimal solution

Activity-Selection Problem

- ◆ Problem: get your money's worth out of a festival
 - » Buy a wristband that lets you onto any ride
 - » Lots of rides, each starting and ending at different times
 - » Your goal: ride as many rides as possible
 - Another, alternative goal that we don't solve here: maximize time spent on rides

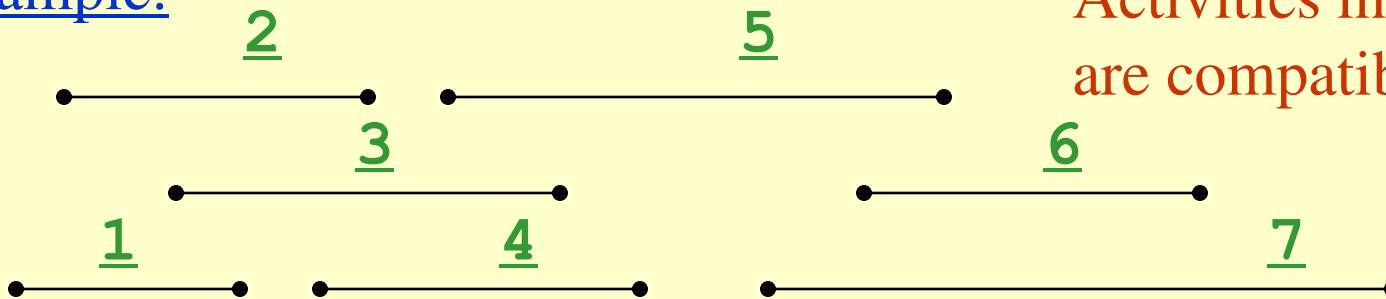
The *activity selection problem is Given below*

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

Activity-selection Problem

- ◆ Input: Set S of n activities, a_1, a_2, \dots, a_n .
 - » s_i = start time of activity i .
 - » f_i = finish time of activity i .
- ◆ Output: Subset A of maximum number of compatible activities.
 - » Two activities are compatible, if their intervals don't overlap.

Example:



Activities in each line
are compatible.

Optimal Substructure

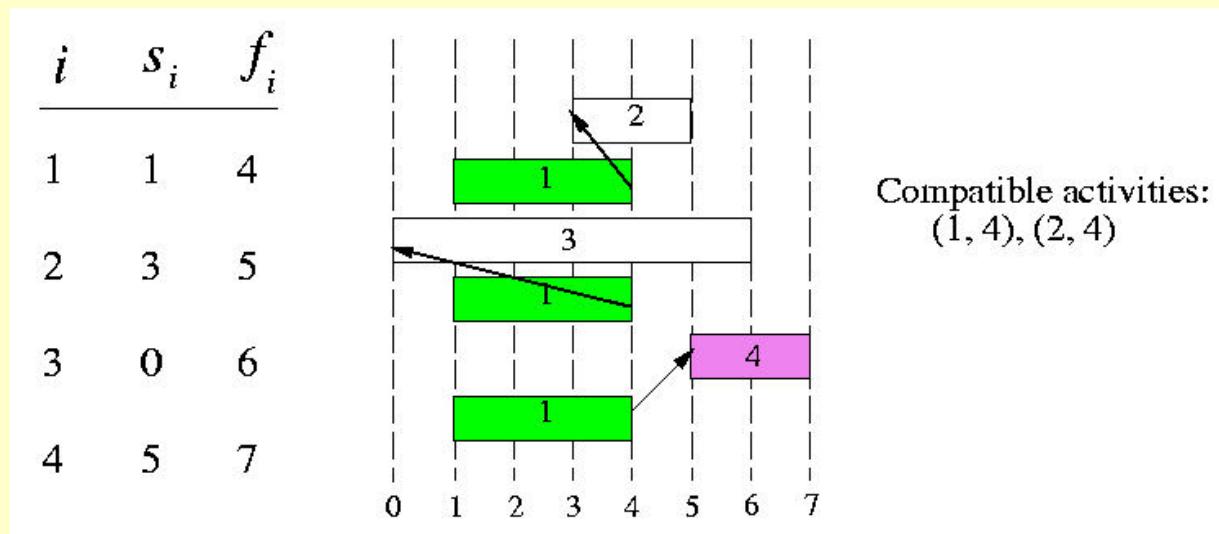
- ◆ Assume activities are sorted by finishing times.
 - » $f_1 \leq f_2 \leq \dots \leq f_n$.
- ◆ Suppose an optimal solution includes activity a_k .
 - » This generates two subproblems.
 - » Selecting from a_1, \dots, a_{k-1} , activities compatible with one another, and that finish before a_k starts (compatible with a_k).
 - » Selecting from a_{k+1}, \dots, a_n , activities compatible with one another, and that start after a_k finishes.
 - » The solutions to the two subproblems must be optimal.
 - Prove using the cut-and-paste approach.

Greedy-choice Property

- ◆ The problem also exhibits the **greedy-choice property**.
 - » There is an optimal solution to the subproblem S_{ij} , that includes the activity with the smallest finish time in set S_{ij} .
 - » Can be proved easily.
- ◆ Hence, **there is an optimal solution to S that includes a_1** .
- ◆ Therefore, **make this greedy choice** without solving subproblems first and evaluating them.
- ◆ Solve the subproblem that ensues as a result of making this greedy choice.
- ◆ Combine the greedy choice and the solution to the subproblem.

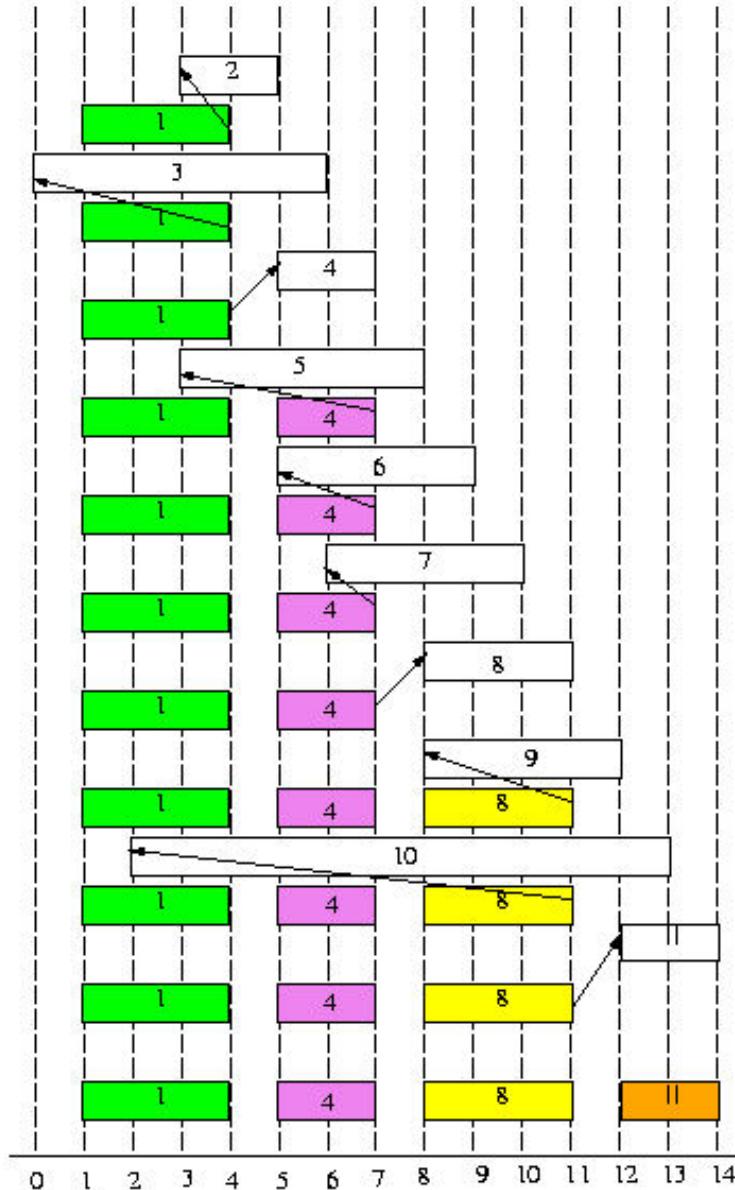
A Greedy Algorithm

- ◆ A **greedy algorithm** always makes the choice that looks best at the moment.
- ◆ **An Activity-Selection Problem:** Given a set $S = \{1, 2, \dots, n\}$ of n proposed activities, with a start time s_i and a finish time f_i for each activity i , select a maximum-size set of mutually compatible activities.
 - » If selected, activity i takes place during the half-open time interval $[s_i, f_i)$.
 - » Activities i and j are **compatible** if $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap (i.e., $s_i \geq f_j$ or $s_j \geq f_i$).

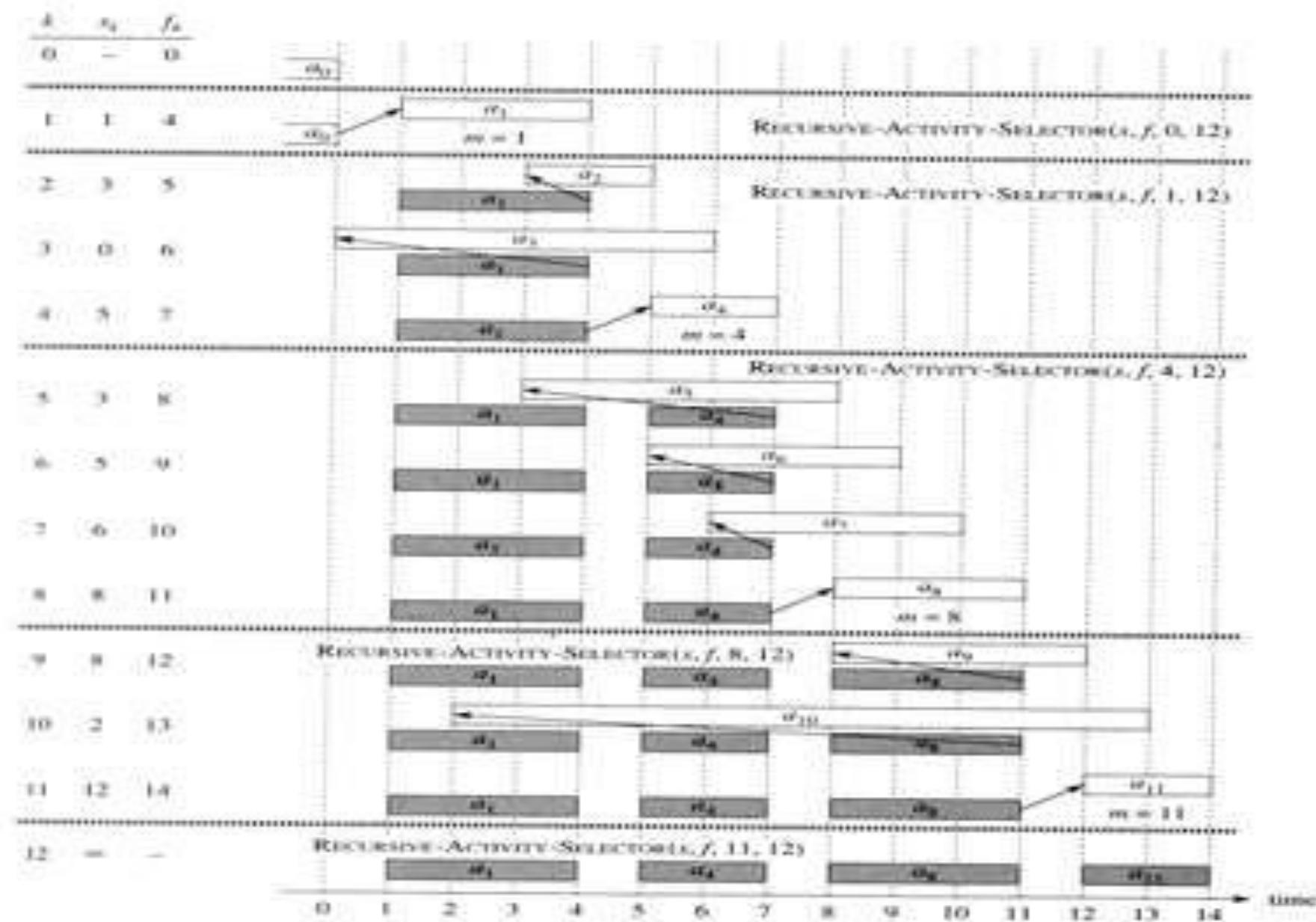


Activity Selection

i	s_i	f_i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14



1. Sort f_i
2. Select the first activity.
3. Pick the first activity i such that $s_i \geq f_j$ where activity j is the most recently selected activity.



Iterative algorithm

GREEDY-ACTIVITY-SELECTOR(s, f)

```
1   $n \leftarrow \text{length}[s]$ 
2   $A \leftarrow \{a_1\}$ 
3   $i \leftarrow 1$ 
4  for  $m \leftarrow 2$  to  $n$ 
5      do if  $s_m \geq f_i$ 
6          then  $A \leftarrow A \cup \{a_m\}$ 
7           $i \leftarrow m$ 
8  return  $A$ 
```

f_k is always max –finish time of any activity in A

- Line 2-3 select activity a_1
- Line 4-7 find earliest activity in S_{ij}
- Line 6-7 add activity a_m to A
- Time is $O(n)$

Typical Steps

- ◆ Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
- ◆ Prove that there's always an optimal solution that makes the greedy choice, so that the greedy choice is always safe.
- ◆ Show that greedy choice and optimal solution to subproblem \Rightarrow optimal solution to the problem.
- ◆ Make the greedy choice and **solve top-down**.
- ◆ May have to **preprocess** input to put it into **greedy order**.
 - » Example: Sorting activities by finish time.

Activity Selection:

A Greedy Algorithm

- ◆ So actual algorithm is simple:
 - » Sort the activities by finish time
 - » Schedule the first activity
 - » Then schedule the next activity in sorted list which starts after previous activity finishes
 - » Repeat until no more activities
- ◆ Intuition is even more simple:
 - » Always pick the shortest ride available at the time

GREEDY-ACTIVITY-SELECTOR(s, f)

```
1   $n \leftarrow \text{length}[s]$ 
2   $A \leftarrow \{a_1\}$ 
3   $i \leftarrow 1$ 
4  for  $m \leftarrow 2$  to  $n$ 
5      do if  $s_m \geq f_i$ 
6          then  $A \leftarrow A \cup \{a_m\}$ 
7           $i \leftarrow m$ 
8  return  $A$ 
```

Elements of Greedy Algorithms

- ◆ Greedy-choice Property.
 - » A globally optimal solution can be arrived at by making a locally optimal (greedy) choice.
- ◆ Optimal Substructure.

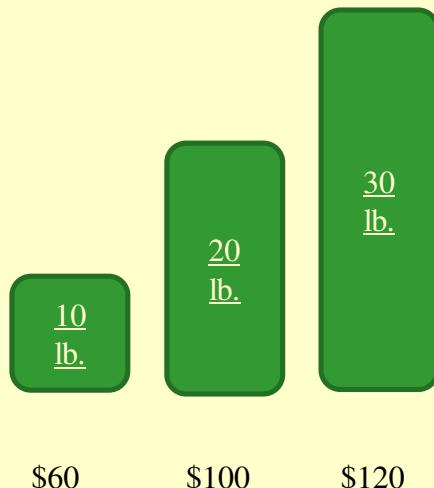
Knapsack Problem

Fractional knapsack and 0-1 knapsack

- ◆ After you break into a house, how to choose the items you put into your knapsack, to maximize your income.
- ◆ Each item has a weight and a value
- ◆ Your knapsack has a maximum weight
- ◆ 0-1 knapsack
 - » You can only take an item or leave it there
- ◆ Fractional knapsack
 - » You can take part of an item

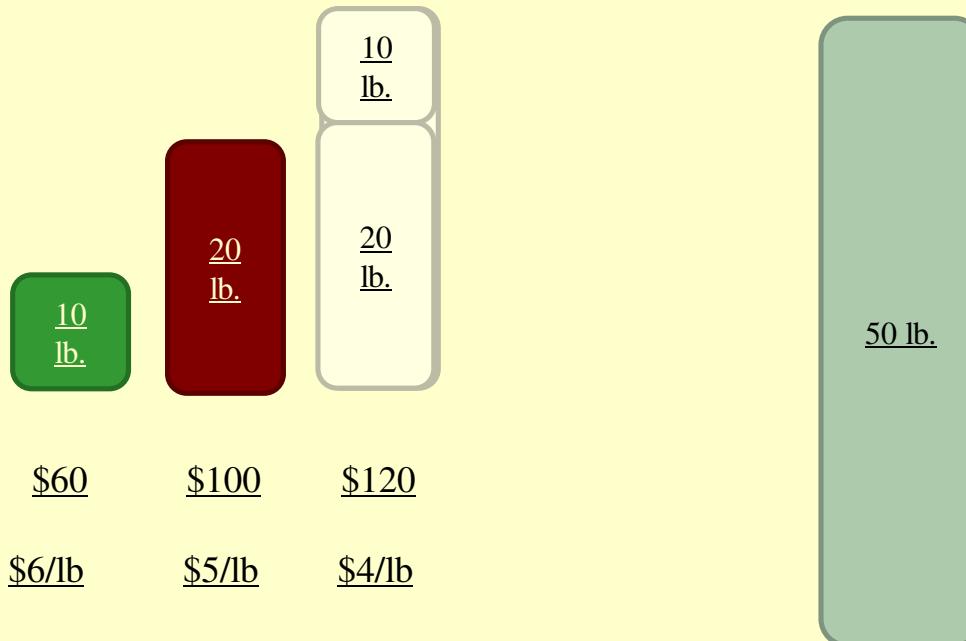
Fractional knapsack

- ◆ The **fractional knapsack problem** is a classic problem that can be solved by greedy algorithms
- ◆ E.g.
 - » your knapsack can contain 50 lb. Stuff;
 - » the items are as in the figure
 - » What is your algorithm?



