

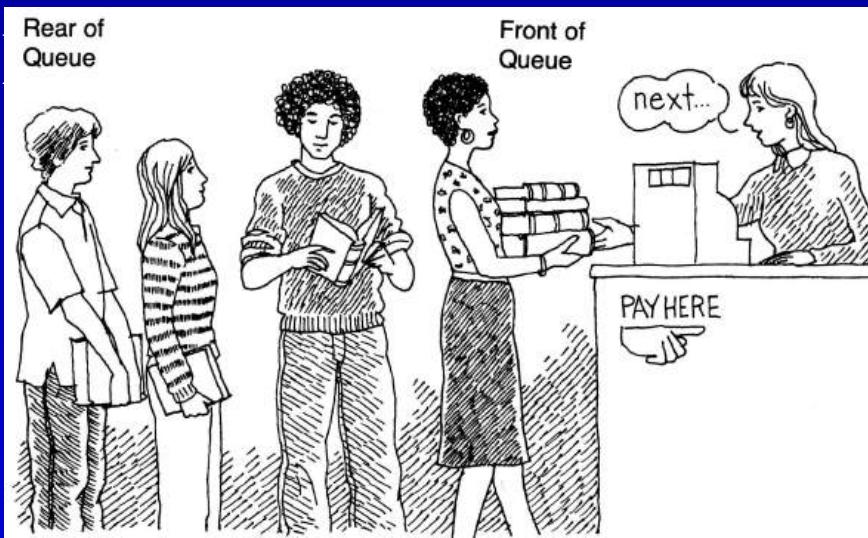
# Queues

Data Structures

# What is a queue?

- It is an ordered group of homogeneous items of elements.
- Queues have two ends:
  - Elements are added at one end.
  - Elements are removed from the other end.
- The element added first is also removed first

(FIFO:



# Queue Specification

- Definitions: (provided by the user)
  - *MAX\_ITEMS*: Max number of items that might be on the queue
  - *ItemType*: Data type of the items on the queue
- Operations
  - MakeEmpty
  - Boolean IsEmpty
  - Boolean IsFull
  - Enqueue (*ItemType* newItem)
  - Dequeue (*ItemType&* item)

# Enqueue (ItemType newItem)

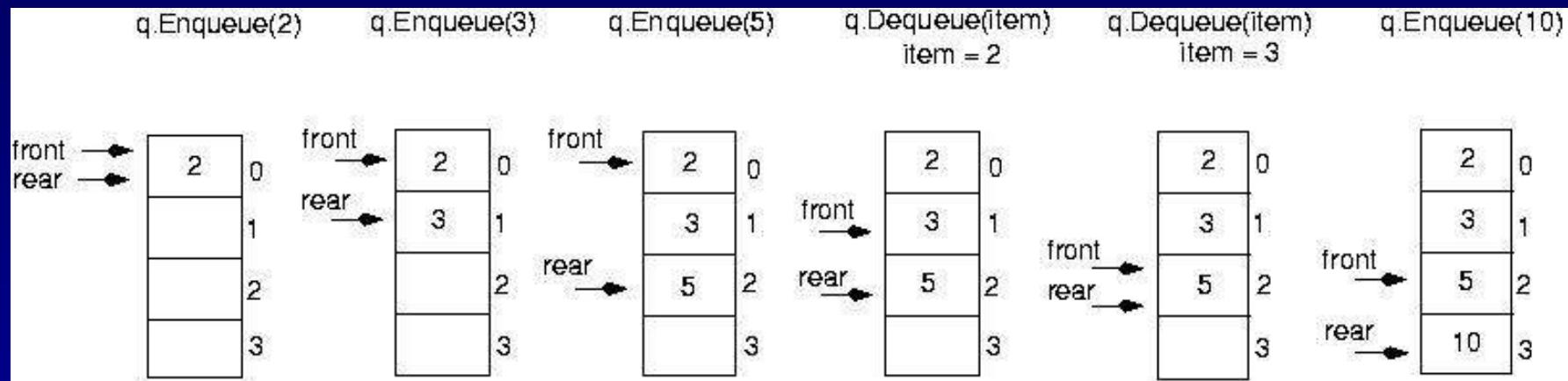
- *Function:* Adds newItem to the rear of the queue.
- *Preconditions:* Queue has been initialized and is not full.
- *Postconditions:* newItem is at rear of queue.

# Dequeue (ItemType& item)

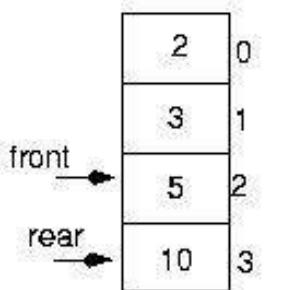
- *Function:* Removes front item from queue and returns it in item.
- *Preconditions:* Queue has been initialized and is not empty.
- *Postconditions:* Front element has been removed from queue and item is a copy of removed element.

# Implementation issues

- Implement the queue as a *circular structure*.
- How do we know if a queue is full or empty?
- Initialization of *front* and *rear*.
- Testing for a *full* or *empty* queue.



q.Enqueue(20) ???

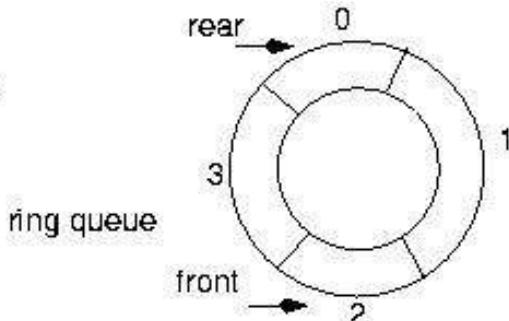
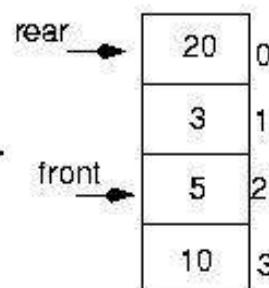


Let the queue elements  
"wrap around"

```

if(rear == maxQue -1)
    rear = 0;
else
    rear = rear + 1;
or
rear = (rear + 1) % maxQue;

```

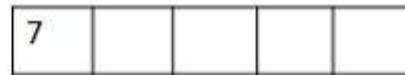




[0] [1] [2] [3] [4]

front= -1 rear= -1

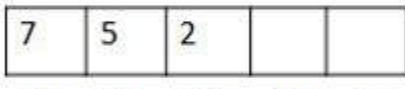
Empty queue



[0] [1] [2] [3] [4]

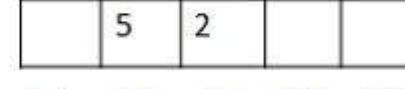
front=0 rear=0

One element added to queue



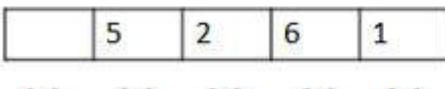
[0] [1] [2] [3] [4]

front = 0 rear= 2



[0] [1] [2] [3] [4]

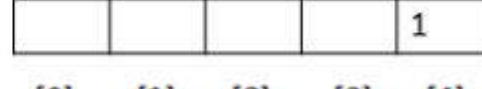
front= 1 rear= 2



[0] [1] [2] [3] [4]

front = 1 rear= 4

Two elements added to queue



[0] [1] [2] [3] [4]

front = 4 rear= 4

Three elements deleted from queue

```
@Override
public void insertitem(int item) {

    if(r==a.length-1)
    {
        System.out.println("queue is full");
    }
    else
    {

        if(f==-1)
        {
            f=0;
        }
        r++;
        a[r]=item;
        System.out.println(a[r]);
    }
}
```

```
@Override  
public void deleteitem() {  
    if(f==-1)  
    {  
        System.out.println("queue is underflow");  
    }  
    else  
    {  
  
        System.out.println("deleted item is"+a[f]);  
  
        f++;  
    }  
}
```

```
public void display()
{
    for(int i=f;i<=r;i++)
    {
        System.out.println("data of queue is"+a[i]);
    }
}
```

# CIRCULAR QUEUE

- A circular queue is similar to a linear queue as it is also based on the FIFO (First In First Out) principle except that the last position is connected to the first position in a circular queue that forms a circle. It is also known as a *Ring Buffer*.

# Operations on Circular Queue

- **Front:** It is used to get the front element from the Queue.
- **Rear:** It is used to get the rear element from the Queue.
- **enQueue(value):** This function is used to insert the new value in the Queue. The new element is always inserted from the rear end.
- **deQueue():** This function deletes an element from the Queue. The deletion in a Queue always takes

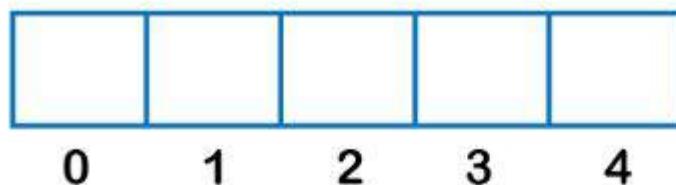
- Enqueue operation
- The steps of enqueue operation are given below:
  - First, we will check whether the Queue is full or not.
  - Initially the front and rear are set to -1. When we insert the first element in a Queue, front and rear both are set to 0.
  - When we insert a new element, the rear gets incremented, i.e.,  $rear=rear+1$ .
  -

- Dequeue Operation
- The steps of dequeue operation are given below:
- First, we check whether the Queue is empty or not. If the queue is empty, we cannot perform the dequeue operation.
- When the element is deleted, the value of front gets decremented by 1.
- If there is only one element left which is to be deleted, then the front and rear are reset to -1.
-

```
void insertCQ(int val) {  
    if ((front == 0 && rear == n-1) || (front == rear+1)) {  
        cout<<"Queue Overflow \n";  
        return;  
    }  
    if (front == -1) {  
        front = 0;  
        rear = 0;  
    }  
    else {  
        if (rear == n - 1)  
            rear = 0;  
        else  
            rear = rear + 1;  
    }  
    cqueue[rear] = val ;  
}
```

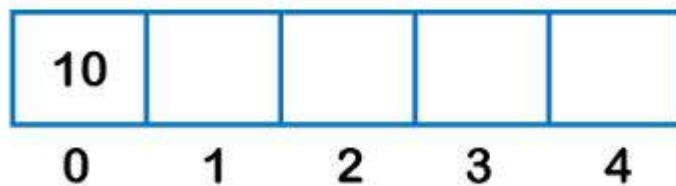
- **Algorithm to delete an element from the circular queue**
- **Step 1:** IF FRONT = -1  
    Write " UNDERFLOW "  
    Goto Step 4  
    [END of IF]
- **Step 2:** SET VAL = QUEUE[FRONT]
- **Step 3:** IF FRONT = REAR  
    SET FRONT = REAR = -1  
    ELSE  
        IF FRONT = MAX -1  
            SET FRONT = 0  
        ELSE  
            SET FRONT = FRONT + 1  
        [END of IF]  
    [END OF IF]
- **Step 4:** EXIT

```
void deleteCQ() {
    if (front == -1) {
        cout<<"Queue Underflow\n";
        return ;
    }
    cout<<"Element deleted from queue is : "<<cqueue[front]<<endl;
    if (front == rear) {
        front = -1;
        rear = -1;
    }
    else {
        if (front == n - 1)
            front = 0;
        else
            front = front + 1;
    }
}
```



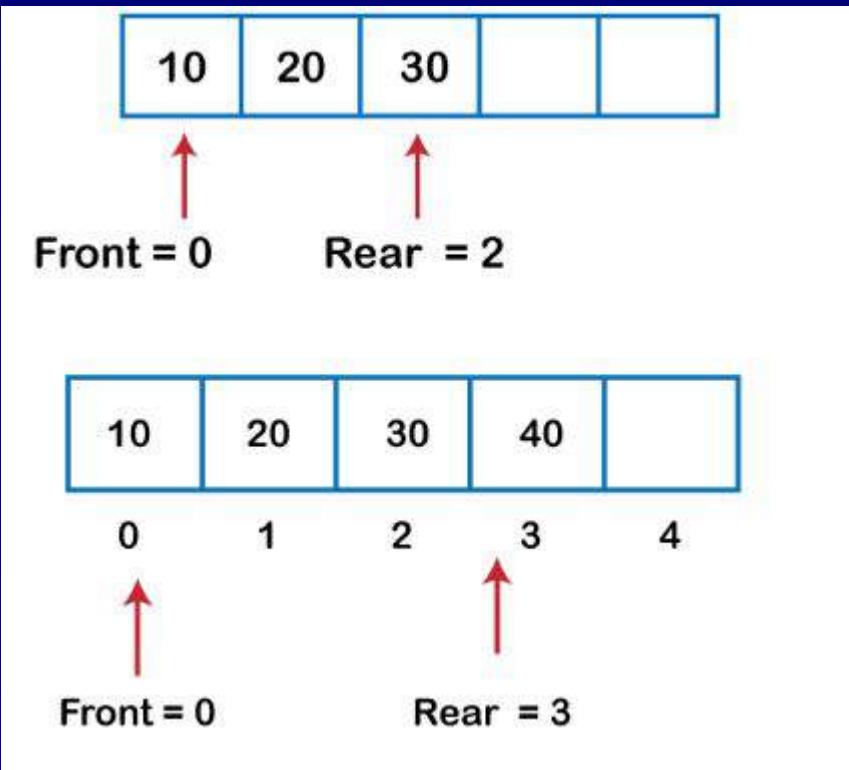
**Front = -1**

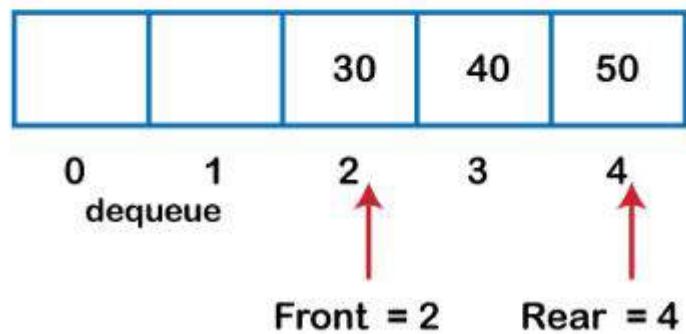
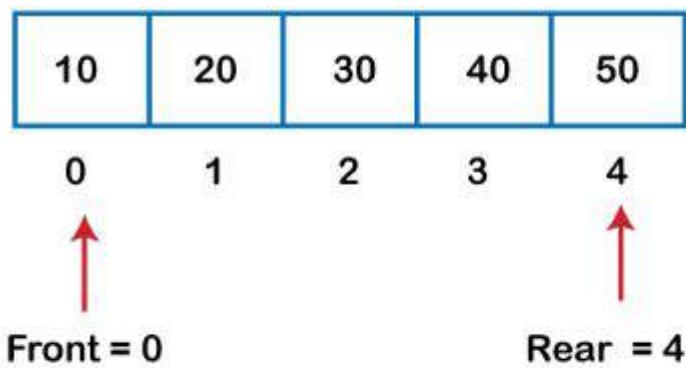
**Rear = -1**



**Front = 0**

**Rear = 0**





60		30	40	50
----	--	----	----	----

0 1 2 3 4

 Rear       Front

60	70	30	40	50
----	----	----	----	----

0 1 2 3 4

 Rear       Front

# Priority Queue

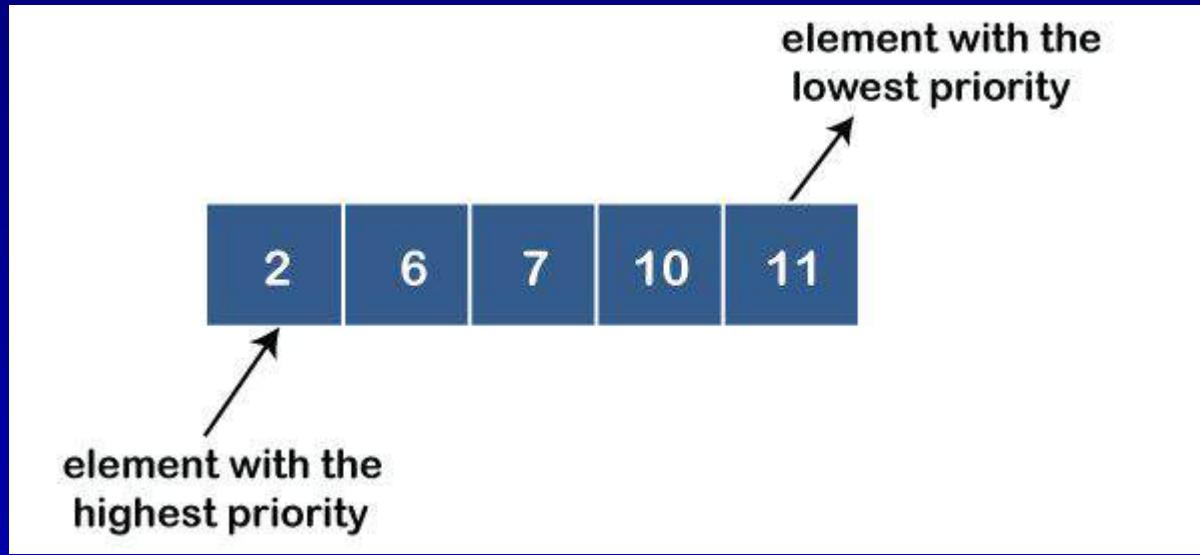
- A priority queue is an abstract data type that behaves similarly to the normal queue except that each element has some priority, i.e., the element with the highest priority would come first in a priority queue. The priority of the elements in a priority queue will determine the order in which elements are removed from the priority queue.

- For example, suppose we have some values like 1, 3, 4, 8, 14, 22 inserted in a priority queue with an ordering imposed on the values is from least to the greatest. Therefore, the 1 number would be having the highest priority while 22 will be having the lowest priority.

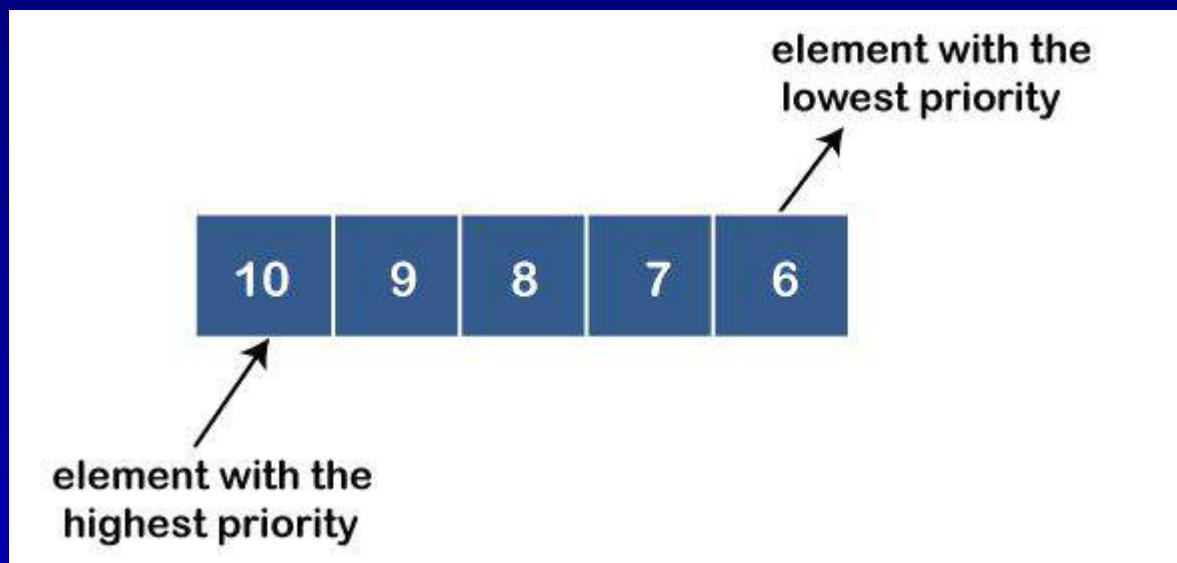
- Characteristics of a Priority queue
- A priority queue is an extension of a queue that contains the following characteristics:
- Every element in a priority queue has some priority associated with it.
- An element with the higher priority will be deleted before the deletion of the lesser priority.
- If two elements in a priority queue have the same priority, they will be arranged using the FIFO principle.

- Let's understand the priority queue through an example.
- We have a priority queue that contains the following values:
  - 1, 3, 4, 8, 14, 22
- All the values are arranged in ascending order. Now, we will observe how the priority queue will look after performing the following operations:
- **poll()**: This function will remove the highest priority element from the priority queue. In the above priority queue, the '1' element has the highest priority, so it will be removed from the priority queue.
- **add(2)**: This function will insert '2' element in a priority queue. As 2 is the smallest element among all the numbers so it will obtain the highest priority.
- **poll()**: It will remove '2' element from the priority queue as it has the highest priority queue.
- **add(5)**: It will insert 5 element after 4 as 5 is larger than 4 and lesser than 8, so it will obtain the third highest priority in a priority queue.

- Types of Priority Queue
- **There are two types of priority queue:**
- **Ascending order priority queue:** In ascending order priority queue, a lower priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in an ascending order like 1,2,3,4,5; therefore, the smallest number, i.e., 1 is given as the highest priority in a priority queue.

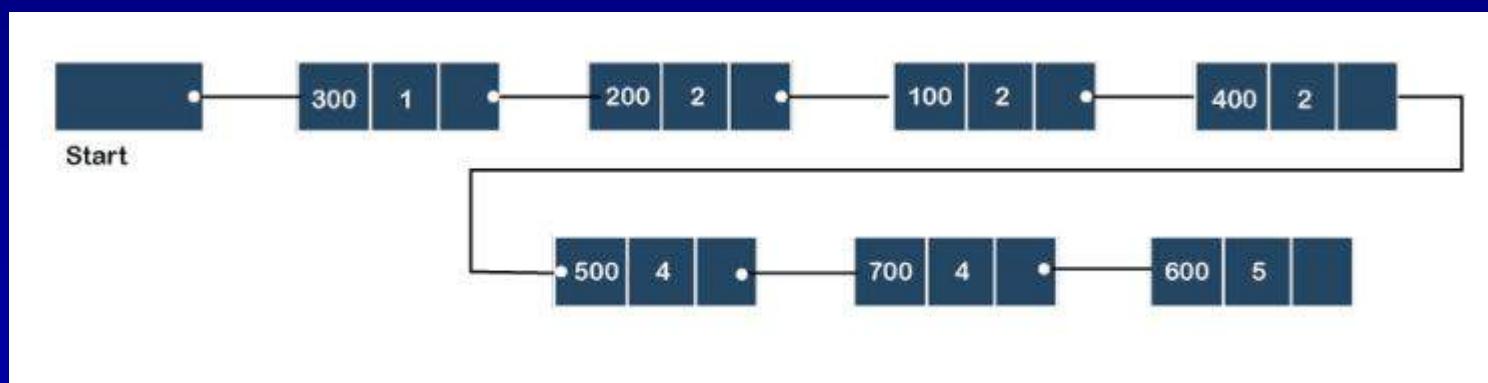


- **Descending order priority queue:** In descending order priority queue, a higher priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in descending order like 5, 4, 3, 2, 1; therefore, the largest number, i.e., 5 is given as the highest priority in a priority queue.



- In the case of priority queue, lower priority number is considered the higher priority, i.e., lower priority number = higher priority.
- Step 1: In the list, lower priority number is 1, whose data value is 333, so it will be inserted in the list as shown in the below diagram:
- Step 2: After inserting 333, priority number 2 is having a higher priority, and data values associated with this priority are 222 and 111. So, this data will be inserted based on the FIFO principle; therefore 222 will be added first and then 111.

- **Step 3:** After inserting the elements of priority 2, the next higher priority number is 4 and data elements associated with 4 priority numbers are 444, 555, 777. In this case, elements would be inserted based on the FIFO principle; therefore, 444 will be added first, then 555, and then 777.
- **Step 4:** After inserting the elements of priority 4, the next higher priority number is 5, and the value associated with priority 5 is 666, so it will be inserted at the end of the queue.



<b>Stack</b>	<b>Queue</b>
A Linear List Which allows insertion or deletion of an element at one end only is called as Stack	A Linear List Which allows insertion at one end and deletion at another end is called as Queue
Since insertion and deletion of an element are performed at one end of the stack, the elements can only be removed in the opposite order of insertion.	Since insertion and deletion of an element are performed at opposite end of the queue, the elements can only be removed in the same order of insertion.
Stack is called as Last In First Out (LIFO) List.	Queue is called as First In First Out (FIFO) List.
The most and least accessible elements are called as TOP and BOTTOM of the stack	Insertion of element is performed at FRONT end and deletion is performed from REAR end
Example of stack is arranging plates in one above one.	Example is ordinary queue in provisional store.
Insertion operation is referred as PUSH and deletion operation is referred as POP	Insertion operation is referred as ENQUEUE and deletion operation is referred as DQUEUE
Function calling in any languages uses Stack	Task Scheduling by Operating System uses queue



---

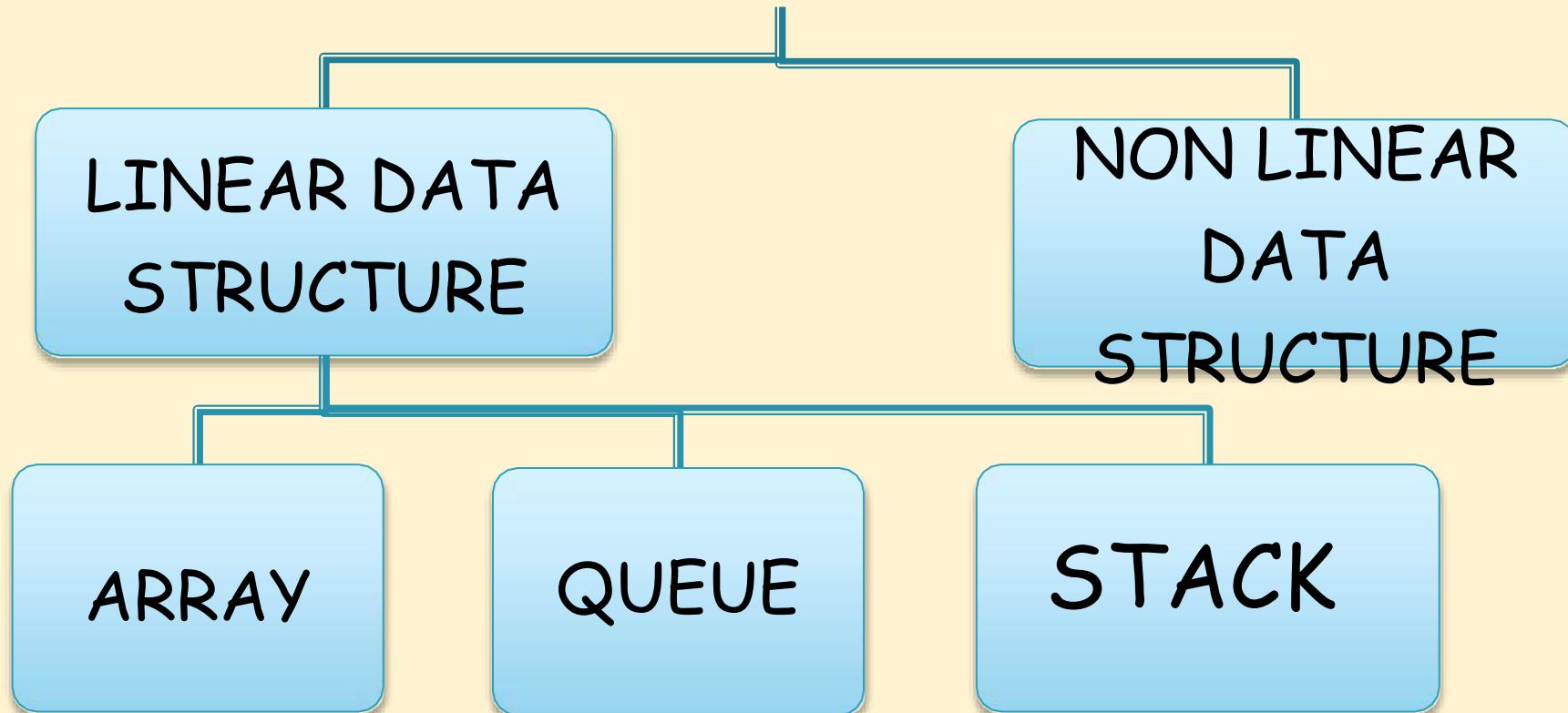
PP SAVANI  
UNIVERSITY

LECTURE-1

# STACKS

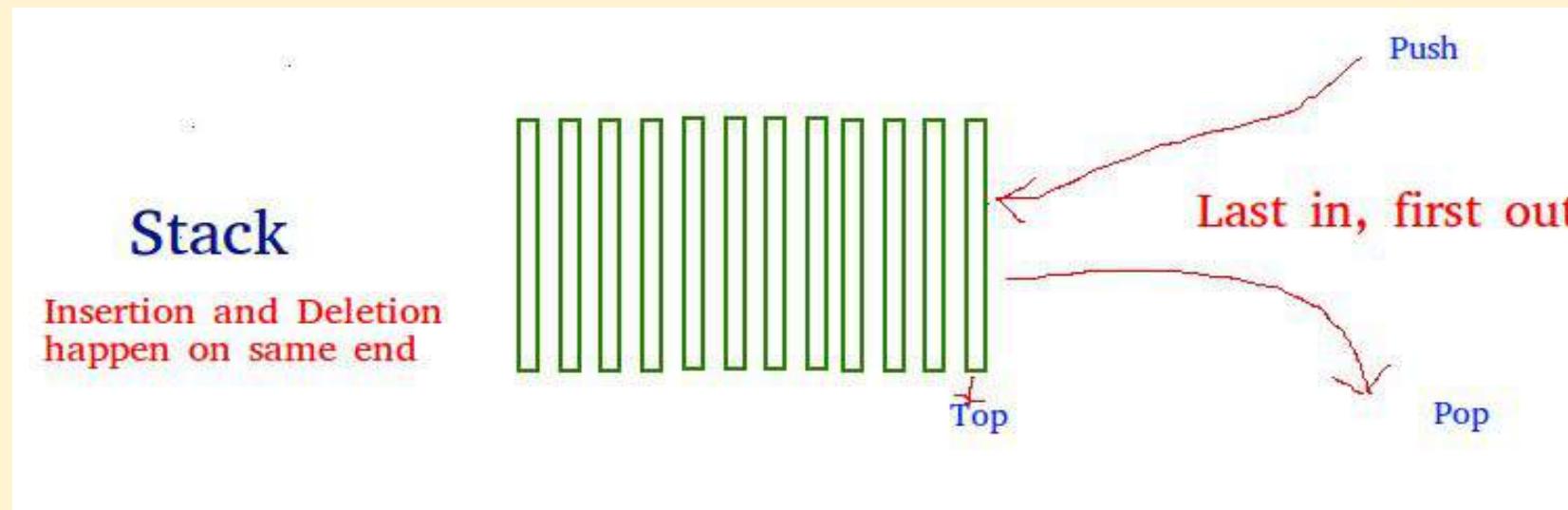


# DATA STRUCTURE



# STACK

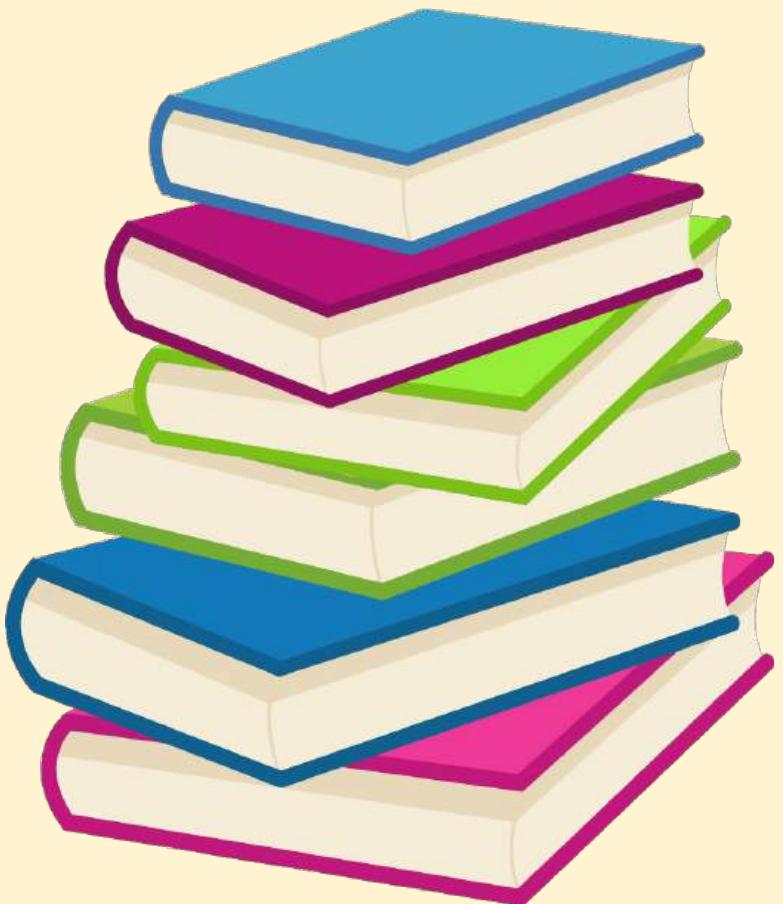
- A linear data structure following a particular order in which the operations are performed.
- The order may be LIFO(Last In First Out) or FILO(First In Last Out).





P P SAVANI  
UNIVERSITY

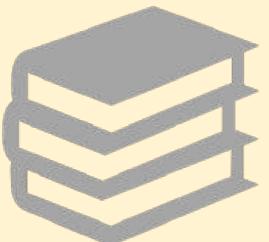
## EXAMPLES OF STACK



# Real Life Example

- There are many real-life examples of a stack.
- Consider an example of plates stacked over one another. The plate which is at the top is the first one to be removed, i.e. the plate which has been placed at the bottommost position remains in the stack for the longest period of time.
- So, it can be simply seen to follow LIFO(Last In First Out)/FILO(First In Last Out) order.

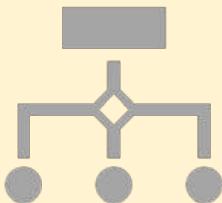
# Stack Specification



## Definitions: (provided by the user)

*MAX\_ITEMS*: Max number of items that might be on the stack

*ItemType*: Data type of the items on the stack



## Operations

Boolean IsEmpty

Boolean IsFull

Push

Pop



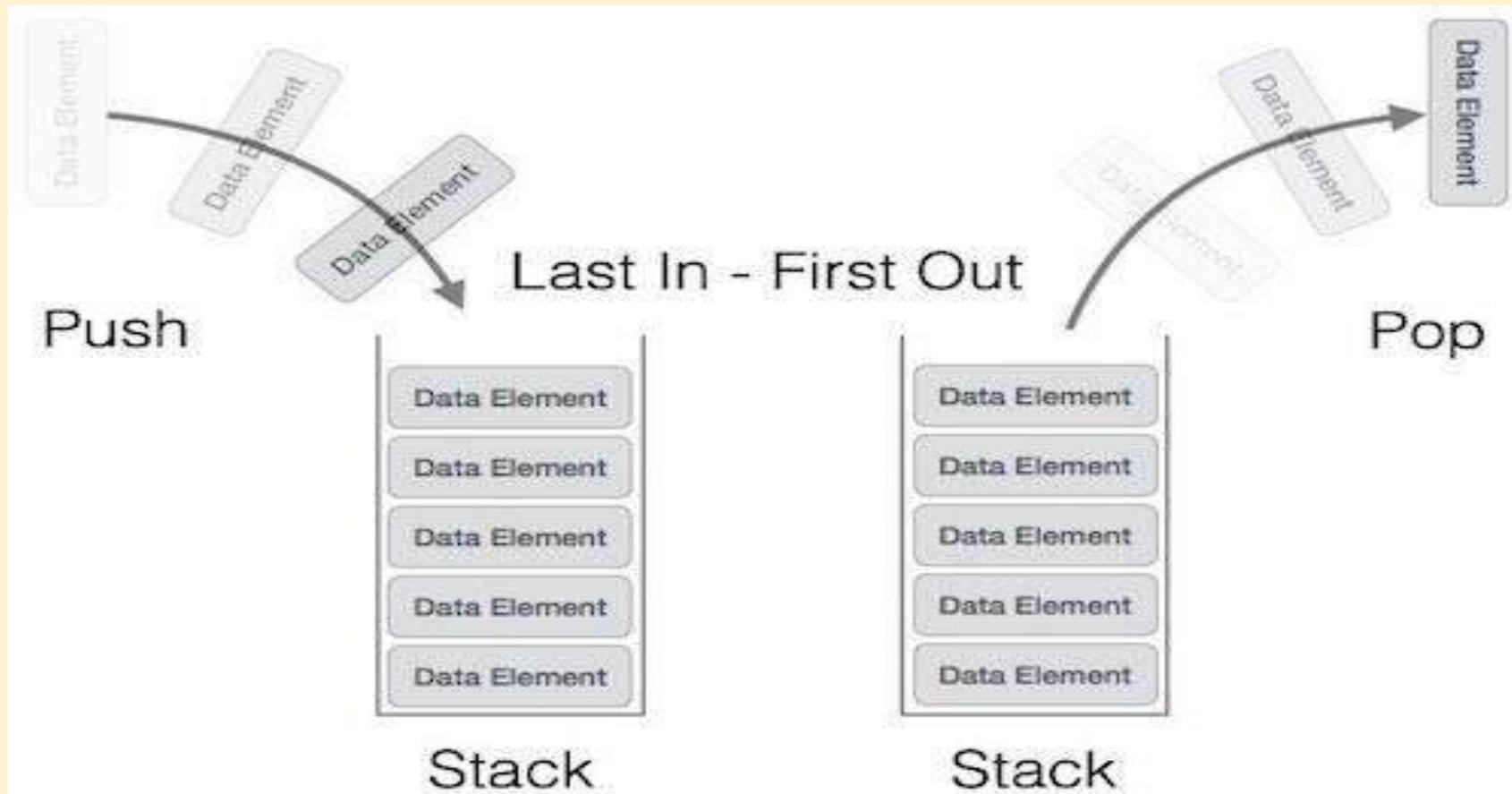
## **Stack overflow**

- The condition resulting from trying to push an element onto a full stack.

## **Stack underflow**

- The condition resulting from trying to pop an empty stack.

# Stack representation



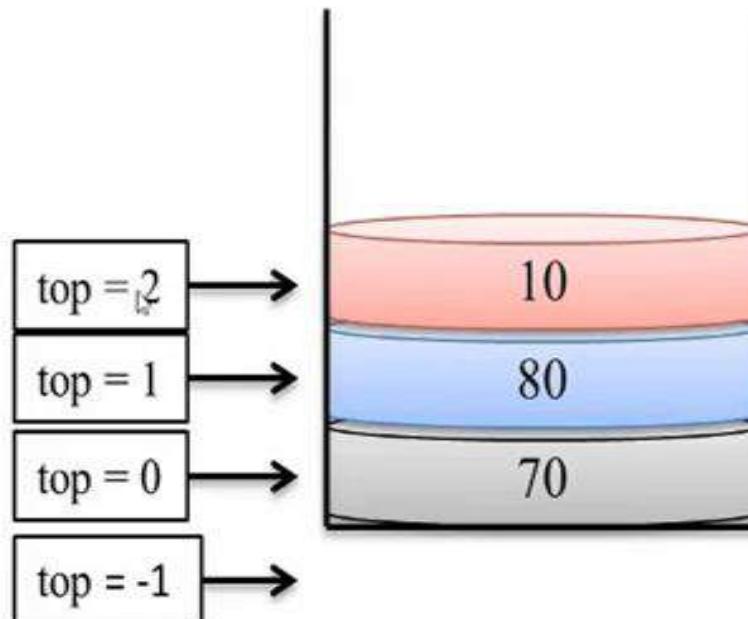
# Implementation of Stack

- A stack can be implemented by means of Arrays and Linked List, in java.
- Stack can either be a fixed size one or it may have a sense of dynamic resizing.
- Implementing stack using arrays, makes it a fixed size stack implementation.

```
int arr[5]; int top = -1;
```

- Insertion and deletion at the top of the stack only.
- Initially when the stack is empty,  $\text{top} = -1$
- For Push operation, first the value of  $\text{top}$  is increased by 1 and then the new element is pushed at the position of  $\text{top}$ .

Push (70)  
Push (80)  
Push (10)

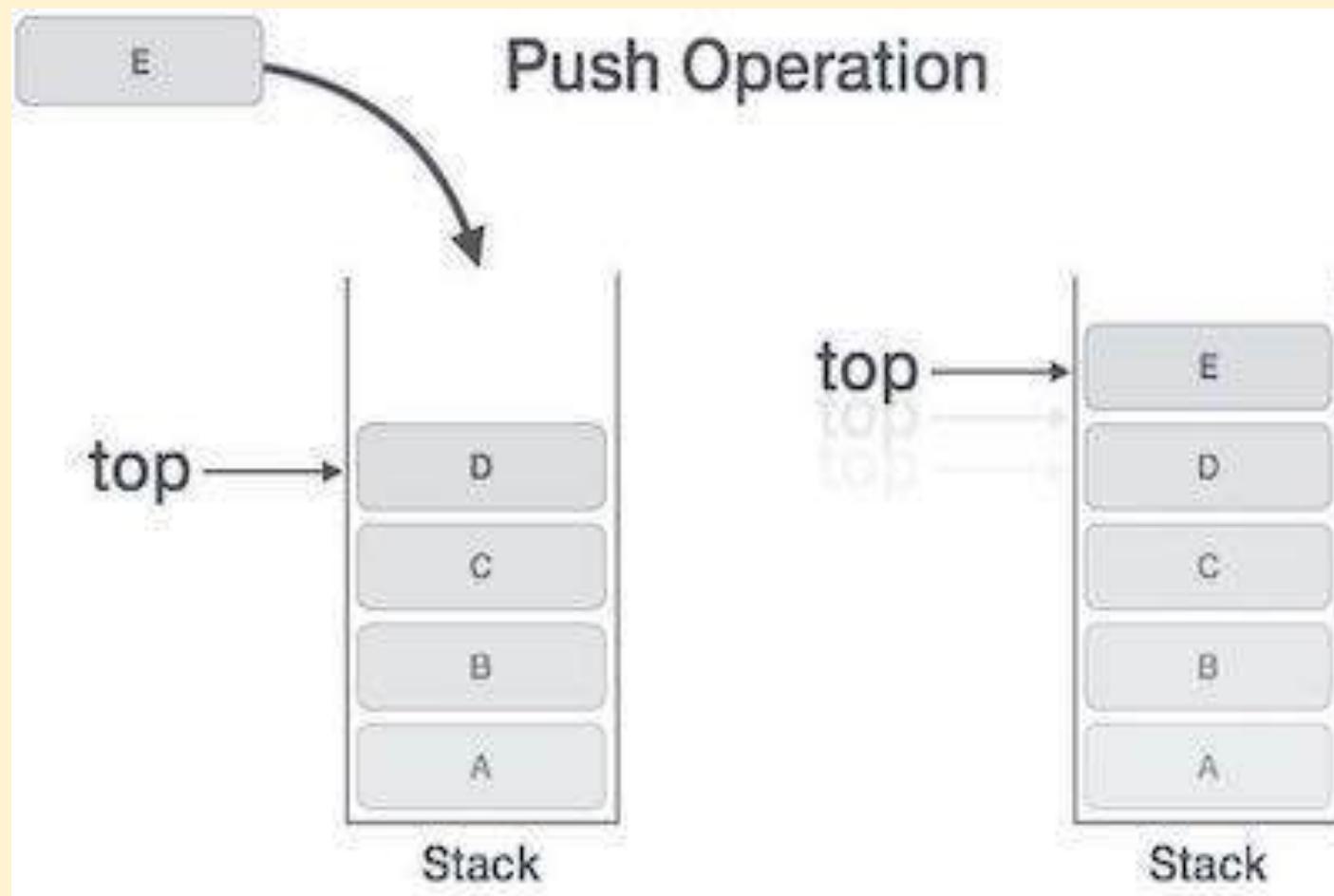


```
if (top == -1)
    printf("Stack is Empty");
```

# Basic Operations

---

- Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations –
- **push()** – Pushing (storing) an element on the stack.
- **pop()** – Removing (accessing) an element from the stack.





# Push Operation

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps –

**Step 1 – Check if the stack is full.**

**Step 2 – If the stack is full, produce an error and exit.**

**Step 3 – If the stack is not full, increment top to point next empty space.**

**Step 4 – Add data element to the stack location, where top is pointing.**

**Step 5 – Return success.**

If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.

# Algorithm for PUSH Operation

- A simple algorithm for Push operation can be derived as follows –

```
begin procedure push: stack, data
```

```
if stack is full
```

```
return null
```

```
endif
```

```
Else
```

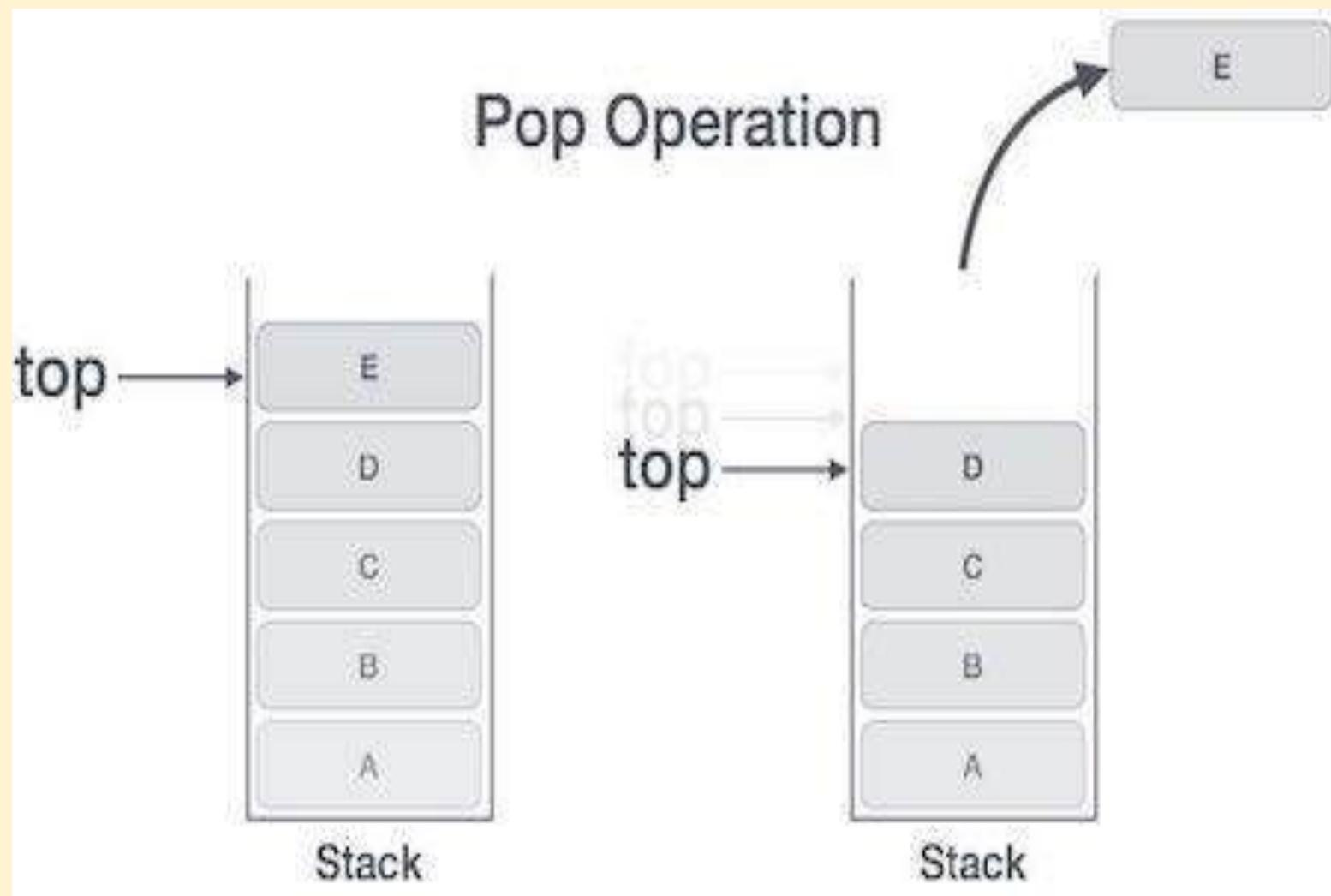
```
begin
```

```
    top ← top + 1
```

```
    stack[top] ← data
```

```
End else
```

```
end procedure
```



# Pop Operation

- Accessing the content while removing it from the stack, is known as a Pop Operation.
- In an array implementation of pop() operation, the data element is not actually removed, instead **top** is decremented to a lower position in the stack to point to the next value.
- But in linked-list implementation, pop() actually removes data element and deallocates memory space.

- A Pop operation may involve the following steps –

**Step 1** – Checks if the stack is empty.

**Step 2** – If the stack is empty, produces an error and exit.

**Step 3** – If the stack is not empty, accesses the data element at which **top** is pointing.

**Step 4** – Decreases the value of top by 1.

**Step 5** – Returns success.



# Additional Operations

To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks –

- **peek()** – get the top data element of the stack, without removing it.
- **isFull()** – check if stack is full.
- **isEmpty()** – check if stack is empty.

# isempty()

Algorithm isempty()

if top is 1

return true

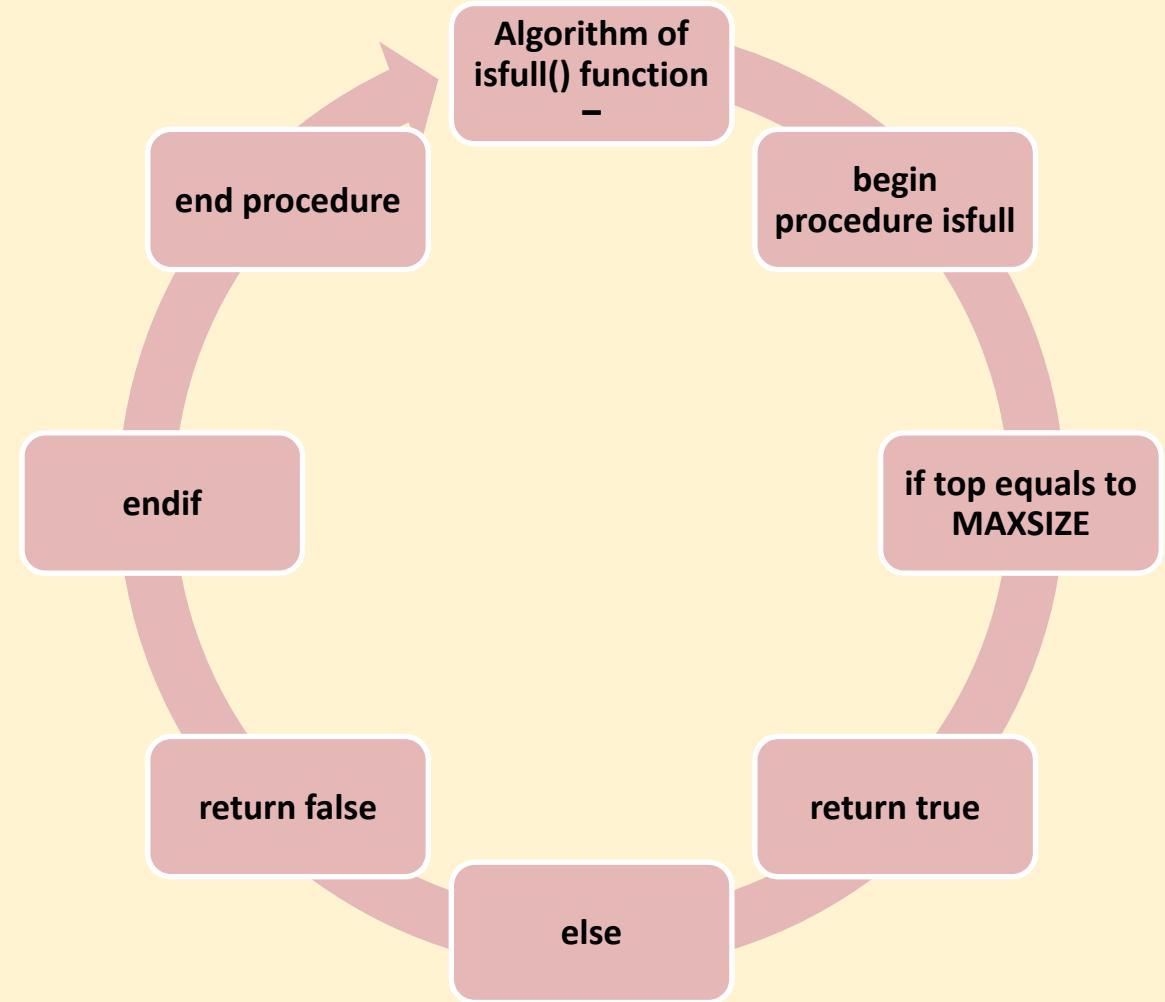
else

return false

endif

end procedure

# isfull()



# peek()



Algorithm of peek() function –



begin procedure peek



return stack[top]



end procedure

# PROGRAMS

1. Write a program to implement stack using arrays.

**\* Definition of Done:**

- a) The program should display a menu (1) Push (2) Pop (3) Peep (4) Exit.
- b) The program should define functions for the menu item listed above.



# ARRAY IMPLEMENTATION

```
boolean isEmpty()
```

```
{  
    return (top < 0);  
}
```

```
boolean push(int x)
```

```
{  
    if (top >= (MAX - 1))  
    {  
        System.out.println("Stack Overflow");  
        return false;  
    }  
    else {  
        a[++top] = x;  
        System.out.println(x + " pushed into stack");  
        return true;  
    } }
```

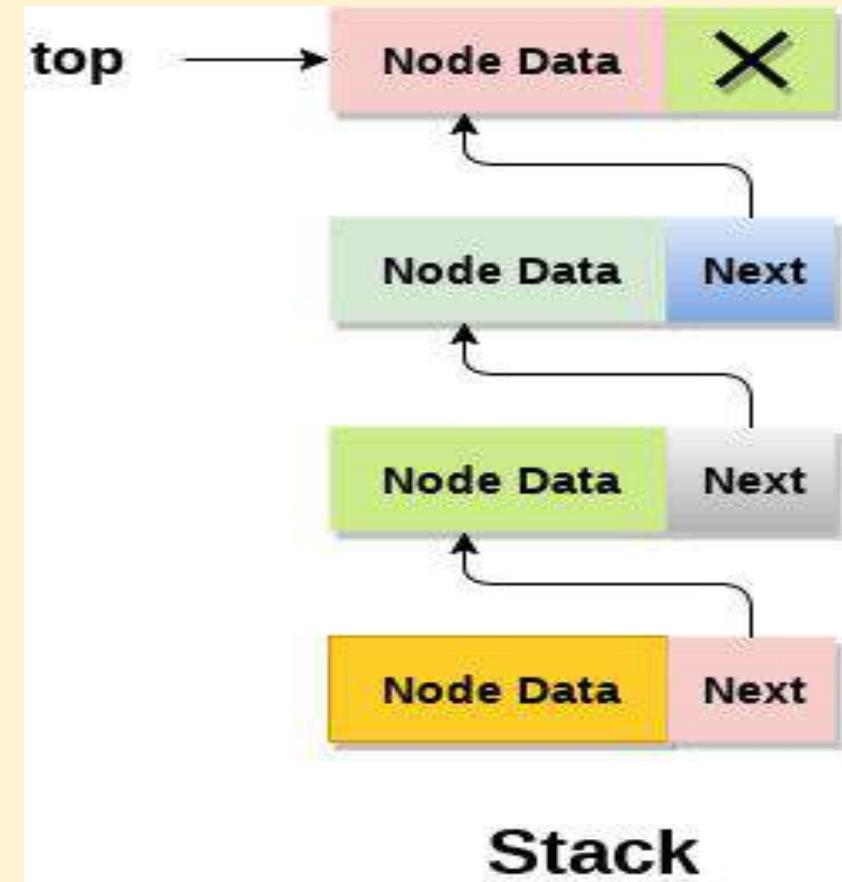
```
int pop()
```

```
{  
    if (top < 0) {  
        System.out.println("Stack Underflow");  
        return 0;  
    }  
    else {  
        int x = a[top--];  
        return x;  
    } }
```

```
int peek()
```

```
{  
    if (top < 0) {  
        System.out.println("Stack Underflow");  
        return 0;  
    }  
    else {  
        int x = a[top];  
        return x;  
    } }
```

# Stack implementation as linked list





# LINKED LIST IMPLEMENTATION

```
public void push(int data)
{
StackNode n;
n= new StackNode();
n.setData(data);
if (top == null)
{
    top = n;
}
else {
    StackNode temp = top;
    top = n;
    n.setNext(temp);
}
System.out.println(data + " pushed to stack");
}
```

```
public int pop()
{
int popped = Integer.MIN_VALUE;
if (top == null) {
    System.out.println("Stack is Empty");
}
else {
    popped = top.getData();
    top = top.getNext();
    size--;
}
return popped;
}
public int peek()
{
if (top == null) {
    System.out.println("Stack is empty");
    return Integer.MIN_VALUE;
}
else {
    return top.getData();
}
}
```



# LINKED LIST IMPLEMENTATION

```
class StackNode
{
    private int data;
    private StackNode next;
    public StackNode()
    {
        data=0;
        next=null;
    }
    StackNode(int d, StackNode n)
    {data=d;
    next=n;}
    public void setData(int d)
    {data=d;}
    public void setNext(StackNode n)
    {next=n;}
    public int getData()
    {return(data);}
    public StackNode getNext()
    {return(next);}
}
```

```
class StackLinkedList
{
    private int size;
    private StackNode top;
    public StackLinkedList()
    {int size=0;
    top=null;
    }

    public boolean isEmpty()
    {
        if (top==null)
        return(true);
        else
        return(false);
    }
}
```

```
public void push(int data)
{
    StackNode n;
    n= new StackNode();
    n.setData(data);
    if (top == null)
    {
        top = n;
    }
    else {
        StackNode temp = top;
        top = n;
        n.setNext(temp);
    }
    System.out.println(data + " pushed to stack");
}
```

```
public int pop()
{
    int popped = Integer.MIN_VALUE;
    if (top == null) {
        System.out.println("Stack is Empty");
    }
    else {
        popped = top.getData();
        top = top.getNext();
        size--;
    }
    return popped;
}
public int peek()
{
    if (top == null) {
        System.out.println("Stack is empty");
        return Integer.MIN_VALUE;
    }
    else {
        return top.getData();
    }
}
```



---

P P SAVANI  
U N I V E R S I T Y

# APPLICATIONS OF STACKS



# SCANNING ARITHMETIC EXPRESSIONS

- The way to write arithmetic expression is known as a **notation**.
- Arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression. These notations are –
- Infix Notation
- Prefix (Polish) Notation
- Postfix (Reverse-Polish) Notation

# Infix Notation

- We write expression in **infix** notation, e.g.  $a - b + c$ , where operators are used **in-between** operands.
- It is easy for us humans to read, write, and speak in infix notation but the same does not go well with computing devices.
- An algorithm to process infix notation could be difficult and costly in terms of time and space consumption.

# Prefix Notation

- In this notation, operator is **prefixed** to operands, i.e. operator is written ahead of operands.
- For example, **+ab**. This is equivalent to its infix notation **a + b**.
- Prefix notation is also known as **Polish Notation**.

# Postfix Notation

- This notation style is known as **Reversed Polish Notation**.
- In this notation style, the operator is **postfixed** to the operands i.e., the operator is written after the operands.
- For example, **ab+**. This is equivalent to its infix notation **a + b**.



# Table depicting all Notations

S.no	Infix	Prefix	Postfix
1	$a + b$	$+ a b$	$a b +$
2	$(a + b) * c$	$* + a b c$	$a b + c *$
3	$a * (b + c)$	$* a + b c$	$a b c + *$
4	$a / b + c / d$	$+ / a b / c d$	$a b / c d / +$
5	$(a + b) * (c + d)$	$* + a b + c d$	$a b + c d + *$
6	$((a + b) * c) - d$	$- * + a b c d$	$a b + c * d -$



# Parsing Expressions

To parse any arithmetic expression, we need to take care of –

- Operator precedence
- Associativity

# Precedence

- When an operand is in between two different operators, which operator will take the operand first, is decided by the precedence of an operator over others.
- For example –As multiplication operation has precedence over addition,  $b * c$  will be evaluated first. A table of operator precedence is provided later.

# Associativity

- Associativity describes the rule where operators with the same precedence appear in an expression.
- For example, in expression  $a + b - c$ , both  $+$  and  $-$  have the same precedence, then which part of the expression will be evaluated first, is determined by associativity of those operators.
- Here, both  $+$  and  $-$  are left associative, so the expression will be evaluated as  $(a + b) - c$ .



# Operator precedence & associativity table

S. NO.	OPERATOR	PRECEDENCE	ASSOCIATIVITY
1	Exponentiation ^	Highest	Right Associative
2	Multiplication ( * ) & Division ( / )	Second Highest	Left Associative
3	Addition ( + ) & Subtraction ( - )	Lowest	Left Associative



# Conversion of expressions

- INFIX TO POSTFIX
- INFIX TO PREFIX
- PREFIX TO INFIX
- PREFIX TO POSTFIX
- POSTFIX TO PREFIX
- POSTFIX TO INFIX



1. Scan the infix expression from left to right.

2. If the scanned character is an operand, output it.

3. Else,

.....3.1 If the precedence of the scanned operator is greater than the precedence of the operator in the stack(or the stack is empty or the stack contains a '(', push it.

.....3.2 Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator.

After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)

4. If the scanned character is an '(', push it to the stack.

5. If the scanned character is an ')', pop the stack and and output it until a '(' is encountered, and discard both the parenthesis.

6. Repeat steps 2-6 until infix expression is scanned.

7. Print the output

8. Pop and output from the stack until it is not empty



# INFIX TO POSTFIX

Infix Expression : A+B*(C^D-E)				
Token	Action	Result	Stack	Notes
A	Add A to the result	A		
+	Push + to stack	A	+	
B	Add B to the result	AB	+	
*	Push * to stack	AB	* +	* has higher precedence than +
(	Push ( to stack	AB	( * +	
C	Add C to the result	ABC	( * +	
^	Push ^ to stack	ABC	^ ( * +	
D	Add D to the result	ABCD	^ ( * +	
-	Pop ^ from stack and add to result	ABCD^	( * +	- has lower precedence than ^
	Push - to stack	ABCD^	- ( * +	
E	Add E to the result	ABCD^E	- ( * +	
)	Pop - from stack and add to result	ABCD^E-	( * +	Do process until ( is popped from stack
	Pop ( from stack	ABCD^E-	* +	
	Pop * from stack and add to result	ABCD^E-*	+	Given expression is iterated, do Process till stack is not Empty, It will give the final result
	Pop + from stack and add to result	ABCD^E-*+		
Postfix Expression : ABCD^E-*+				



# INFIX TO POSTFIX

- Convert following infix expressions into postfix expressions
  - $(a-b)^*(d/e)$
  - $(a+b^d)/(e-f)+g$
  - $A^*(B+D)/E-F^*(G+H/K)$



# INFIX TO PREFIX

To convert an infix to postfix we use the same to convert Infix to Prefix.

- Step 1: Reverse the infix expression i.e  $A+B*C$  will become  $C*B+A$ . Note while reversing each '(' will become ')' and each ')' becomes '('.
- Step 2: Obtain the postfix expression of the modified expression i.e  $CB*A+$ .
- Step 3: Reverse the postfix expression. Hence in our example prefix is  $+A*BC$ .



# INFIX TO PREFIX

Infix Expression : A+B*(C^D-E)				
Reverse Infix expression: )E-D^C(*B+A				
Reverse brackets: (E-D^C)*B+A				
Token	Action	Result	Stack	Notes
(	Push ( to stack		(	
E	Add E to the result	E	(	
-	Push - to stack	E	( -	
D	Add D to the result	ED	( -	
^	Push ^ to stack	ED	( - ^	
C	Add C to the result	EDC	( - ^	
)	Pop ^ from stack and add to result	EDC^	( -	Do process until ( is popped from stack
	Pop - from stack and add to result	EDC^-	(	
	Pop ( from stack	EDC^-		
*	Push * to stack	EDC^-	*	
B	Add B to the result	EDC^-B	*	
+	Pop * from stack and add to result	EDC^-B		- has lower precedence than ^
	Push + to stack	EDC^-B*	+	
A	Add A to the result	EDC^-B*A	+	
	Pop + from stack and add to result	EDC^-B*A+		Given expression is iterated, do Process till stack is not Empty, It will give the final result
Prefix Expression (Reverse Result): +A*B^-CDE				



# INFIX TO PREFIX

- Convert following infix expressions into prefix expressions
  - $(a-b)^*(d/e)$
  - $(a+b^d)/(e-f)+g$
  - $A^*(B+D)/E-F^*(G+H/K)$

# Applications of Stack

- **Expression Evaluation**

Stack is used to evaluate prefix, postfix and infix expressions.

- **Expression Conversion**

An expression can be represented in prefix, postfix or infix notation. Stack can be used to convert one form of expression to another.

- **Syntax Parsing**

Many compilers use a stack for parsing the syntax of expressions, program blocks etc. before translating into low level code.



- **Backtracking**

Suppose we are finding a path for solving maze problem. We choose a path and after following it we realize that it is wrong. Now we need to go back to the beginning of the path to start with new path. This can be done with the help of stack.

- **Parenthesis Checking**

Stack is used to check the proper opening and closing of parenthesis.

- **String Reversal**

Stack is used to reverse a string. We push the characters of string one by one into stack and then pop character from stack.

- **Function Call**

Stack is used to keep information about the active functions or subroutines.



# STACKS OPERATIONS AND APPLICATIONS

# PROGRAM-1

1. Write a program to implement stack using arrays.
  - **Definition of Done:**
    - a) The program should display a menu (1) Push (2) Pop (3) Peep (4) Exit.
    - b) The program should define functions for the menu item listed above.