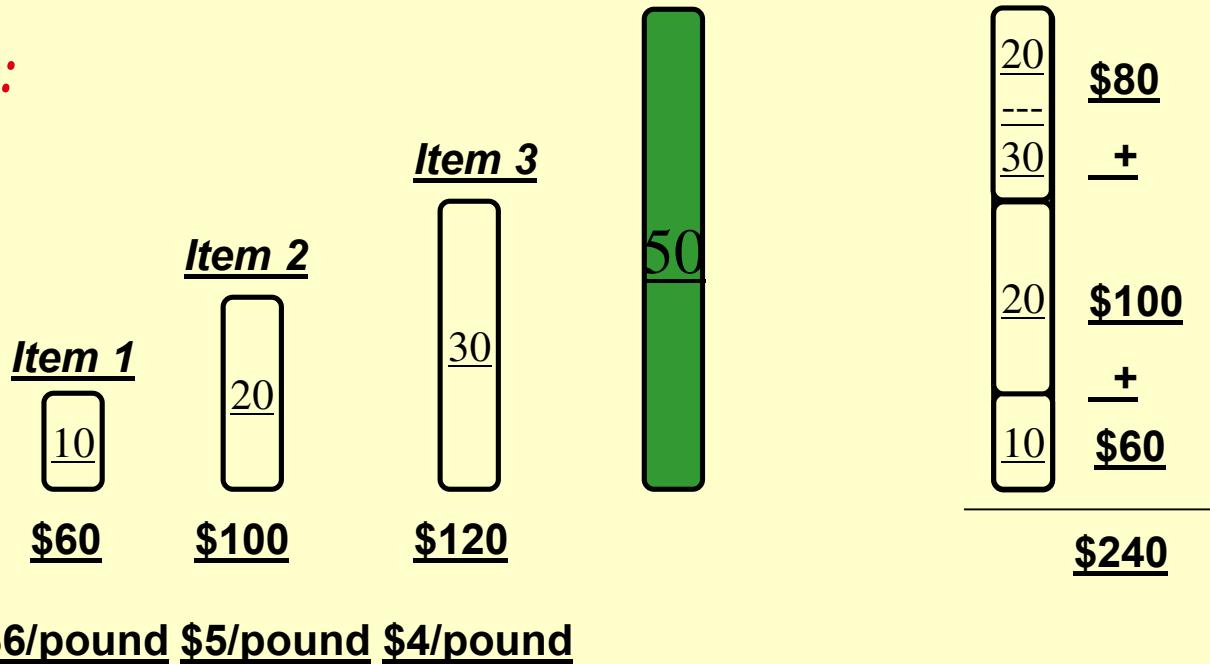
Fractional knapsack

- ◆ A greedy algorithm for **fractional knapsack problem**
- ◆ Greedy choice: choose the maximum value/lb. item



Fractional Knapsack - Example

◆ *E.g.:*



Greedy algorithm for fractional knapsack problem

- ◆ Compute Ratio= value/weight for each item
- ◆ Sort items by their ratio into decreasing order
 - » **Call the remaining item with the highest ratio the Most Valuable Item (MVI)**
- ◆ Iteratively:
 - » If the weight limit can not be reached by adding MVI
 - Select MVI
 - » Otherwise select MVI partially until weight limit

Example

<u>item</u>	<u>Weight (LB)</u>	<u>Value (\$)</u>	<u>\$ / LB</u>
1	2	2	1
2	4	3	0.75
3	3	3	1
4	5	6	1.2
5	2	4	2
6	6	9	1.5

- ◆ Weight limit: 10

Here

- ◆ Ratio = Value/Weight

$$Item\ 1 = 2/2 = 1$$

$$Item\ 2 = 3/4 = 0.75$$

$$Item\ 3 = 3/3 = 1$$

$$Item\ 4 = 6/5 = 1.2$$

$$Item\ 5 = 4/2 = 2$$

$$Item\ 6 = 9/6 = 1.5$$

Example

item	Weight (LB)	Value (\$)	\$ / LB
5	2	4	2
6	6	9	1.5
4	5	6	1.2
1	2	2	1
3	3	3	1
2	4	3	0.75

- ◆ Weight limit: 10
- ◆ Take item 5
 - » 2 LB, \$4
- ◆ Take item 6
 - » $(2+6)=8$ LB, $(4+9)=\$13$
- ◆ Take 2 LB of item 4
 - $(8+2)=10$ LB,
 - $(2/5)*6=2.4 + 13 = 15.4$

Example -2

- ◆ Find the optimal solution for the fractional problem using the Greedy approach. Total Weight=60

Item	Weight	Value	Ratio=Value/Weight
1	5	30	$30/5=6$
2	10	40	$40/10=4$
3	15	45	$45/15=3$
4	22	77	$77/22=3.5$
5	25	90	$90/25=3.6$

- ◆ Step:1 Calculate the **ratio= value/weight** for each item
- ◆ Step :2 **Sort the items on the basis of the ratio** in **descending order.**
- ◆ Step:3 Starting from the item with **the highest ratio**, start **putting the items into the KnapSack** . Take as much items as you can.

After sorting we have

Item	Weight	Value	Ratio=Value/Weight
1	5	30	$30/5=6$
2	10	40	$40/10=4$
5	25	90	$90/25= 3.6$
4	22	77	$77/22=3.5$
3	15	45	$45/15 =3$

Weight	Item in the KnapSack	Cost
Total= 60	Null	0
60- 5=55	1	30
55- 10=45	1,2	$30+40= 70$
45- 25=20	1,2,5	$70+90= 160$
0	1,2,5,4	$160+(20/22)*77= 230$ Here value of 20 item out 22 is calculate as $(20/22)*77$

- ◆ Therefore we have items 1,2,5,4 in the Knapsack with optimal value 230

The Knapsack Problem

- ◆ The famous *knapsack problem*:
 - » A thief breaks into a museum. Fabulous paintings, sculptures, and jewels are everywhere. The thief has a good eye for the value of these objects, and knows that each will fetch hundreds or thousands of dollars on the clandestine art collector's market. But, the thief has only brought a single knapsack to the scene of the robbery, and can take away only what he can carry. What items should the thief take to maximize the haul?

0-1 Knapsack problem

- ◆ Given a knapsack with maximum capacity W , and a set S consisting of n items
- ◆ Each item i has some weight w_i and benefit value b_i (all w_i , b_i and W are integer values)
- ◆ Problem: How to pack the knapsack to achieve maximum total value of packed items?

0-1 Knapsack problem: a picture

<u>Items</u>	<u>Weight</u> w_i	<u>Benefit value</u> b_i
	2	3
	3	4
	4	5
	5	8
<u>This is a knapsack</u>		
<u>Max weight: $W = 20$</u>		
<u>$W = 20$</u>		
	9	10

The Knapsack Problem

- ◆ More formally, the *0-1 knapsack problem*:
 - » The thief must choose among n items, where the i th item worth v_i dollars and weighs w_i pounds
 - » Carrying at most W pounds, maximize value
 - Note: assume v_i , w_i , and W are all integers
 - “0-1” b/c each item must be taken or left in entirety
- ◆ A variation, the *fractional knapsack problem*:
 - » Thief can take fractions of items
 - » Think of items in 0-1 problem as gold ingots, in fractional problem as buckets of gold dust

0-1 Knapsack problem

- ◆ Problem, in other words, is to find

$$\max \sum_{i \in T} b_i \text{ subject to } \sum_{i \in T} w_i \leq W$$

- The problem is called a “0-1” problem, because each item must be entirely accepted or rejected.
- Just another version of this problem is the “*Fractional Knapsack Problem*”, where we can take fractions of items.

0-1 Knapsack problem: brute-force approach

Let's first solve this problem with a straightforward algorithm

- ◆ Since there are n items, there are 2^n possible combinations of items.
- ◆ We go through all combinations and find the one with the most total value and with total weight less or equal to W
- ◆ Running time will be $O(2^n)$

0-1 Knapsack problem: brute-force approach

- ◆ Can we do better?
- ◆ Yes, with an algorithm based on dynamic programming
- ◆ We need to carefully identify the subproblems

Let's try this:

If items are labeled $1..n$, then a subproblem would be to find an optimal solution for $S_k = \{items\ labeled\ 1,\ 2,\ ..\ k\}$

Defining a Subproblem

If items are labeled $1..n$, then a subproblem would be to find an optimal solution for $S_k = \{items\ labeled\ 1, 2, .. k\}$

- ◆ This is a valid subproblem definition.
- ◆ The question is: can we describe the final solution (S_n) in terms of subproblems (S_k)?
- ◆ Unfortunately, we can't do that. Explanation follows....

Defining a Subproblem

$w_1 = 2$	$w_2 = 4$	$w_3 = 5$	$w_4 = 3$	
$b_1 = 3$	$b_2 = 5$	$b_3 = 8$	$b_4 = 4$	

?

Max weight: $W = 20$

For S_4 :

Total weight: 14;
total benefit: 20

$w_1 = 2$	$w_2 = 4$	$w_3 = 5$	$w_4 = 9$
$b_1 = 3$	$b_2 = 5$	$b_3 = 8$	$b_4 = 10$

For S_5 :

Total weight: 20
total benefit: 26

Item	Weight w_i	Benefit b_i
#		
1	2	3
2	3	4
3	4	5
4	5	8
5	9	10

Solution for S_4 is
not part of the
solution for S_5 !!!

Defining a Subproblem

(continued)

- ◆ As we have seen, the solution for S_4 is not part of the solution for S_5
- ◆ So our definition of a subproblem is flawed and we need another one!
- ◆ Let's add another parameter: w , which will represent the exact weight for each subset of items
- ◆ The subproblem then will be to compute $B[k, w]$

Recursive Formula for subproblems

■ Recursive formula for subproblems:

$$B[k, w] = \begin{cases} B[k - 1, w] & \text{if } w_k > w \\ \max\{ B[k - 1, w], B[k - 1, w - w_k] + b_k \} & \text{else} \end{cases}$$

- ◆ It means, that the best subset of S_k that has total weight w is one of the two:
 - 1) the best subset of S_{k-1} that has total weight w , or
 - 2) the best subset of S_{k-1} that has total weight $w-w_k$ plus the item k

Recursive Formula

$$B[k, w] = \begin{cases} B[k-1, w] & \text{if } w_k > w \\ \max\{ B[k-1, w], B[k-1, w-w_k] + b_k \} & \text{else} \end{cases}$$

- ◆ The best subset of S_k that has the total weight w , either contains item k or not.
- ◆ First case: $w_k > w$. Item k can't be part of the solution, since if it was, the total weight would be $> w$, which is unacceptable
- ◆ Second case: $w_k \leq w$. Then the item k can be in the solution, and we choose the case with greater value

The Knapsack Problem

And Optimal Substructure

- ◆ Both variations exhibit optimal substructure
- ◆ To show this for the 0-1 problem, consider the most valuable load weighing at most W pounds
 - » *If we remove item j from the load, what do we know about the remaining load?*
 - » A: remainder must be the most valuable load weighing at most $W - w_j$ that thief could take from museum, excluding item j

Solving The Knapsack Problem

- ◆ The optimal solution to the *fractional knapsack* problem can be found with a greedy algorithm
 - » *How?*
- ◆ The optimal solution to the 0-1 problem cannot be found with the same greedy strategy
 - » Greedy strategy: take in order of dollars/pound
 - » Example: 3 items weighing 10, 20, and 30 pounds, knapsack can hold 50 pounds
 - *Suppose item 2 is worth \$100. Assign values to the other items so that the greedy strategy will fail*

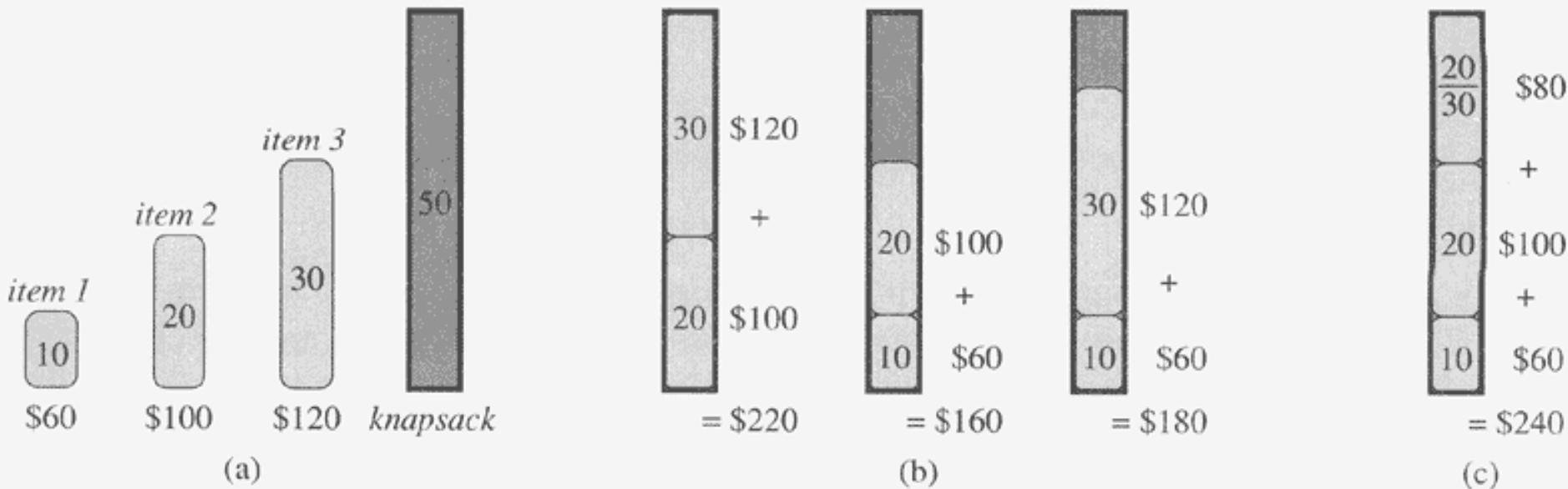


Figure 16.2 The greedy strategy does not work for the 0-1 knapsack problem. (a) The thief must select a subset of the three items shown whose weight must not exceed 50 pounds. (b) The optimal subset includes items 2 and 3. Any solution with item 1 is suboptimal, even though item 1 has the greatest value per pound. (c) For the fractional knapsack problem, taking the items in order of greatest value per pound yields an optimal solution.

The Knapsack Problem:

Greedy Vs. Dynamic

- ◆ The fractional problem can be solved greedily
- ◆ The 0-1 problem cannot be solved with a greedy approach
 - » As you have seen, however, it can be solved with dynamic programming

0-1 Knapsack Algorithm

for $w = 0$ to W

$$B[0,w] = 0$$

for $i = 0$ to n

$$B[i,0] = 0$$

for $w = 0$ to W

if $w_i \leq w$ // item i can be part of the solution

$$\text{if } b_i + B[i-1, w-w_i] > B[i-1, w]$$

$$B[i, w] = b_i + B[i-1, w - w_i]$$

else

$$B[i, w] = B[i-1, w]$$

else $B[i, w] = B[i-1, w]$ // $w_i > w$

Running time

for $w = 0$ to W

$O(W)$

$B[0,w] = 0$

for $i = 0$ to n

Repeat n times

$B[i,0] = 0$

for $w = 0$ to W

$O(W)$

< the rest of the code >

What is the running time of this algorithm?

$O(n*W)$

Remember that the brute-force algorithm
takes $O(2^n)$

Example

Let's run our algorithm on the following data:

$n = 4$ (# of elements)

$W = 8$ (max weight)

Elements (weight, benefit):

(4,10), (1,3), (2,6), (3,5)

Example (2)

<u>w</u>	<u>i</u>	0	1	2	3	4
0	0					
1	0					
2	0					
3	0					
4	0					
5	0					

for w = 0 to W
 B[0,w] = 0

Example (3)

<u>W</u>	<u>i</u>	0	1	2	3	4
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					
5	0					

for i = 0 to n
 B[i,0] = 0

Example (4)

<u>w</u>	<u>i</u>	0	1	2	3	4
0		0	0	0	0	0
1		0	→ 0			
2		0				
3		0				
4		0				
5		0				

Items:

1: (4,10)

2: (1,3)

3: (2,6)

4: (3,5)

$i=1$

$b_i=3$

$w_i=2$

$w=1$

$w-w_i=-1$

if $w_i \leq w$ // item i can be part of the solution

 if $b_i + B[i-1, w-w_i] > B[i-1, w]$

 $B[i, w] = b_i + B[i-1, w-w_i]$

 else

 $B[i, w] = B[i-1, w]$

 else $B[i, w] = B[i-1, w]$ // $w_i > w$

Example (5)

<u>w</u>	<u>i</u>	0	1	2	3	4
0	0	0	0	0	0	0
1	1	0	0			
2	2	0	3			
3	3	0				
4	4	0				
5	5	0				

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=1$

$b_i=3$

$w_i=2$

$w=2$

$w - w_i = 0$

if $w_i \leq w$ // item i can be part of the solution

 if $b_i + B[i-1, w-w_i] > B[i-1, w]$

 $B[i, w] = b_i + B[i-1, w - w_i]$

 else

 $B[i, w] = B[i-1, w]$

 else $B[i, w] = B[i-1, w]$ // $w_i > w$

Example (6)

<u>w</u>	<u>i</u>	0	1	2	3	4
0	0	0	0	0	0	0
1	0	0				
2	0	3				
3	0	3				
4	0					
5	0					

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=1$

$b_i=3$

$w_i=2$

$w=3$

$w-w_i=1$

if $w_i \leq w$ // item i can be part of the solution

 if $b_i + B[i-1, w-w_i] > B[i-1, w]$

 $B[i, w] = b_i + B[i-1, w - w_i]$

 else

 $B[i, w] = B[i-1, w]$

 else $B[i, w] = B[i-1, w]$ // $w_i > w$

Example (7)

<u>w</u>	<u>i</u>	0	1	2	3	4
0	0	0	0	0	0	0
1	0	0				
2	0	3				
3	0	3				
4	0	3				
5	0					

Items:
<u>1: (2,3)</u>
<u>2: (3,4)</u>
<u>3: (4,5)</u>
<u>4: (5,6)</u>
<u>$i=1$</u>
<u>$b_i=3$</u>
<u>$w_i=2$</u>
<u>$w=4$</u>
<u>$w-w_i=2$</u>

if $w_i \leq w$ // item i can be part of the solution

 if $b_i + B[i-1, w-w_i] > B[i-1, w]$

 $B[i, w] = b_i + B[i-1, w - w_i]$

 else

 $B[i, w] = B[i-1, w]$

 else $B[i, w] = B[i-1, w]$ // $w_i > w$

Example (8)

<u>w</u>	<u>i</u>	0	1	2	3	4
0	0	0	0	0	0	0
1	0	0				
2	0	3				
3	0	3				
4	0	3				
5	0	3				

<u>Items:</u>
<u>1: (2,3)</u>
<u>2: (3,4)</u>
<u>3: (4,5)</u>
<u>4: (5,6)</u>

$i=1$
 $b_i=3$
 $w_i=2$
 $w=5$
 $w-w_i=2$

if $w_i \leq w$ // item i can be part of the solution

 if $b_i + B[i-1, w-w_i] > B[i-1, w]$

 $B[i, w] = b_i + B[i-1, w - w_i]$

 else

 $B[i, w] = B[i-1, w]$

 else $B[i, w] = B[i-1, w]$ // $w_i > w$

Example (9)

<u>w</u>	<u>i</u>	0	1	2	3	4
0	0	0	0	0	0	0
1	0	0	→ 0			
2	0	3				
3	0	3				
4	0	3				
5	0	3				

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

i=2

b_i=4

w_i=3

w=1

w-w_i=-2

if w_i ≤ w // item i can be part of the solution

 if b_i + B[i-1, w-w_i] > B[i-1, w]

 B[i, w] = b_i + B[i-1, w-w_i]

 else

 B[i, w] = B[i-1, w]

else B[i, w] = B[i-1, w] // w_i > w

Example (10)

<u>w</u>	<u>i</u>	0	1	2	3	4
0	0	0	0	0	0	
1	0	0	0			
2	0	3 → 3				
3	0	3				
4	0	3				
5	0	3				

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

i=2

b_i=4

w_i=3

w=2

w-w_i=-1

if w_i ≤ w // item i can be part of the solution

 if b_i + B[i-1, w-w_i] > B[i-1, w]

 B[i, w] = b_i + B[i-1, w-w_i]

 else

 B[i, w] = B[i-1, w]

 else B[i, w] = B[i-1, w] // w_i > w

Example (11)

<u>w</u>	<u>i</u>	0	1	2	3	4
0	0	0	0	0	0	0
1	0	0	0			
2	0	3	3			
3	0	3	4			
4	0	3				
5	0	3				

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=2$

$b_i=4$

$w_i=3$

$w=3$

$w-w_i=0$

if $w_i \leq w$ // item i can be part of the solution

 if $b_i + B[i-1, w-w_i] > B[i-1, w]$

 $B[i, w] = b_i + B[i-1, w - w_i]$

 else

 $B[i, w] = B[i-1, w]$

 else $B[i, w] = B[i-1, w]$ // $w_i > w$

Example (12)

<u>w</u>	<u>i</u>	0	1	2	3	4
0	0	0	0	0	0	0
1	0	0	0			
2	0	3	3			
3	0	3	4			
4	0	3	4			
5	0	3				

<u>Items:</u>	
1:	(2,3)
2:	(3,4)
3:	(4,5)
4:	(5,6)

$\underline{i=2}$
 $\underline{b_i=4}$
 $\underline{w_i=3}$
 $\underline{w=4}$
 $\underline{w-w_i=1}$

if $w_i \leq w$ // item i can be part of the solution

if $b_i + B[i-1, w-w_i] > B[i-1, w]$

$B[i, w] = b_i + B[i-1, w - w_i]$

else

$B[i, w] = B[i-1, w]$

else $B[i, w] = B[i-1, w]$ // $w_i > w$

Example (13)

<u>w</u>	<u>i</u>	0	1	2	3	4
0	0	0	0	0	0	0
1	0	0	0			
2	0	3	3			
3	0	3	4			
4	0	3	4			
5	0	3	7			

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=2$

$b_i=4$

$w_i=3$

$w=5$

$w - w_i = 2$

if $w_i \leq w$ // item i can be part of the solution

 if $b_i + B[i-1, w-w_i] > B[i-1, w]$

 $B[i, w] = b_i + B[i-1, w - w_i]$

 else

 $B[i, w] = B[i-1, w]$

 else $B[i, w] = B[i-1, w]$ // $w_i > w$

Example (14)

<u>w</u>	<u>i</u>	0	1	2	3	4
0	0	0	0	0	0	0
1	0	0	0	0 → 0		
2	0	3	3	3 → 3		
3	0	3	4	4 → 4		
4	0	3	4			
5	0	3	7			

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

i=3

b_i=5

w_i=4

w=1..3

if w_i ≤ w // item i can be part of the solution

 if b_i + B[i-1, w-w_i] > B[i-1, w]

 B[i, w] = b_i + B[i-1, w-w_i]

 else

 B[i, w] = B[i-1, w]

else B[i, w] = B[i-1, w] // w_i > w

Example (15)

<u>w</u>	<u>i</u>	0	1	2	3	4
0	0	0	0	0	0	0
1	0	0	0	0		
2	0	3	3	3		
3	0	3	4	4		
4	0	3	4	5		
5	0	3	7			

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=3$

$b_i=5$

$w_i=4$

$w=4$

$w - w_i = 0$

if $w_i \leq w$ // item i can be part of the solution

 if $b_i + B[i-1, w-w_i] > B[i-1, w]$

 $B[i, w] = b_i + B[i-1, w - w_i]$

 else

 $B[i, w] = B[i-1, w]$

 else $B[i, w] = B[i-1, w]$ // $w_i > w$

Example (15)

<u>w</u>	<u>i</u>	0	1	2	3	4
0	0	0	0	0	0	
1	0	0	0	0		
2	0	3	3	3		
3	0	3	4	4		
4	0	3	4	5		
5	0	3	7	7	7	

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$$\underline{i=3}$$

$$\underline{b_i=5}$$

$$\underline{w_i=4}$$

$$\underline{w=5}$$

$$\underline{w - w_i = 1}$$

if $w_i \leq w$ // item i can be part of the solution

 if $b_i + B[i-1, w-w_i] > B[i-1, w]$

 $B[i, w] = b_i + B[i-1, w - w_i]$

 else

 $B[i, w] = B[i-1, w]$

 else $B[i, w] = B[i-1, w]$ // $w_i > w$

Example (16)

<u>w</u>	<u>i</u>	0	1	2	3	4
0	0	0	0	0	0	0
1	0	0	0	0	0	0
2	0	3	3	3	3	3
3	0	3	4	4	4	4
4	0	3	4	5	5	5
5	0	3	7	7		

Items:

<u>1: (2,3)</u>
<u>2: (3,4)</u>
<u>3: (4,5)</u>
<u>4: (5,6)</u>

$$\underline{i=3}$$

$$\underline{b_i=5}$$

$$\underline{w_i=4}$$

$$\underline{w=1..4}$$

if $w_i \leq w$ // item i can be part of the solution

 if $b_i + B[i-1, w-w_i] > B[i-1, w]$

 $B[i, w] = b_i + B[i-1, w-w_i]$

 else

 $B[i, w] = B[i-1, w]$

 else $B[i, w] = B[i-1, w]$ // $w_i > w$

Example (17)

w	i	0	1	2	3	4
0	0	0	0	0	0	0
1	0	0	0	0	0	0
2	0	3	3	3	3	3
3	0	3	4	4	4	4
4	0	3	4	5	5	5
5	0	3	7	7	7	7

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

i=3

b_i=5

w_i=4

w=5

if w_i <= w // item i can be part of the solution

if b_i + B[i-1, w-w_i] > B[i-1, w]

B[i,w] = b_i + B[i-1, w- w_i]

else

B[i,w] = B[i-1,w]

else B[i,w] = B[i-1,w] // w_i > w

Comments

- ◆ This algorithm only finds the max possible value that can be carried in the knapsack
- ◆ To know the items that make this maximum value, an addition to this algorithm is necessary
- ◆ Please see LCS algorithm from the previous lecture for the example how to extract this data from the table we built



The 0/1 Knapsack Problem

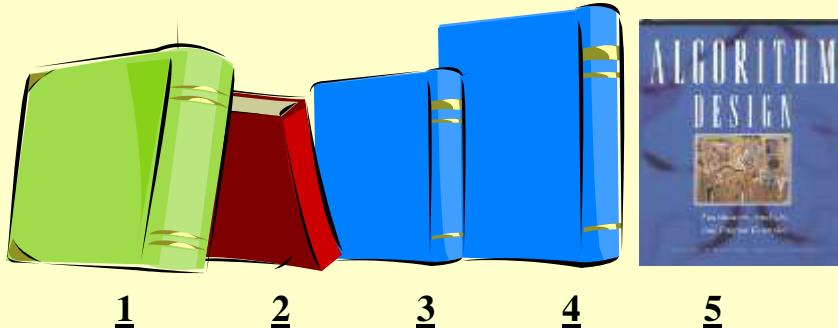
- ◆ Given: A set S of n items, with each item i having
 - » w_i - a positive weight
 - » b_i - a positive benefit
- ◆ Goal: Choose items with maximum total benefit but with weight at most W .
- ◆ If we are **not** allowed to take fractional amounts, then this is the **0/1 knapsack problem**.
 - » In this case, we let T denote the set of items we take
 - » Objective: maximize $\sum_{i \in T} b_i$
 - » Constraint: $\sum_{i \in T} w_i \leq W$

Example



- ◆ Given: A set S of n items, with each item i having
 - » b_i - a positive “benefit”
 - » w_i - a positive “weight”
- ◆ Goal: Choose items with maximum total benefit but with weight at most W .

Items:

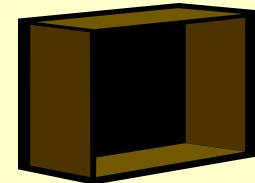


1 2 3 4 5

<u>Weight:</u>	<u>4 in</u>	<u>2 in</u>	<u>2 in</u>	<u>6 in</u>	<u>2 in</u>
<u>Benefit:</u>	<u>\$20</u>	<u>\$3</u>	<u>\$6</u>	<u>\$25</u>	<u>\$80</u>

Dynamic
Programming

“knapsack”



box of width 9 in

Solution:

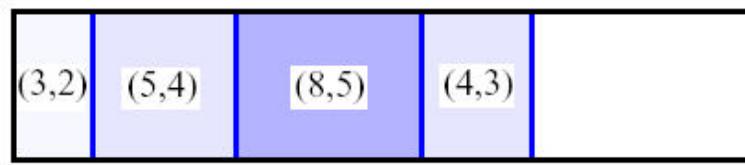
- item 5 (\$80, 2 in)
- item 3 (\$6, 2in)
- item 1 (\$20, 4in)

A 0/1 Knapsack Algorithm, First Attempt

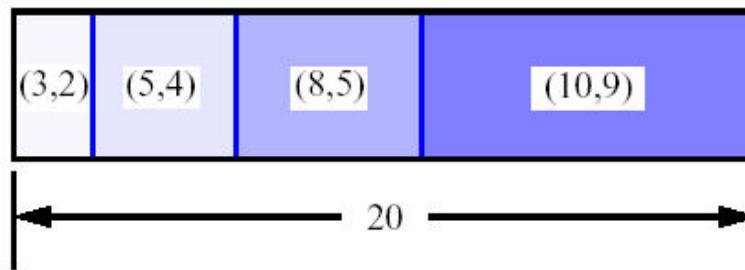


- ◆ S_k : Set of items numbered 1 to k.
- ◆ Define $B[k] =$ best selection from S_k .
- ◆ Problem: does not have subproblem optimality:
 - » Consider set $S=\{(3,2),(5,4),(8,5),(4,3),(10,9)\}$ of (benefit, weight) pairs and total weight $W = 20$

Best for S_4 :



Best for S_5 :



A 0/1 Knapsack Algorithm,

Second Attempt



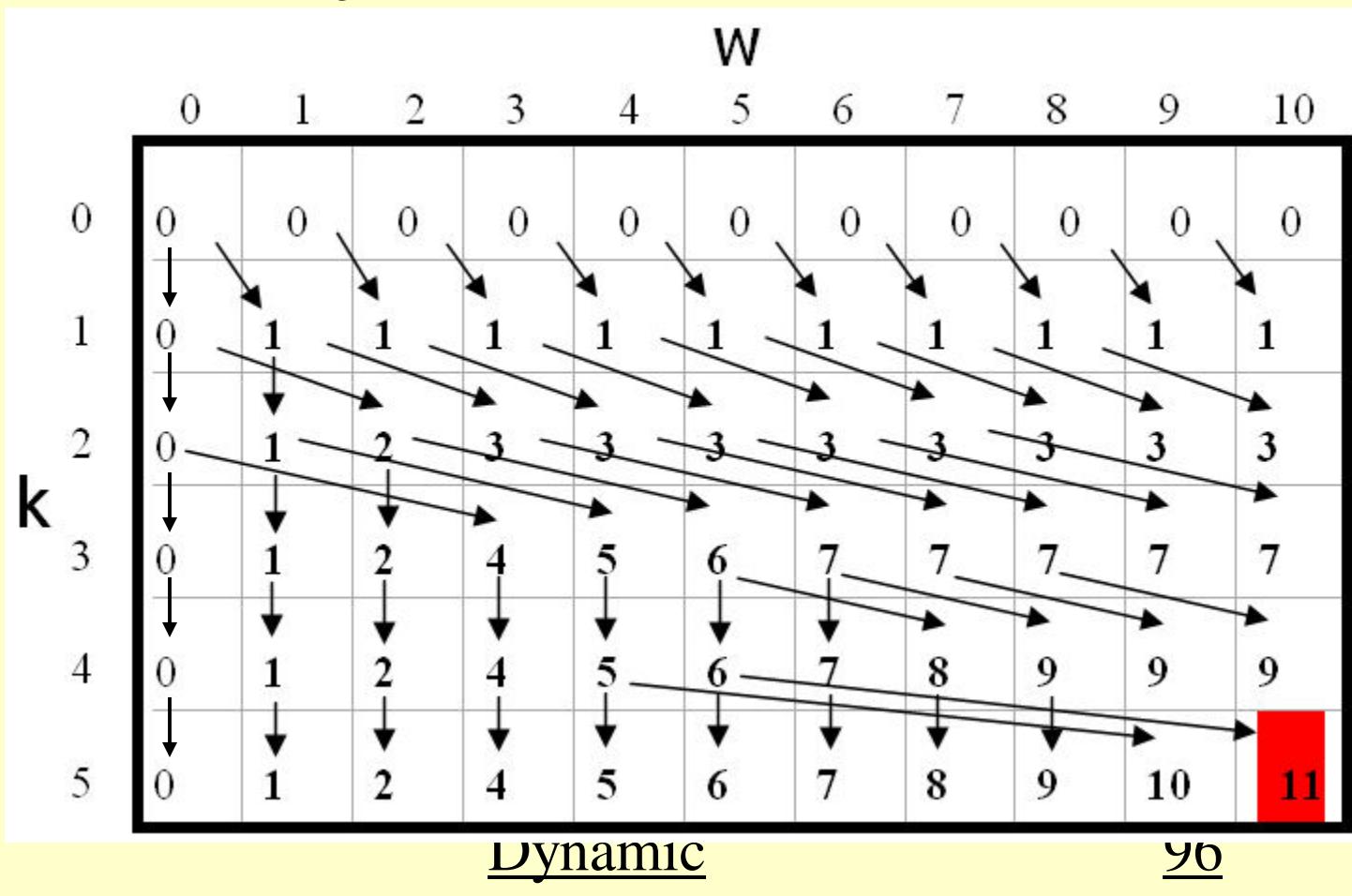
- ◆ S_k : Set of items numbered 1 to k.
- ◆ Define $B[k, w]$ to be the best selection from S_k with weight at most w
- ◆ Good news: this does have subproblem optimality.

$$B[k, w] = \begin{cases} B[k - 1, w] & \text{if } w_k > w \\ \max\{ B[k - 1, w], B[k - 1, w - w_k] + b_k \} & \text{else} \end{cases}$$

- ◆ I.e., the best subset of S_k with weight at most w is either
 - » the best subset of S_{k-1} with weight at most w or
 - » the best subset of S_{k-1} with weight at most $w - w_k$ plus item k

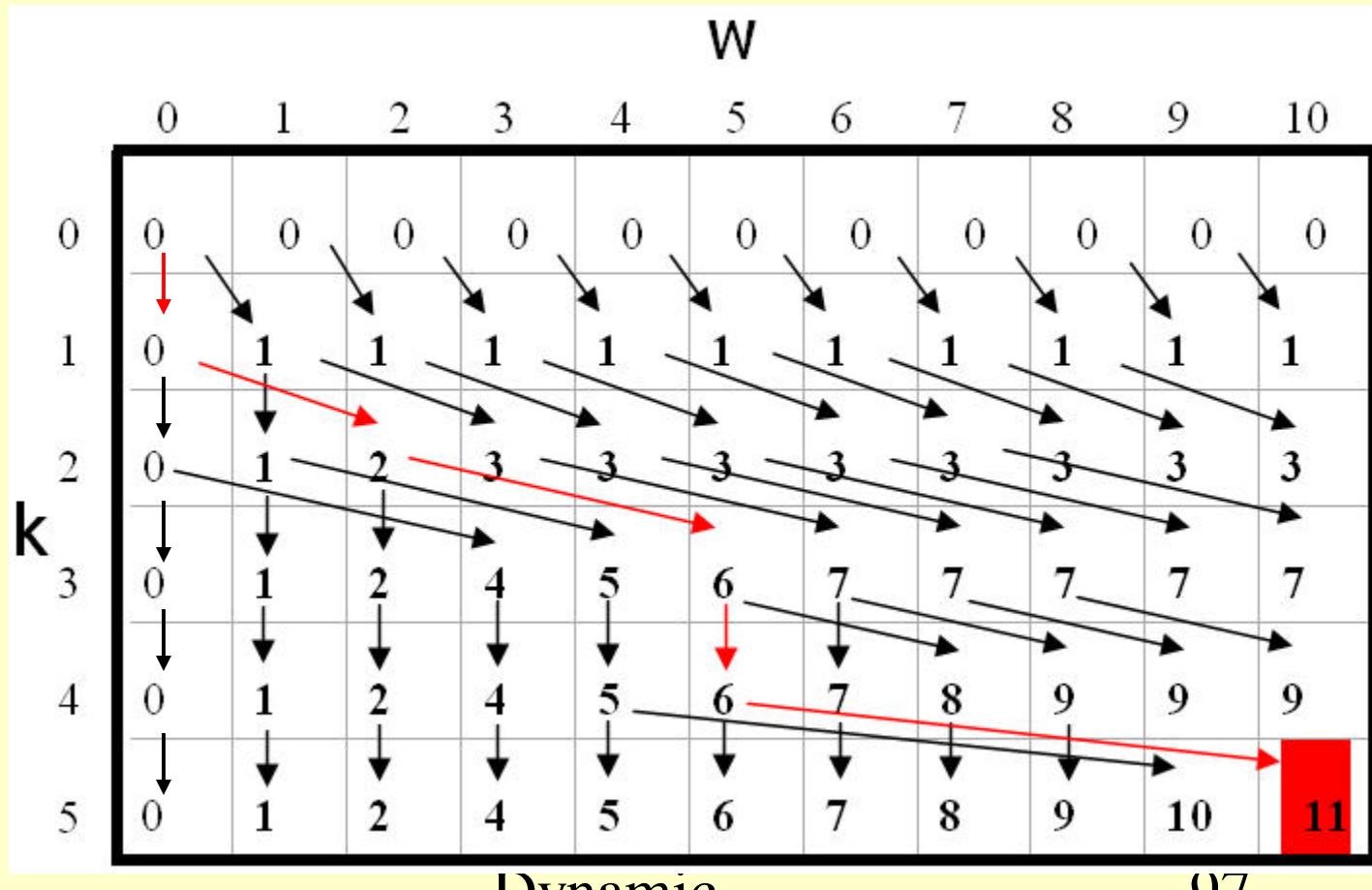
0/1 Knapsack Algorithm

- Consider set $S=\{(1,1),(2,2),(4,3),(2,2),(5,5)\}$ of (benefit, weight) pairs and total weight $W = 10$