# ARRAY IMPLEMENTATION

```
boolean isEmpty()
{
    return (top < 0);
}

boolean push(int x)
{
    if (top >= (MAX - 1))
    {
        System.out.println("Stack Overflow");
        return false;
    }
    else {
        a[++top] = x;
        System.out.println(x + " pushed into stack");
        return true;
    }
}
```

```
int pop()
{
    if (top < 0) {
        System.out.println("Stack Underflow");
        return 0;
    }
    else {
        int x = a[top--];
        return x;
    }
}

int peek()
{
    if (top < 0) {
        System.out.println("Stack Underflow");
        return 0;
    }
    else {
        int x = a[top];
        return x;
    }
}
```



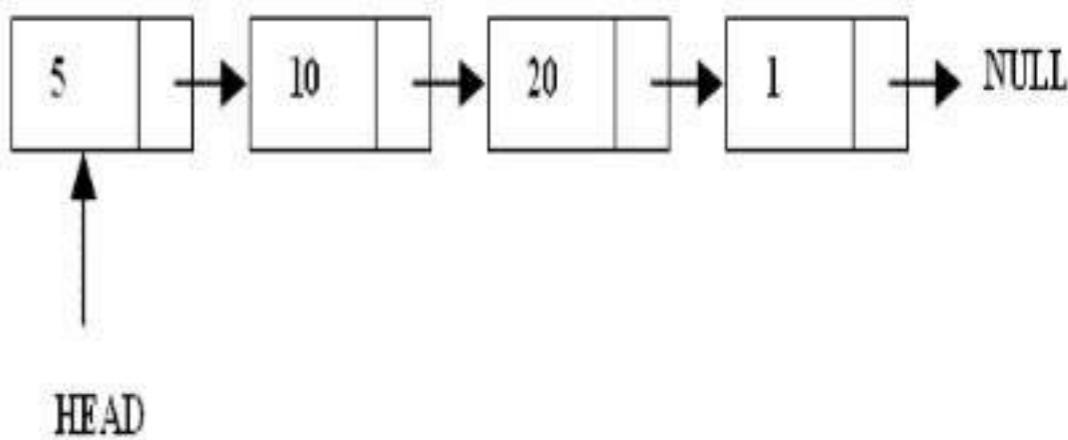
---

PP SAVANI  
UNIVERSITY

ALL THE BEST!!

# What are Linked Lists

- A linked list is a linear data structure.
- Nodes make up linked lists.
- Nodes are structures made up of data and a pointer to another node.
- Usually the pointer is called next.



# Arrays Vs Linked Lists

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

# Types of lists

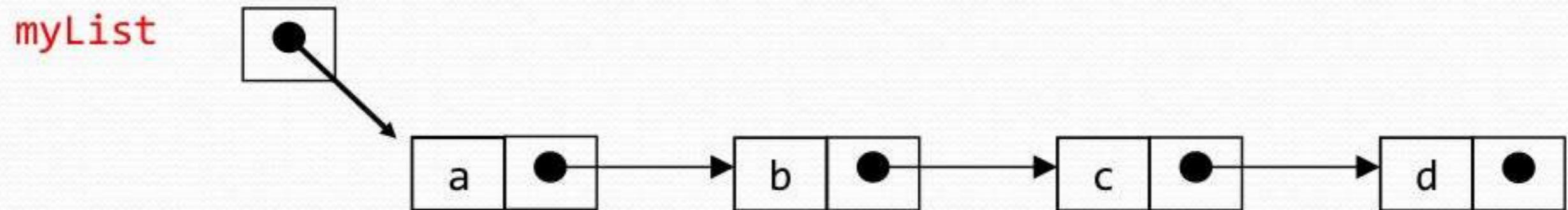
- There are two basic types of linked list
- Singly Linked list
- Doubly linked list

# Singly Linked List

- Each node has only one link part
- Each link part contains the address of the next node in the list
- Link part of the last node contains NULL value which signifies the end of the node

# Schematic representation

- Here is a singly-linked list (SLL):



- Each node contains a value(data) and a pointer to the next node in the list
- **myList** is the header pointer which points at the first node in the list

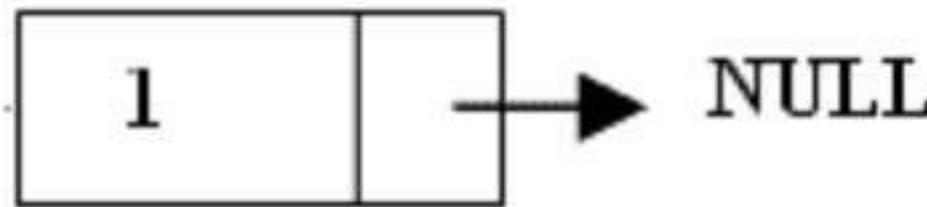
# Basic Operations on a list

- Creating a List
- Inserting an element in a list
- Deleting an element from a list
- Searching a list
- Reversing a list

# Creating a node

```
struct node{  
    int data;           // A simple node of a linked list  
    node*next;  
}*start;           //start points at the first node  
start=NULL ;      initialised to NULL at beginning
```

```
node* create( int num) //say num=1 is passed from main
{
    node*ptr;
    ptr= new node; //memory allocated dynamically
    if(ptr==NULL)
        'OVERFLOW' // no memory available
        exit(1);
    else
    {
        ptr->data=num;
        ptr->next=NULL;
        return ptr;
    }
}
```



## To be called from main() as:-

```
void main()
{
    node* ptr;
    int data;
    cin>>data;
    ptr=create(data);
}
```

# Inserting the node in a SLL

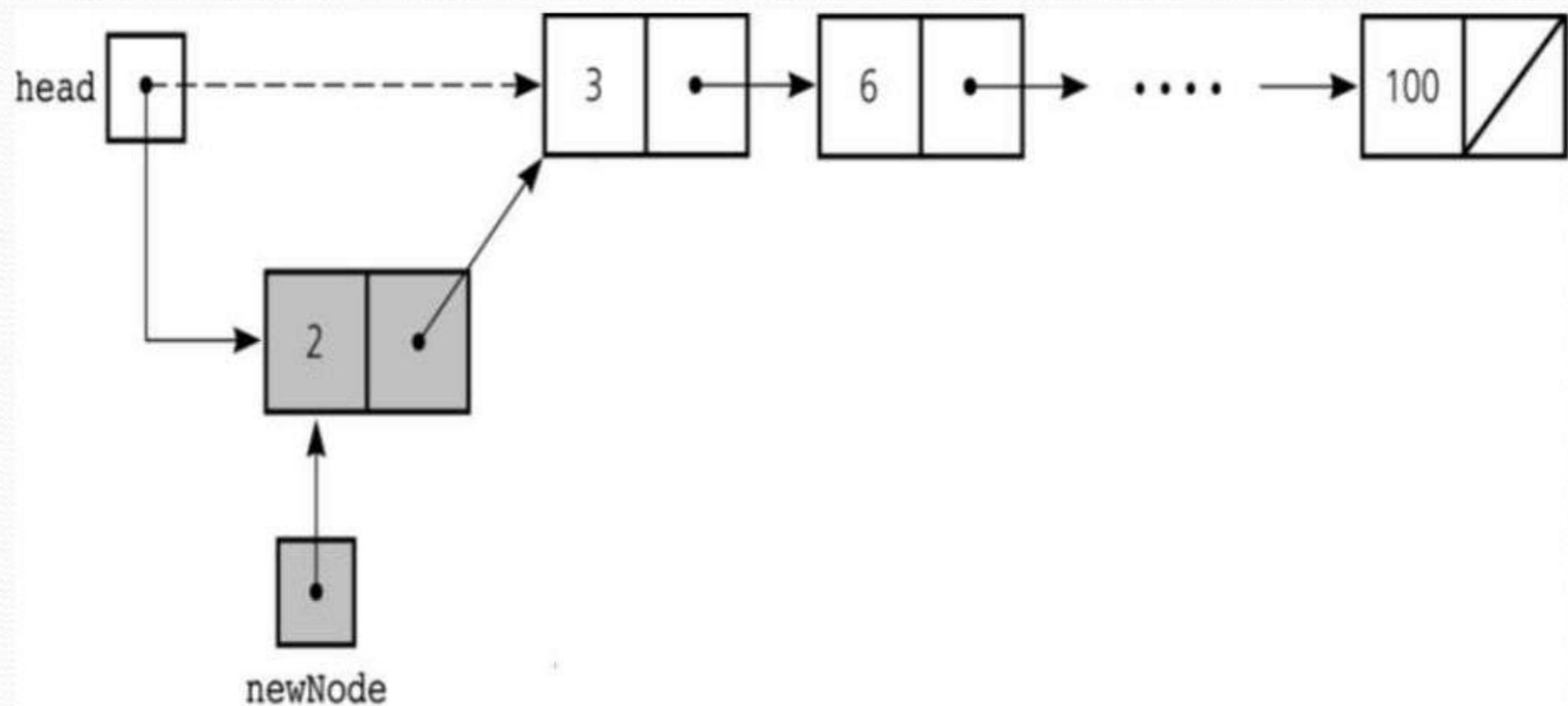
There are 3 cases here:-

- Insertion at the beginning
- Insertion at the end
- Insertion after a particular node

# Insertion at the beginning

There are two steps to be followed:-

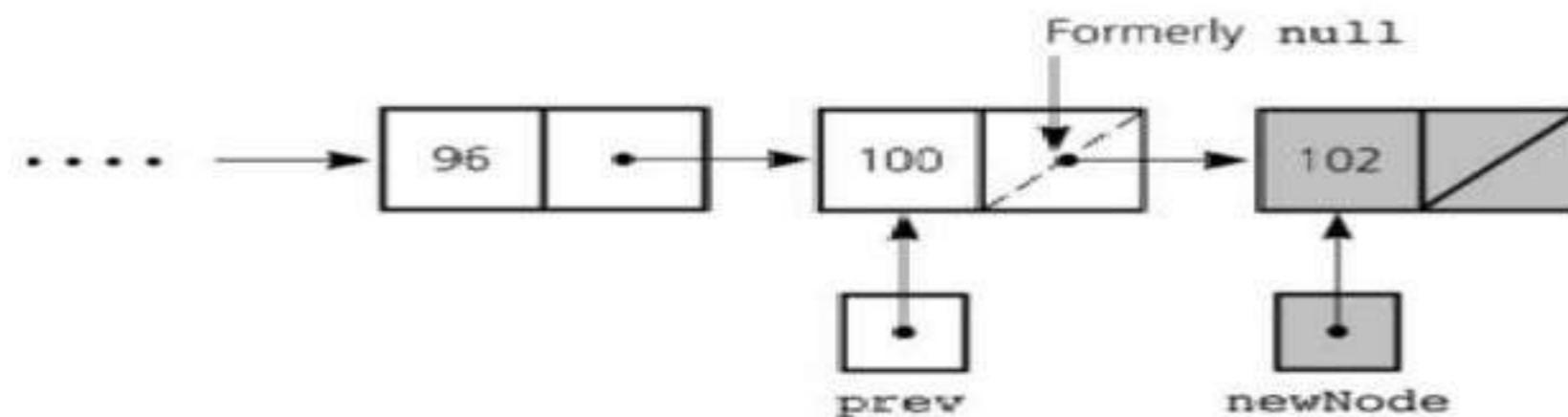
- a) Make the next pointer of the node point towards the first node of the list
- b) Make the start pointer point towards this new node
- If the list is empty simply make the start pointer point towards the new node;



```
void insert_beg(node* p)
{
node* temp;
    if(start==NULL) //if the list is empty
    {
        start=p;
        cout<<"\nNode inserted successfully at the
beginning";
    }
else {
    temp=start;
    start=p;
    p->next=temp; //making new node point at
}
the first node of the list
}
```

# Inserting at the end

Here we simply need to make the next pointer of the last node point to the new node

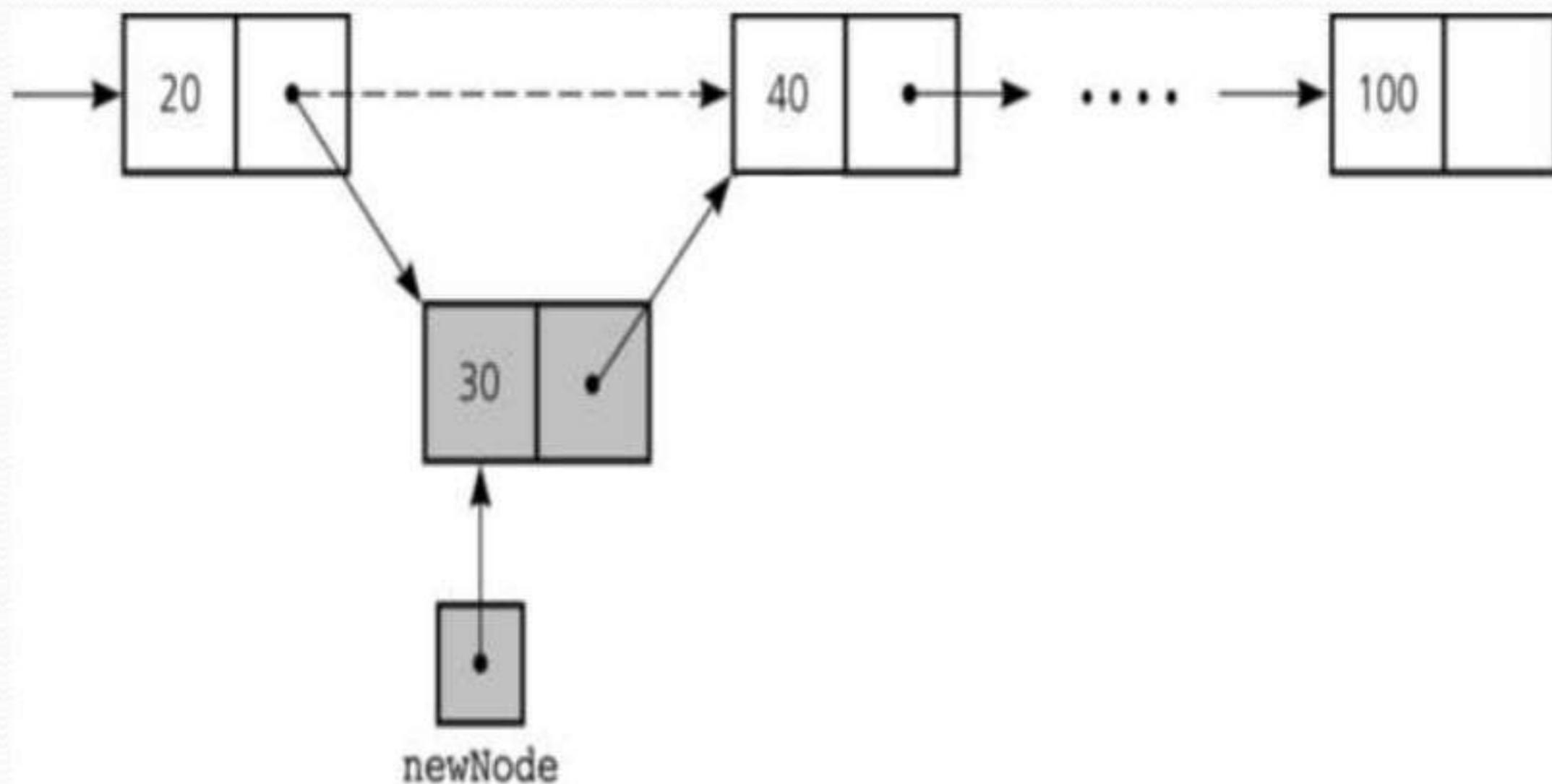


```
void insert_end(node* p)
{
node *q=start;
    if(start==NULL)
    {
        start=p;
        cout<<"\nNode inserted successfully at the end...!!!\n";
    }
else{
    while(q->link!=NULL)
        q=q->link;
    q->next=p;
}
}
```

# Inserting after an element

Here we again need to do 2 steps :-

- Make the next pointer of the node to be inserted point to the next node of the node after which you want to insert the node
- Make the next pointer of the node after which the node is to be inserted, point to the node to be inserted



```
void insert_after(int c,node* p)
{
node* q;
q=start;
    for(int i=1;i<c;i++)
    {
        q=q->link;
        if(q==NULL)
            cout<<"Less than "<<c<<" nodes in the list...!!!";
    }
p->link=q->link;
q->link=p;
cout<<"\nNode inserted successfully";
}
```

# Deleting a node in SLL

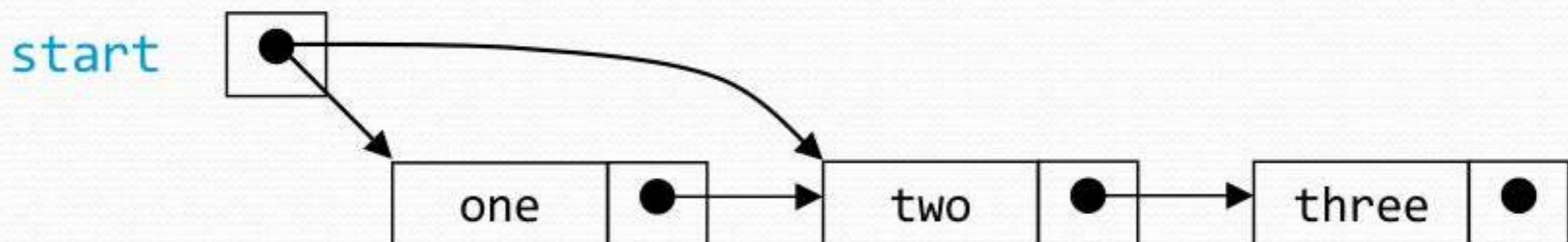
Here also we have three cases:-

- Deleting the first node
- Deleting the last node
- Deleting the intermediate node

# Deleting the first node

Here we apply 2 steps:-

- Making the start pointer point towards the 2<sup>nd</sup> node
- Deleting the first node using **delete** keyword

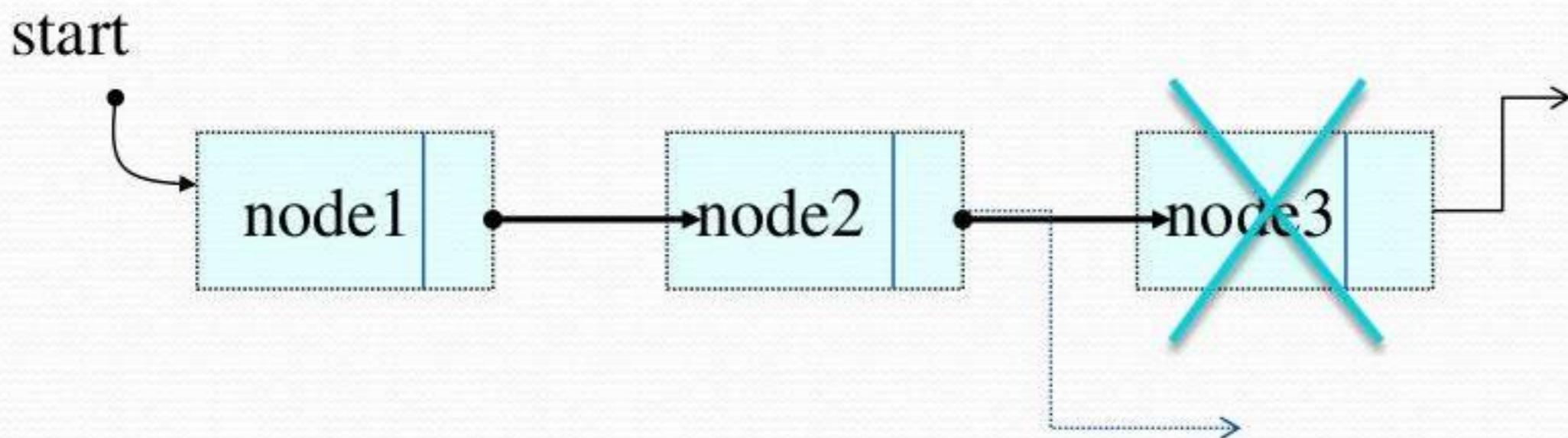


```
void del_first()
{
    if(start==NULL)
        cout<<"\nError.....List is empty\n";
    else
    {
        node* temp=start;
        start=temp->link;
        delete temp;
        cout<<"\nFirst node deleted successfully....!!!";
    }
}
```

# Deleting the last node

Here we apply 2 steps:-

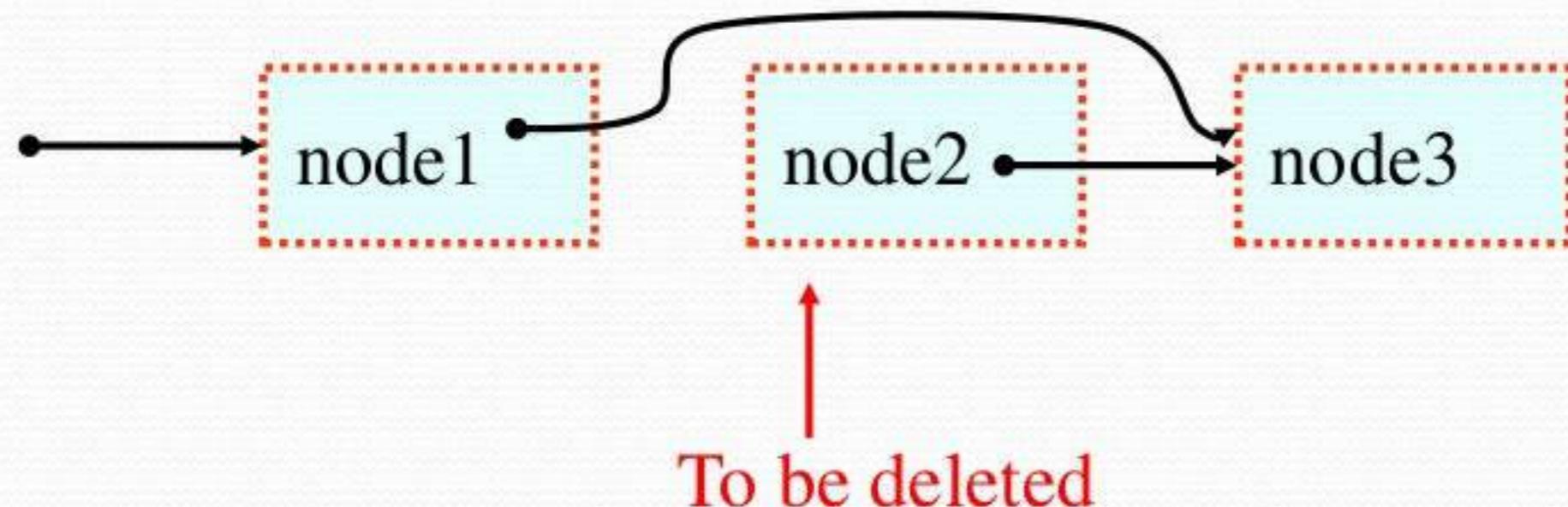
- Making the second last node's next pointer point to NULL
- Deleting the last node via `delete` keyword



```
void del_last()
{
    if(start==NULL)
        cout<<"\nError....List is empty";
    else
    {
        node* q=start;
        while(q->link->link!=NULL)
            q=q->link;
        node* temp=q->link;
        q->link=NULL;
        delete temp;
        cout<<"\nDeleted successfully...";
    }
}
```

# Deleting a particular node

Here we make the next pointer of the node previous to the node being deleted ,point to the successor node of the node to be deleted and then delete the node using **delete** keyword



```
void del(int c)
{
node* q=start;
    for(int i=2;i<c;i++)
    {
        q=q->link;
        if(q==NULL)
            cout<<"\nNode not found\n";
    }
if(i==c)
{
    node* p=q->link;      //node to be deleted
    q->link=p->link;      //disconnecting the node p
    delete p;
    cout<<"Deleted Successfully";
}
}
```

# Searching a SLL

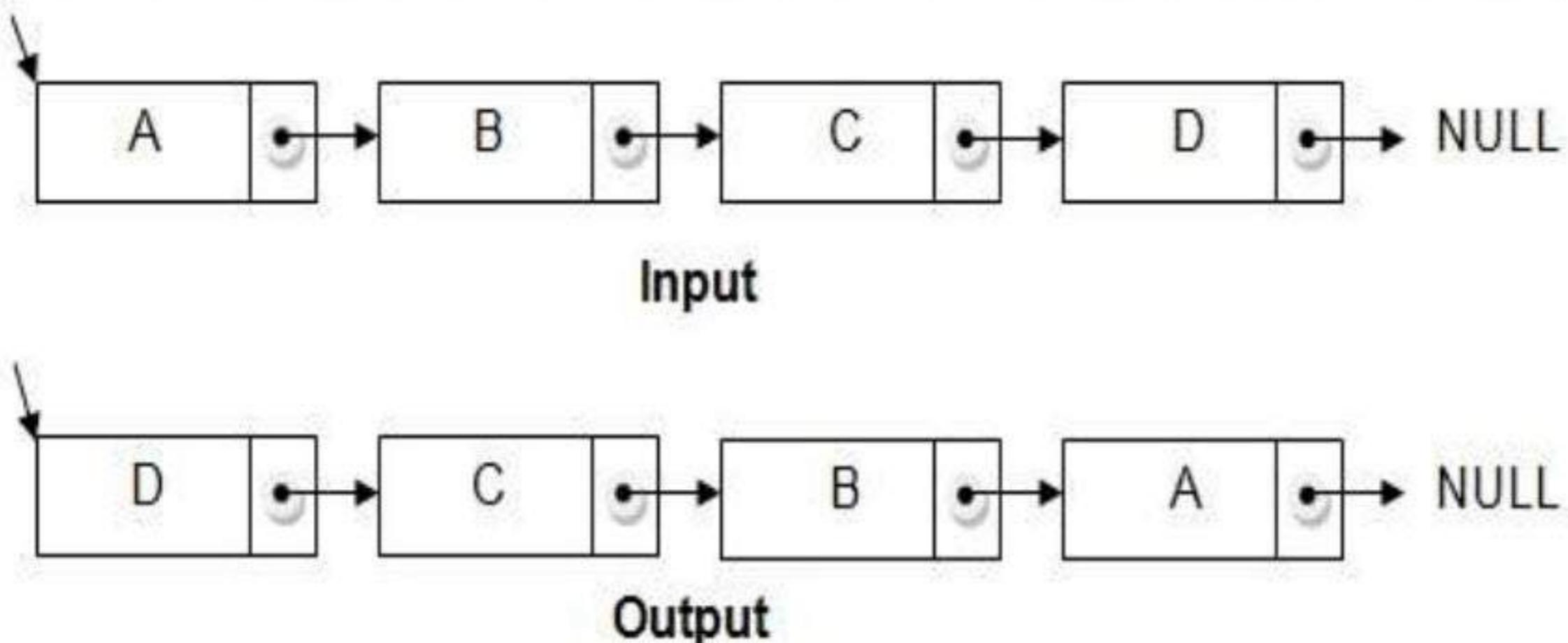
- Searching involves finding the required element in the list
- We can use various techniques of searching like linear search or binary search where binary search is more efficient in case of Arrays
- But in case of linked list since random access is not available it would become complex to do binary search in it
- We can perform simple linear search traversal

In linear search each node is traversed till the data in the node matches with the required value

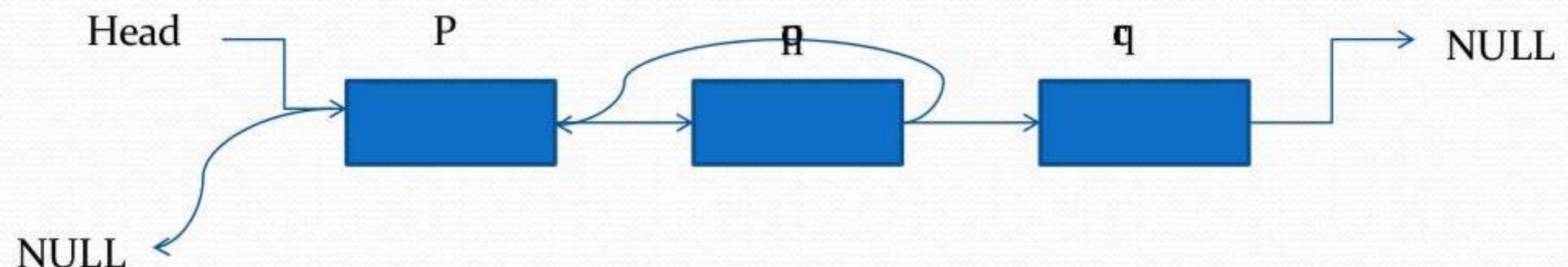
```
void search(int x)
{
    node *temp=start;
    while(temp!=NULL)
    {
        if(temp->data==x)
        {
            cout<<"FOUND "<<temp->data;
            break;
        }
        temp=temp->next;
    }
}
```

# Reversing a linked list

- We can reverse a linked list by reversing the direction of the links between 2 nodes



- We make use of 3 structure pointers say p,q,r
- At any instant q will point to the node next to p and r will point to the node next to q



- For next iteration  $p=q$  and  $q=r$
- At the end we will change head to the last node

# Code

```
void reverse()
{
node*p,*q,*r;
if(start==NULL)
{
cout<<"\nList is empty\n";
return;
}
p=start;
q=p->link;
p->link=NULL;
while(q!=NULL)
{
r=q->link;
q->link=p;
p=q;
q=r;
}
start=p;
cout<<"\nReversed successfully";
}
```

# COMPLEXITY OF VARIOUS OPERATIONS IN ARRAYS AND SLL

Operation	ID-Array Complexity	Singly-linked list Complexity
Insert at beginning	$O(n)$	$O(1)$
Insert at end	$O(1)$	$O(1)$ if the list has <b>tail</b> reference $O(n)$ if the list has no <b>tail</b> reference
Insert at middle	$O(n)$	$O(n)$
Delete at beginning	$O(n)$	$O(1)$
Delete at end	$O(1)$	$O(n)$
Delete at middle	$O(n)$ : $O(1)$ access followed by $O(n)$ shift	$O(n)$ : $O(n)$ search, followed by $O(1)$ delete
Search	$O(n)$ linear search $O(\log n)$ Binary search	$O(n)$
Indexing: What is the element at a given position $k$ ?	$O(1)$	$O(n)$

# Doubly Linked List

1. **Doubly linked list** is a linked data structure that consists of a set of sequentially linked records called nodes.
- 2 . Each node contains three fields ::
  - : one is data part which contain data only.
  - :two other field is links part that are point or references to the previous or to the next node in the sequence of nodes.
3. The beginning and ending nodes' **previous** and **next** links, respectively, point to some kind of terminator, typically a sentinel node or null to facilitate traversal of the list.

## NODE

previous

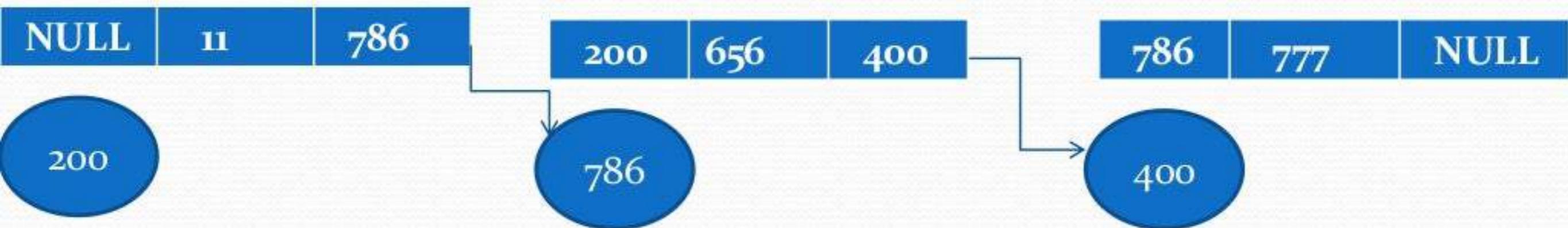
data

next

A

B

C



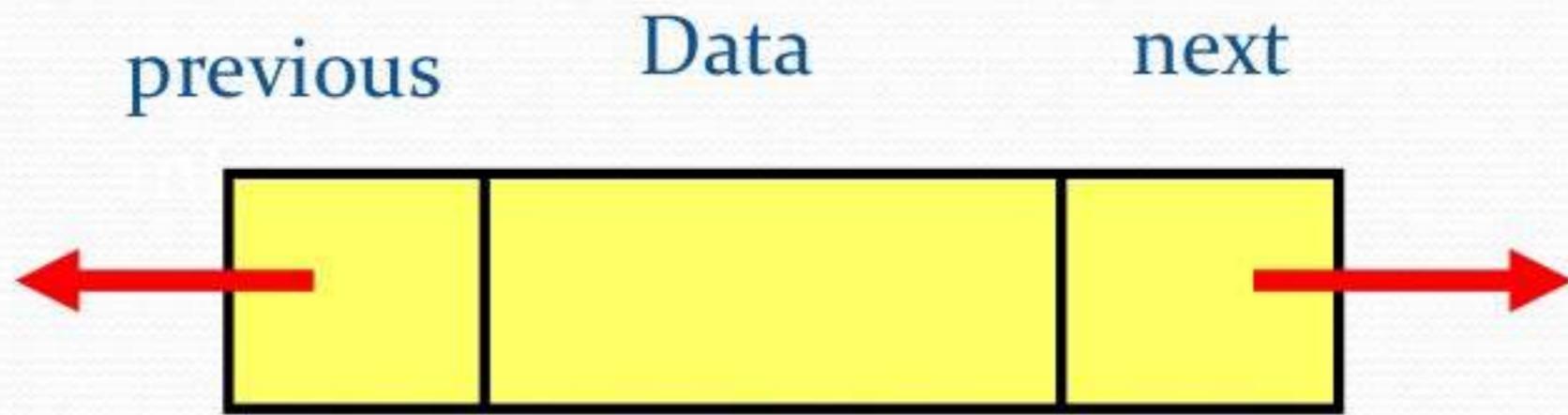
A doubly linked list contain three fields: an integer value, the link to the next node, and the link to the previous node.