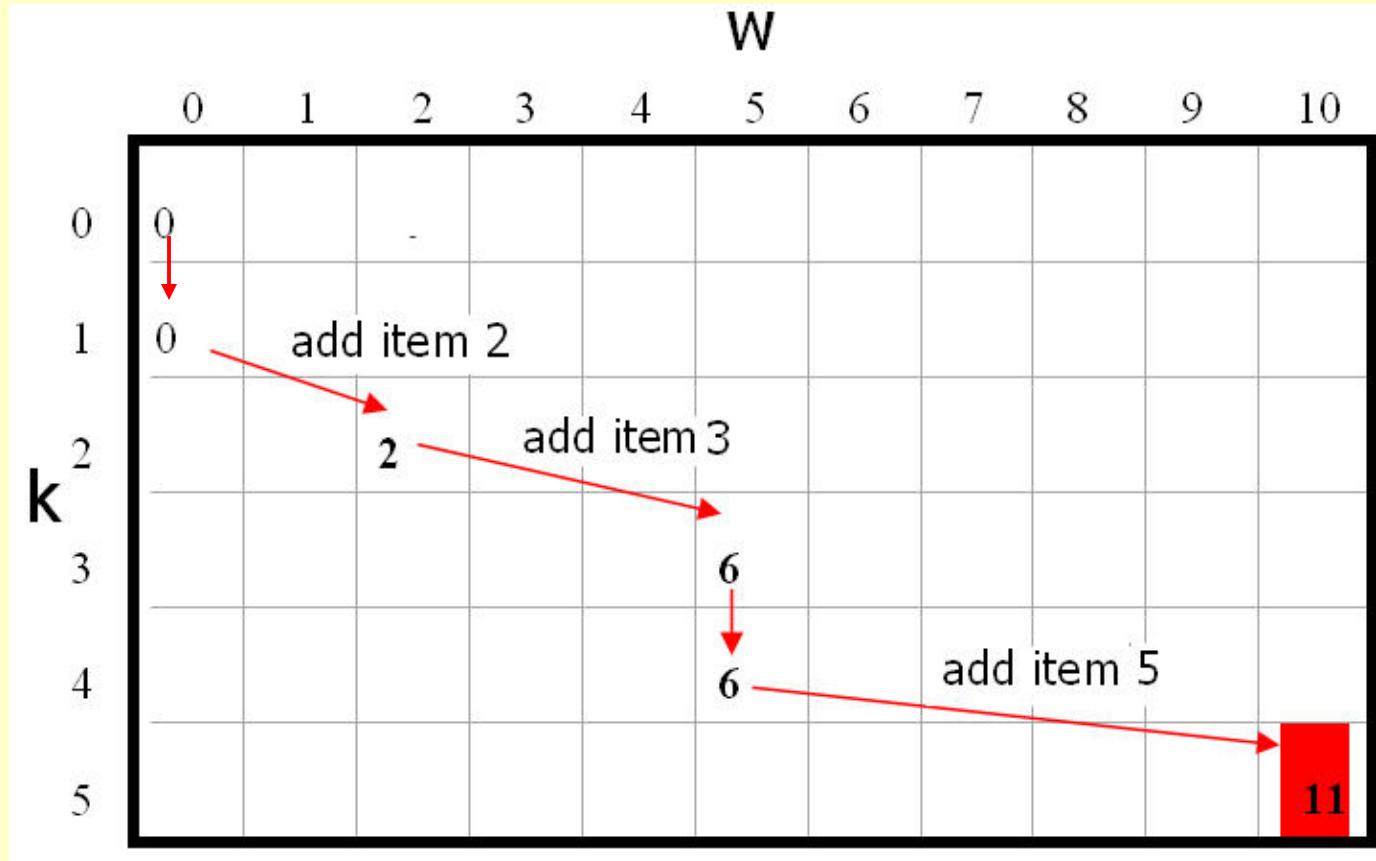
0/1 Knapsack Algorithm

- ◆ Trace back to find the items picked



0/1 Knapsack Algorithm

- ◆ Each diagonal arrow corresponds to adding one item into the bag
- ◆ Pick items 2,3,5
- ◆ $\{(2,2),(4,3),(5,5)\}$ are what you will take away



0/1 Knapsack Algorithm



$$B[k, w] = \begin{cases} B[k-1, w] & \text{if } w_k > w \\ \max\{B[k-1, w], B[k-1, w - w_k] + b_k\} & \text{else} \end{cases}$$

- Recall the definition of $B[k, w]$
- Since $B[k, w]$ is defined in terms of $B[k-1, *]$, we can use two arrays instead of a matrix
- Running time: $O(nW)$.
- Not a polynomial-time algorithm since W may be large
- This is a **pseudo-polynomial** time algorithm

Algorithm *01Knapsack(S, W)*:

Input: set S of n items with benefit b_i and weight w_i ; maximum weight W

Output: benefit of best subset of S with weight at most W

let A and B be arrays of length $W + 1$

for $w \leftarrow 0$ **to** W **do**

$B[w] \leftarrow 0$

for $k \leftarrow 1$ **to** n **do**

copy array B into array A

for $w \leftarrow w_k$ **to** W **do**

if $A[w - w_k] + b_k > A[w]$ **then**

$B[w] \leftarrow A[w - w_k] + b_k$

return $B[W]$

Dynamic Programming

- ◆ Dynamic programming is a useful technique of solving certain kind of problems
- ◆ When the solution can be recursively described in terms of partial solutions, we can store these partial solutions and re-use them as necessary
- ◆ Running time (Dynamic Programming algorithm vs. naïve algorithm):
 - » LCS: **O(m*n)** vs. **O(n * 2^m)**
 - » 0-1 Knapsack problem: **O(W*n)** vs. **O(2ⁿ)**

NP COMPLETENESS

Module 6

Tractable & Intractable Problems

- We will be looking at :
 - What is a P and NP problem
 - NP-Completeness
 - The question of whether P=NP

Polynomial Time (P)

- Most of the algorithms we have looked at so far have been **polynomial-time algorithms**
- On inputs of size n , their worst-case running time is $O(n^k)$ for some constant k
- The question is asked can all problems be solved in polynomial time?
- From what we've covered to date the answer is obviously no. There are many examples of problems that cannot be solved by any computer no matter how much time is involved
- There are also problems that can be solved, but not in time $O(n^k)$ for any constant k

Non-Polynomial Time (NP)

- Another class of problems are called NP problems
- These are problems that we have yet to find efficient algorithms in Polynomial Time for, but given a solution we can verify that solution in polynomial time
- Can these problems be solved in polynomial time?
- It has not been proved if these problems can be solved in polynomial time, or if they would require superpolynomial time
- This so-called $P \neq NP$ question is one which is widely researched and has yet to be settled

Tractable and Intractable

- Generally we think of problems that are solvable by polynomial time algorithms as being tractable, and problems that require superpolynomial time as being intractable.
- Sometimes the line between what is an ‘easy’ problem and what is a ‘hard’ problem is a fine one.
- For example, “Find the shortest path from vertex x to vertex y in a given weighted graph”. This can be solved efficiently without much difficulty.
- However, if we ask for the longest path (without cycles) from x to y , we have a problem for which no one knows a solution better than an exhaustive search

Deterministic v Non-Deterministic

- Let us now define some terms
 - **P:** The set of all problems that can be solved by deterministic algorithms in polynomial time
- By *deterministic* we mean that at any time during the operation of the algorithm, there is only one thing that it can do next
- A nondeterministic algorithm, when faced with a choice of several options, has the power to “guess“ the right one.
- Using this idea we can define NP problems as,
 - **NP:**The set of all problems that can be solved by nondeterministic algorithms in polynomial time.

NP-Completeness

- Poly time algorithm: input size n (in some encoding), worst case running time – $O(n^c)$ for some constant c .
- Three classes of problems
 - P: problems solvable in poly time.
 - NP: problems verifiable in poly time.
 - NPC: problems in NP and as hard as any problem in NP.

NP-Completeness (verifiable)

- Verifiable in poly time: given a certificate of a solution, could verify the certificate is correct in poly time.
- Examples (their definitions come later):
 - Hamiltonian-cycle, given a certificate of a sequence (v_1, v_2, \dots, v_n) , easily verified in poly time.
 - 3-CNF, given a certificate of an assignment 0s, 1s, easily verified in poly time.
 - (so try each instance, and verify it, but 2^n instances)
- Why not defined as “solvable in exponential time?” or “Non Poly time”?

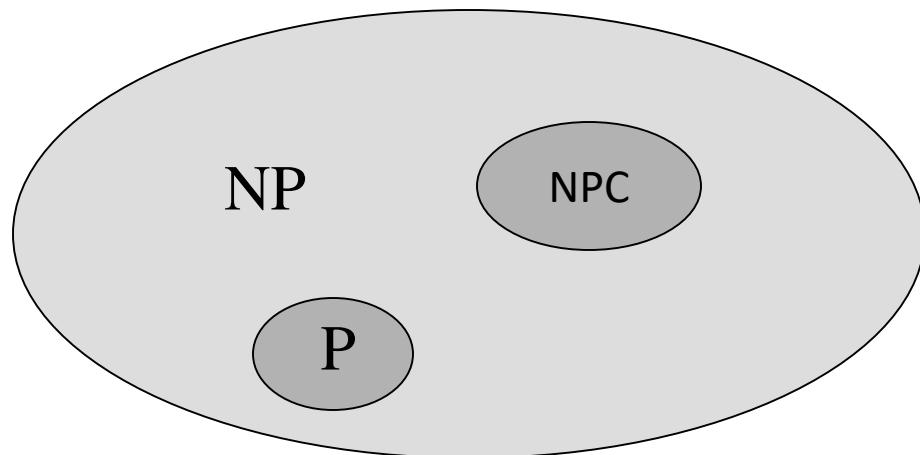
NP-Completeness (why NPC?)

- A problem $p \in \text{NP}$, and any other problem $p' \in \text{NP}$ can be translated as p in poly time.
- So if p can be solved in poly time, then all problems in NP can be solved in poly time.
- All current known NP hard problems have been proved to be NPC.

Relation among P, NP, NPC

- $P \subseteq NP$ (Sure)
- $NPC \subseteq NP$ (sure)
- $P = NP$ (or $P \subset NP$, or $P \neq NP$) ???
- $NPC = NP$ (or $NPC \subset NP$, or $NPC \neq NP$) ???
- $P \neq NP$: one of the deepest, most perplexing open research problems in (theoretical) computer science since 1971.

View of Theoretical Computer Scientists on P, NP, NPC



$$P \subset NP, NPC \subset NP, P \cap NPC = \emptyset$$

Why discussion on NPC

- If a problem is proved to be NPC, a good evidence for its intractability (hardness).
- Not waste time on trying to find efficient algorithm for it
- Instead, focus on design approximate algorithm or a solution for a special case of the problem
- Some problems looks very easy on the surface, but in fact, is hard (NPC).

Decision VS. Optimization Problems

- Decision problem: solving the problem by giving an answer “YES” or “NO”
- Optimization problem: solving the problem by finding the optimal solution.
- Examples:
 - SHORTEST-PATH (optimization)
 - Given G, u, v , find a path from u to v with fewest edges.
 - PATH (decision)
 - Given G, u, v , and k , whether exist a path from u to v consisting of at most k edges.

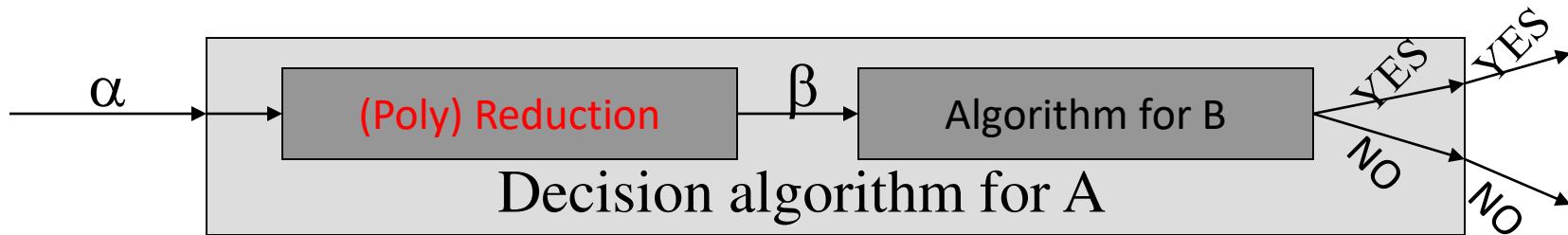
Decision VS. Optimization Problems (Cont.)

- Decision is easier (i.e., no harder) than optimization
- If there is an algorithm for an optimization problem, the algorithm can be used to solve the corresponding decision problem.
 - Example: SHORTEST-PATH for PATH
- If optimization is easy, its corresponding decision is also easy. Or in another way, if provide evidence that decision problem is hard, then the corresponding optimization problem is also hard.
- NPC is confined to decision problem. (also applicable to optimization problem.)
 - Another reason is that: easy to define reduction between decision problems.

(Poly) reduction between decision problems

- Problem (class) and problem instance
- Instance α of decision problem A and instance β of decision problem B
- A reduction from A to B is a transformation with the following properties:
 - The transformation takes poly time
 - The answer is the same (the answer for α is YES if and only if the answer for β is YES).

Implication of (poly) reduction



1. If decision algorithm for B is poly, so does A.
A is no harder than B (or B is no easier than A)
2. If A is hard (e.g., NPC), so does B.
3. How to prove a problem B to be NPC ??

(at first, prove B is in NP, which is generally easy.)

- 3.1 find a already proved NPC problem A
- 3.2 establish an (poly) reduction from A to B

Question: What is and how to prove the first NPC problem?

Circuit-satisfiability problem.

Discussion on Poly time problems

- $\Theta(n^{100})$ vs. $\Theta(2^n)$
 - Reasonable to regard a problem of $\Theta(n^{100})$ as intractable, however, very few practical problem with $\Theta(n^{100})$.
 - Most poly time algorithms require much less.
 - Once a poly time algorithm is found, more efficient algorithm may follow soon.
- Poly time keeps same in many different computation models, e.g., poly class of serial random-access machine \equiv poly class of abstract Turing machine \equiv poly class of parallel computer (#processors grows polynomially with input size)
- Poly time problems have nice closure properties under addition, multiplication and composition.

Encoding impact on complexity

- The problem instance must be represented in a way the program (or machine) can understand.
- General encoding is “binary representation”.
- Different encoding will result in different complexities.
- Example: an algorithm, only input is integer k , running time is $\Theta(k)$.
 - If k is represented in *unary*: a string of k 1s, the running time is $\Theta(k) = \Theta(n)$ on length- n input, **poly on n** .
 - If k is represented in *binary*: the input length $n = \lfloor \log k \rfloor + 1$, the running time is $\Theta(k) = \Theta(2^n)$, **exponential on n** .
- Ruling out *unary*, other encoding methods are same.

Examples of encoding and complexity

- Given integer n , check whether n is a composite.
- Dynamic programming for subset-sum.

Class P Problems

- Let n = the length of binary encoding of a problem (i.e., input size), $T(n)$ is the time to solve it.
- A problem is *poly-time solvable* if $T(n) = O(n^k)$ for some constant k .
- Complexity class **P**=set of problems that are *poly-time solvable*.

Poly Time Verification

- PATH problem: Given $\langle G, u, v, k \rangle$, whether exists a path from u to v with at most k edges?
- Moreover, also given a path p from u to v , verify whether the length of p is at most k ?
- Easy or not?

Of course, very easy.

Poly Time Verification, encoding, and language

- Hamiltonian cycles
 - A simple path containing every vertex.
 - HAM-CYCLE={<G>: G is a Hamiltonian graph, i.e. containing Hamiltonian cycle}.
 - Suppose n is the length of **encoding** of G.
 - HAM-CYCLE can be considered as a **Language** after encoding, i.e. a subset of Σ^* where $\Sigma=\{0,1\}^*$.
- The naïve algorithm for determining HAM-CYCLE runs in $\Omega(m!)=\Omega(2^m)$ time, where m is the number of vertices, $m \approx n^{1/2}$.
- However, given an ordered sequence of m vertices (called “certificate”), let you verify whether the sequence is a Hamiltonian cycle. Very easy. In $O(n^2)$ time.

Class NP problems

- For a problem p , given its certificate, the certificate can be verified in poly time.
- Call this kind of problem an NP one.
- Complement set/class: Co-NP.
 - Given a set S (as a universal) and given a subset A
 - The complement is that $S-A$.
 - When NP problems are represented as languages (i.e. a set), we can discuss their complement set, i.e., Co-NP.

Relation among P, NP and co-NP = {L: $\overline{L} \in \text{NP}$ where $\overline{L} = \Sigma^* - L$ }

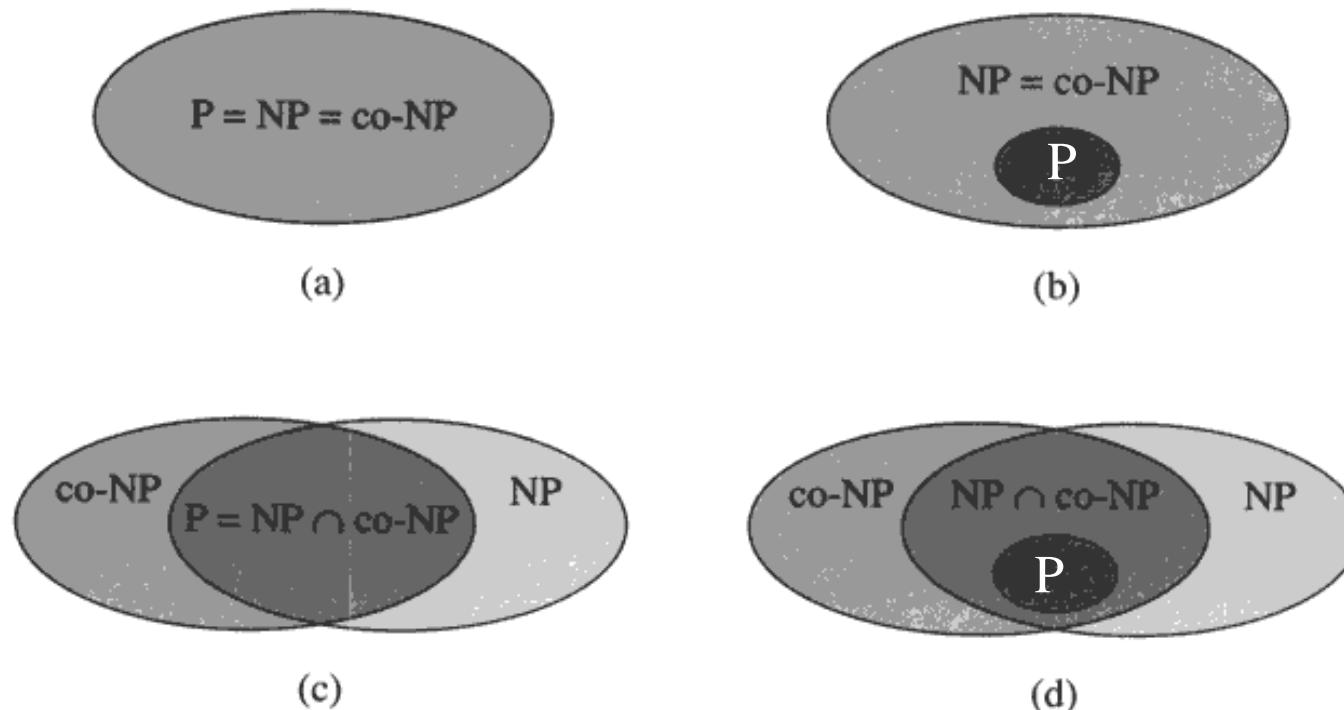


Figure 34.3 Four possibilities for relationships among complexity classes. In each diagram, one region enclosing another indicates a proper-subset relation. (a) $P = NP = \text{co-NP}$. Most researchers regard this possibility as the most unlikely. (b) If NP is closed under complement, then $NP = \text{co-NP}$, but it need not be the case that $P = NP$. (c) $P = NP \cap \text{co-NP}$, but NP is not closed under complement. (d) $NP \neq \text{co-NP}$ and $P \neq NP \cap \text{co-NP}$. Most researchers regard this possibility as the most likely.