# DLL's compared to SLL's

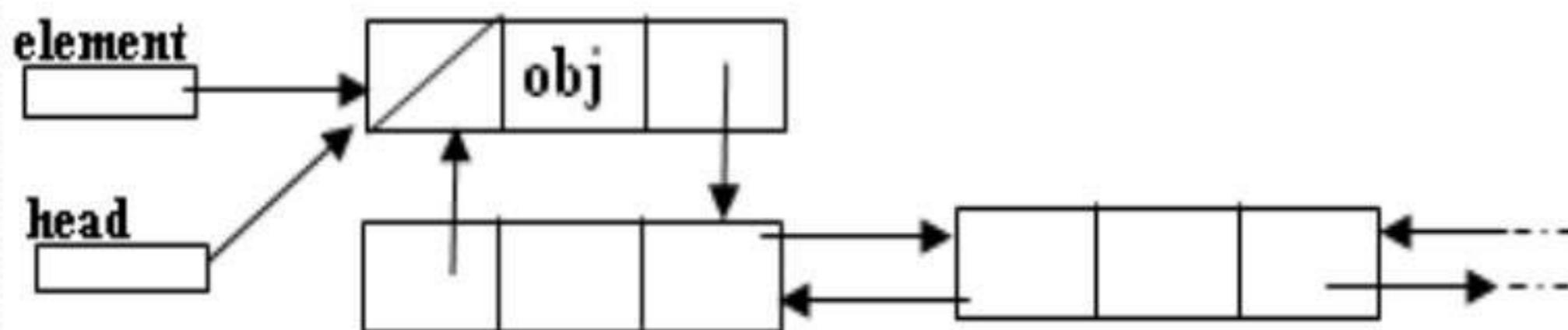
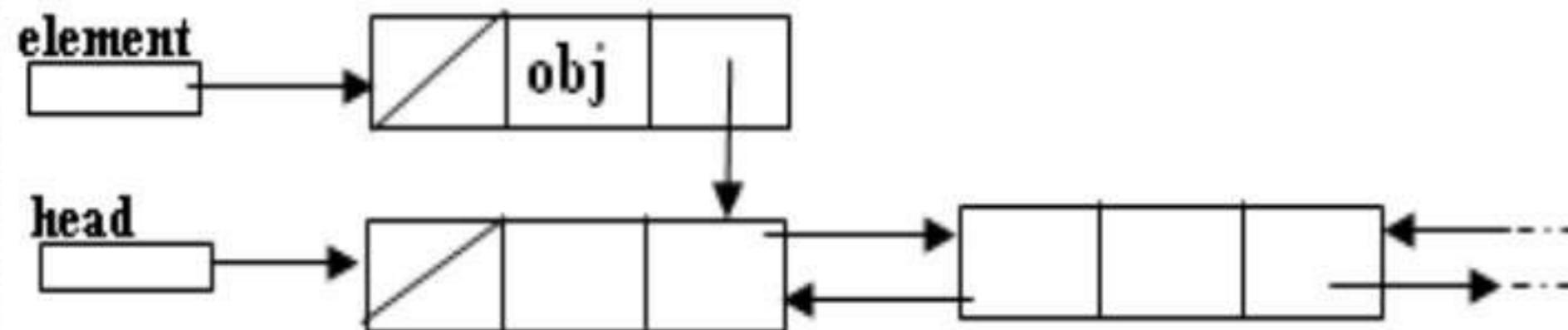
- Advantages:
  - Can be traversed in either direction (may be essential for some programs)
  - Some operations, such as deletion and inserting before a node, become easier
- Disadvantages:
  - Requires more space
  - List manipulations are slower (because more links must be changed)
  - Greater chance of having bugs (because more links must be manipulated)

# Structure of DLL

```
struct node
{
    int data;
    node*next;
    node*previous; //holds the address of previous node
};
```

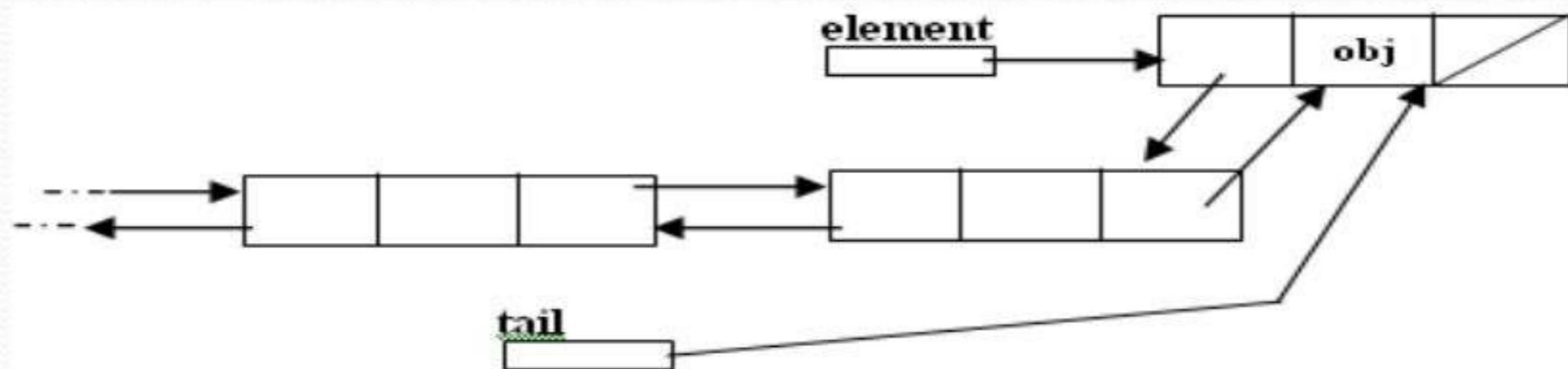
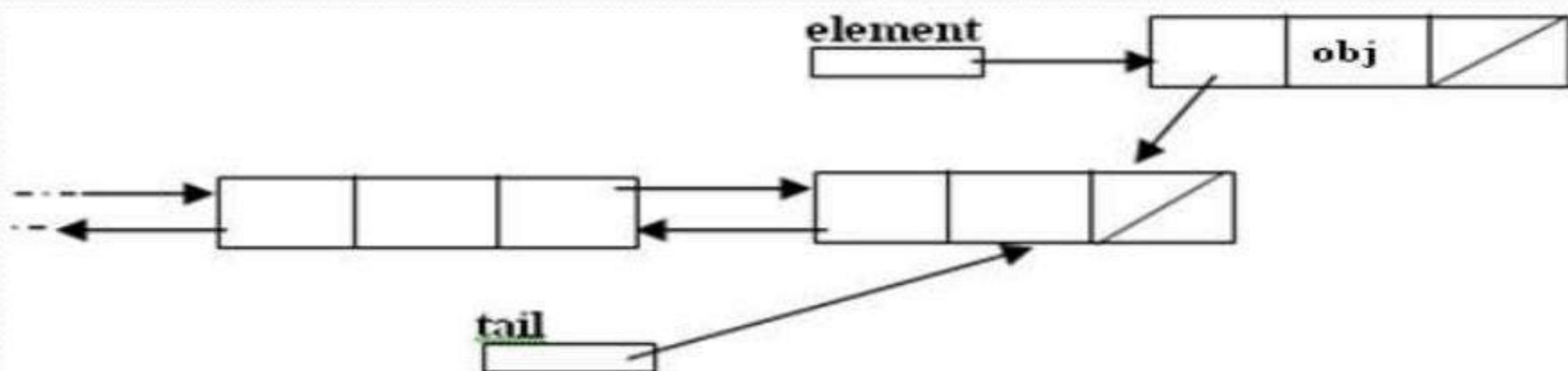


# Inserting at beginning



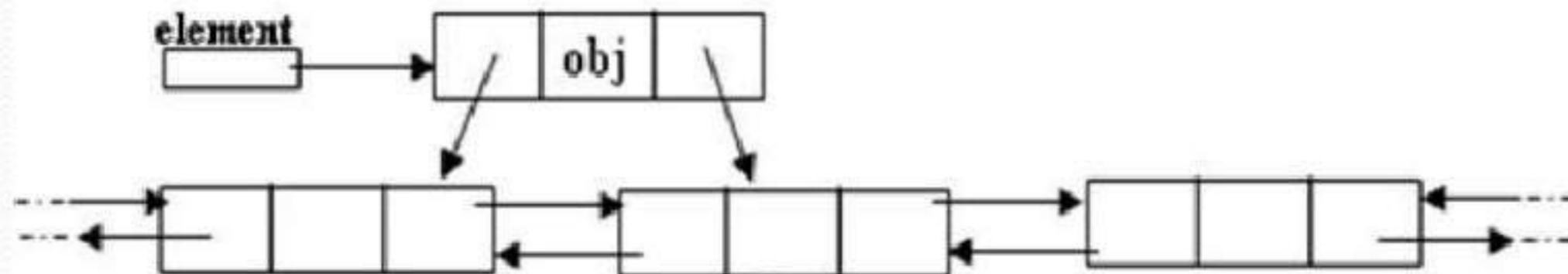
```
void insert_beg(node *p)
{
    if(start==NULL)
    {
        start=p;
        cout<<"\nNode inserted successfully at the beginning\m";
    }
    else
    {
        node* temp=start;
        start=p;
        temp->previous=p; //making 1st node's previous point to the
                           //new node
        p->next=temp;     //making next of the new node point to the
                           //1st node
        cout<<"\nNode inserted successfully at the beginning\n";
    }
}
```

# Inserting at the end

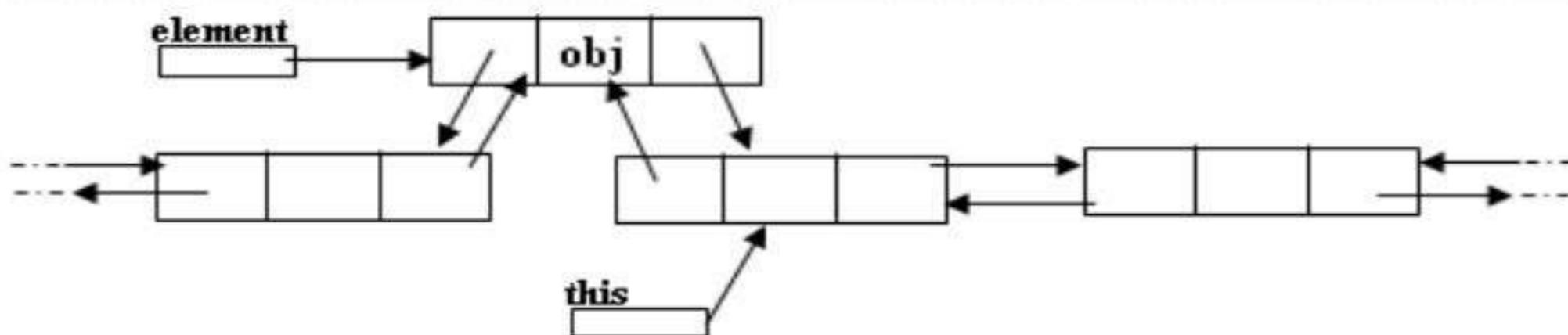


```
void insert_end(node* p)
{
    if(start==NULL)
    {
        start=p;
        cout<<"\nNode inserted successfully at the end";
    }
    else
    {
        node* temp=start;
        while(temp->next!=NULL)
        {
            temp=temp->next;
        }
        temp->next=p;
        p->previous=temp;
        cout<<"\nNode inserted successfully at the end\n";
    }
}
```

# Inserting after a node



Making next and previous pointer of the node to be inserted point accordingly

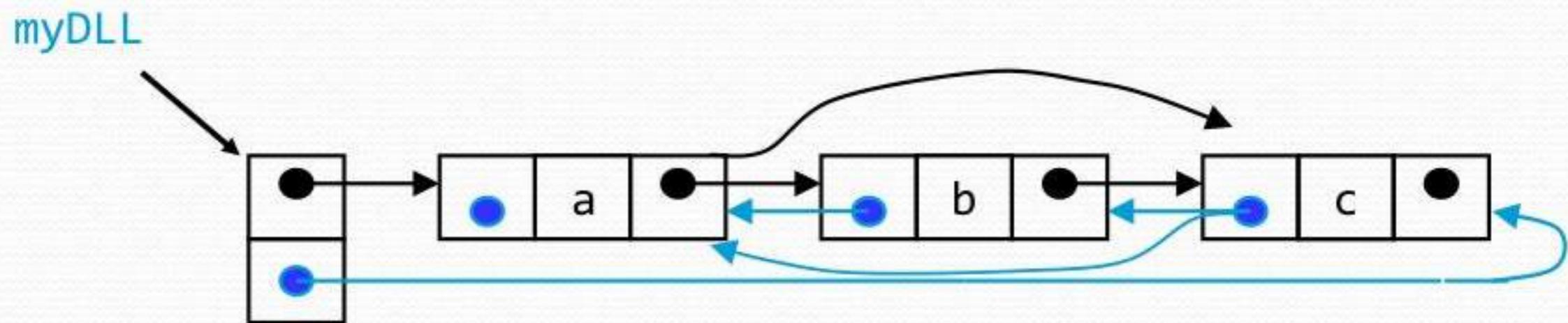


Adjusting the next and previous pointers of the nodes b/w which the new node accordingly

```
void insert_after(int c,node* p)
{
    temp=start;
    for(int i=1;i<c-1;i++)
    {
        temp=temp->next;
    }
    p->next=temp->next;
    temp->next->previous=p;
    temp->next=p;
    p->previous=temp;
    cout<<"\nInserted successfully";
}
```

# Deleting a node

- Node deletion from a DLL involves changing *two* links
- In this example, we will delete node b



- We don't have to do anything about the links in node b
- Garbage collection will take care of deleted nodes
- Deletion of the first node or the last node is a special case

```
void del_at(int c)
{
    node*s=start;
    {
        for(int i=1;i<c-1;i++)
        {
            s=s->next;
        }
        node* p=s->next;
        s->next=p->next;
        p->next->previous=s;
        delete p;
        cout<<"\nNode number "<<c<<" deleted successfully";
    }
}
```

# APPLICATIONS OF LINKED LIST

1. Applications that have an MRU list (a linked list of file names)
2. The cache in your browser that allows you to hit the BACK button (a linked list of URLs)
3. Undo functionality in Photoshop or Word (a linked list of state)
4. A stack, hash table, and binary tree can be implemented using a doubly linked list.

# Circular Linked List

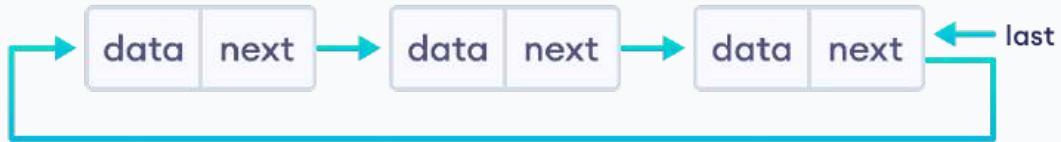
In this article, you will learn what circular linked list is and its types with implementation.

A circular linked list is a type of [linked list](#) in which the first and the last nodes are also connected to each other to form a circle.

There are basically two types of circular linked list:

## 1. Circular Singly Linked List

Here, the address of the last node consists of the address of the first node.



Circular Linked List Representation

## 2. Circular Doubly Linked List

Here, in addition to the last node storing the address of the first node, the first node will also store the address of the last node.

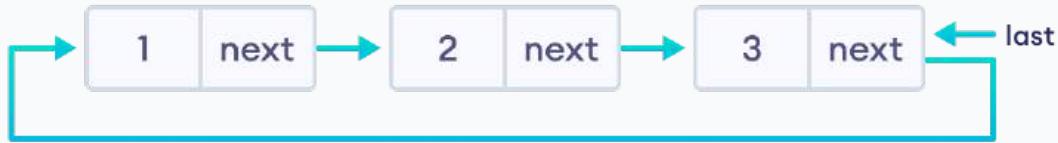


Circular Doubly Linked List Representation

**Note:** We will be using the singly circular linked list to represent the working of circular linked list.

## Representation of Circular Linked List

Let's see how we can represent a circular linked list on an algorithm/code.  
Suppose we have a linked list:



Initial circular linked list

Here, the single node is represented as

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

Each struct node has a data item and a pointer to the next struct node.

Now we will create a simple circular linked list with three items to understand how this works.

```
/* Initialize nodes */  
struct node *last;  
struct node *one = NULL;  
struct node *two = NULL;  
struct node *three = NULL;  
  
/* Allocate memory */  
one = malloc(sizeof(struct node));  
two = malloc(sizeof(struct node));  
three = malloc(sizeof(struct node));  
  
/* Assign data values */  
one->data = 1;  
two->data = 2;  
three->data = 3;  
  
/* Connect nodes */  
one->next = two;  
two->next = three;  
three->next = one;
```

```
/* Save address of third node in last */  
last = three;
```

In the above code, one, two, and three are the nodes with data items 1, 2, and 3 respectively.

#### For node one

- next stores the address of two (there is no node before it)

#### For node two

- next stores the address of three

#### For node three

- next stores NULL (there is no node after it)
- next points to node one

---

## Insertion on a Circular Linked List

We can insert elements at 3 different positions of a circular linked list:

1. [Insertion at the beginning](#)
2. [Insertion in-between nodes](#)
3. [Insertion at the end](#)

Suppose we have a circular linked list with elements 1, 2, and 3.



Initial circular linked list

Let's add a node with value 6 at different positions of the circular linked list we made above. The first step is to create a new node.

- allocate memory for newNode
- assign the data to newNode



New Node

New node

---

## 1. Insertion at the Beginning

- store the address of the current first node in the `newNode` (i.e. pointing the `newNode` to the current first node)
- point the last node to `newNode` (i.e making `newNode` as head)



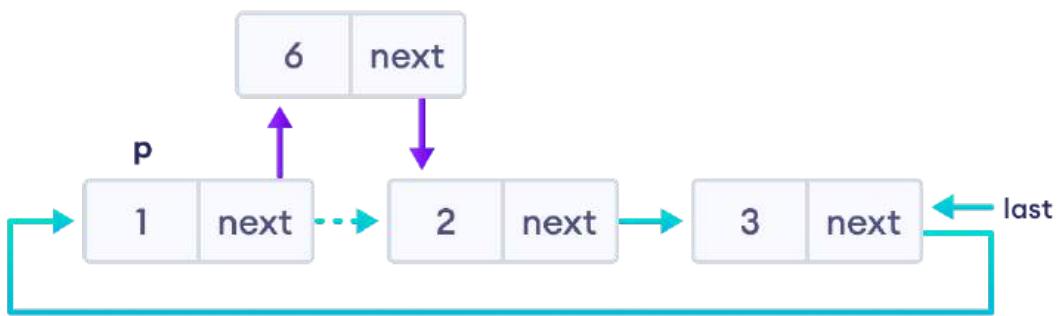
Insert at the beginning

---

## 2. Insertion in between two nodes

Let's insert `newNode` after the first node.

- travel to the node given (let this node be `p`)
- point the `next` of `newNode` to the node next to `p`
- store the address of `newNode` at `next` of `p`

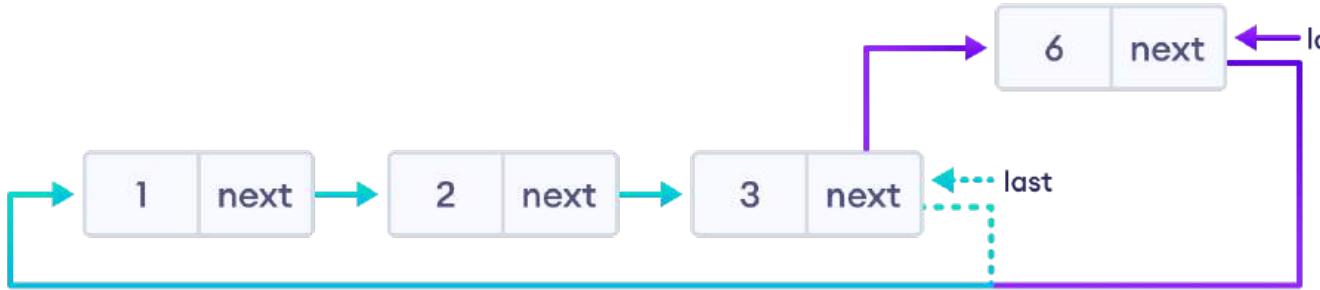


Insertion at a node

---

### 3. Insertion at the end

- store the address of the head node to `next` of `newNode` (making `newNode` the last node)
- point the current last node to `newNode`
- make `newNode` as the last node

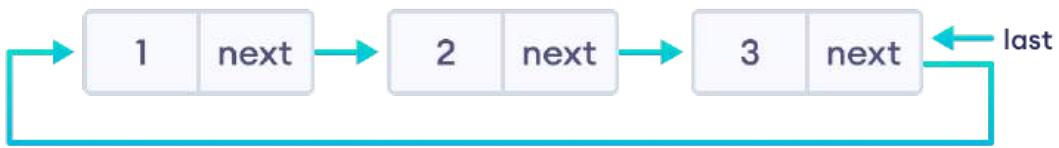


Insert at the end

---

## Deletion on a Circular Linked List

Suppose we have a double-linked list with elements 1, 2, and 3.



Initial circular linked list

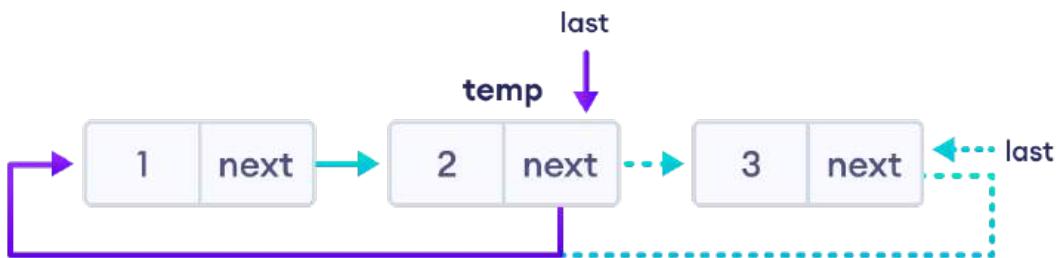
---

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

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

### 2. If last node is to be deleted

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



Delete the last node

### 3. If any other nodes are to be deleted

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



Delete a specific node

# Data Structure Graph

# Graphs

- 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 .
- A graph  $G$  is defined as follows:

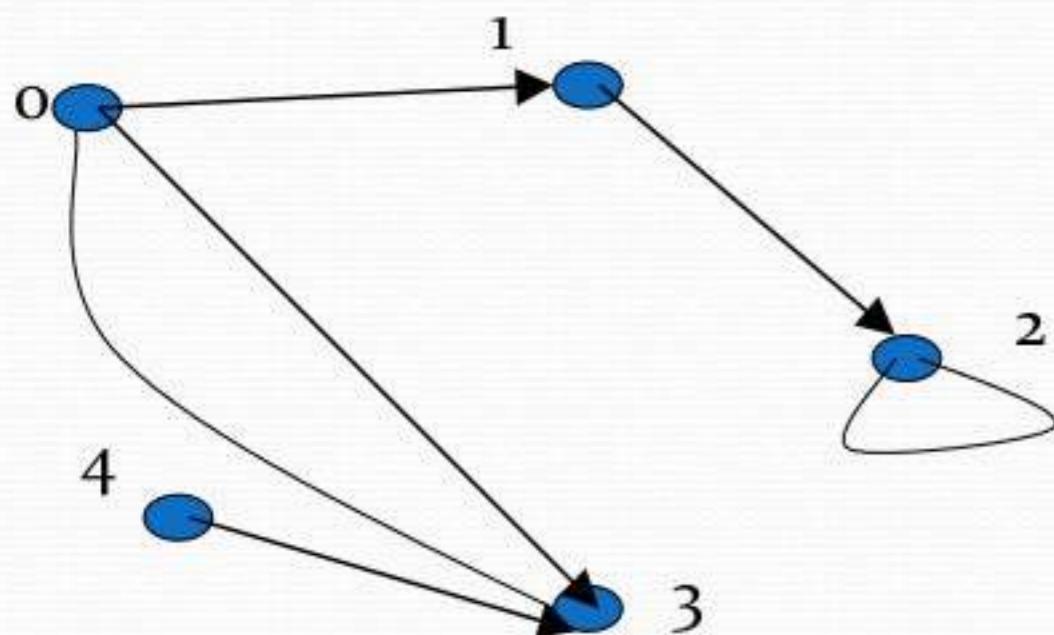
$$G=(V,E)$$

$V(G)$ : a finite, nonempty set of vertices

$E(G)$ : a set of edges (pairs of vertices)

# Examples of Graphs

- $V = \{0, 1, 2, 3, 4\}$
- $E = \{(0,1), (1,2), (0,3), (3,0), (2,2), (4,3)\}$



When  $(x,y)$  is an edge,  
we say that  $x$  is *adjacent to*  $y$ , and  $y$   
is *adjacent from*  $x$ .

0 is adjacent to 1.  
1 is not adjacent to 0.  
2 is adjacent from 1.

## Directed vs. Undirected Graphs

- Undirected edge has no orientation (no arrow head)
- Directed edge has an orientation (has an arrow head)
- Undirected graph – all edges are undirected
- Directed graph – all edges are directed

**u** ————— **v**

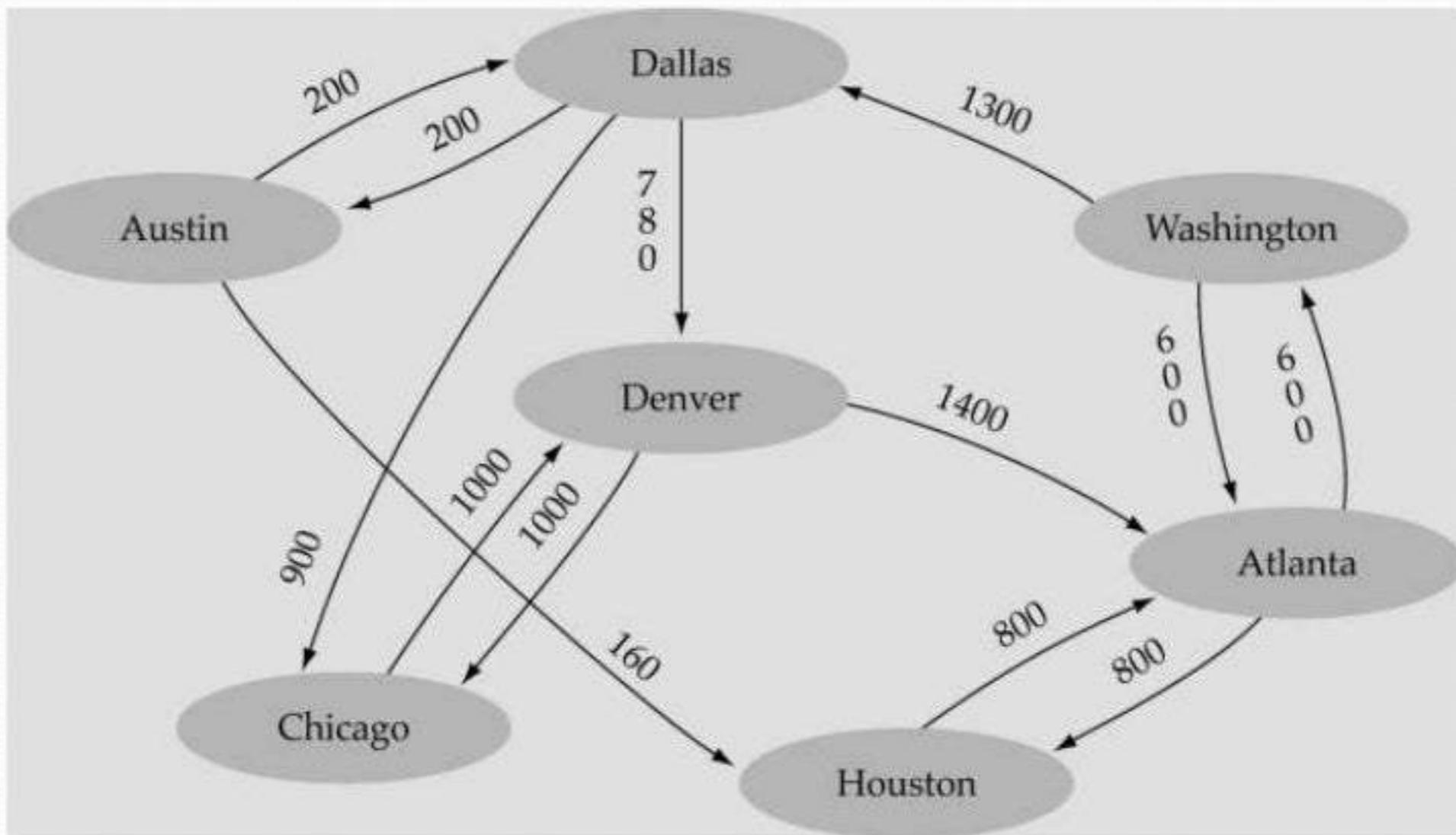
undirected edge

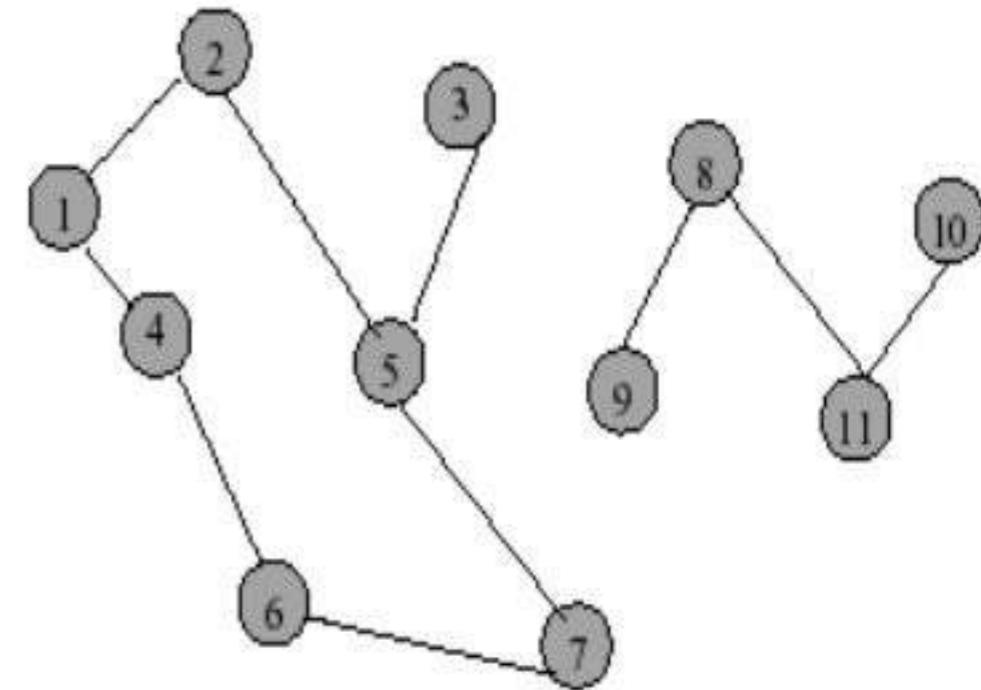
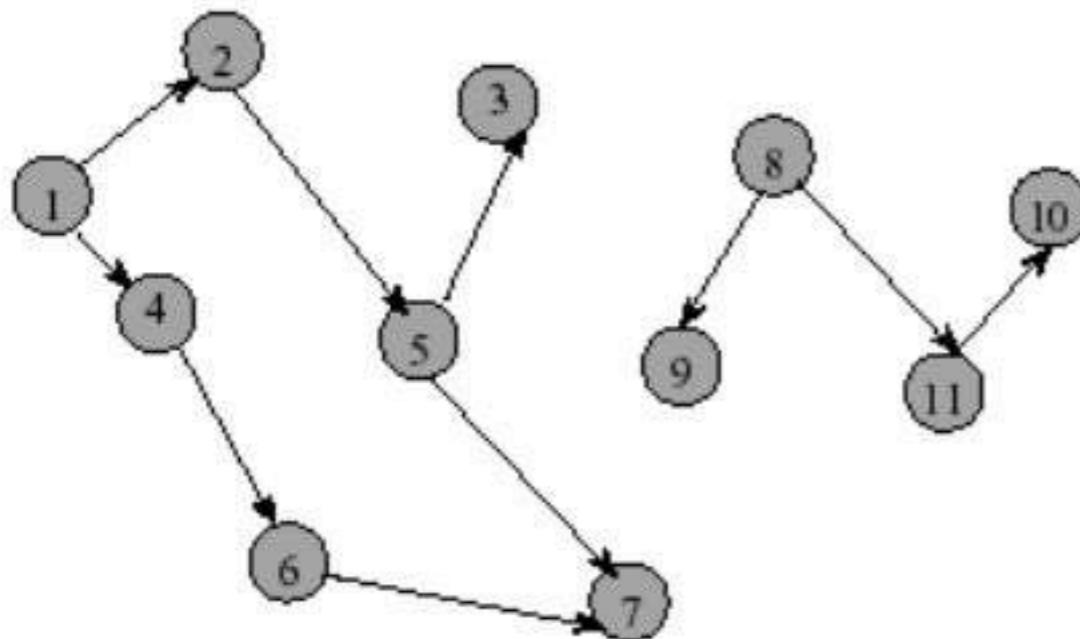
**u** → **v**

directed edge

## Weighted graph:

-a graph in which each edge carries a value





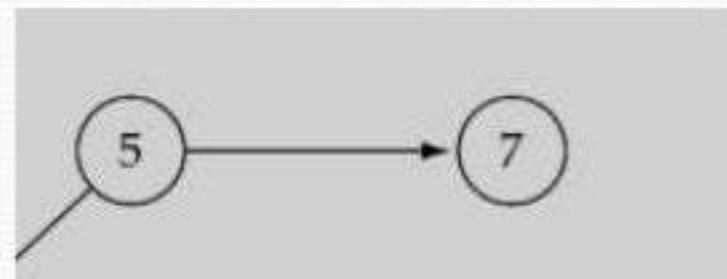
Directed graph

## Directed Graph

- Directed edge  $(i, j)$  is **incident to** vertex  $j$  and **incident from** vertex  $i$
- Vertex  $i$  is **adjacent to** vertex  $j$ , and vertex  $j$  is **adjacent from** vertex  $i$

# Graph terminology

- **Adjacent nodes:** two nodes are adjacent if they are connected by an edge

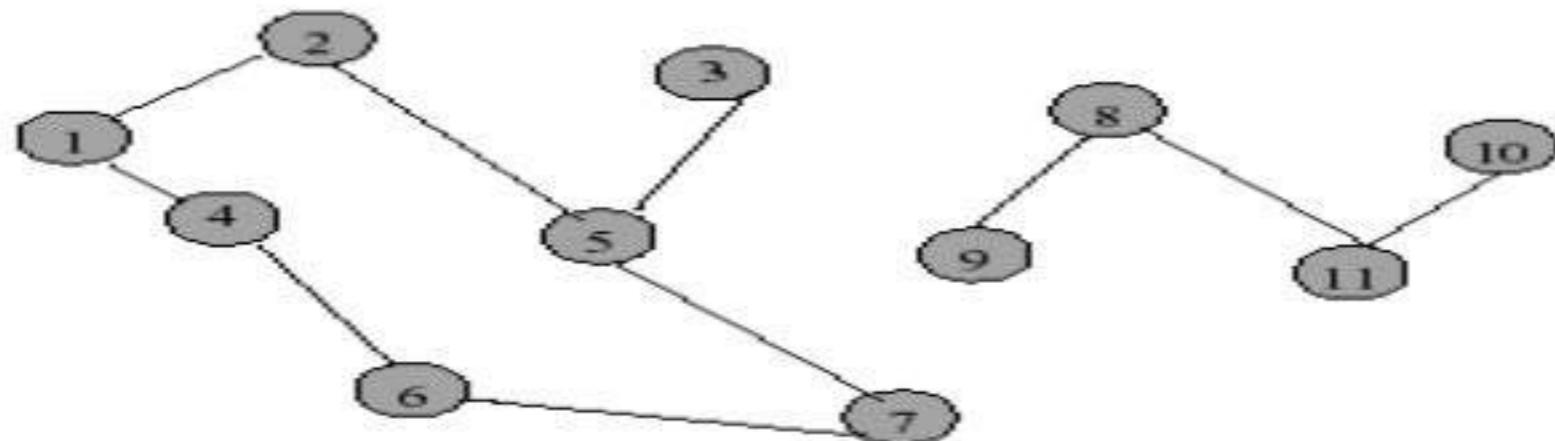


5 is adjacent to 7  
7 is adjacent from

- **Path:** a sequence of vertices that connect two nodes in a graph
- A **simple path** is a path in which all vertices, except possibly in the first and last, are different.
- **Complete graph:** a graph in which every vertex is directly connected to every other vertex

Continued...

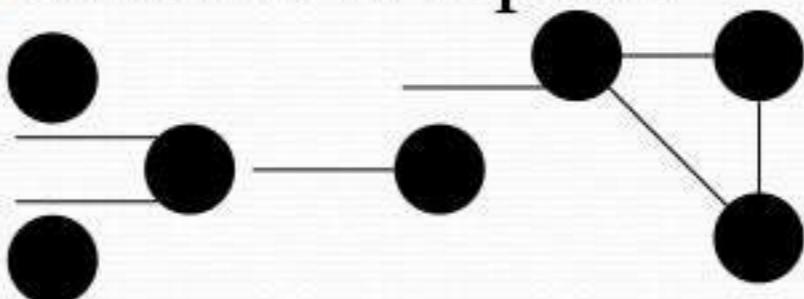
- A **cycle** is a simple path with the same start and end vertex.
- The **degree** of vertex  $i$  is the no. of edges incident on vertex  $i$ .



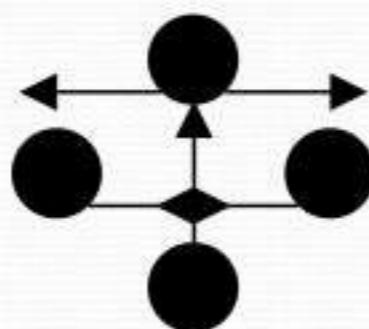
e.g.,  $\text{degree}(2) = 2$ ,  $\text{degree}(5) = 3$ ,  $\text{degree}(3) = 1$

## Continued...

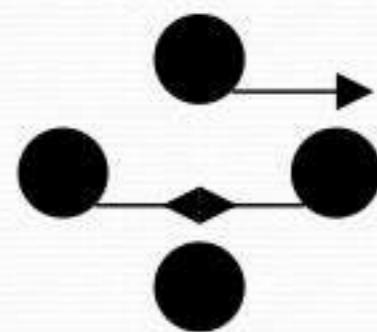
Undirected graphs are *connected* if there is a path between any two vertices



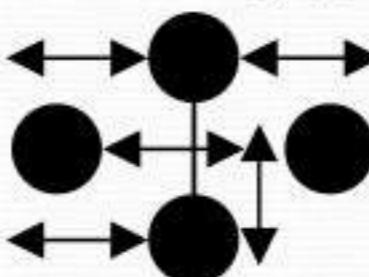
Directed graphs are *strongly connected* if there is a path from any one vertex to any other



Directed graphs are *weakly connected* if there is a path between any two vertices, *ignoring direction*



A *complete* graph has an edge between every pair of vertices



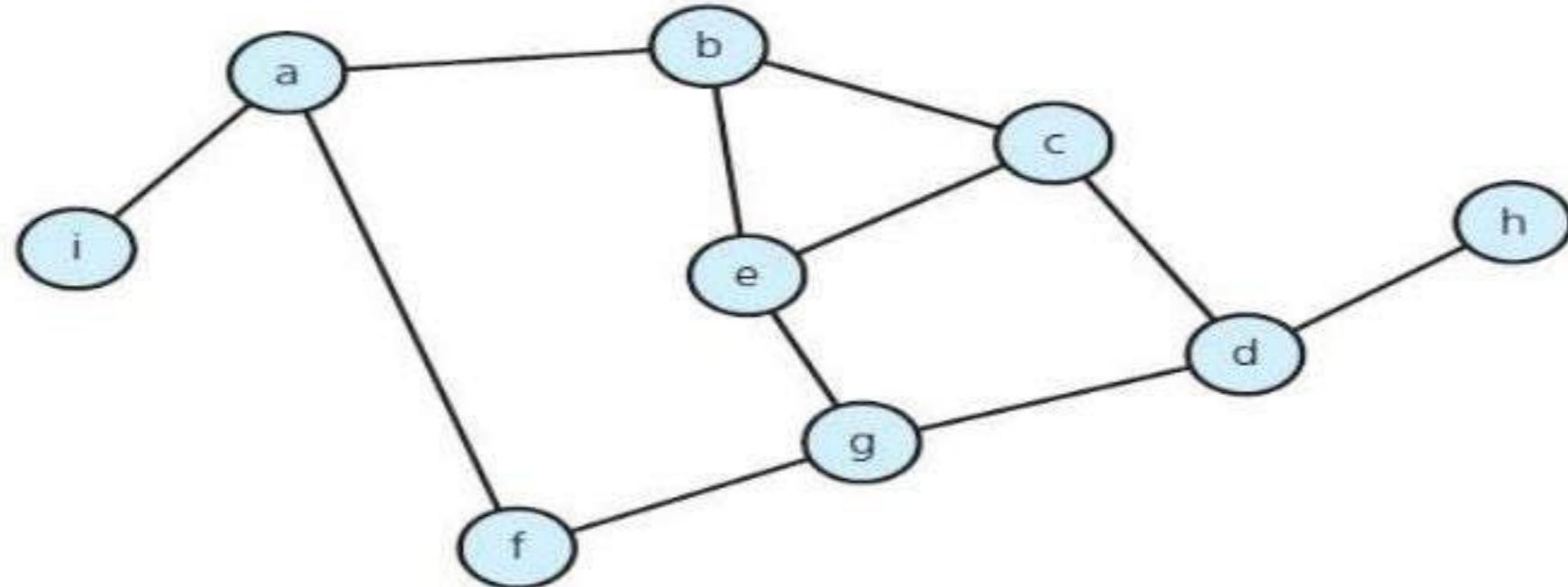
# Continued...

- *Loops*: edges that connect a vertex to itself
- *Paths*: sequences of vertices  $p_0, p_1, \dots, p_m$  such that each adjacent pair of vertices are connected by an edge
- A *simple path* is a path in which all vertices, except possibly in the first and last, are different.
- *Multiple Edges*: two nodes may be connected by  $> 1$  edge
- *Simple Graphs*: have no loops and no multiple edges

# Graph Properties

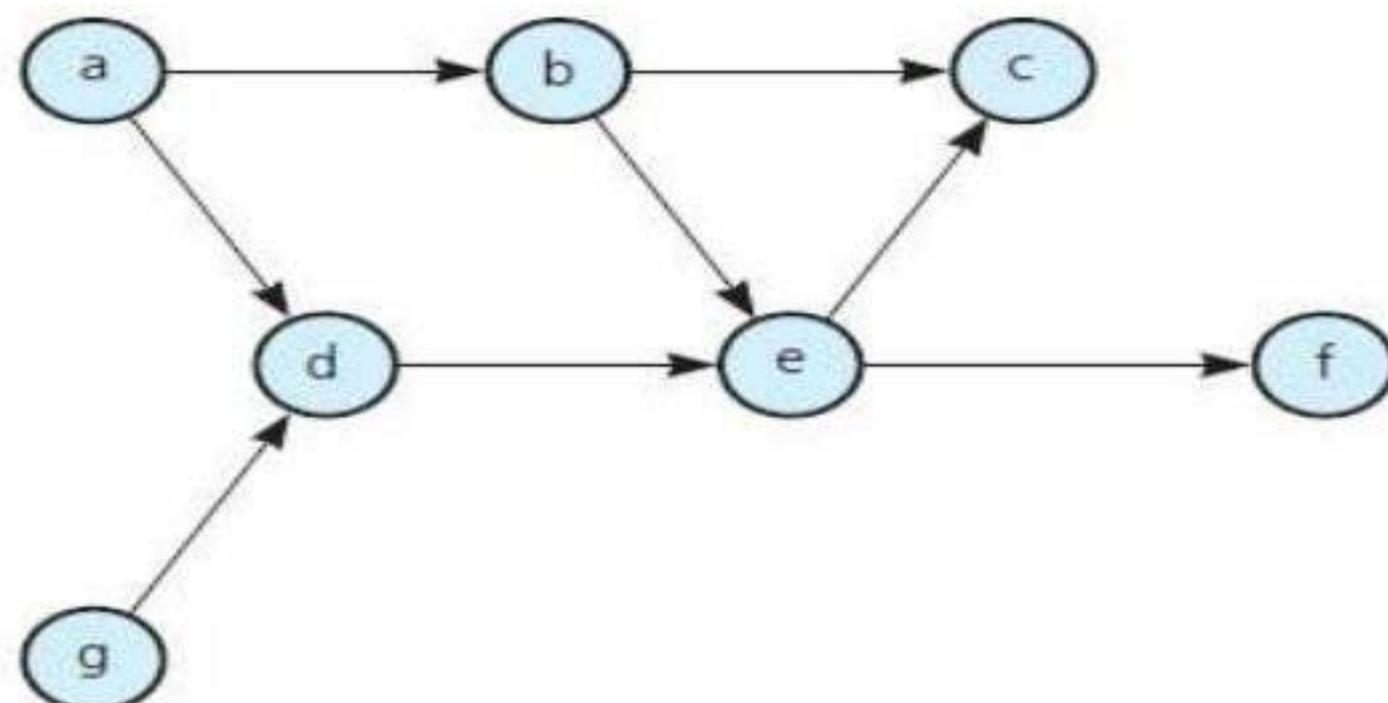
## Number of Edges – Undirected Graph

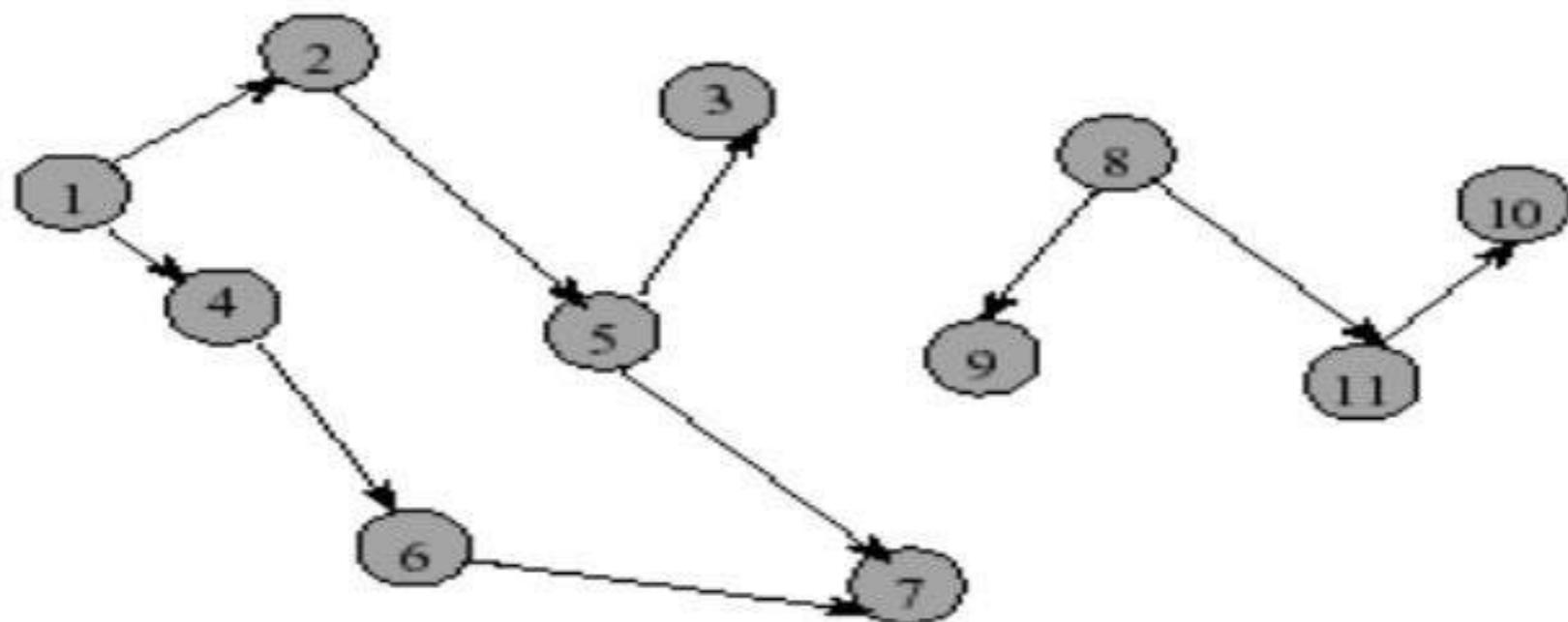
- The no. of possible pairs in an n vertex graph is  $n^*(n-1)$
- Since edge  $(u,v)$  is the same as edge  $(v,u)$ , the number of edges in an undirected graph is  $n^*(n-1)/2$ .



# Number of Edges - Directed Graph

- The no. of possible pairs in an n vertex graph is  $n^*(n-1)$
- Since edge  $(u,v)$  is not the same as edge  $(v,u)$ , the number of edges in a directed graph is  $n^*(n-1)$
- Thus, the number of edges in a directed graph is  $\leq n^*(n-1)$





- **In-degree** of vertex  $i$  is the **number of edges incident to  $i$**  (i.e., the number of incoming edges).  
e.g.,  $\text{indegree}(2) = 1$ ,  $\text{indegree}(8) = 0$
- **Out-degree** of vertex  $i$  is the **number of edges incident from  $i$**  (i.e., the number of outgoing edges).  
e.g.,  $\text{outdegree}(2) = 1$ ,  $\text{outdegree}(8) = 2$

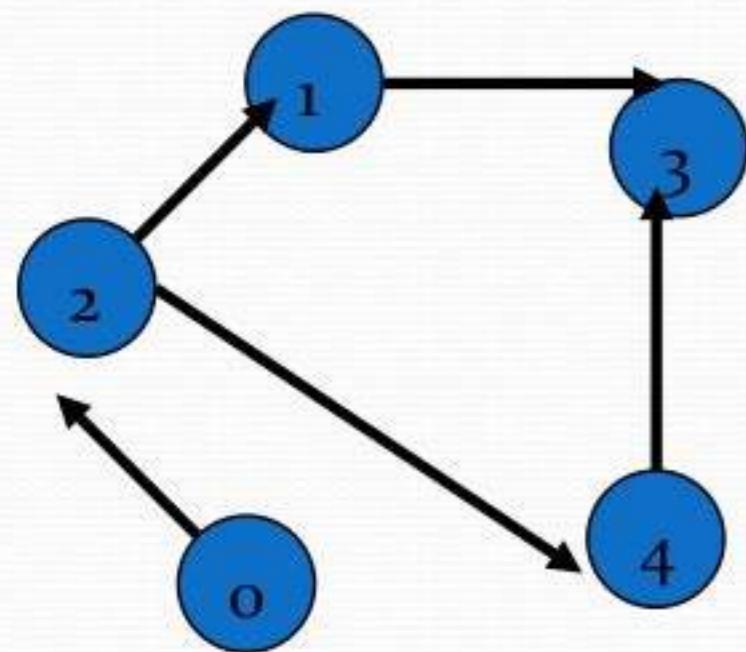
# Graph Representation

- For graphs to be computationally useful, they have to be conveniently represented in programs
- There are two computer representations of graphs:
  - Adjacency matrix representation
  - Adjacency lists representation

- *Adjacency Matrix*

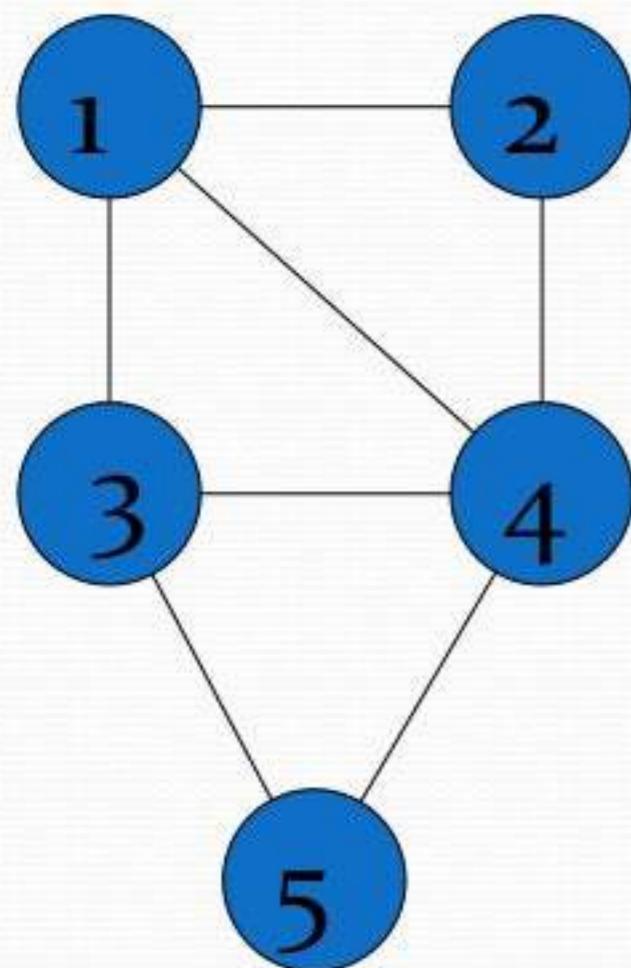
- A square grid of boolean values
- If the graph contains  $N$  vertices, then the grid contains  $N$  rows and  $N$  columns
- For two vertices numbered  $I$  and  $J$ , the element at row  $I$  and column  $J$  is true if there is an edge from  $I$  to  $J$ , otherwise false

# Adjacency Matrix



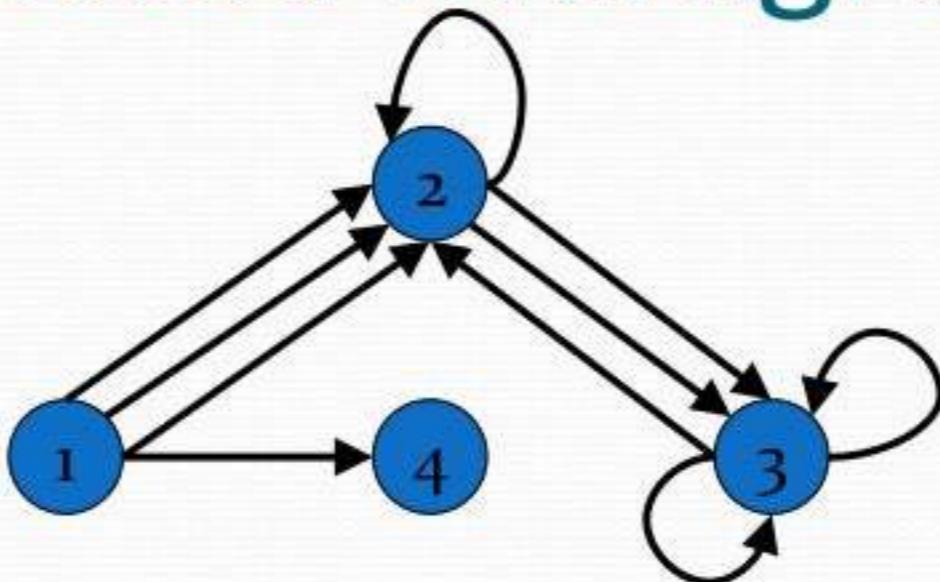
	0	1	2	3	4
0	false	false	true	false	false
1	false	false	false	true	false
2	false	true	false	false	true
3	false	false	false	false	false
4	false	false	false	true	false

# Adjacency Matrix



	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>
<b>1</b>	0	1	1	1	0
<b>2</b>	1	0	0	1	0
<b>3</b>	1	0	0	1	1
<b>4</b>	1	1	1	0	1
<b>5</b>	0	0	1	1	0

# Adjacency Matrix -Directed Multigraphs



A:

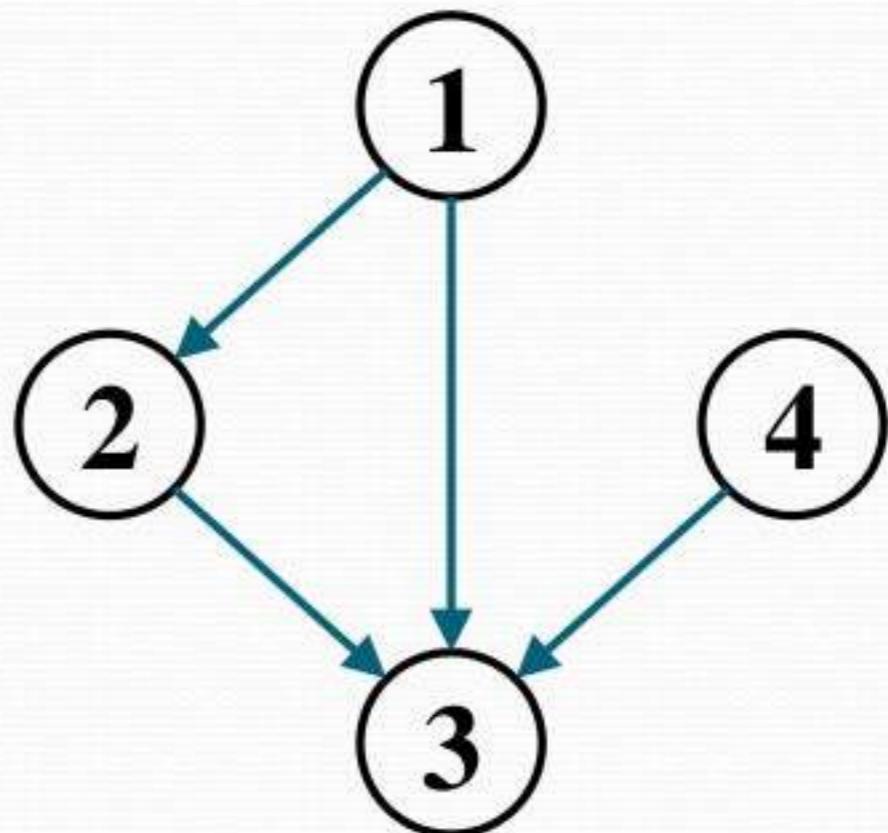
$$\begin{pmatrix} 0 & 3 & 0 & 1 \\ 0 & 1 & 2 & 0 \\ 0 & 1 & 2 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

# Adjacency Lists Representation

- A graph of  $n$  nodes is represented by a one-dimensional array  $L$  of linked lists, where
  - $L[i]$  is the linked list containing all the nodes adjacent from node  $i$ .
  - The nodes in the list  $L[i]$  are in no particular order

# Graphs: Adjacency List

- Adjacency list: for each vertex  $v \in V$ , store a list of vertices adjacent to  $v$
- Example:
  - $\text{Adj}[1] = \{2, 3\}$
  - $\text{Adj}[2] = \{3\}$
  - $\text{Adj}[3] = \{\}$
  - $\text{Adj}[4] = \{3\}$
- Variation: can also keep a list of edges coming *into* vertex

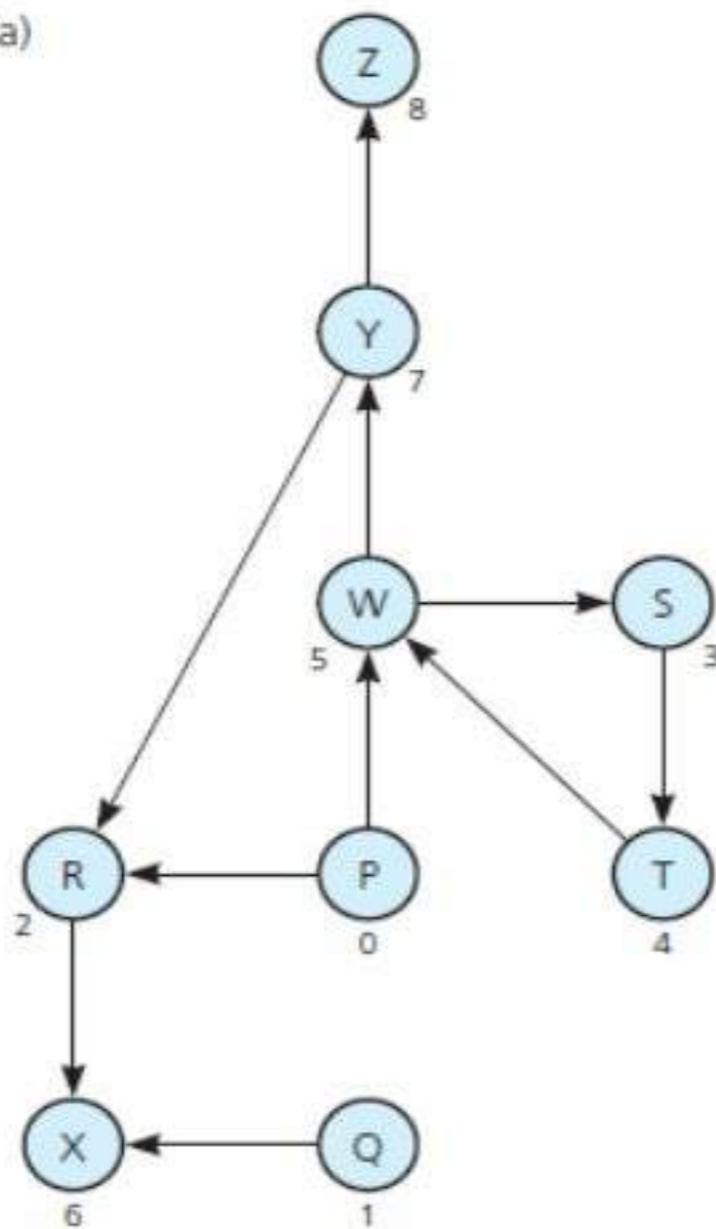


# Graphs: Adjacency List

- How much storage is required?
  - The *degree* of a vertex  $v$  = # incident edges
    - Directed graphs have in-degree, out-degree
  - For directed graphs, # of items in adjacency lists is
$$\sum \text{out-degree}(v) = |E|$$
For undirected graphs, # items in adjacency lists is
$$\sum \text{degree}(v) = 2 |E|$$
- So: Adjacency lists take  $O(V+E)$  storage

# Implementing Graphs

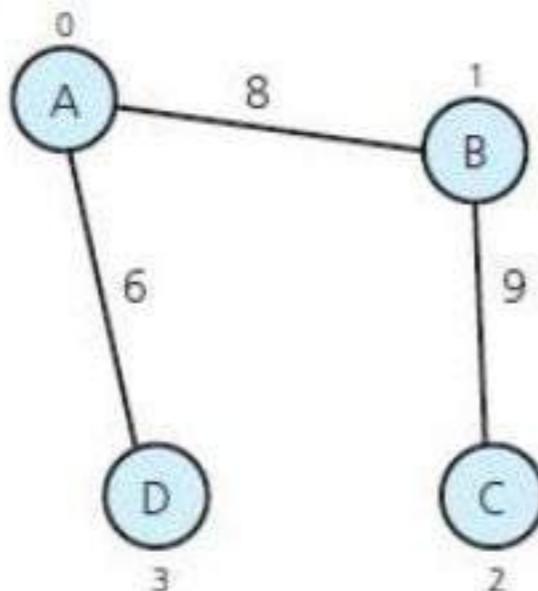
(a)



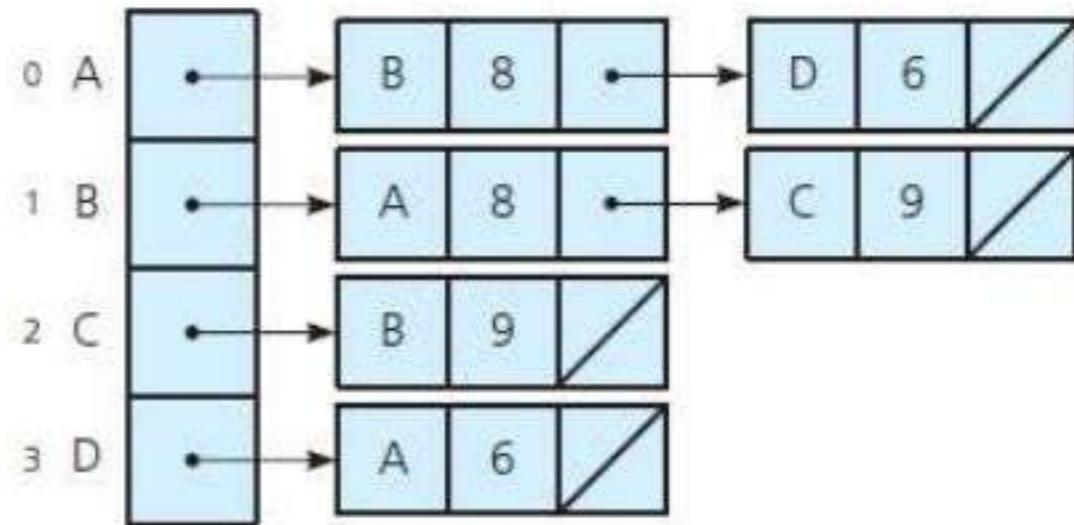
(b)

# Implementing Graphs

(a)



(b)



- (a) A weighted undirected graph and  
(b) its adjacency list



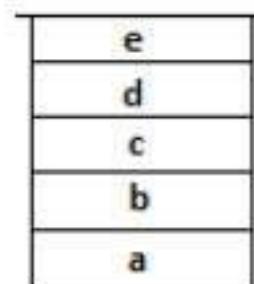
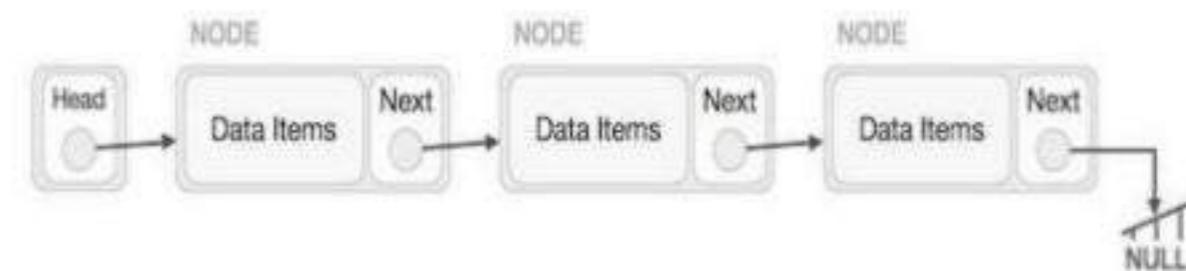
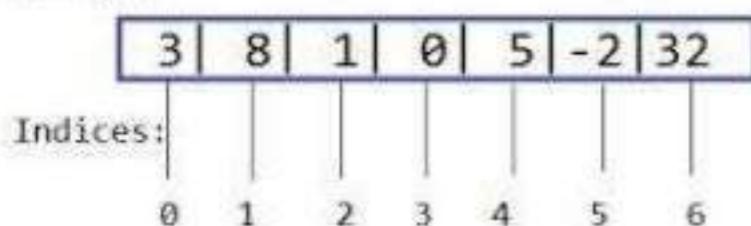
# Thank you!!!!