NP-completeness and Reducibility

- A (class of) problem P_1 is **poly-time reducible** to P_2 , written as $P_1 \leq_p P_2$ if there exists a poly-time function $f: P_1 \rightarrow P_2$ such that for any instance of $p_1 \in P_1$, p_1 has “YES” answer if and only if answer to $f(p_1)$ ($\in P_2$) is also “YES”.
- *Theorem 34.3:* (page 985)
 - For two problems P_1, P_2 , if $P_1 \leq_p P_2$ then $P_2 \in P$ implies $P_1 \in P$.

NP-completeness and Reducibility (cont.)

- A problem p is **NP-complete** if
 1. $p \in \text{NP}$ and
 2. $p' \leq_p p$ for every $p' \in \text{NP}$.(if p satisfies 2, then p is said **NP-hard**.)

Theorem 34.4 (page 986)

if any NP-complete problem is poly-time solvable, then $P=NP$. Or say: if any problem in NP is not poly-time solvable, then no NP-complete problem is poly-time solvable.

NP-completeness proof basis

- *Lemma 34.8 (page 995)*
 - If X is a problem (class) such that $P' \leq_p X$ for some $P' \in \text{NPC}$, then X is NP-hard. Moreover, if $X \in \text{NP}$, then $X \in \text{NPC}$.
- Steps to prove X is NP-complete
 - Prove $X \in \text{NP}$.
 - Given a certificate, the certificate can be verified in poly time.
 - Prove X is NP-hard.
 - Select a known NP-complete P' .
 - Describe a transformation function f that maps every instance x of P' into an instance $f(x)$ of X .
 - Prove f satisfies that the answer to $x \in P'$ is YES if and only if the answer to $f(x) \in X$ is YES for all instance $x \in P'$.
 - Prove that the algorithm computing f runs in poly-time.

NPC proof –Formula Satisfiability (SAT)

- SAT definition
 - n boolean variables: x_1, x_2, \dots, x_n .
 - M boolean connectives: any boolean function with one or two inputs and one output, such as $\wedge, \vee, \neg, \rightarrow, \leftrightarrow, \dots$ and
 - Parentheses.
- A SAT ϕ is satisfiable if there exists an true assignment which causes ϕ to evaluate to 1.
- $SAT = \{<\phi> : \phi \text{ is a satisfiable boolean formula}\}$.
- The historical honor of the first NP-complete problem ever shown.

SAT is NP-complete

- *Theorem 34.9:* (page 997)
 - SAT is NP-complete.
- Proof:
 - SAT belongs to NP.
 - Given a satisfying assignment, the verifying algorithm replaces each variable with its value and evaluates the formula, **in poly time**.
 - SAT is NP-hard (show $\text{CIRCUIT-SAT} \leq_p \text{SAT}$).

SAT is NP-complete (cont.)

- CIRCUIT-SAT \leq_p SAT, i.e., any instance of circuit satisfiability can be reduced in poly time to an instance of formula satisfiability.
- Intuitive induction:
 - Look at the gate that produces the circuit output.
 - Inductively express each of gate's inputs as formulas.
 - Formula for the circuit is then obtained by writing an expression that applies the gate's function to its input formulas.
- Unfortunately, this is not a poly reduction
 - Shared formula (the gate whose output is fed to 2 or more inputs of other gates) cause the size of generated formula to grow exponentially.

SAT is NP-complete (cont.)

- Correct reduction:
 - For every wire x_i of C, give a variable x_i in the formula.
 - Every gate can be expressed as $x_o \leftrightarrow (x_{i_1} \theta x_{i_2} \theta \dots \theta x_{i_l})$
 - The final formula ϕ is the AND of the circuit output variable and conjunction of all clauses describing the operation of each gate. ([example Figure 34.10](#))
- Correctness of the reduction
 - Clearly the reduction can be done in poly time.
 - C is satisfiable if and only if ϕ is satisfiable.
 - If C is satisfiable, then there is a satisfying assignment. This means that each wire of C has a well-defined value and the output of C is 1. Thus the assignment of wire values to variables in ϕ makes each clause in ϕ evaluate to 1. So ϕ is 1.
 - The reverse proof can be done in the same way.

Example of reduction of CIRCUIT-SAT to SAT

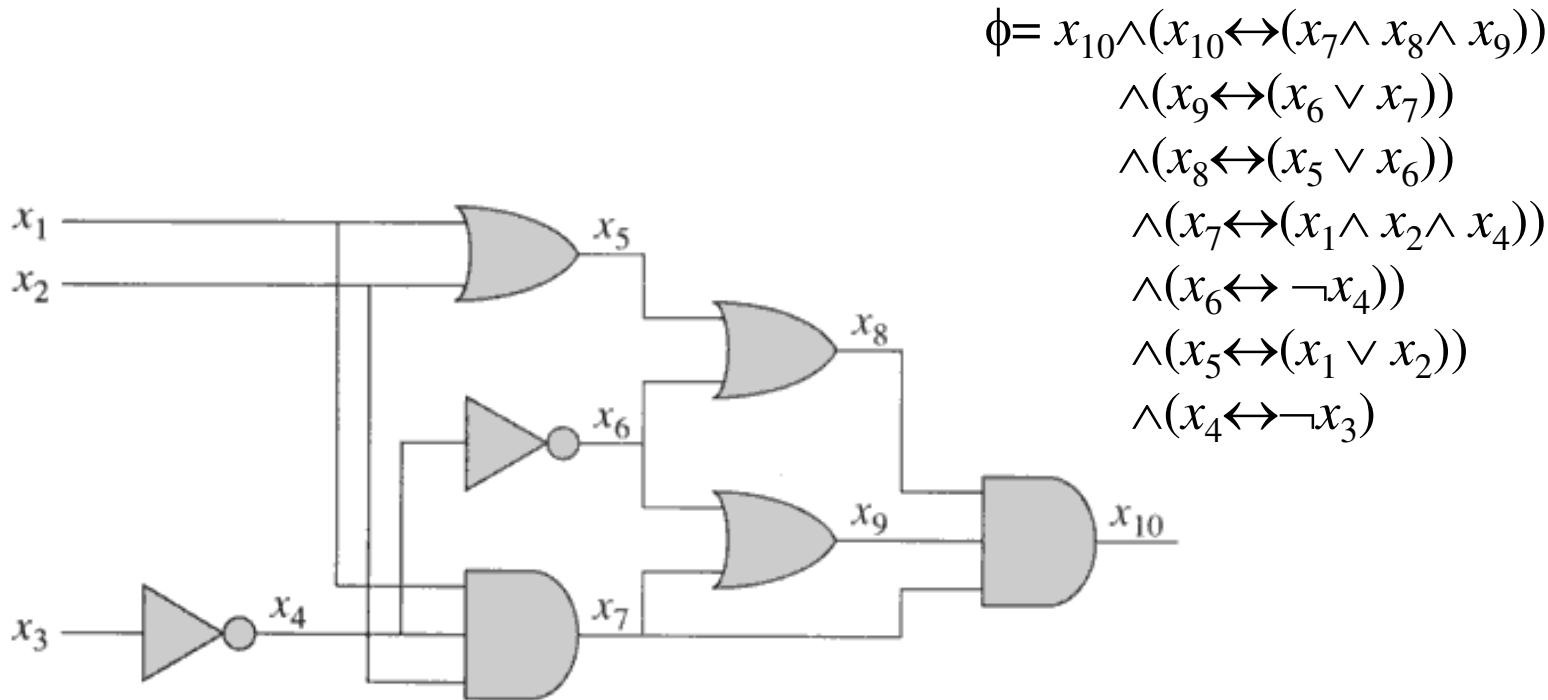


Figure 34.10 Reducing circuit satisfiability to formula satisfiability. The formula produced by the reduction algorithm has a variable for each wire in the circuit.

INCORRECT REDUCTION: $\phi = x_{10} = x_7 \wedge x_8 \wedge x_9 = (x_1 \wedge x_2 \wedge x_4) \wedge (x_5 \vee x_6) \wedge (x_6 \vee x_7) = (x_1 \wedge x_2 \wedge x_4) \wedge ((x_1 \vee x_2) \vee \neg x_4) \wedge (\neg x_4 \vee (x_1 \wedge x_2 \wedge x_4)) = \dots$

NPC Proof – 3-CNF Satisfiability

- 3-CNF definition
 - A *literal* in a boolean formula is an occurrence of a variable or its negation (x_i or $\neg x_i$).
 - CNF (Conjunctive Normal Form) is a boolean formula expressed as AND of clauses, each of which is the OR of one or more literals.
 - 3-CNF is a CNF in which each clause has exactly 3 distinct literals.
- 3-CNF-SAT: whether a given 3-CNF is satisfiable?

$$\phi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_1 \vee x_2 \vee x_3)$$

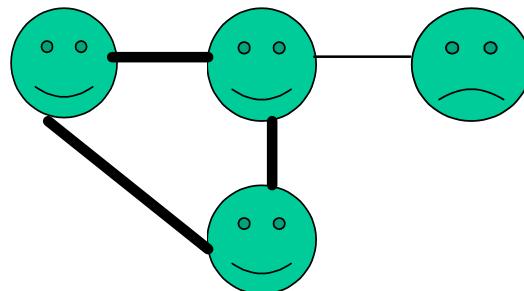
$\phi = (c1) \wedge (c2) \wedge (c3)$, where $c1, c2, c3$ are clauses

CLIQUE is NP-Complete

In an undirected graph, a **subgraph wherein each node is connected to each other node in the subgraph is a clique.**

Give an integer k and a graph, the problem is to determine there is a **clique of size k** anywhere in the graph.

A clique of size k is called a **k -clique**.



A graph with a 3-clique

First, prove that CLIQUE is in NP

It is simple to find a verifier for CLIQUE, using the nodes in the clique as the certificate:

- 1. Verify that the clique is a subgraph of the correct size
- 2. Check that the graph contains edges connecting the nodes in the clique.
- If 1 and 2 are both true, accept, otherwise reject.

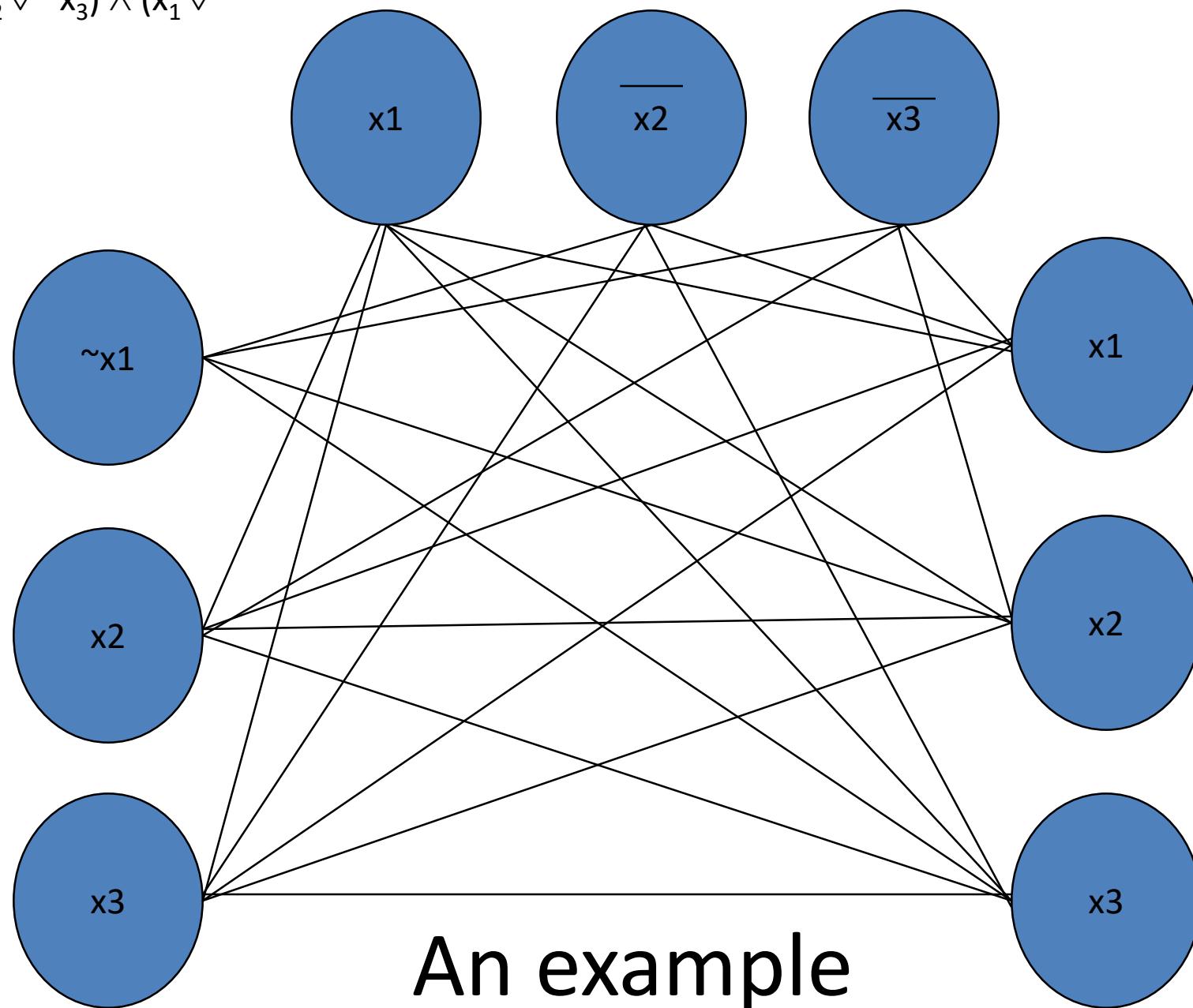
Now, a polynomial time reduction to 3SAT

- We will construct a graph G based on a formula ϕ .
- Let k be the number of clauses in ϕ .
- G will have a clique of size k iff ϕ is satisfiable.

Construction of G

- Each clause in ϕ will be represented as a set of three nodes, one for each literal.
- The nodes will all be connected, except that no nodes from the same clause will be connected and no two nodes with contradictory labels will be connected (i.e. x_1 and $\sim x_1$ shall not share an edge)

$$\phi = (\neg x_1 \vee x_2 \vee x_3) \wedge \\ (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_1 \vee \\ x_2 \vee x_3)$$



Proof

Theorem: ϕ is satisfiable iff G has a k -clique:

Assume ϕ is satisfiable. In **each clause, at least one literal is true.**

Select one node which has a true literal as its label from each clause.

The **nodes selected form a k -clique since the nodes are selected one from each clause.**

The labels are connected since they cannot be contradictory. (cont)

Finishing the proof(the converse)

Assume G has a k-clique.

Assign each of the literals in the labels of the clique to be true(1).

Because nodes from the same clause are not connected, a k-clique must visit k clauses.

No contradictions will be encountered since contradictory labels are not connected.

Thus each clause contains at least one true literal, which means that ϕ is satisfied.

Algorithm Design Analysis

Table of Contents

- Class NP
- Class NPC
- Approximation Algorithms

- Class of problems for which “yes” can be verified in polynomial time.
- A ***Verification*** algorithm takes as an input, a problem instance, and a certificate and decides whether it is a yes-instance.
- $A(x,y) = 1$; iff, y is a valid certificate.

input certificate
instance

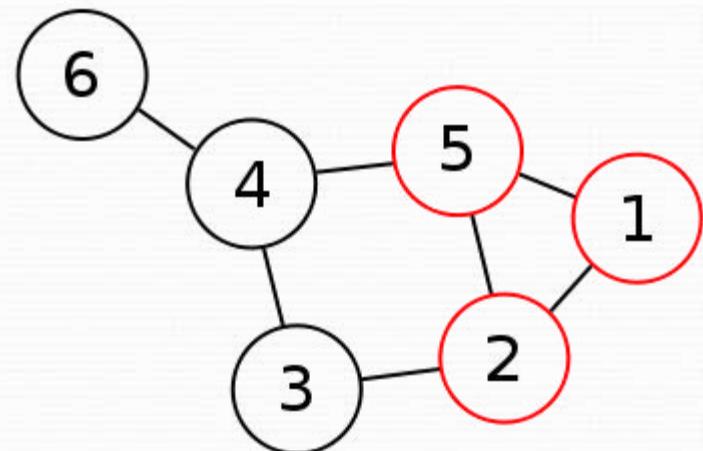
- Before we elaborate on this

Optimization problems and Decision problems

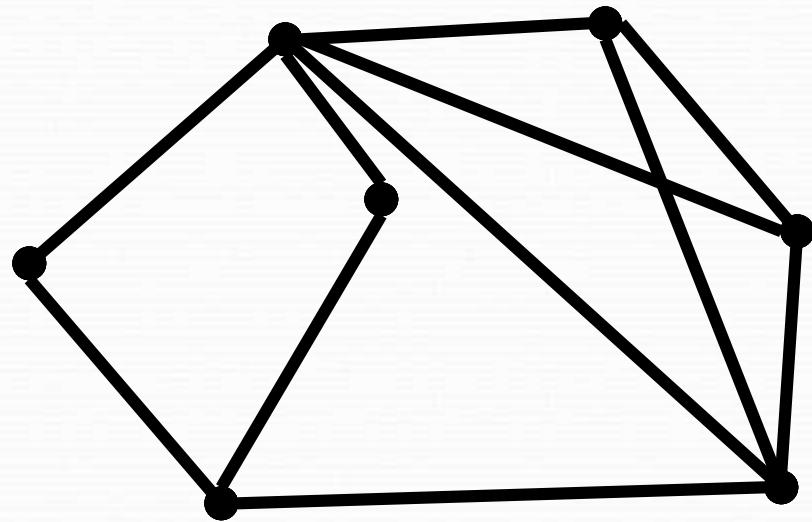
- Clique
- Vertex cover
- Back to verification

Clique problem

- Given an undirected graph $G = (V, E)$, a clique is a subset V' of V such that every pair of vertices is connected by an edge in E .
- In the graph to the right, vertices 1, 2, 5 form a clique since each has an edge to all the others.

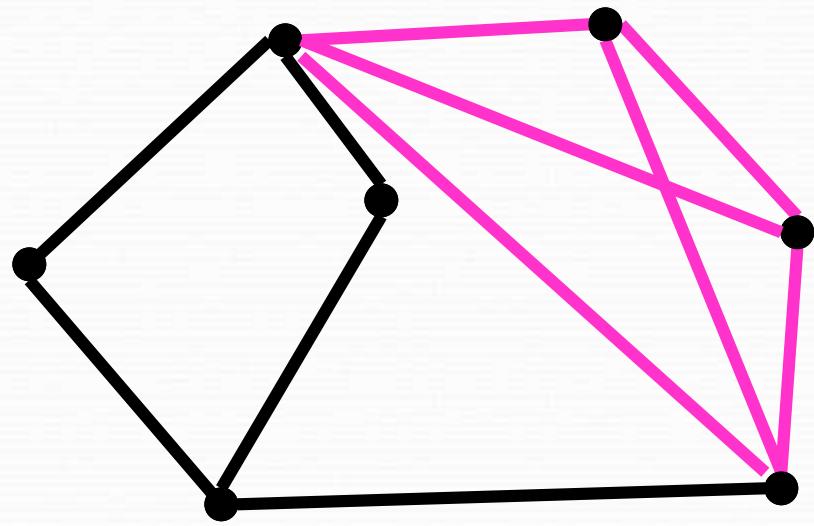


G

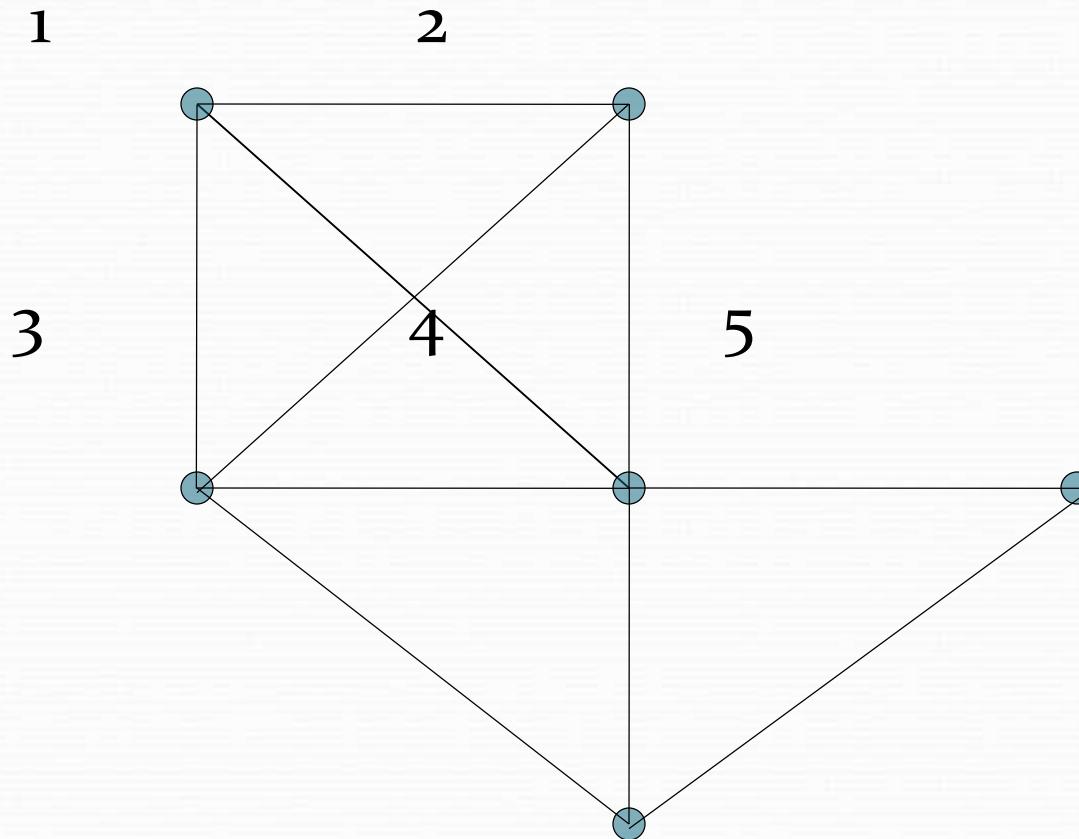


Green ovals represent CLIQUE
for this graph

G



Clique



Clique

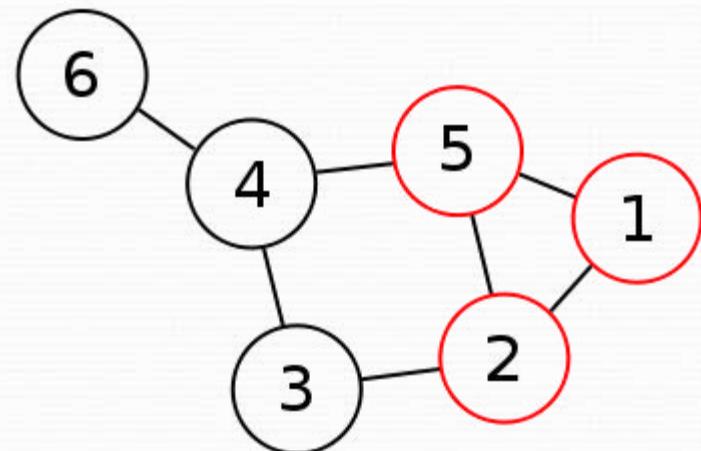
- Optimization version: Given an undirected graph G , find a largest size Clique in it.
- Decision version: Given an undirected graph G , does there exist a clique of size at most k ?

“Yes” verification

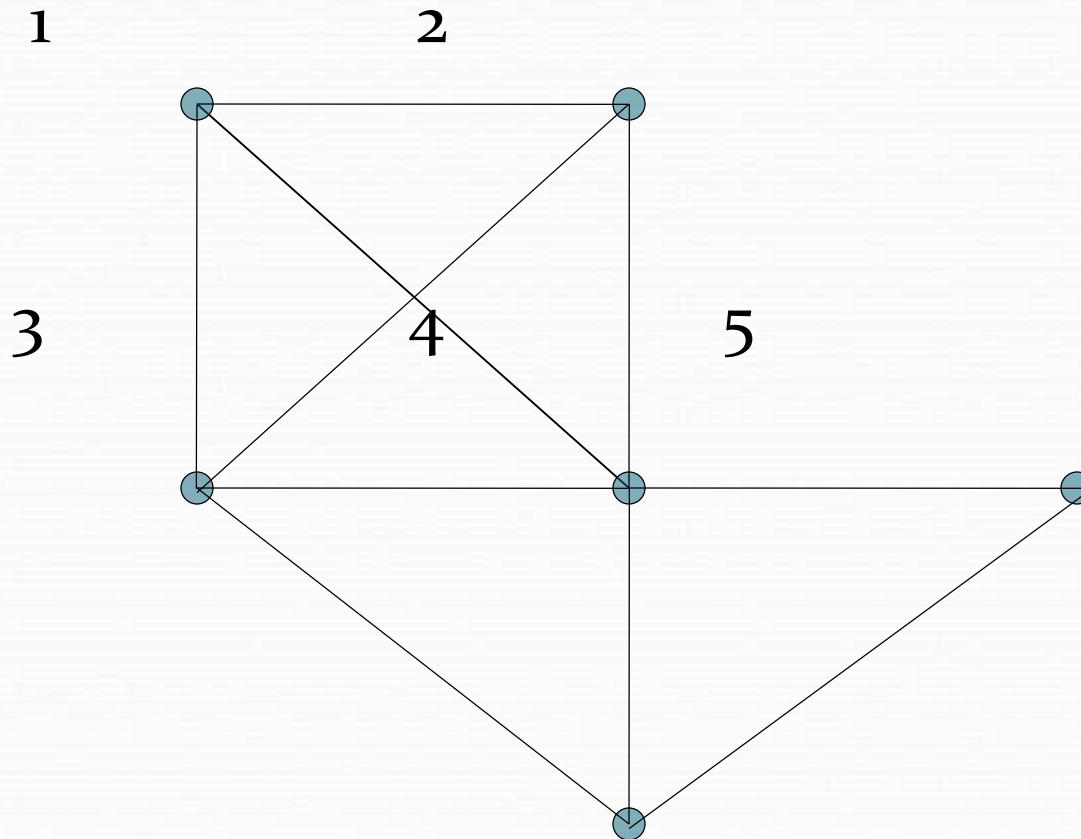
- Given a set of vertices V' in G , does V' form a Clique in G ?

Vertex Cover problem

- Given an undirected graph $G = (V, E)$, a vertex cover is a subset V' of V such that every edge in E has at least one end point in V' .
- In the graph to the right, vertices 1,2,5 form a clique since each has an edge to all the others.



Vertex Cover,



Vertex Cover

- Optimization version: Given an undirected graph G , find a smallest size vertex cover in it.
- Decision version: Given an undirected graph G , does there exist a vertex cover of size at least k ?

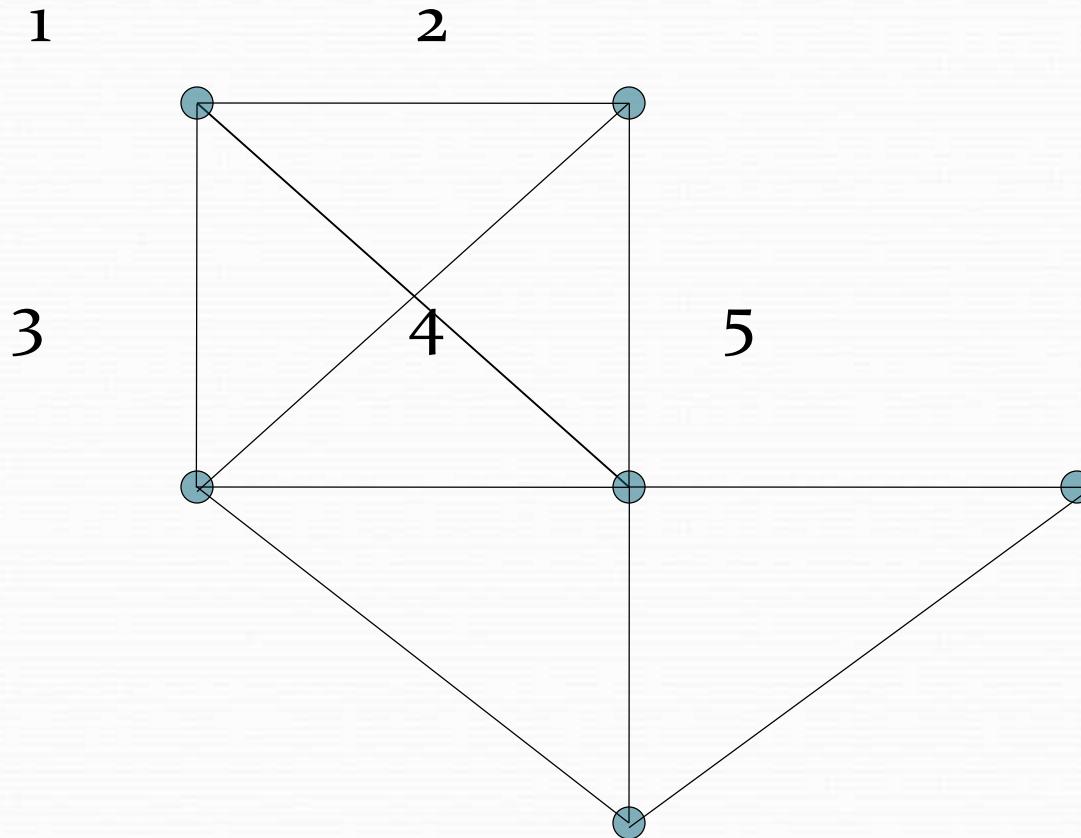
“Yes” verification

- Given a set of vertices V' in G , does V' form a vertex cover in G ?

Hamiltonian Cycle Problem

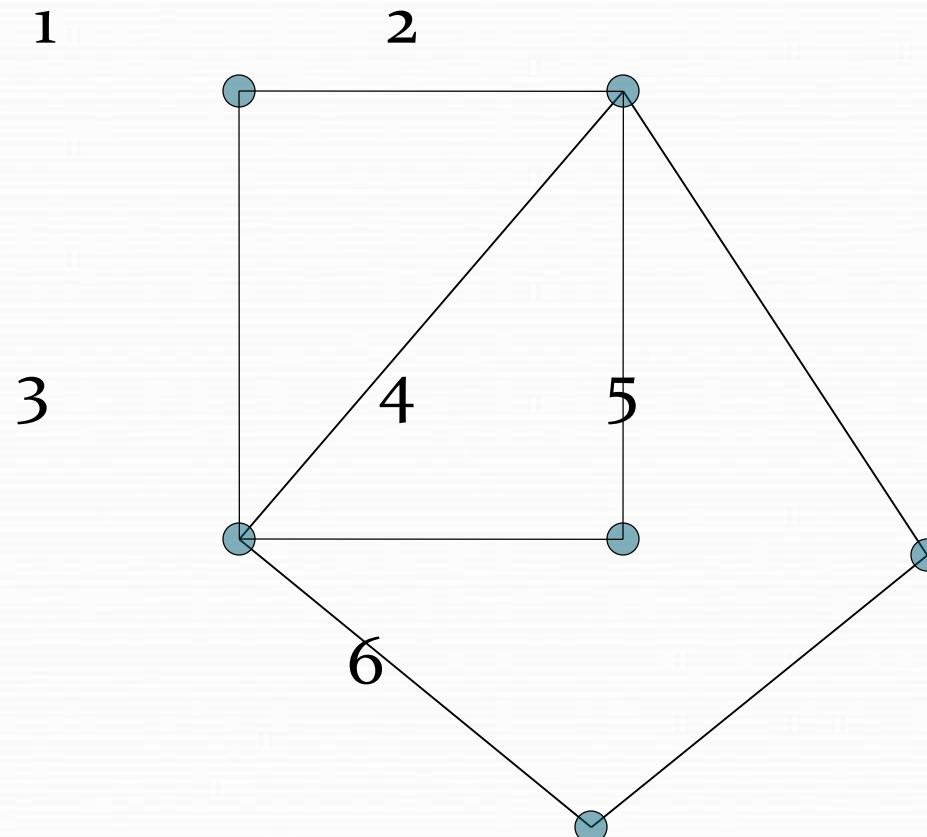
- A **Hamiltonian cycle** of an undirected graph, G , is a simple cycle that contains every vertex.
- For instance, consider the graph G

Forming an HC,



- However, for the graph G' , there does not exist any Hamiltonian cycle.

- 1) with vertices 1,2,3,4 – not an HC
- 2) with vertices 1,2,5,6,3 – again not an HC



Hamiltonian Cycle Problem

- The Hamiltonian-cycle problem: given a graph G, find a Hamiltonian cycle in it?

Ham _cycle = { $\langle G \rangle$: G is hamiltonian}.

- Decision version: Given an undirected graph G, does it contain a Hamiltonian Cycle?

“Yes” verification

- Given a sequence of vertices V' in G , does V' form a Hamiltonian cycle in G ?

Class NP

- Defined for decision problems.
- The class of decision problems for which there is a polynomially bounded *nondeterministic* algorithm.

“No” verification is not easy

- Examples:
 - Clique
 - Vertex Cover
 - Hamiltonian

Class Co- NP

- Class of problems for which “No” can be verified in polynomial time.
- In other words, it is the set of problems whose complement is in NP.

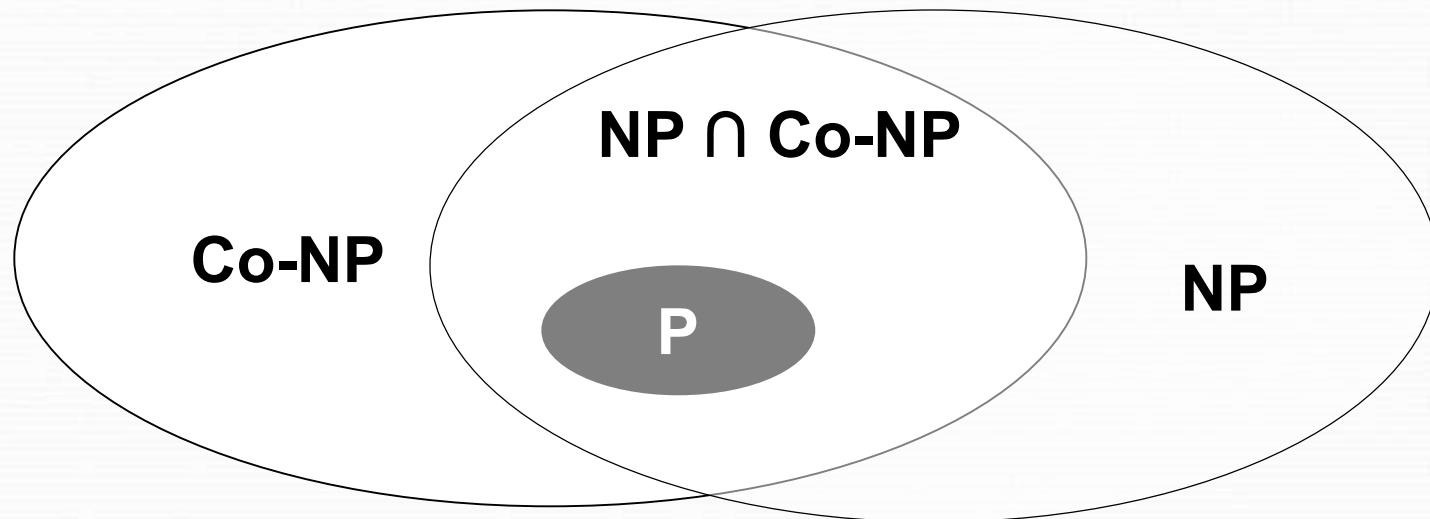
Shortest Path

- “yes” and “no” both the certificates exist and can be verified in polynomial time. Thus SP is in the intersection.
- My **verification algorithm** will compute the shortest path in polynomial time and without explicitly using the certificate will answer “yes” or “no” as the case may be.

Class P is in the intersection

- Similarly, for every problem in P, both the certificates exist and can be verified in polynomial time.
- The **verification algorithm** being the polynomial time algorithm to solve the problem itself.
- Hence P is in the intersection.

Relation between class P, NP and Co-NP



NP-COMPLETENESS

Instructor: Ms. Neelima Gupta

Class NPC

- NP-completeness does not apply directly to optimization problems, but to decision problems.
- As we have seen we can obtain a decision version of an optimization problem, which will be “easier” or at least, “no harder” than the optimization problem.
- Optimization problem is the harder of the two.