# Tree

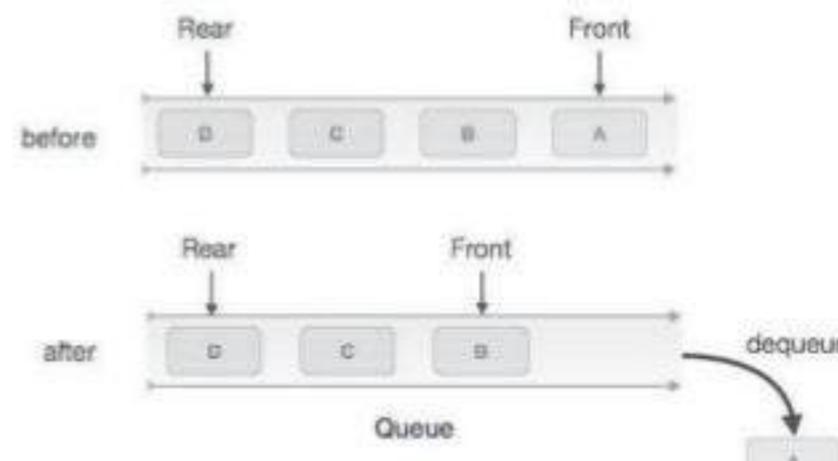
Unit 6

# So far we discussed Linear data structures like

Array :



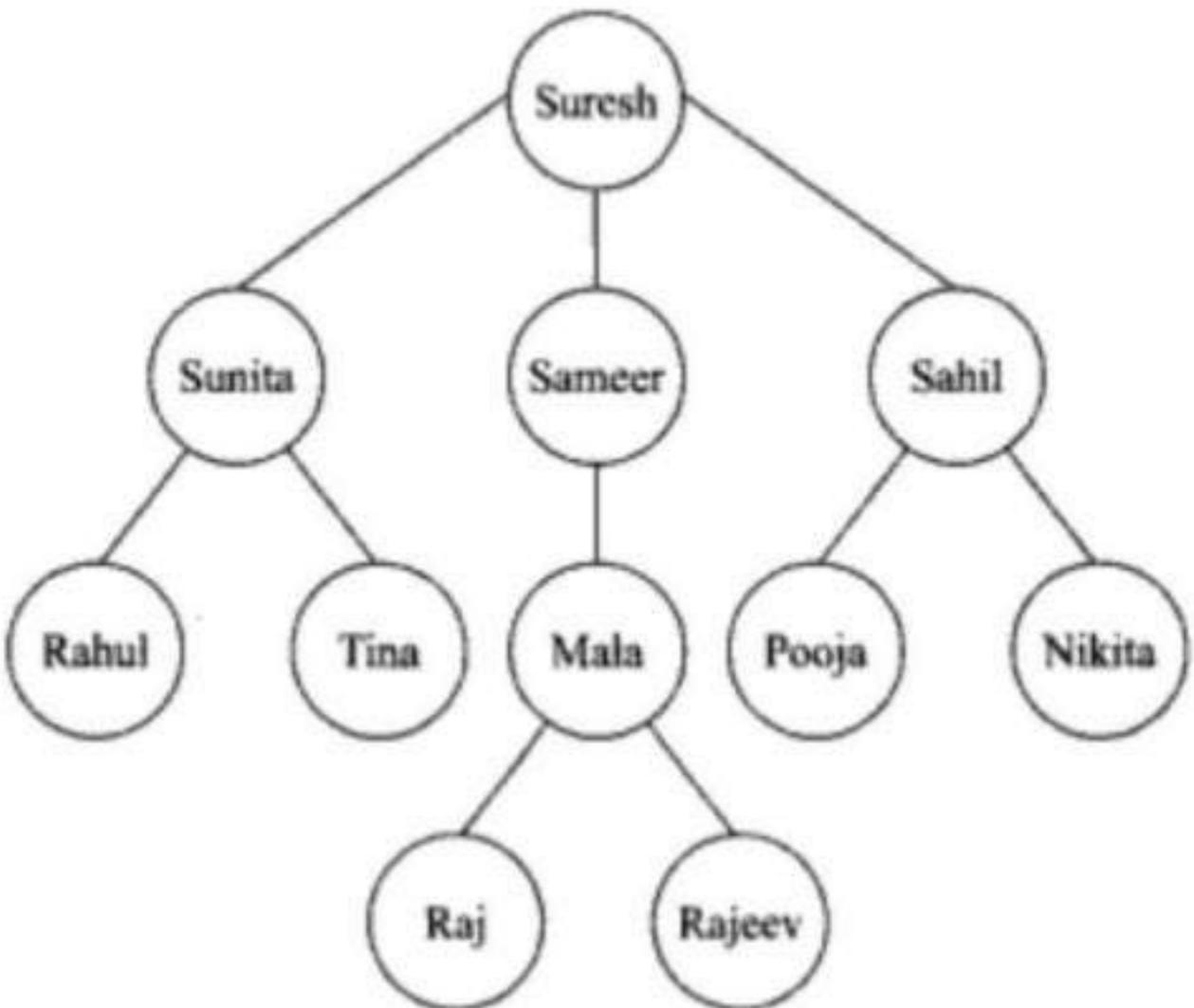
stack



Queue Dequeue

# Introduction to trees

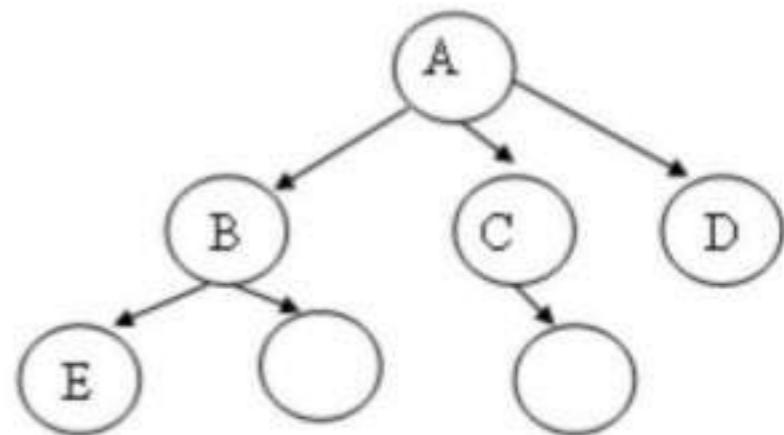
- So far we have discussed mainly linear data structures – strings, arrays, lists, stacks and queues
- Now we will discuss a non-linear data structure called **tree**.
- Trees are mainly used to represent data containing a hierarchical relationship between elements, for example, records, family trees and table of contents.
- Consider a parent-child relationship



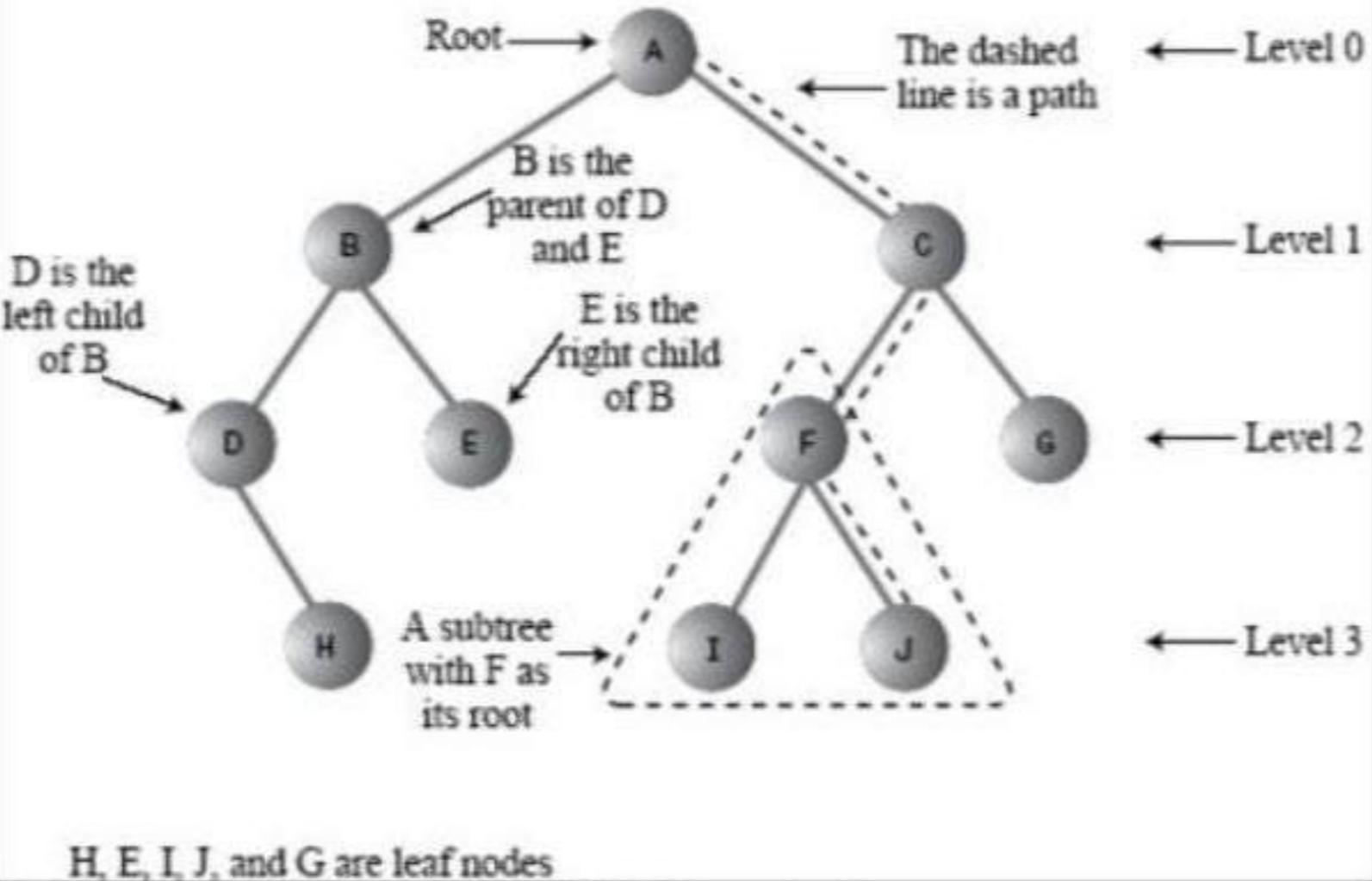
**Fig. 8.1 A Hypothetical Family Tree**

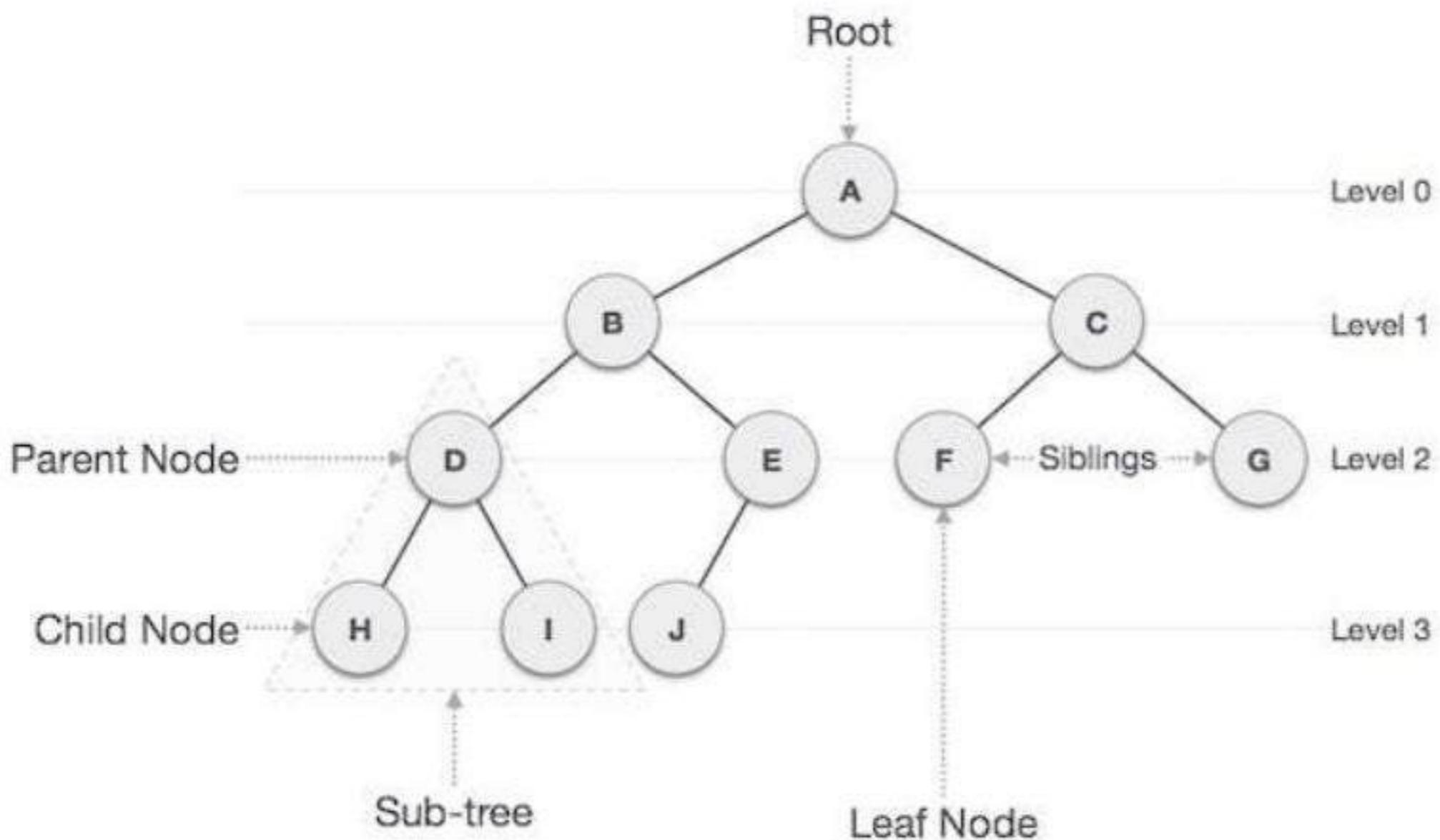
# Tree

- A tree is an abstract model of a hierarchical structure that consists of nodes with a parent-child relationship.
  - Tree is a sequence of **nodes**
  - There is a starting node known as a **root** node
  - Every node other than the root has a **parent** node.
  - Nodes may have any number of children



A has 3 children, B, C, D  
A is parent of B



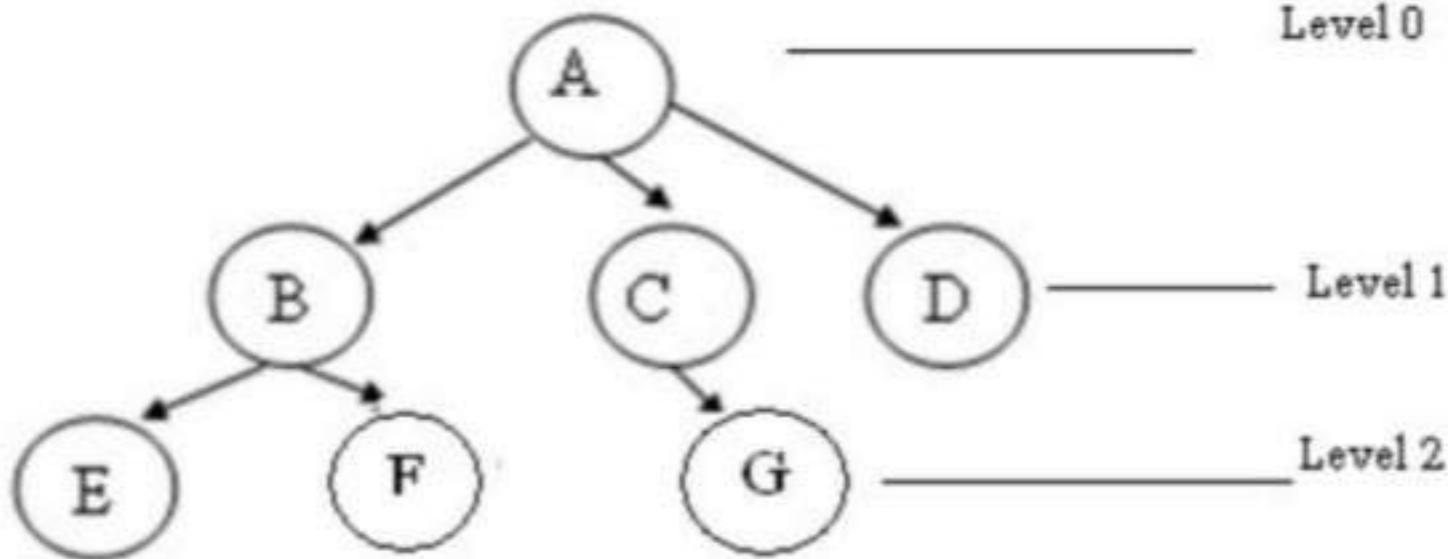


# Some Key Terms:

- **Root** – Node at the top of the tree is called root.
- **Parent** – Any node except root node has one edge upward to a node called parent.
- **Child** – Node below a given node connected by its edge downward is called its child node.
- **Sibling** – Child of same node are called siblings
- **Leaf** – Node which does not have any child node is called leaf node.
- **Sub tree** – Sub tree represents descendants of a node.
- **Levels** – Level of a node represents the generation of a node. If root node is at level 0, then its next child node is at level 1, its grandchild is at level 2 and so on.
- **keys** – Key represents a value of a node based on which a search operation is to be carried out for a node.

# Some Key Terms:

- Degree of a node:
  - The degree of a node is the number of children of that node
- Degree of a Tree:
  - The degree of a tree is the maximum degree of nodes in a given tree
- Path:
  - It is the sequence of consecutive edges from source node to destination node.
- Height of a node:
  - The height of a node is the max path length form that node to a leaf node.
- Height of a tree:
  - The height of a tree is the height of the root
- Depth of a tree:
  - Depth of a tree is the max level of any leaf in the tree



- ✓ A is the root node
- ✓ B is the parent of E and F
- ✓ D is the sibling of B and C
- ✓ E and F are children of B
- ✓ E, F, G, D are external nodes or leaves
- ✓ A, B, C are internal nodes
- ✓ Depth of F is 2
- ✓ the height of tree is 2
- ✓ the degree of node A is 3
- ✓ The degree of tree is 3

# Characteristics of trees

- Non-linear data structure
- Combines advantages of an ordered array
- Searching as fast as in ordered array
- Insertion and deletion as fast as in linked list
- Simple and fast

# Application

- Directory structure of a file store
- Structure of an arithmetic expressions
- Used in almost every 3D video game to determine what objects need to be rendered.
- Used in almost every high-bandwidth router for storing router-tables.
- used in compression algorithms, such as those used by the .jpeg and .mp3 file-formats.

FOR Further detail [Click Here](#)

# Introduction To Binary Trees

- A **binary tree**, is a tree in which no node can have more than two children.
- Consider a binary tree T, here 'A' is the root node of the binary tree T.
- 'B' is the left child of 'A' and 'C' is the right child of 'A'
  - i.e A is a father of B and C.
  - The node B and C are called **siblings**.
- Nodes D,H,I,F,J are leaf node

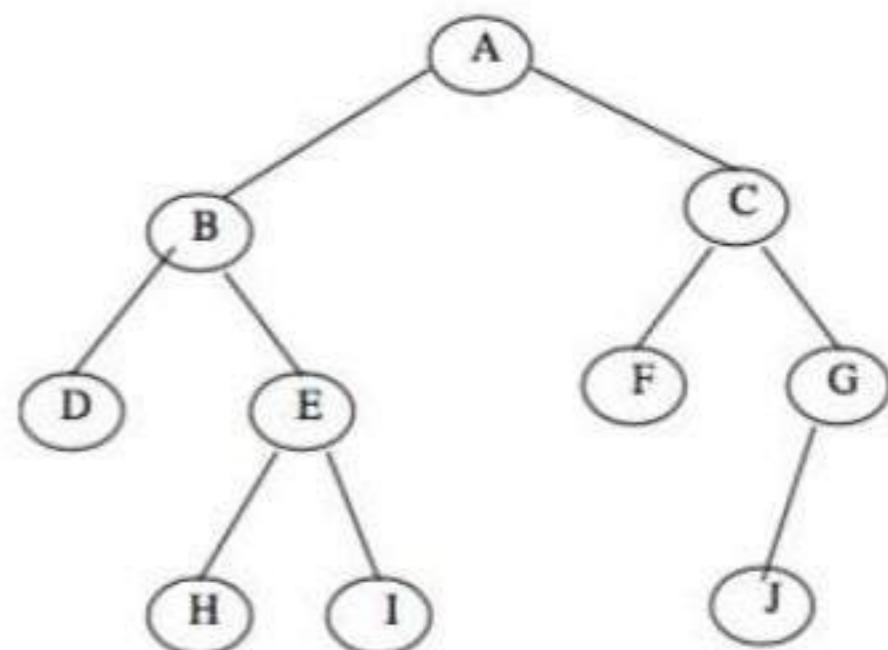
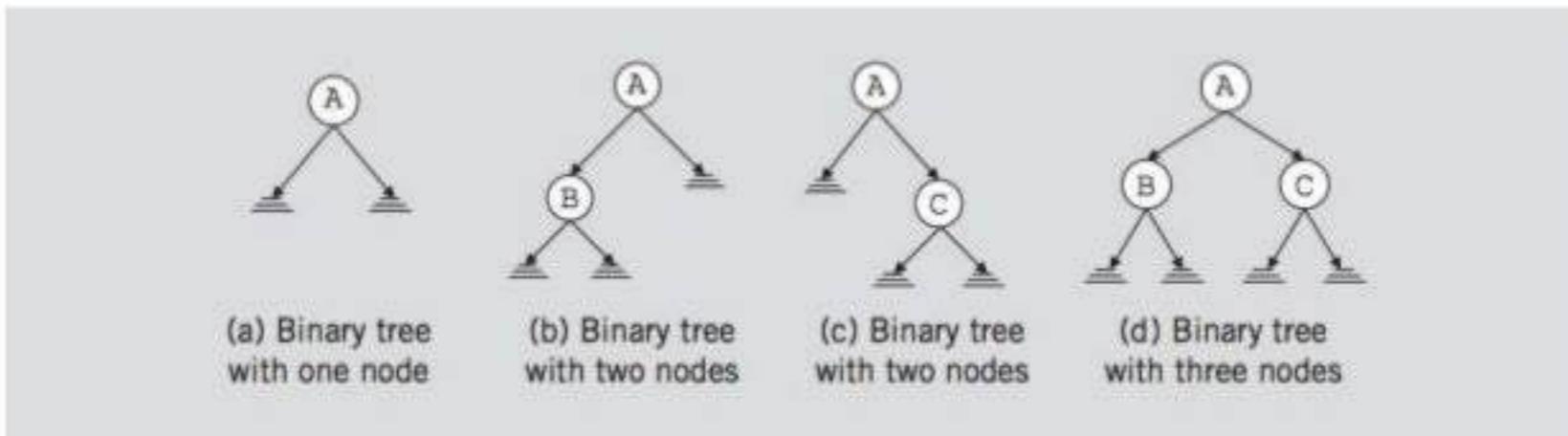


Fig. 8.3. Binary tree

# Binary Trees

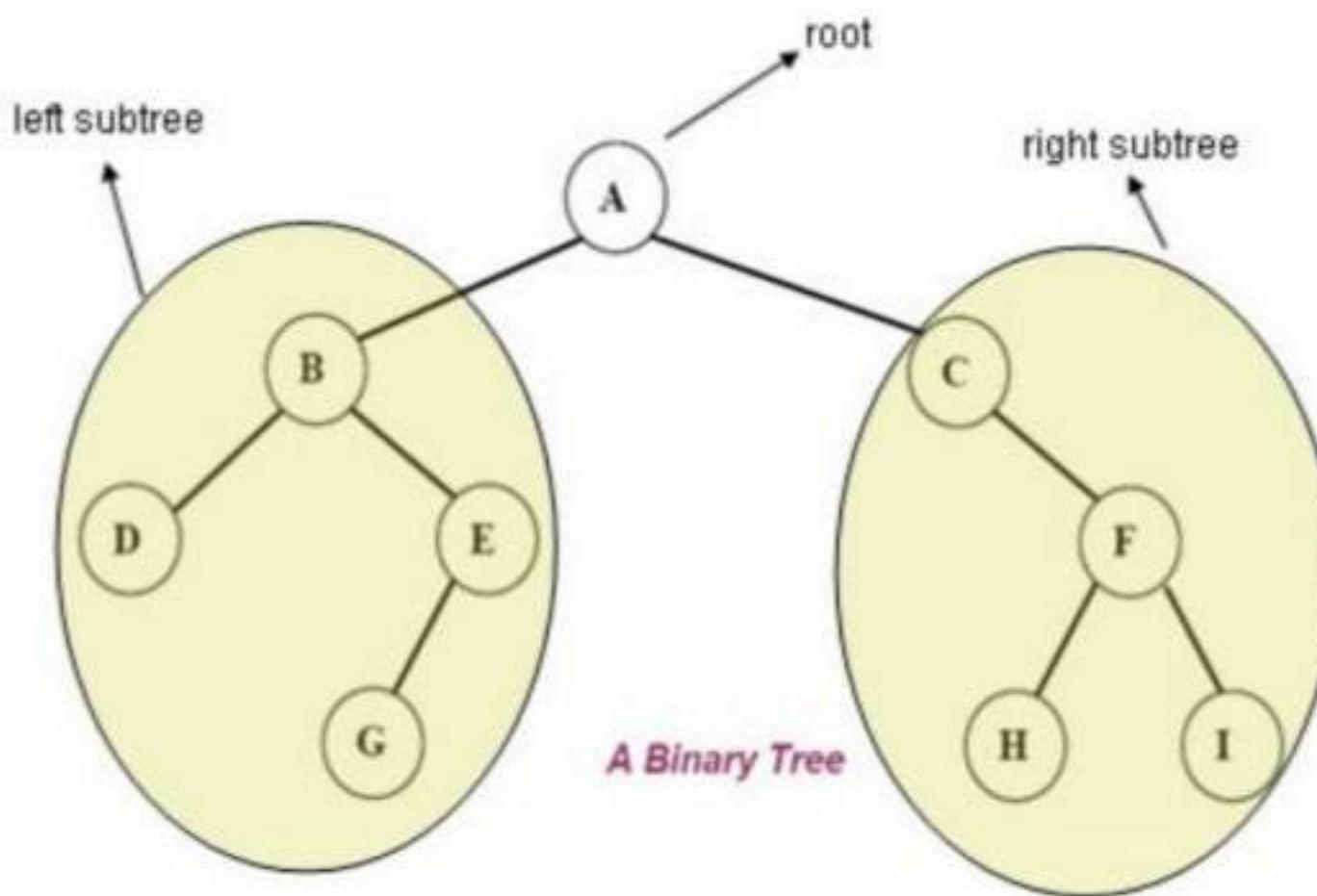
- A **binary tree**,  $T$ , is either empty or such that
  - $T$  has a special node called the **root** node
  - $T$  has two sets of nodes  $L_T$  and  $R_T$ , called the **left subtree** and **right subtree** of  $T$ , respectively.
  - $L_T$  and  $R_T$  are binary trees.



# Binary Tree

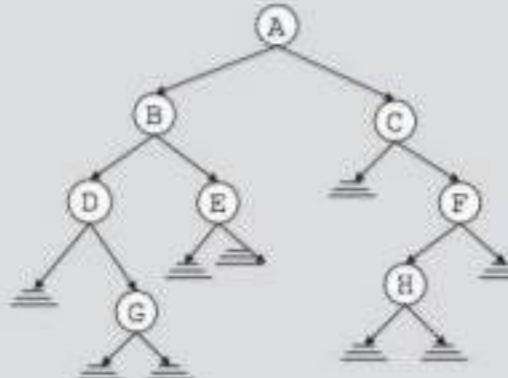
- A binary tree is a finite set of elements that are either empty or is partitioned into three disjoint subsets.
- The first subset contains a single element called the **root** of the tree.
- The other two subsets are themselves binary trees called the **left** and **right sub-trees** of the original tree.
- A left or right sub-tree can be empty.
- Each element of a binary tree is called a **node** of the tree.

The following figure shows a binary tree with 9 nodes where A is the root



## Binary Tree

- The root node of this binary tree is A.
- The left sub tree of the root node, which we denoted by  $L_A$ , is the set  $L_A = \{B, D, E, G\}$  and the right sub tree of the root node,  $R_A$  is the set  $R_A = \{C, F, H\}$
- The root node of  $L_A$  is node B, the root node of  $R_A$  is C and so on



# Binary Tree Properties

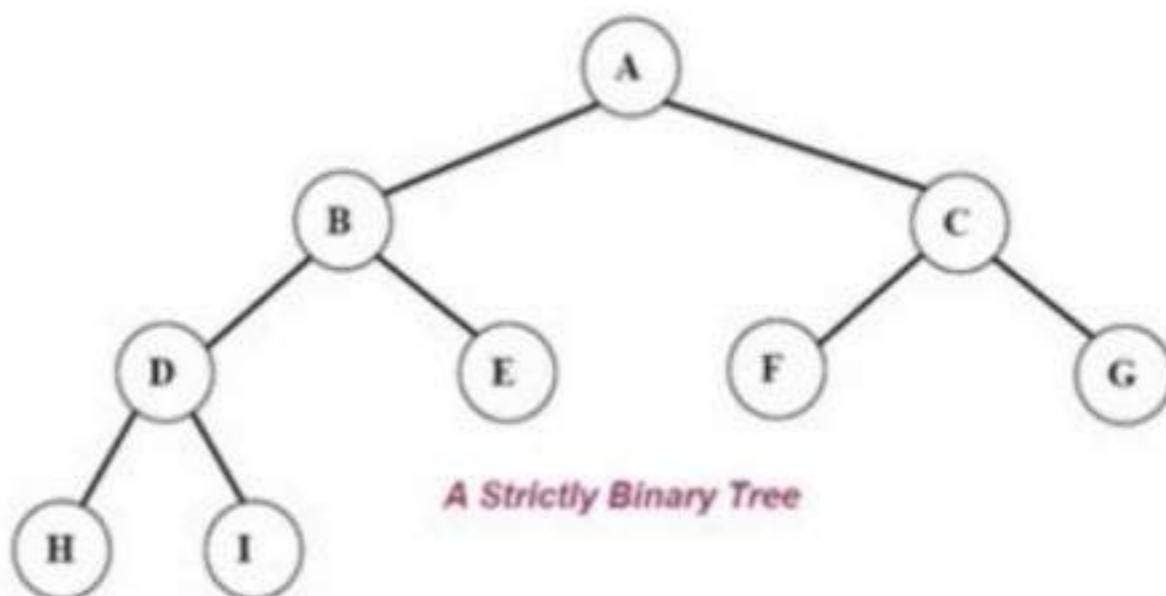
- If a binary tree contains  $m$  nodes at level  $L$ , it contains at most  $2m$  nodes at level  $L+1$
- Since a binary tree can contain at most 1 node at level 0 (the root), it contains at most  $2L$  nodes at level  $L$ .

# Types of Binary Tree

- Complete binary tree
- Strictly binary tree
- Almost complete binary tree

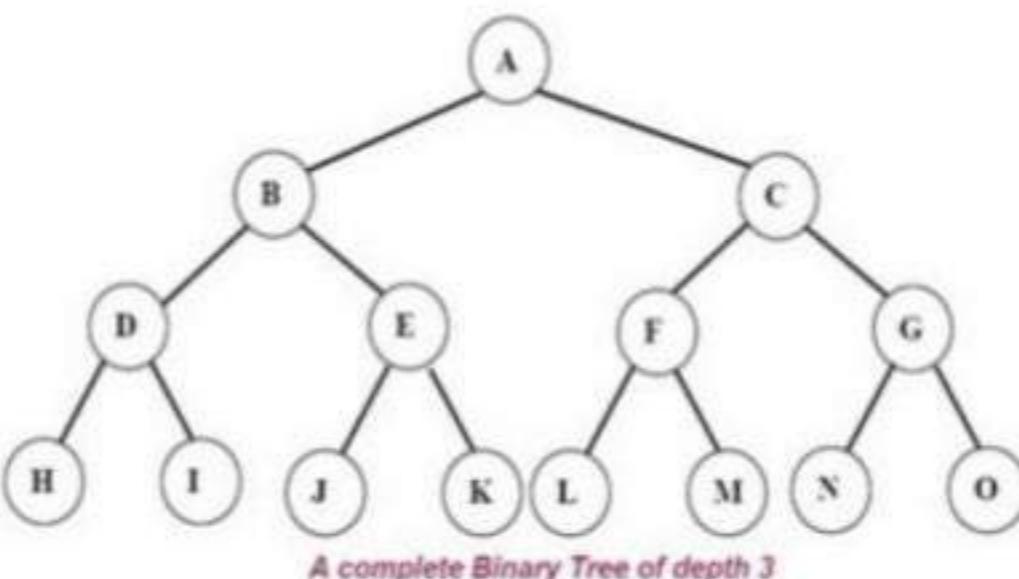
# Strictly binary tree

- If every non-leaf node in a binary tree has nonempty left and right sub-trees, then such a tree is called a **strictly binary tree**.
- **Or, to put it another way,** all of the nodes in a strictly binary tree are of degree zero or two, never degree one.
- A strictly binary tree with  $N$  leaves always contains  $2N - 1$  nodes.