Defining NPC

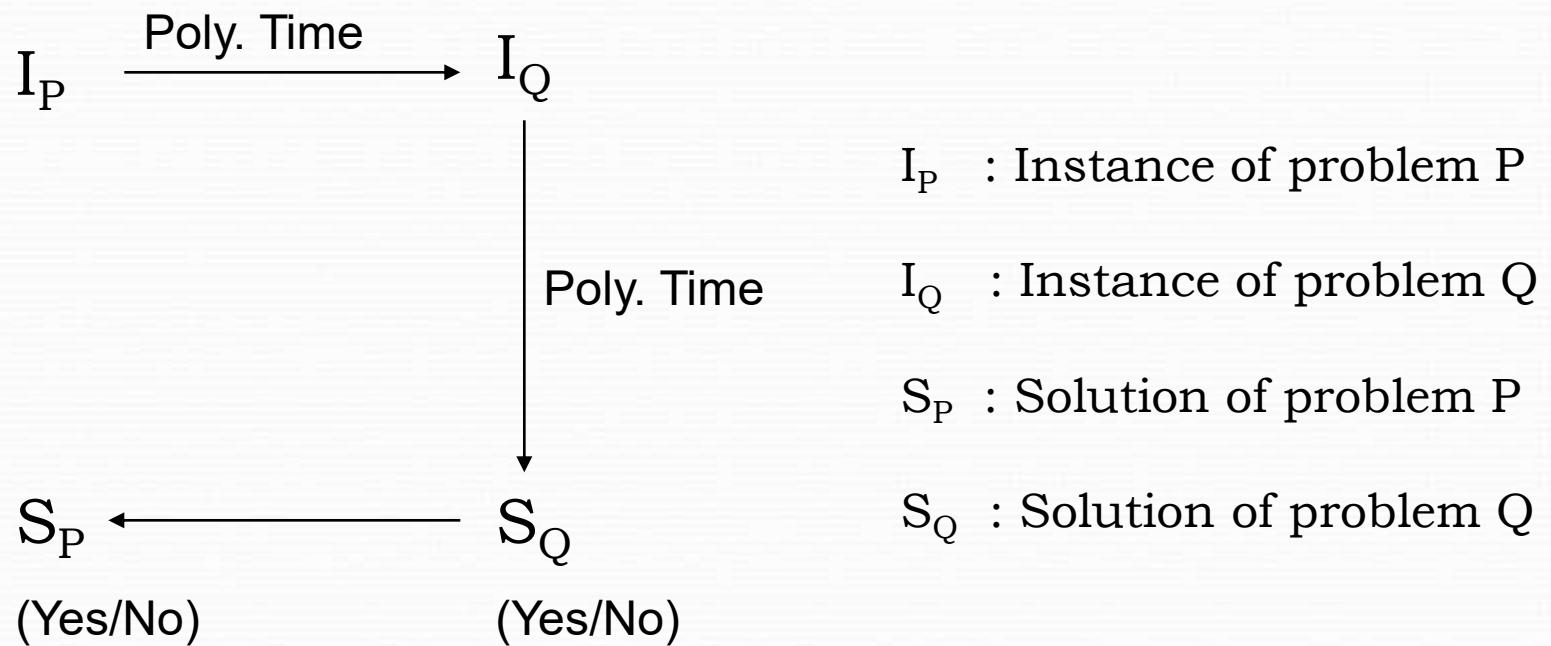
- Let Q be any problem at hand, it belongs to class NPC, if,
 - ✓ $Q \in NP$ &
 - ✓ Q is NP hard

NP hard

\mathcal{Q} belongs to class NP-hard

1. If $\forall P \in NP$, and
 2. $P \leq_p Q$

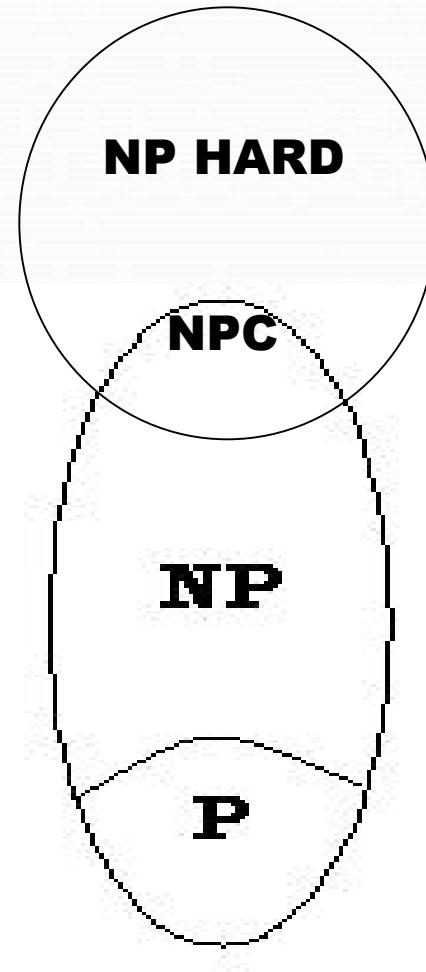
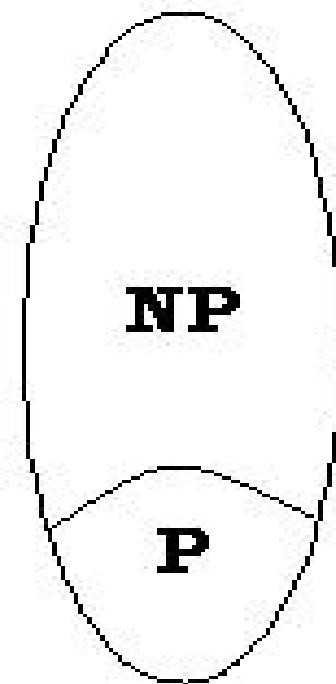
In step 2. we are mapping an instance of problem P to an instance of problem Q.



Transformation Characteristics :

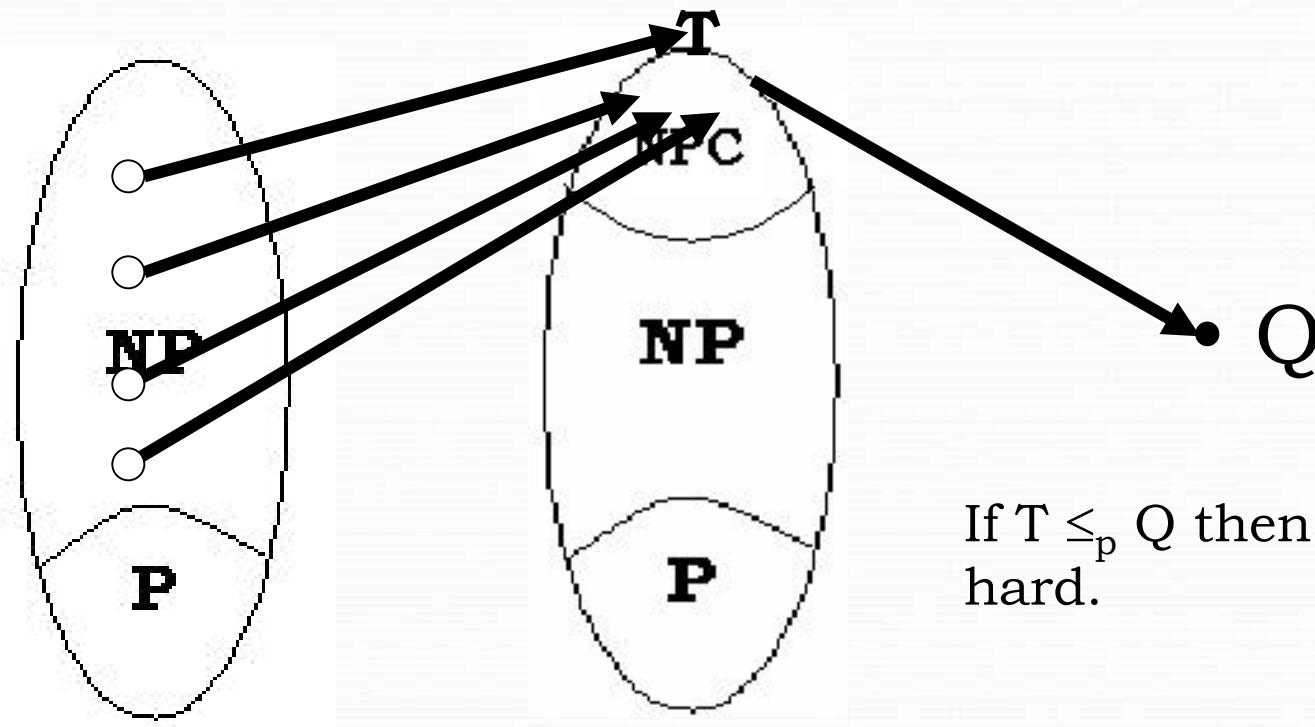
- If $A(Q)$ is yes then $A(P)$ is yes
- Vice versa
- It should be done in polynomial time

Diagrammatically



If all problems $R \in NP$ are reducible to T , then T is NP-Hard

And $T \in NP$ then T is NPC.



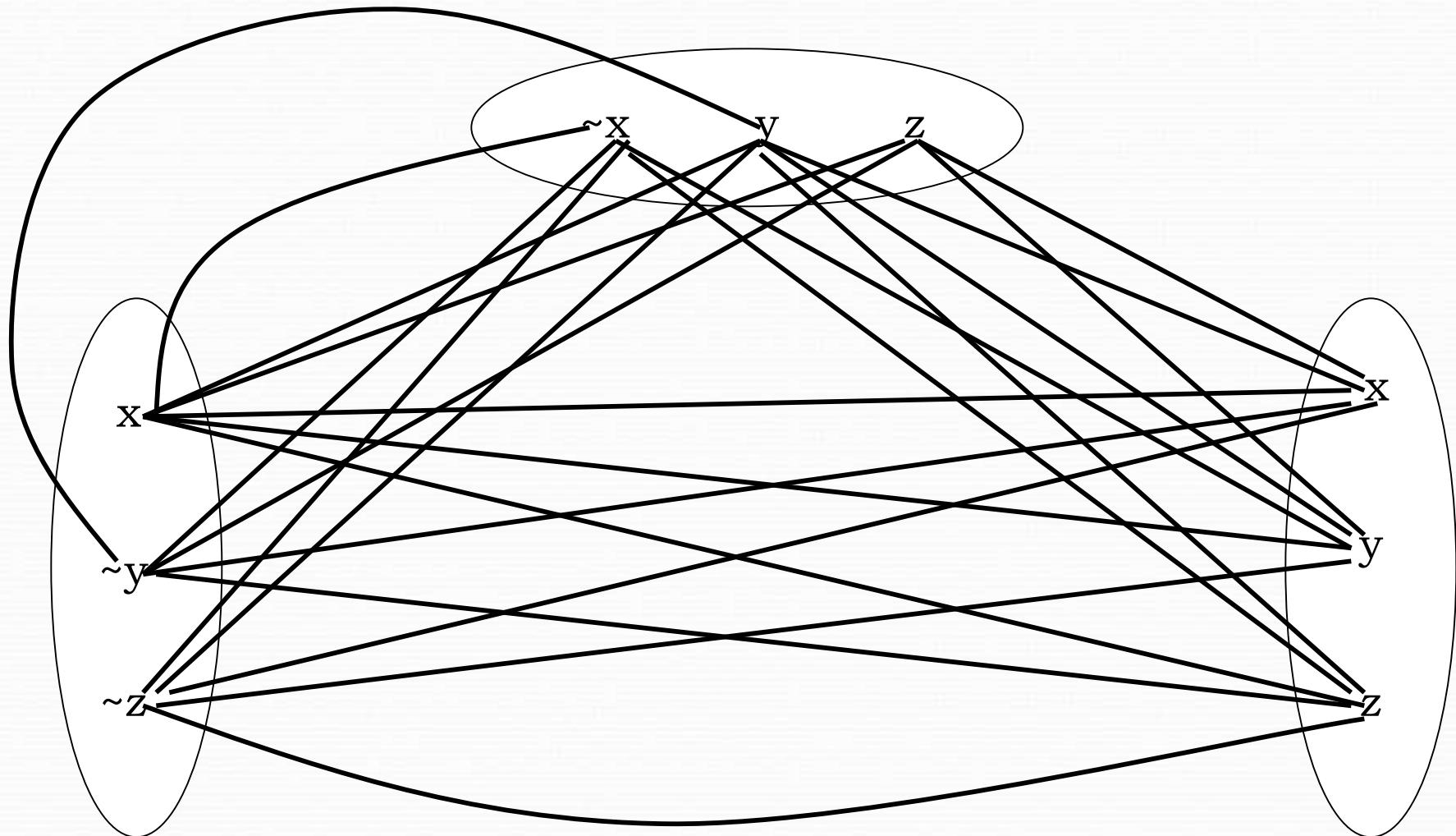
If $T \leq_p Q$ then Q is NP hard.

3CNF \leq_p Clique (Needs to be improved)

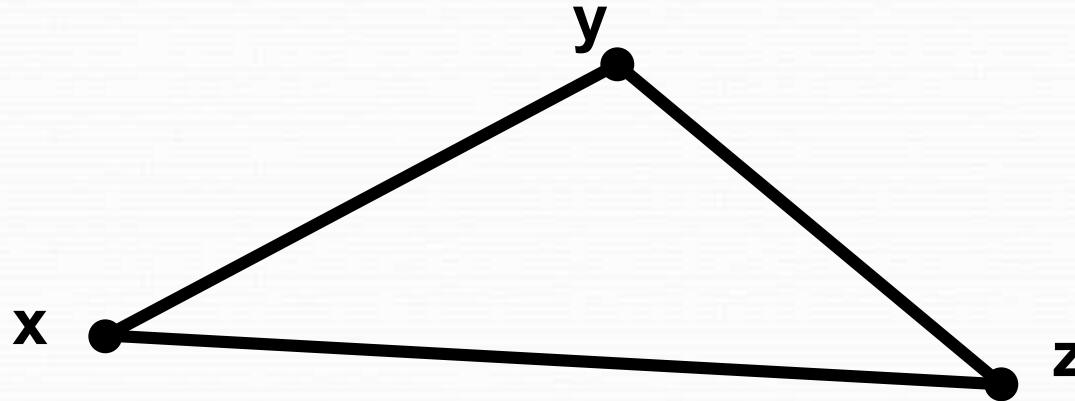
- 3CNF problem : n variables constituting k clauses
- 3CNF is a satisfiability problem, i.e.
there exists an assignment : expression results in T.
- From Cook's Th. :The satisfiability problem is NPC.
- Hence, 3CNF is NPC.
- If 3CNF reduces to Clique then clique is NP hard.

Let the expression in 3CNF be: $(\sim x \vee y \vee z) \wedge (x \vee \sim y \vee \sim z) \wedge (x \vee y \vee z)$

Expression → Graph



Clique thus formed:

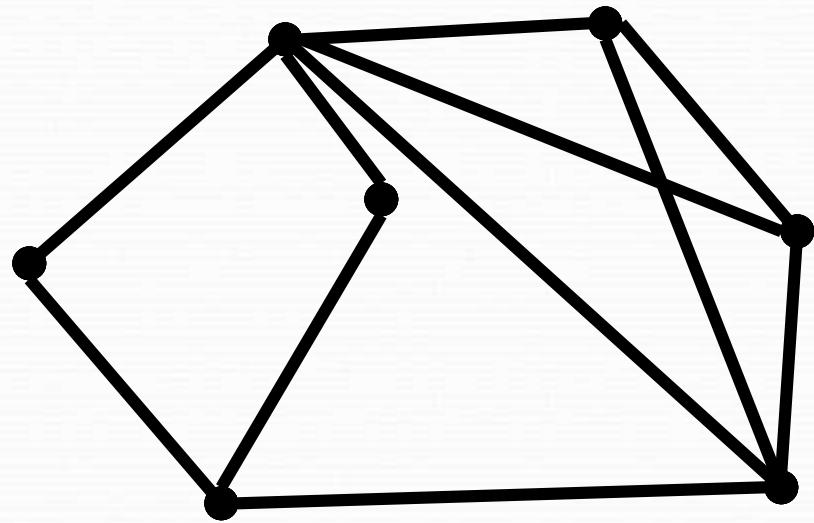


Note:- There are many other possible cliques in previous mapping. This is one of the possible cliques.

Clique \leq_p Vertex Cover

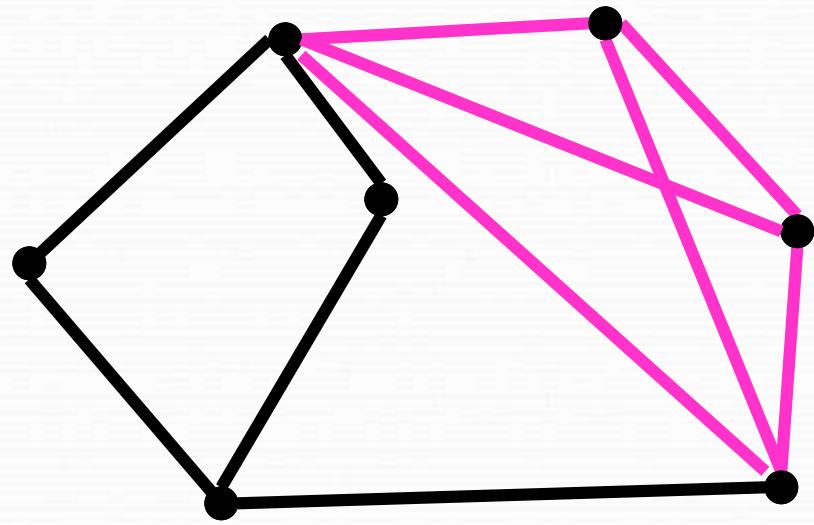
- Let the instance of Clique (I_c) be $\langle G, k \rangle$.
- Reducing it to instance of VC (I_{vc}) be $\langle G', |V|-k \rangle$ where G' : $E(G') =$ Edges b/w vertex pair not present in G and $|V|-k$ is the vertex cover.
- Catch behind this choice : Because it works...!!!