# 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.
- A **complete binary tree** of depth **d** is called strictly binary tree if all of whose leaves are at level **d**.
- A complete binary tree has  $2^d$  nodes at every depth **d** and  $2^d - 1$  non leaf nodes



# Almost complete binary tree

- An almost complete binary tree is a tree where for a right child, there is always a left child, but for a left child there may not be a right child.

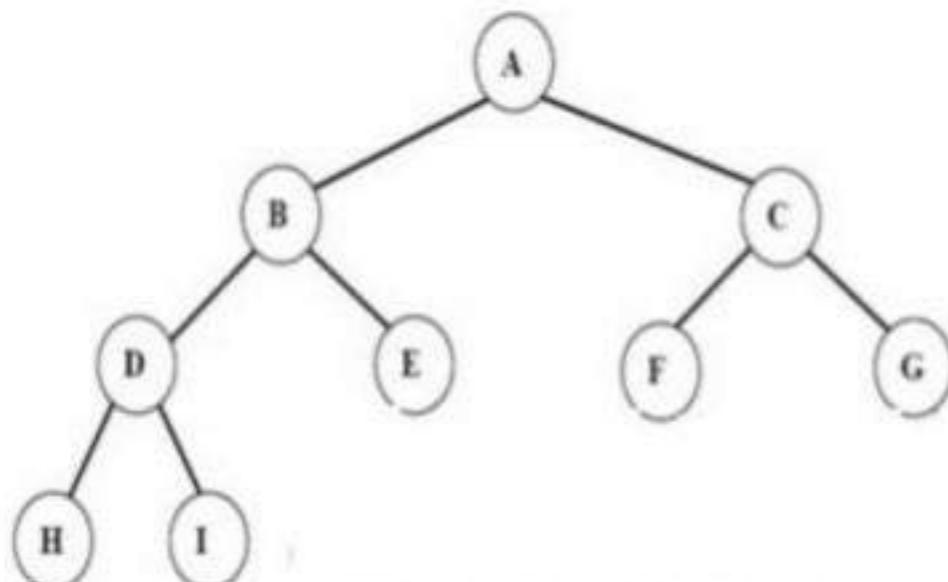


Fig Almost complete binary tree.

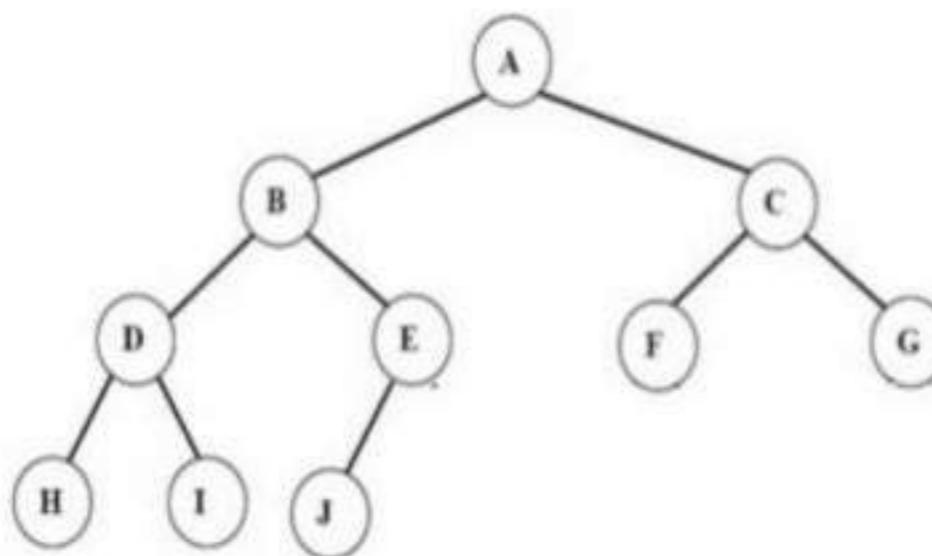


Fig Almost complete binary tree but not strictly binary tree.  
Since node E has a left son but not a right son.

## Operations on Binary tree:

- ✓ **father(n,T):** Return the parent node of the node n in tree T. If n is the root, NULL is returned.
- ✓ **LeftChild(n,T):** Return the left child of node n in tree T. Return NULL if n does not have a left child.
- ✓ **RightChild(n,T):** Return the right child of node n in tree T. Return NULL if n does not have a right child.
- ✓ **Info(n,T):** Return information stored in node n of tree T (ie. Content of a node).
- ✓ **Sibling(n,T):** return the sibling node of node n in tree T. Return NULL if n has no sibling.
- ✓ **Root(T):** Return root node of a tree if and only if the tree is nonempty.
- ✓ **Size(T):** Return the number of nodes in tree T
- ✓ **MakeEmpty(T):** Create an empty tree T
- ✓ **SetLeft(S,T):** Attach the tree S as the left sub-tree of tree T
- ✓ **SetRight(S,T):** Attach the tree S as the right sub-tree of tree T.
- ✓ **Preorder(T):** Traverses all the nodes of tree T in preorder.
- ✓ **postorder(T):** Traverses all the nodes of tree T in postorder
- ✓ **Inorder(T):** Traverses all the nodes of tree T in inorder.

## C representation for Binary tree:

```
struct bnode  
{  
    int info;  
    struct bnode *left;  
    struct bnode *right;  
};  
struct bnode *root=NUI
```

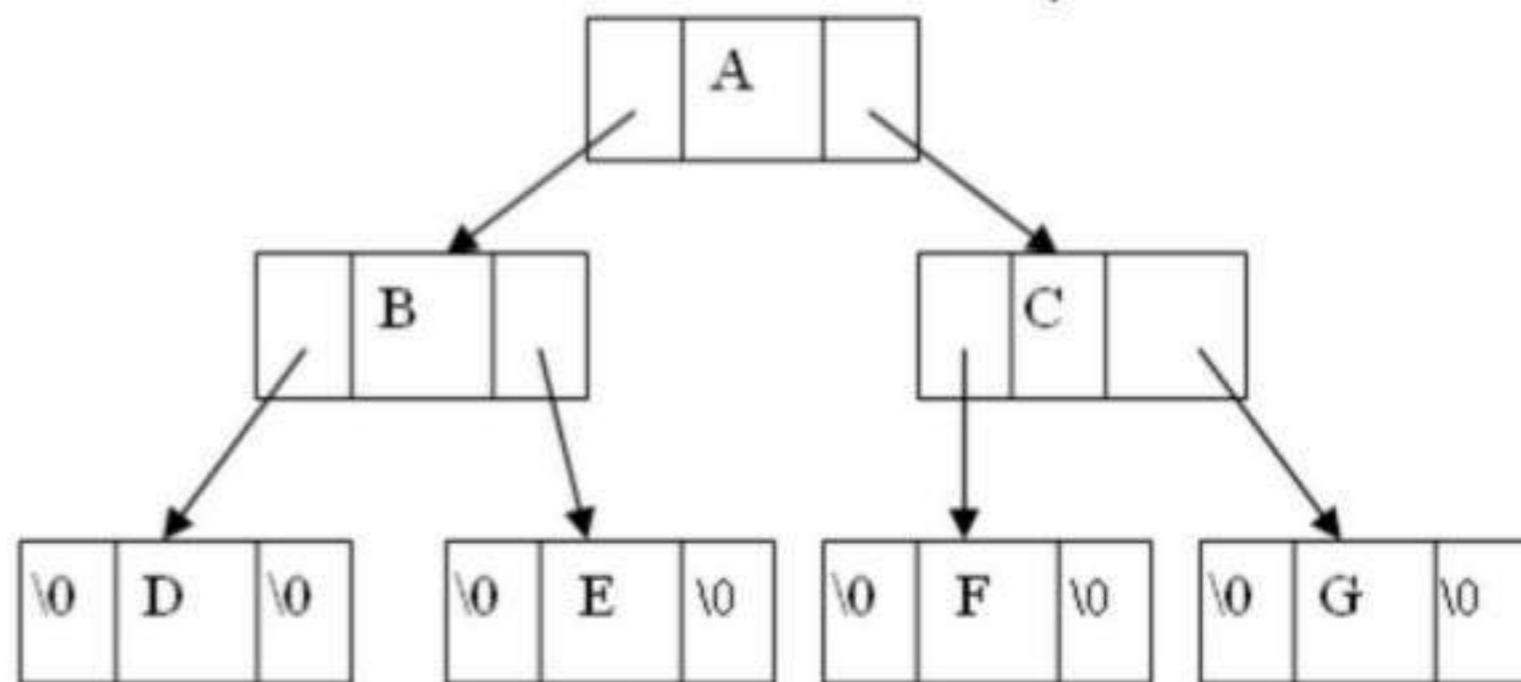


Fig: Structure of Binary tree

# Tree traversal

- Traversal is a process to visit all the nodes of a tree and may print their values too.
- All nodes are connected via edges (links) we always start from the root (head) node.
- There are three ways which we use to traverse a tree
  - In-order Traversal
  - Pre-order Traversal
  - Post-order Traversal
- Generally we traverse a tree to search or locate given item or key in the tree or to print all the values it contains.

## Pre-order, In-order, Post-order

- Pre-order

**<root><left><right>**

- In-order

**<left><root><right>**

- Post-order

**<left><right><root>**

# Pre-order Traversal

- The preorder traversal of a nonempty binary tree is defined as follows:
  - Visit the root node
  - Traverse the left sub-tree in preorder
  - Traverse the right sub-tree in preorder

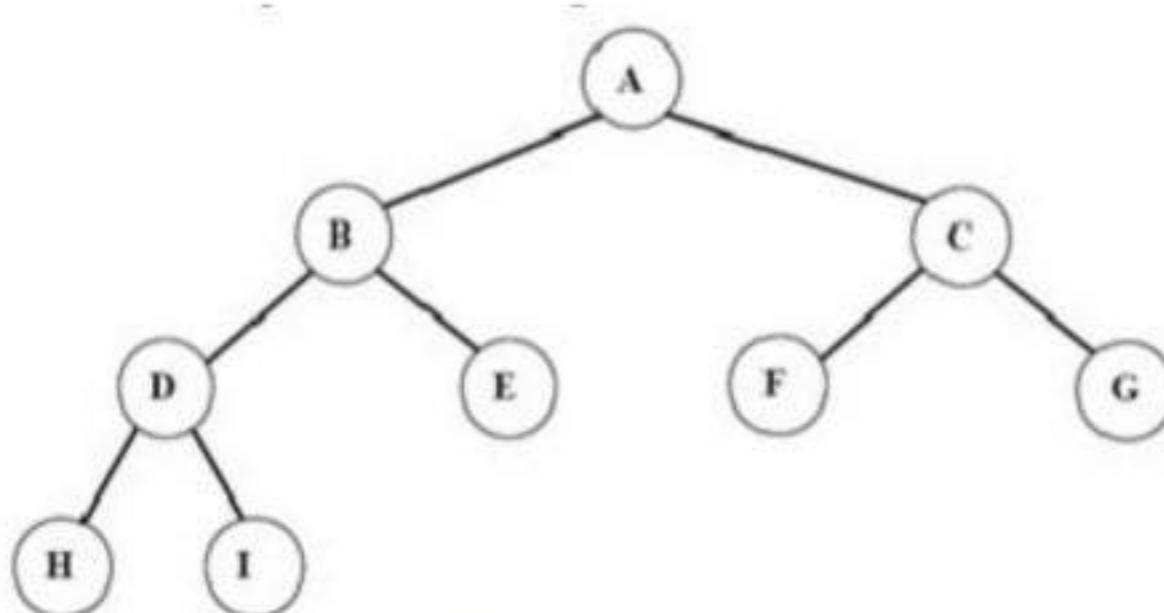


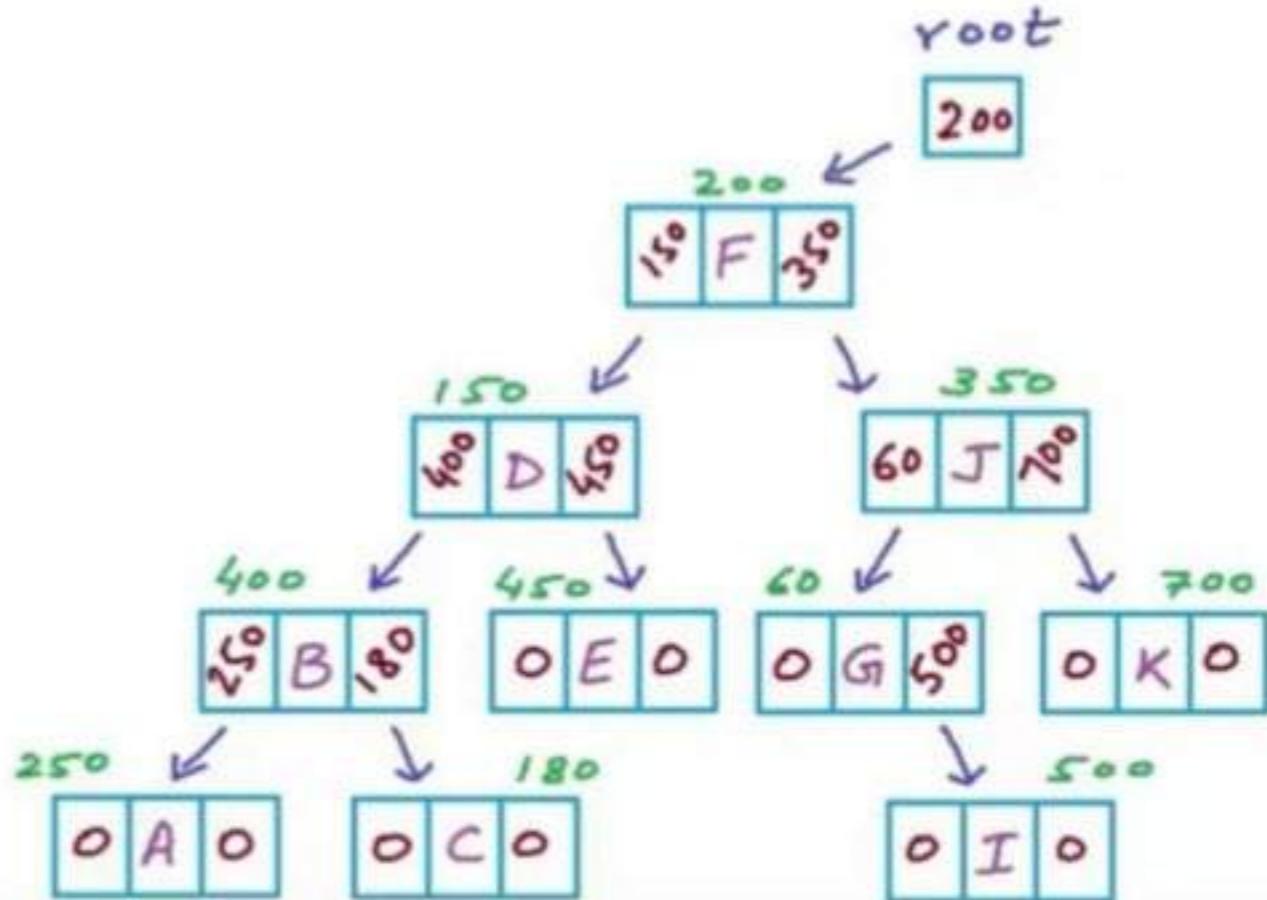
fig Binary tree

The preorder traversal output of the given tree is: A B D H I E C F G

The preorder is also known as depth first order.

# Pre-order Pseudocode

```
struct Node{  
    char data;  
    Node *left;  
    Node *right;  
}  
void Preorder(Node *root)  
{  
    if (root==NULL) return;  
    printf ("%c", root->data);  
    Preorder(root->left);  
    Preorder(root->right);  
}
```



# In-order traversal

- The in-order traversal of a nonempty binary tree is defined as follows:
  - Traverse the left sub-tree in in-order
  - Visit the root node
  - Traverse the right sub-tree in inorder
- The in-order traversal output of the given tree is  
H D I B E A F C G

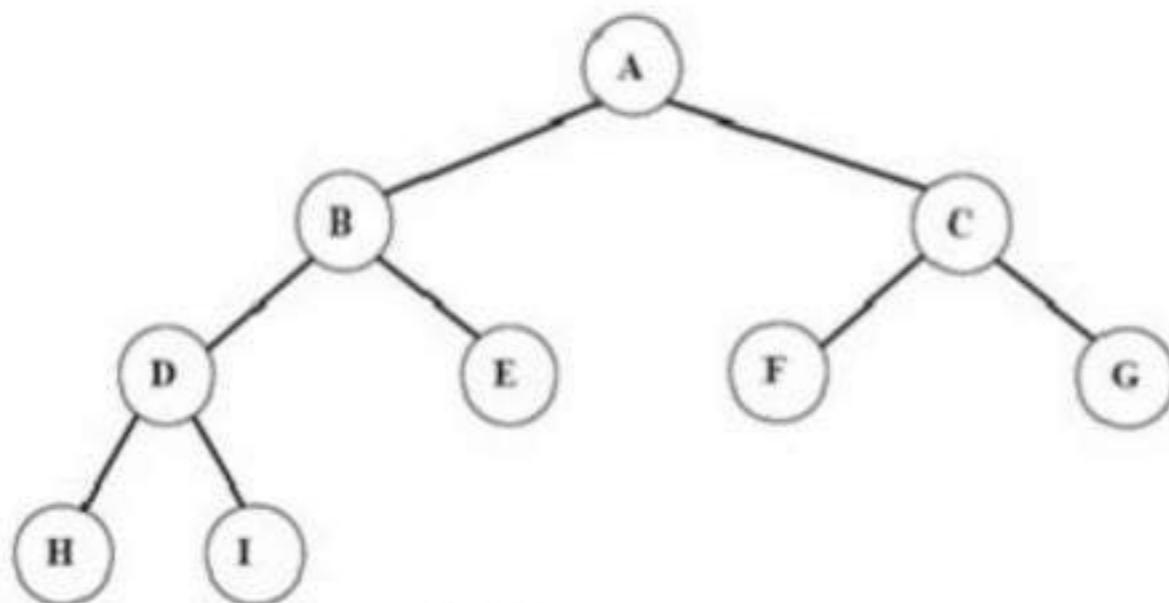
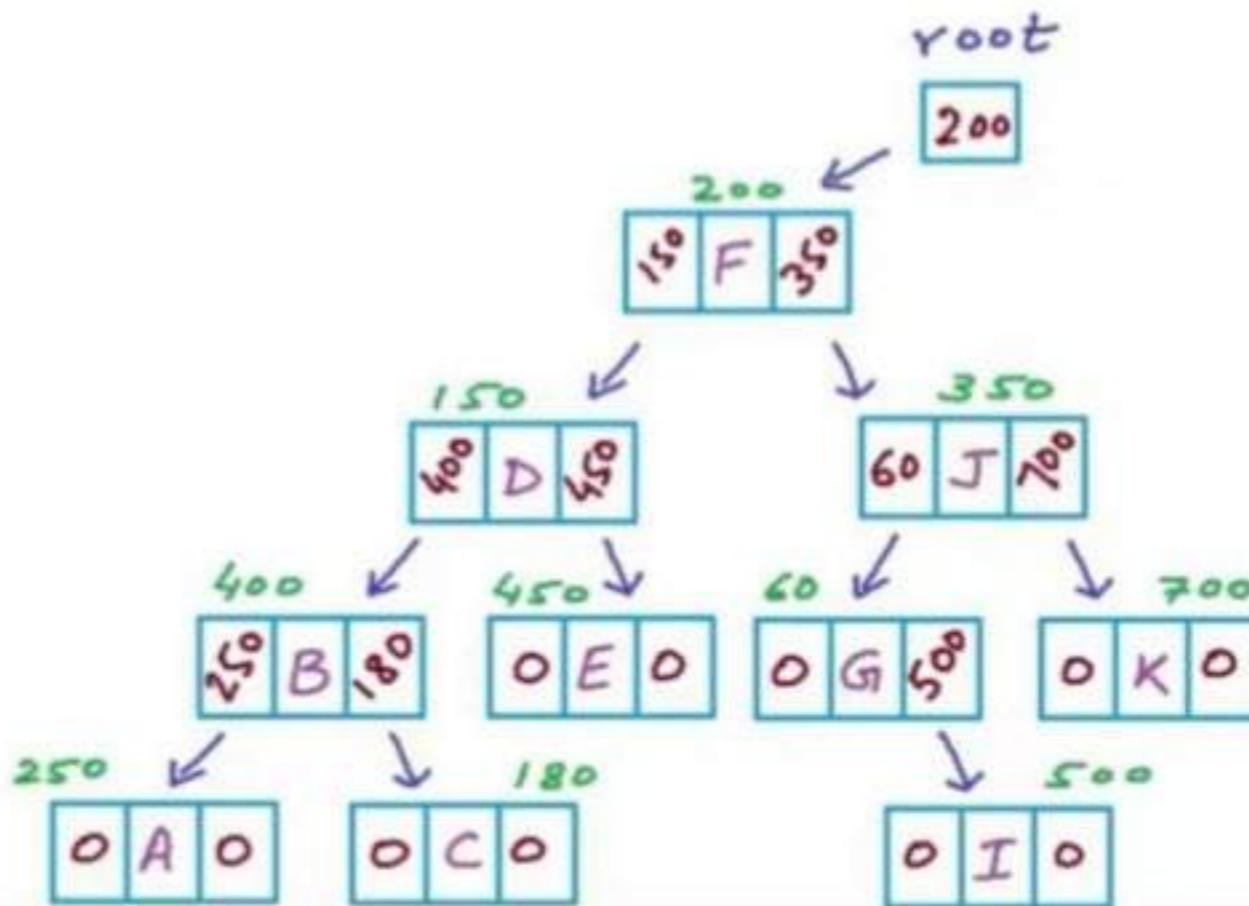


fig Binary tree

# In-order Pseudocode

```
struct Node{  
    char data;  
    Node *left;  
    Node *right;  
}  
void Inorder(Node *root)  
{  
    if (root==NULL) return;  
    Inorder(root->left);  
    printf ("%c", root->data);  
    Inorder(root->right);  
}
```



# Post-order traversal

- The in-order traversal of a nonempty binary tree is defined as follows:
  - Traverse the left sub-tree in post-order
  - Traverse the right sub-tree in post-order
  - Visit the root node
- The in-order traversal output of the given tree is  
H I D E B F G C A

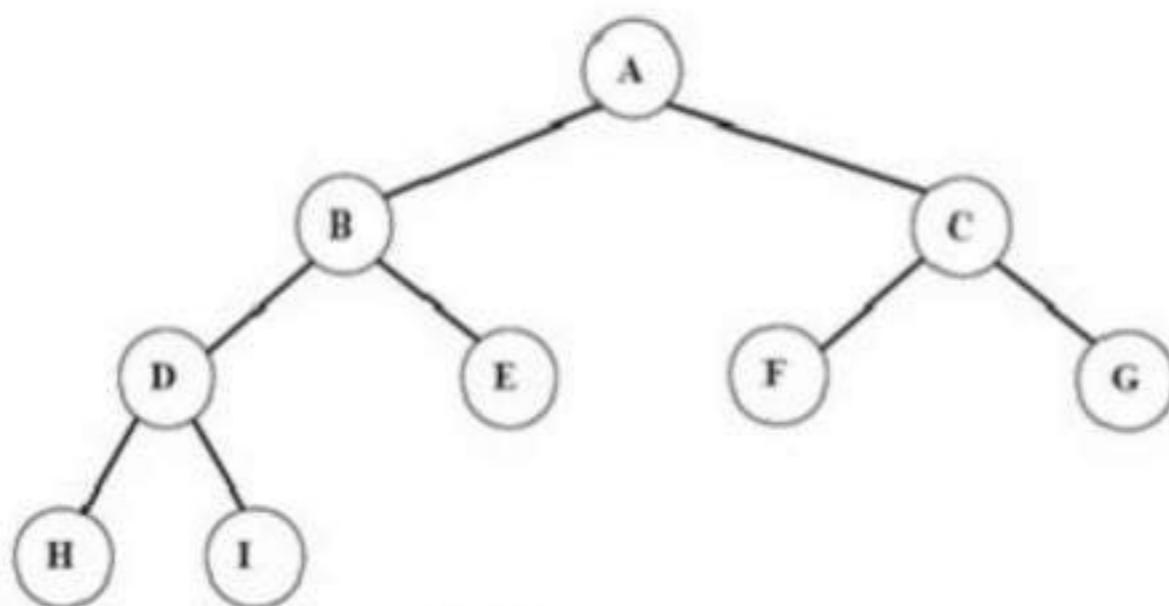
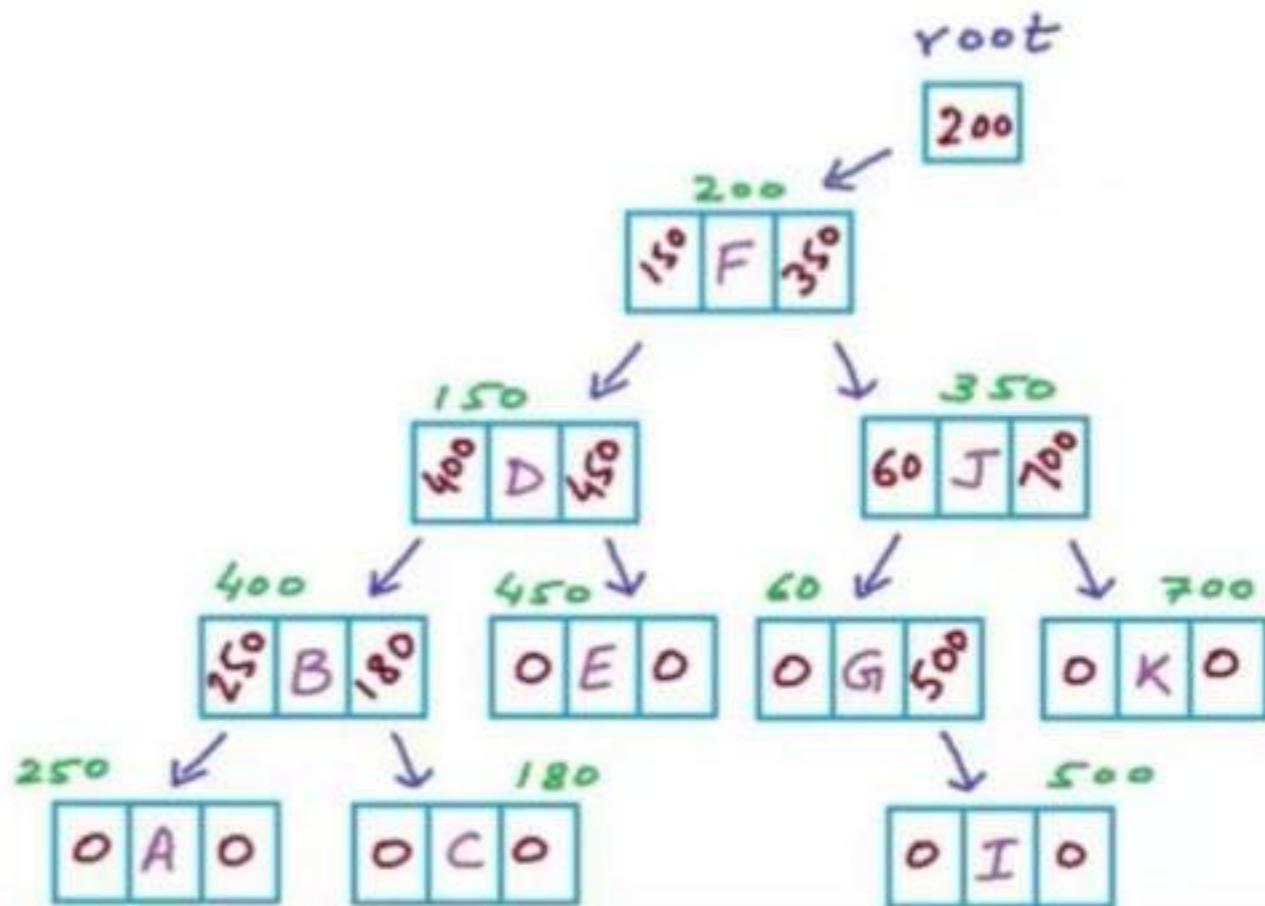


fig Binary tree

# Post-order Pseudocode

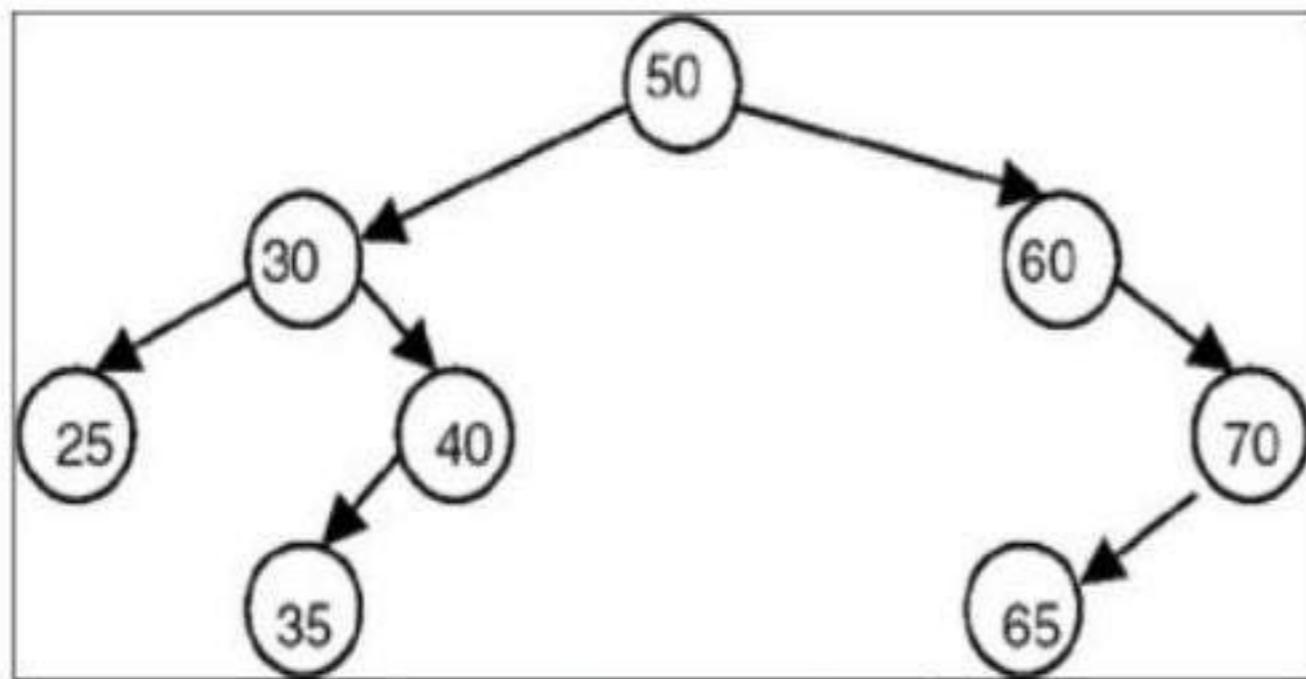
```
struct Node{  
    char data;  
    Node *left;  
    Node *right;  
}  
void Postorder(Node *root)  
{  
    if (root==NULL) return;  
    Postorder(root->left);  
    Postorder(root->right);  
    printf ("%c", root->data);  
}
```



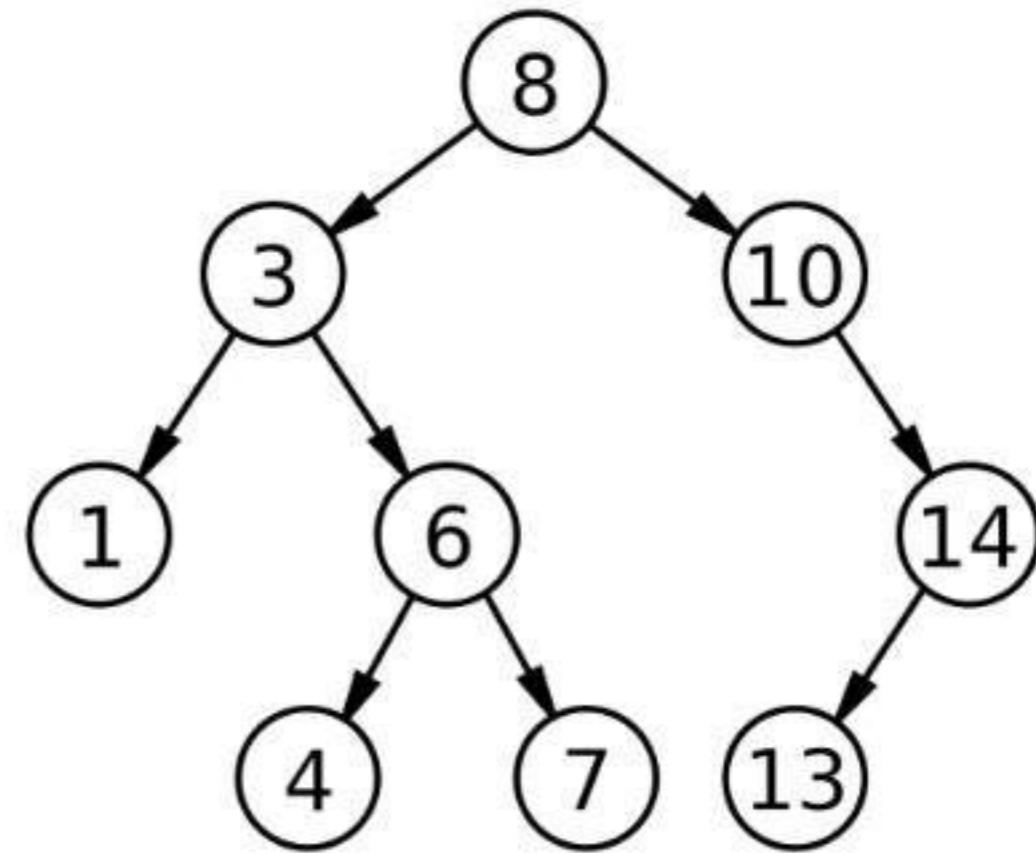
# Binary Search Tree(BST)

- A binary search tree (BST) is a binary tree that is either empty or in which every node contains a key (value) and satisfies the following conditions:
  - All keys in the left sub-tree of the root are smaller than the key in the root node
  - All keys in the right sub-tree of the root are greater than the key in the root node
  - The left and right sub-trees of the root are again binary search trees

# Binary Search Tree(BST)



The binary search tree.