G

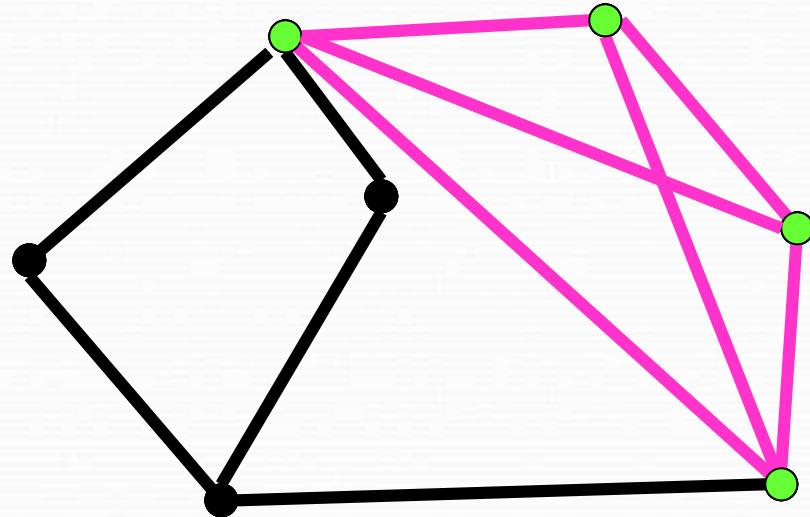


Green ovals represent CLIQUE
for this graph

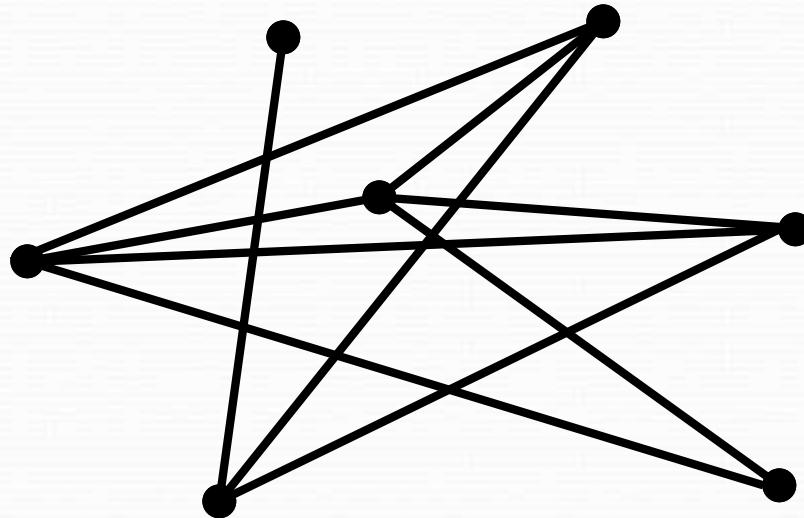
G'



G



G'

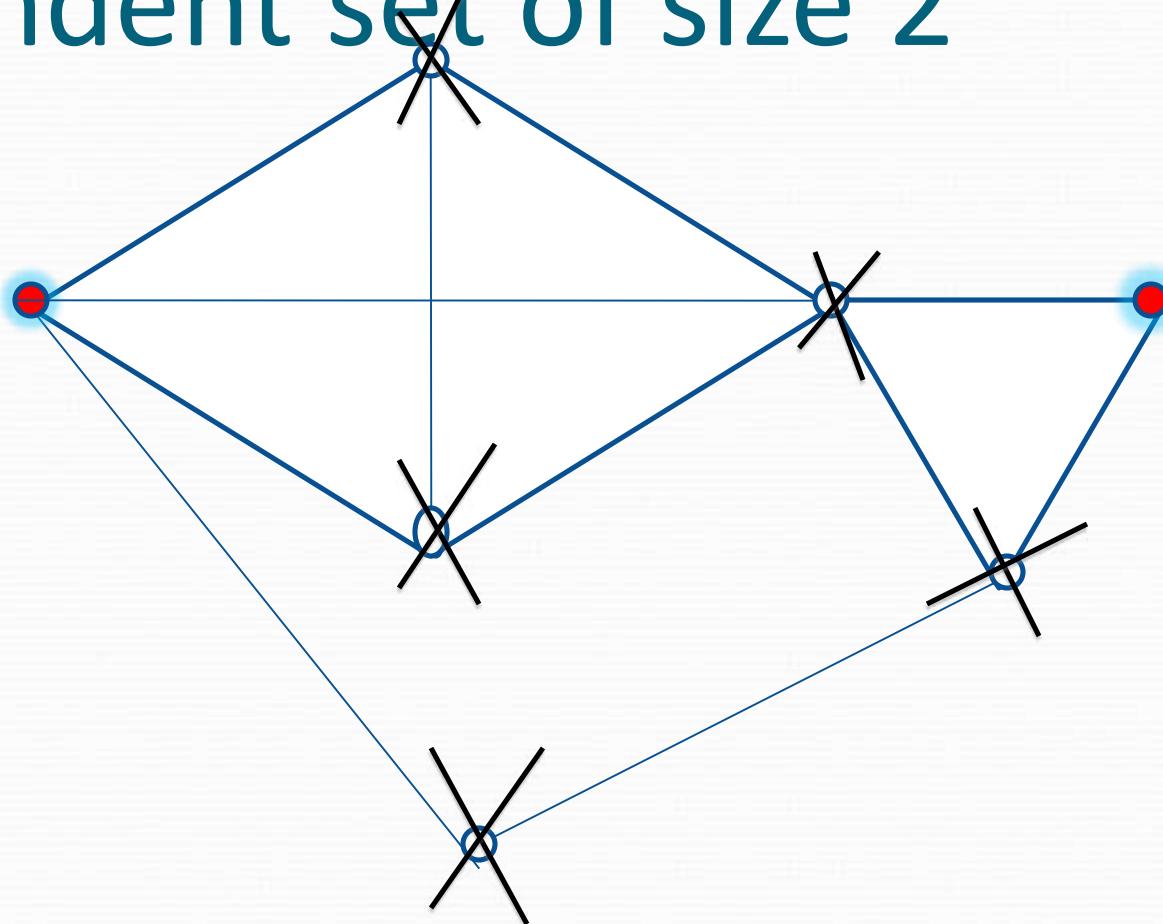


Big ovals represent
the VC for graph G'

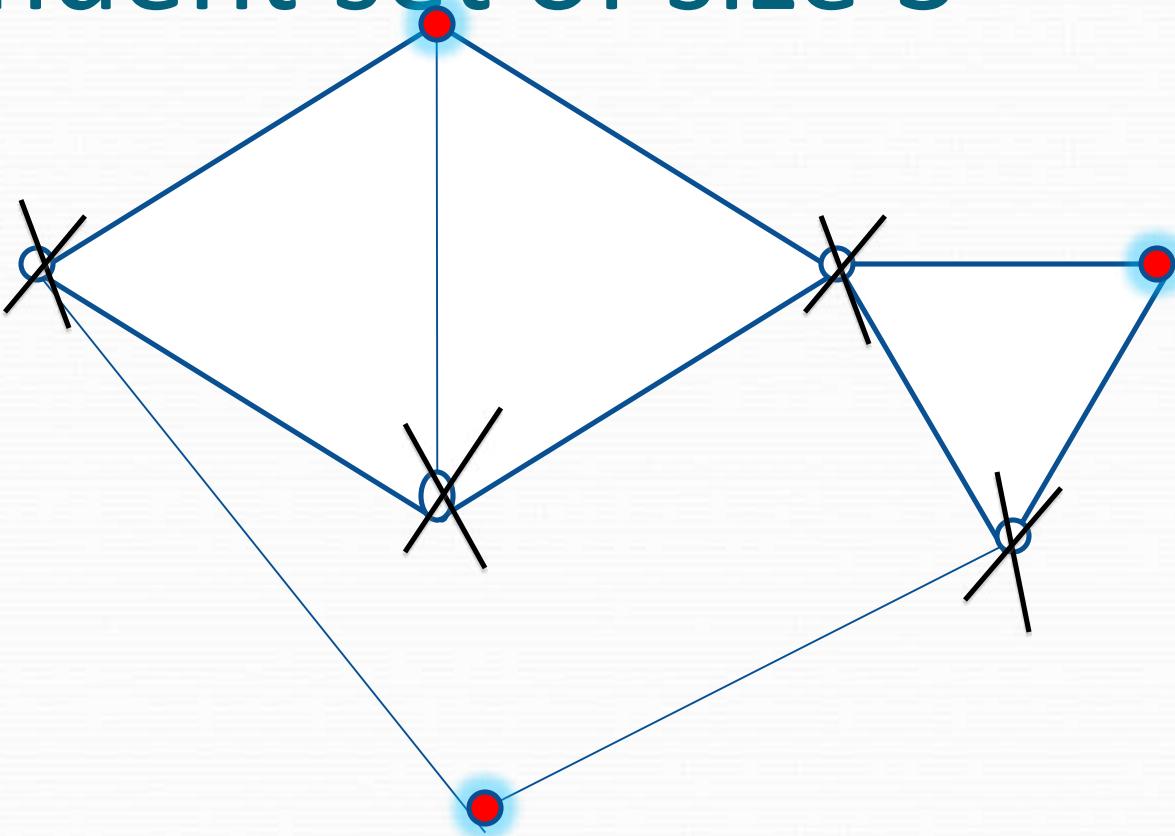
INDEPENDENT SETS

Given an undirected graph, A subset S of V is said to be independent if no two nodes in S are joined by an edge in G .

Independent set of size 2



Independent set of size 3



IS is in NPC

- Assignment : reduce vertex cover to IS.

Constrained Optimization Problems

- An objective function is optimized subject to certain constraints

SUBSET SUM PROBLEM

Subset Sum Problem

Problem Statement

Given a finite set $S = \{x_1, x_2, \dots, x_n\}$ of n elements with non-negative weights w_i and a target t , select a subset S' of items **such that the sum of weights of elements in S' is at most t** (constraint) and,

Optimization version: the sum of weights is maximized.

Decision Version: sum of weights is at least k , for a given k .

Subset Sum Problem

We now prove that Subset Sum Problem is NP-Complete.

i) Subset Sum is in NP.

For an instance $\langle S, t \rangle$,

Let S' be the certificate.

Checking whether elements of S' sum upto t (target) and it can be done in polynomial time.

Subset Sum Problem

ii) Subset Sum is NP Hard

We show this by proving that 3-SAT is reducible to Subset Sum in polynomial time.

Given: 3-SAT formula Φ over variables x_1, x_2, \dots, x_n with clauses C_1, C_2, \dots, C_k , (each containing exactly three distinct literals)

Subset Sum Problem

Without loss of generality, we make the following *2 assumptions*:

- No clause contains both a variable and its negation. WHY?
(Because such a clause would be trivially satisfied.)
- Each variable appears in *at least 1 clause*. WHY?
(Because otherwise, it does not matter what value is assigned to it.)

Subset Sum Problem

Reduction Process - through example

Consider the 3-SAT formula : $\Phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4$

where

$$C_1 = (x_1 \vee x_2' \vee x_3')$$

$$C_2 = (x_1' \vee x_2' \vee x_3')$$

$$C_3 = (x_1' \vee x_2' \vee x_3)$$

$$C_4 = (x_1 \vee x_2 \vee x_3)$$

A satisfying assignment is $\langle x_1=0, x_2=0, x_3=1 \rangle$

- The reduction creates TWO numbers (v_i, v'_i) in set S for each variable x_i and TWO numbers(s_j and s'_j) in S for each Clause C_j
- Create numbers in base 10
- Each number contains $n+k$ digits (one variable or one clause)
- We construct Set S and t as:
 - the least significant digits –k by clause
 - the most significant digits – n by variables

Thus $n+k$ digits

Subset Sum Problem

- **Filling the table**

Construct S and t as follows:

- For each x_i , add 2 integers v_i, v'_i in S.

Both v_i and v'_i **have 1** corresponding to **digit x_i** .

If x_i appears in C_j , the C_j digit in $v_i = 1$

If v'_i appears in C_j , the C_j digit in $v'_i = 1$

- All other digits are zero.

$$(x_1 \vee x_2' \vee x_3') \wedge (x_1' \vee x_2' \vee x_3) \wedge (x_1' \vee x_2' \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$$

	x1	x2	x3	c1	c2	c3	c4
v1	1	0	0	1	0	0	1
v1'	1	0	0	0	1	1	0
v2	0	1	0	0	0	0	1
v2'	0	1	0	1	1	1	0
v3	0	0	1	0	0	1	1
v3'	0	0	1	1	1	0	0
s1	0	0	0	1	0	0	0
s1'	0	0	0	2	0	0	0
s2	0	0	0	0	1	0	0
s2'	0	0	0	0	2	0	0
s3	0	0	0	0	0	1	0
s3'	0	0	0	0	0	2	0
s4	0	0	0	0	0	0	1
s4'	0	0	0	0	0	0	2
t	1	1	1	4	4	4	4

Subset Sum Problem

- Construction of S ant cont.....
- For all C_j , add s_j and s_j' integers in S.

Both (v's &s's) have 0's in all digits other than the one labeled by C_j .

- s_j has a 1 corresponding to C_j , and s_j' has a 2 corresponding to C_j .
- These integers are slack variables(s_j and s_j'), used to get clause labeled digit position to add to the target value of 4.
- Target t has a 1 in each digit labeled by a variable and 4 in each clause-digit.
- $k = t$: Thus the instance of SS requires that the sum of weights of the selected elements be exactly equal (at most and at least) to t.

$$(x_1 \vee x_2' \vee x_3') \wedge (x_1' \vee x_2' \vee x_3) \wedge (x_1' \vee x_2' \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$$

	x1	x2	x3	c1	c2	c3	c4
v1	1	0	0	1	0	0	1
v1'	1	0	0	0	1	1	0
v2	0	1	0	0	0	0	1
v2'	0	1	0	1	1	1	0
v3	0	0	1	0	0	1	1
v3'	0	0	1	1	1	0	0
s1	0	0	0	1	0	0	0
s1'	0	0	0	2	0	0	0
s2	0	0	0	0	1	0	0
s2'	0	0	0	0	2	0	0
s3	0	0	0	0	0	1	0
s3'	0	0	0	0	0	2	0
s4	0	0	0	0	0	0	1
s4'	0	0	0	0	0	0	2
t	1	1	1	4	4	4	4

Subset Sum Problem

Claim: All v_i and v_i' in S are unique

- v_i (v_i') and v_j (v_j') will be different in most significant n digits positions.
- v_i and v_i' will be different in least significant k digit positions. WHY?
(both cannot belong to the same clause)

Subset Sum Problem

Claim: All s_j and s_j' in S are unique

(for reasons similar to v_i and v_i')

Observation: The greatest sum of digits in any digit position is 6. This occurs in clause- digits (v_i and v_i' make a contribution of 3, s_j and s_j' make a contribution of 1 and 2 respectively).

Conclusion: Interpretation is in base 10, so **no carries would be generated.**

- The reduction can be performed in polynomial time.
- The set contains $2n+2k$ values each of which has $n+k$ digits and the time to produce each digit is polynomial in time $n+k$.
- The target t has $n+k$ digits and the reduction produces each in constant time

Solution to 3CNF => Solution to Subset Sum

- Do the following for $i = 1.....n$
- If $x_i = 1$ in the assignment, include v_i in S' , otherwise include v'_i . In the example,
- $x_1=0 \Rightarrow x'_1=1$, v'_1 is selected
- $x_2=0 \Rightarrow x'_2=1$, v'_2 is selected
- $x_3=1 \Rightarrow x_3=1$, v_3 is selected
- Include S_i 's as per the shortfall.

$$(x_1 \vee x_2' \vee x_3') \wedge (x_1' \vee x_2' \vee x_3) \wedge (x_1' \vee x_2' \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$$

Satisfying assignment $x_1=0$, $x_2=0$, $x_3=1$

v_1' , v_2' and v_3 are selected.

	X1	X2	X3	C1	C2	C3	C4
V1	1	0	0	1	0	0	1
V1'	1	0	0	0	1	1	0
V2	0	1	0	0	0	0	1
V2'	0	1	0	1	1	1	0
V3	0	0	1	0	0	1	1
V3'	0	0	1	1	1	0	0
S1	0	0	0	1	0	0	0
S1'	0	0	0	2	0	0	0
S2	0	0	0	0	1	0	0
S2'	0	0	0	0	2	0	0
S3	0	0	0	0	0	1	0
S3'	0	0	0	0	0	2	0
S4	0	0	0	0	0	0	1
S4'	0	0	0	0	0	0	2
t	1	1	1	4	4	4	4

$$(x_1 \vee x_2' \vee x_3') \wedge (x_1' \vee x_2' \vee x_3) \wedge (x_1' \vee x_2' \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$$

Satisfying assignment $x_1=0, x_2=0, x_3=1$

	X1	X2	X3	C1	C2	C3	C4
V1	1	0	0	1	0	0	1
V1'	1	0	0	0	1	1	0
V2	0	1	0	0	0	0	1
V2'	0	1	0	1	1	1	0
V3	0	0	1	0	0	1	1
V3'	0	0	1	1	1	0	0
S1	0	0	0	1	0	0	0
S1'	0	0	0	2	0	0	0
S2	0	0	0	0	1	0	0
S2'	0	0	0	0	2	0	0
S3	0	0	0	0	0	1	0
S3'	0	0	0	0	0	2	0
S4	0	0	0	0	0	0	1
S4'	0	0	0	0	0	0	2
t	1	1	1	4	4	4	4

$$(x_1 \vee x_2' \vee x_3') \wedge (x_1' \vee x_2' \vee x_3) \wedge (x_1' \vee x_2' \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$$

Satisfying assignment $x_1=0, x_2=0, x_3=1$

	X1	X2	X3	C1	C2	C3	C4
V1	1	0	0	1	0	0	1
V1'	1	0	0	0	1	1	0
V2	0	1	0	0	0	0	1
V2'	0	1	0	1	1	1	0
V3	0	0	1	0	0	1	1
V3'	0	0	1	1	1	0	0
S1	0	0	0	1	0	0	0
S1'	0	0	0	2	0	0	0
S2	0	0	0	0	1	0	0
S2'	0	0	0	0	2	0	0
S3	0	0	0	0	0	1	0
S3'	0	0	0	0	0	2	0
S4	0	0	0	0	0	0	1
S4'	0	0	0	0	0	0	2
t	1	1	1	4	4	4	4

$$(x_1 \vee x_2' \vee x_3') \wedge (x_1' \vee x_2' \vee x_3) \wedge (x_1' \vee x_2' \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$$

Satisfying assignment $x_1=0, x_2=0, x_3=1$

	x1	x2	x3	c1	c2	c3	c4
v1	1	0	0	1	0	0	1
v1'	1	0	0	0	1	1	0
v2	0	1	0	0	0	0	1
v2'	0	1	0	1	1	1	0
v3	0	0	1	0	0	1	1
v3'	0	0	1	1	1	0	0
s1	0	0	0	1	0	0	0
s1'	0	0	0	2	0	0	0
s2	0	0	0	0	1	0	0
s2'	0	0	0	0	2	0	0
s3	0	0	0	0	0	1	0
s3'	0	0	0	0	0	2	0
s4	0	0	0	0	0	0	1
s4'	0	0	0	0	0	0	2
t	1	1	1	4	4	4	4

$$(x_1 \vee x_2 \vee x_3) \wedge (x_1' \vee x_2' \vee x_3') \wedge (x_1' \vee x_2' \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$$

Satisfying assignment $x_1=0, x_2=0, x_3=1$

	X1	X2	X3	C1	C2	C3	C4
V1	1	0	0	1	0	0	1
V1'	1	0	0	0	1	1	0
V2	0	1	0	0	0	0	1
V2'	0	1	0	1	1	1	0
V3	0	0	1	0	0	1	1
V3'	0	0	1	1	1	0	0
S1	0	0	0	1	0	0	0
S1'	0	0	0	2	0	0	0
S2	0	0	0	0	1	0	0
S2'	0	0	0	0	2	0	0
S3	0	0	0	0	0	1	0
S3'	0	0	0	0	0	2	0
S4	0	0	0	0	0	0	1
S4'	0	0	0	0	0	0	2
t	1	1	1	4	4	4	4

Solution to SS => Solution to 3CNF

- For each clause j , maximum that can be picked from S_j 's is 3. Therefore at least for one i , at least one of v_i or v'_i must have been picked in S' . Set that to 1.
- Claim: there is no conflict.
- Claim: the assignment is satisfying.
- Proof: do it yourself.

- To prove : If Φ is satisfiable then there exist a set S' subset of S with target t

If $x_i=1$, in the assignment include v_i in S' (by example v_3 included as $x_1=0, x_2=0, x_3=1$)

If $x_i=0$, in this assignment include v_i' in S'