# Binary Search Tree(BST)

- A binary search tree is basically a binary tree, and therefore it can be traversed in inorder, preorder and postorder.
- If we traverse a binary search tree in inorder and print the identifiers contained in the nodes of the tree, we get a sorted list of identifiers in ascending order.

# Why Binary Search Tree?

- Let us consider a problem of searching a list.
- If a list is ordered searching becomes faster if we use contiguous list(array).
- But if we need to make changes in the list, such as inserting new entries or deleting old entries, (**SLOWER!!!!**) because insertion and deletion in a contiguous list requires moving many of the entries every time.

# Why Binary Search Tree?

- So we may think of using a linked list because it permits insertion and deletion to be carried out by adjusting only few pointers.
- But in an n-linked list, there is no way to move through the list other than one node at a time, permitting only sequential access.
- Binary trees provide an excellent solution to this problem. By making the entries of an ordered list into the nodes of a binary search tree, we find that we can search for a key in  $O(\log n)$

# Binary Search Tree(BST)

Time Complexity			
	Array	Linked List	BST
Search	$O(n)$	$O(n)$	$O(\log n)$
Insert	$O(1)$	$O(1)$	$O(\log n)$
Remove	$O(n)$	$O(n)$	$O(\log n)$

# Operations on Binary Search Tree (BST)

- Following operations can be done in BST:
  - **Search( $k$ ,  $T$ ):** Search for key  $k$  in the tree  $T$ . If  $k$  is found in some node of tree then return true otherwise return false.
  - **Insert( $k$ ,  $T$ ):** Insert a new node with value  $k$  in the info field in the tree  $T$  such that the property of BST is maintained.
  - **Delete( $k$ ,  $T$ ):** Delete a node with value  $k$  in the info field from the tree  $T$  such that the property of BST is maintained.
  - **FindMin( $T$ ), FindMax( $T$ ):** Find minimum and maximum element from the given nonempty BST.

# Searching Through The BST

- Compare the target value with the element in the root node
  - ✓ If the target value is **equal**, the search is successful.
  - ✓ If target value is **less**, search the left subtree.
  - ✓ If target value is **greater**, search the right subtree.
  - ✓ If the subtree is **empty**, the search is unsuccessful.

## **C function for BST searching:**

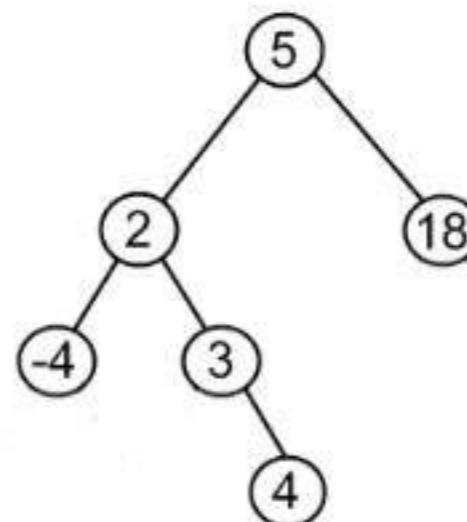
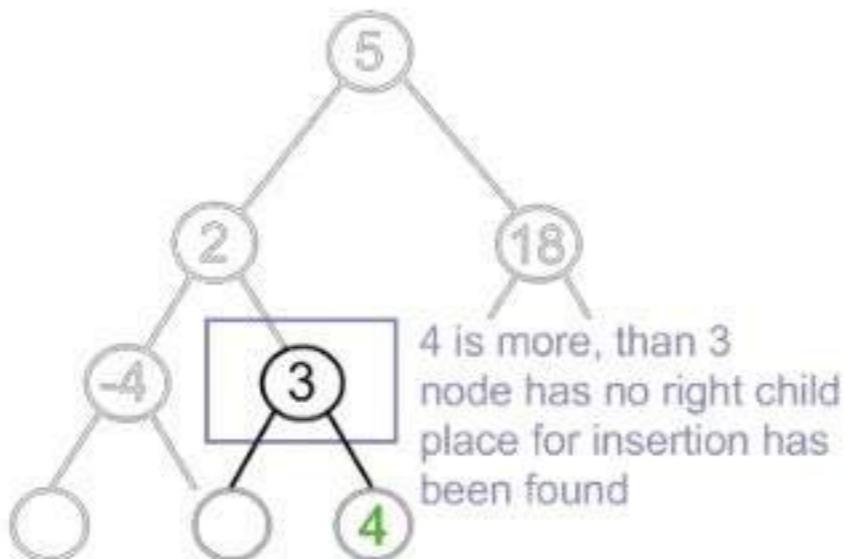
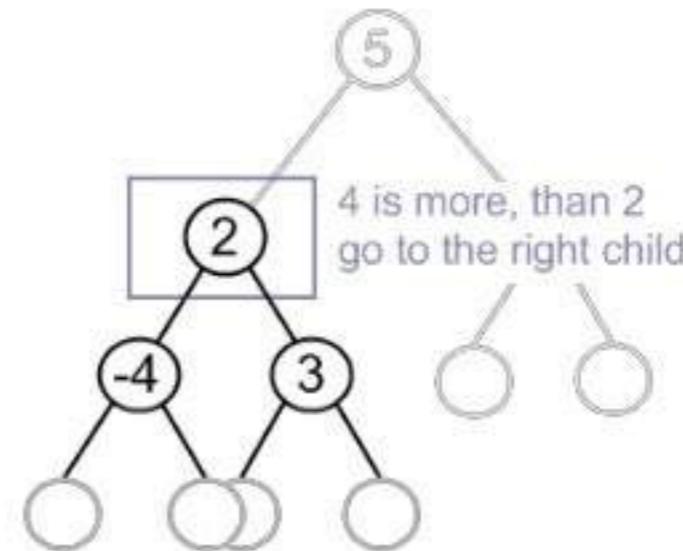
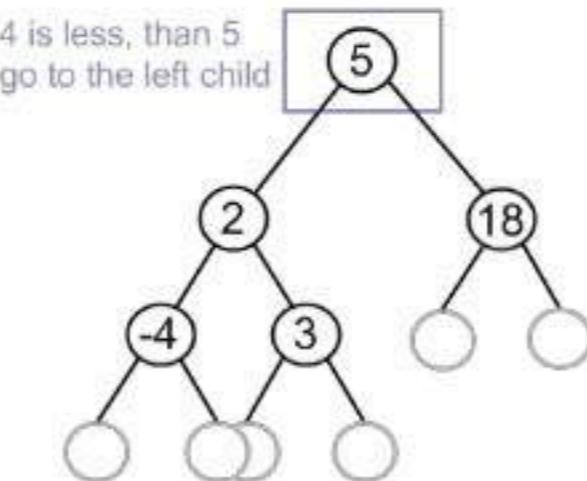
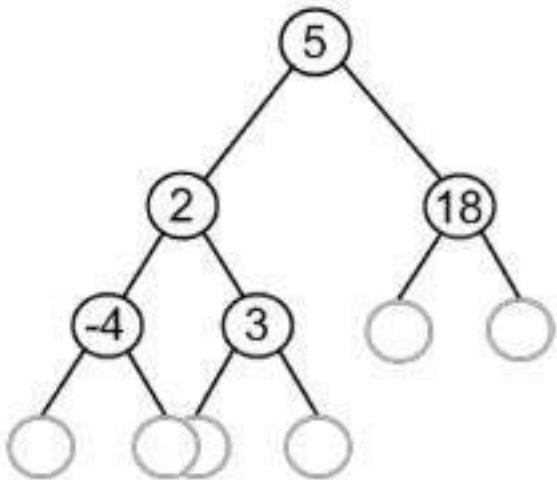
```
void BinSearch(struct bnode *root , int key)
{
    if(root == NULL)
    {
        printf("The number does not exist");
        exit(1);
    }
    else if (key == root->info)
    {
        printf("The searched item is found");
    }
    else if(key < root->info)
        return BinSearch(root->left, key);
    else
        return BinSearch(root->right, key);
}
```

## *Insertion of a node in BST*

- To insert a new item in a tree, we must first verify that its key is different from those of existing elements.
- If a new value is less, than the current node's value, go to the left subtree, else go to the right subtree.
- Following this simple rule, the algorithm reaches a node, which has no left or right subtree.
- By the moment a place for insertion is found, we can say for sure, that a new value has no duplicate in the tree.

# Algorithm for insertion in BST

- Check, whether value in current node and a new value are equal. If so, duplicate is found. Otherwise,
- if a new value is less, than the node's value:
  - if a current node has no left child, place for insertion has been found;
  - otherwise, handle the left child with the same algorithm.
- if a new value is greater, than the node's value:
  - if a current node has no right child, place for insertion has been found;
  - otherwise, handle the right child with the same algorithm.

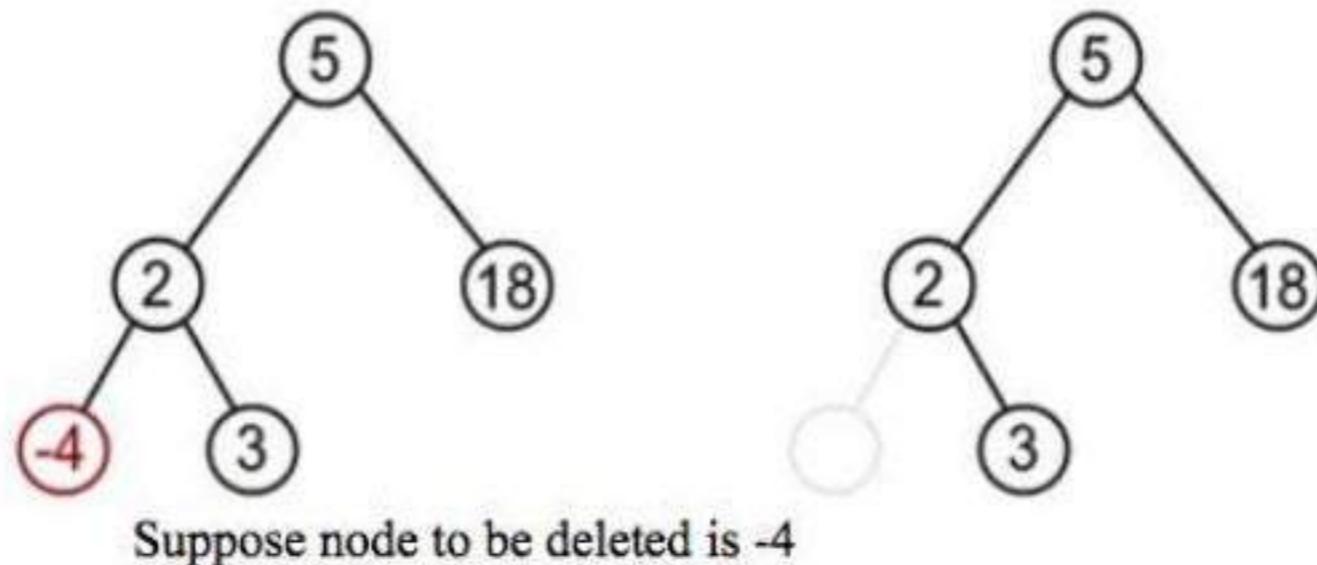


### **C function for BST insertion:**

```
void insert(struct bnode *root, int item)
{
    if(root=NULL)
    {
        root=(struct bnode*)malloc (sizeof(struct bnode));
        root->left=root->right=NULL;
        root->info=item;
    }
    else
    {
        if(item<root->info)
            root->left=insert(root->left, item);
        else
            root->right=insert(root->right, item);
    }
}
```

# Deleting a node from the BST

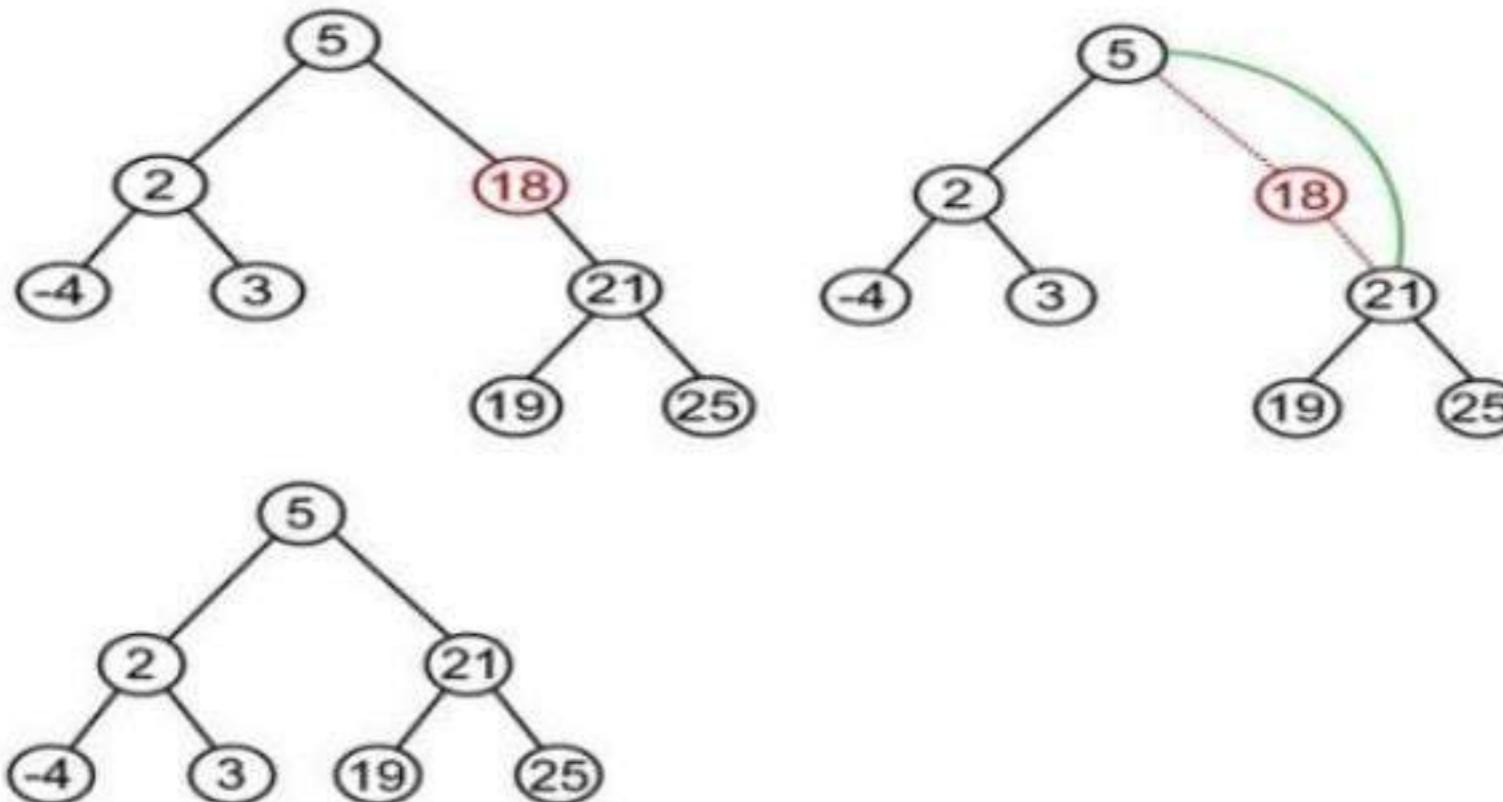
- While deleting a node from BST, there may be three cases:
  - The node to be deleted may be a leaf node:
    - In this case simply delete a node and set null pointer to its parents those side at which this deleted node exist.



# Deleting a node from the BST

2. The node to be deleted has one child

- In this case the child of the node to be deleted is appended to its parent node.
- Suppose node to be deleted is 18



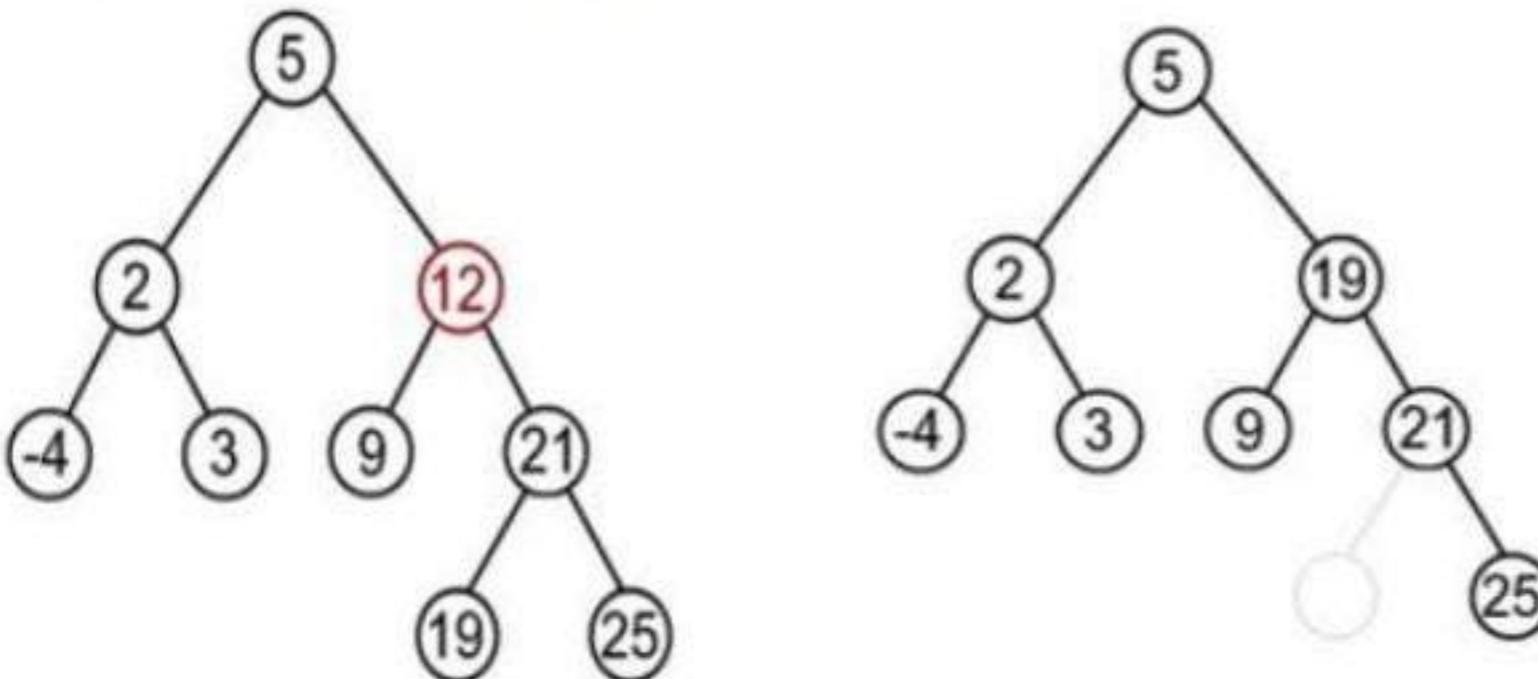
# Deleting a node from the BST

## 3. the node to be deleted has two children:

In this case node to be deleted is replaced by its in-order successor node.

OR

If the node to be deleted is either replaced by its right sub-trees leftmost node or its left sub-trees rightmost node.



Suppose node to be deleted is 12

Find minimum element in the right sub-tree of the node to be removed. In current example it is 19.

## General algorithm to delete a node from a BST:

1. start
2. if a node to be deleted is a leaf nod at left side then simply delete and set null pointer to it's parent's left pointer.
3. If a node to be deleted is a leaf node at right side then simply delete and set null pointer to it's parent's right pointer
4. if a node to be deleted has one child then connect it's child pointer with it's parent pointer and delete it from the tree
5. if a node to be deleted has two children then replace the node being deleted either by
  - a. right most node of it's left sub-tree or
  - b. left most node of it's right sub-tree.
6. End

## The deleteBST function:

```
struct bnode *delete(struct bnode *root, int item)
{
    struct bnode *temp;
    if(root==NULL)
    {
        printf("Empty tree");
        return;
    }
    else if(item<root->info)
        root->left=delete(root->left, item);
    else if(item>root->info)
        root->right=delete(root->right, item);
    else if(root->left!=NULL &&root->right!=NULL) //node has two child
    {
        temp=find_min(root->right);
        root->info=temp->info;
        root->right=delete(root->right, root->info);
    }
    else
    {
        temp=root;
        if(root->left==NULL)
            root=root->right;
        else if(root->right==NULL)
            root=root->left;
        free(temp);
    }
    return(temp);
}
*****find minimum element function*****
struct bnode *find_min(struct bnode *root)
{
    if(root==NULL)
        return0;
    else if(root->left==NULL)
        return root;
    else
        return(find_min(root->left));
}
```

## BFS algorithm

A standard BFS 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 algorithm works as follows:

1. Start by putting any one of the graph's vertices at the back of a queue.
2. Take the front item of the queue and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.
4. Keep repeating steps 2 and 3 until the queue is empty.

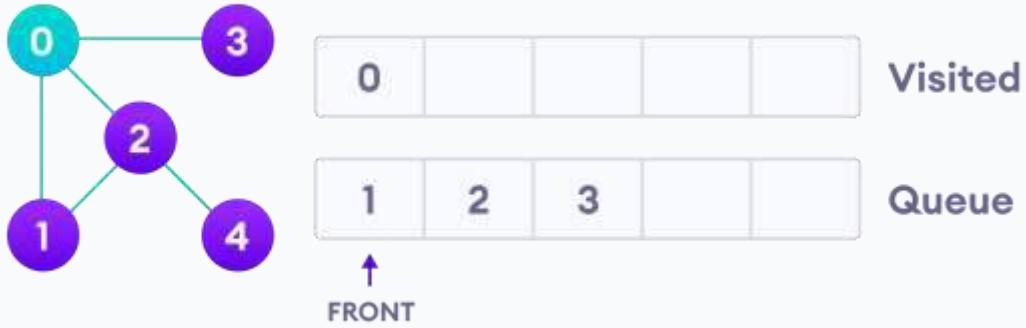
The graph might have two different disconnected parts so to make sure that we cover every vertex, we can also run the BFS algorithm on every node

## BFS example

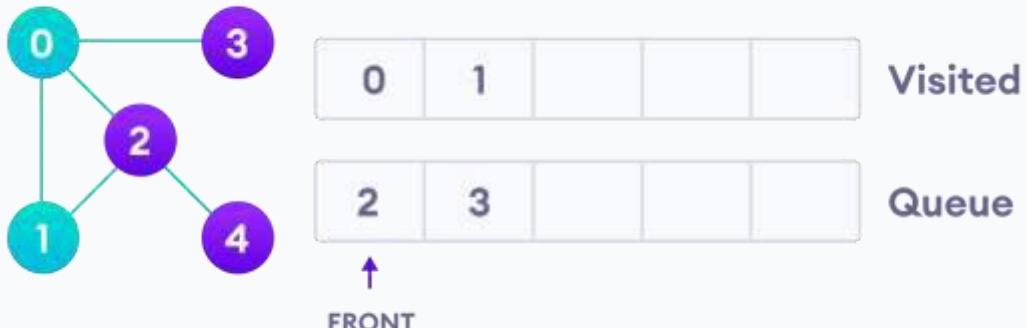
Let's see how the Breadth First Search algorithm works with an example. We use an undirected graph with 5 vertices.



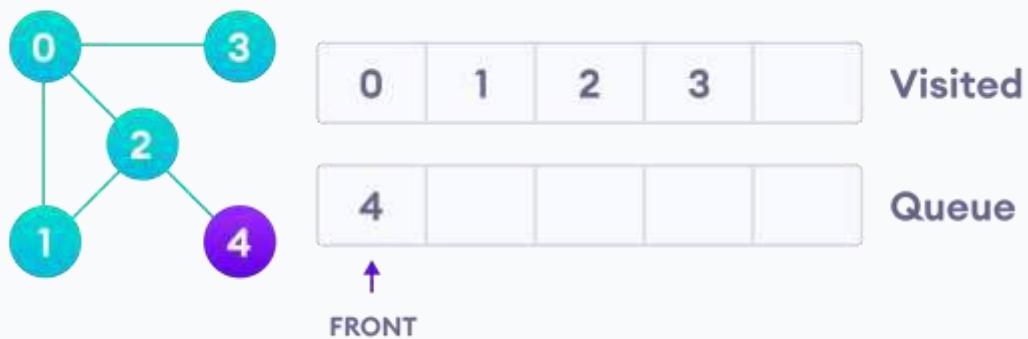
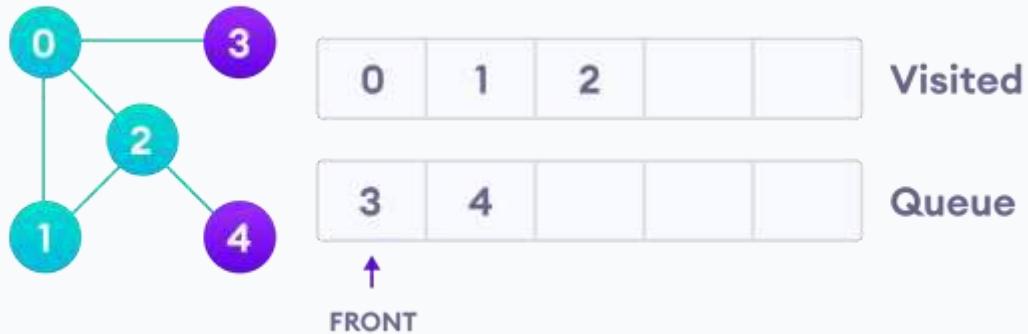
We start from vertex 0, the BFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.



Next, we visit the element at the front of queue i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.



Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the back of the queue and visit 3, which is at the front of the queue.



Only 4 remains in the queue since the only adjacent node of 3 i.e. 0 is already visited. We visit it.



## BFS pseudocode

```
create a queue Q  
mark v as visited and put v into Q  
while Q is non-empty  
    remove the head u of Q  
    mark and enqueue all (unvisited) neighbours of u
```

## Depth First Search Algorithm

A standard DFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

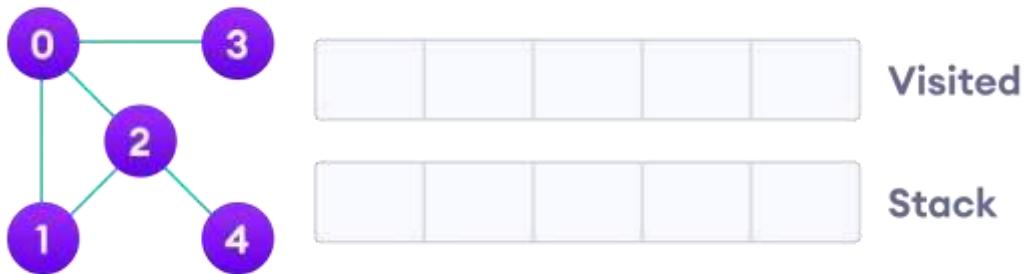
The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The DFS algorithm works as follows:

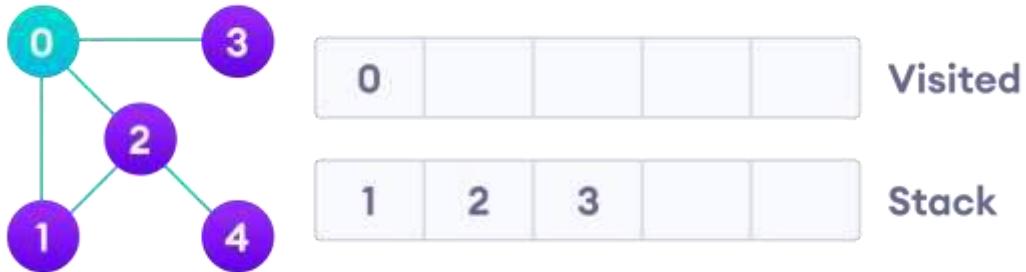
1. Start by putting any one of the graph's vertices on top of a stack.
2. Take the top item of the stack and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
4. Keep repeating steps 2 and 3 until the stack is empty.

## Depth First Search Example

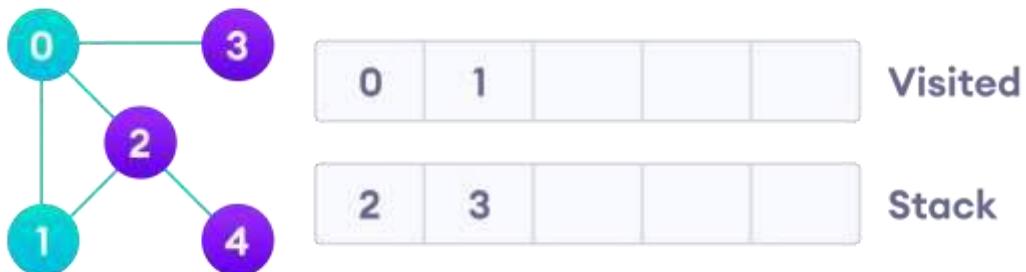
Let's see how the Depth First Search algorithm works with an example. We use an undirected graph with 5 vertices.



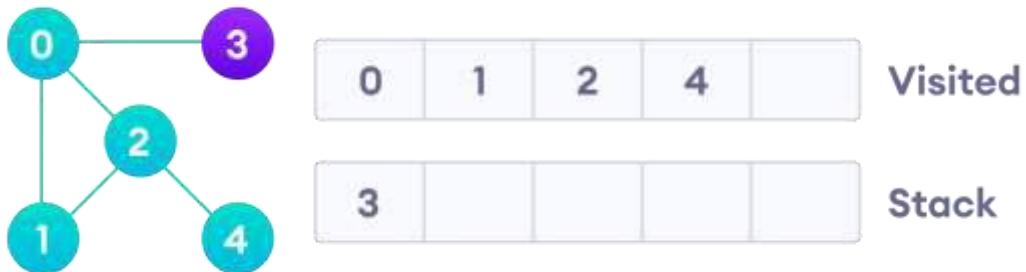
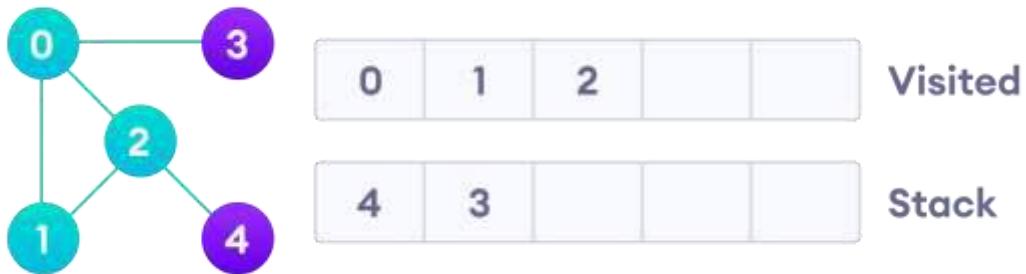
We start from vertex 0, the DFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.



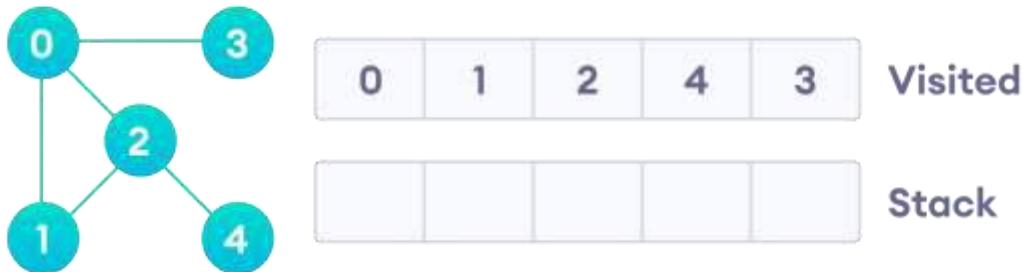
Next, we visit the element at the top of stack i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.



Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.



After we visit the last element 3, it doesn't have any unvisited adjacent nodes, so we have completed the Depth First Traversal of the graph.



## DFS Pseudocode (recursive implementation)

The pseudocode for DFS is shown below. In the `init()` function, notice that we run the DFS function on every node. This is because the graph might have two different disconnected parts so to make sure that we cover every vertex, we can also run the DFS algorithm on every node.

```
DFS(G, u)

    u.visited = true

    for each v ∈ G.Adj[u]

        if v.visited == false

            DFS(G,v)

init() {

    For each u ∈ G

        u.visited = false

    For each u ∈ G

        DFS(G, u)

}
```

# Kruskal's Algorithm

In this tutorial, you will learn how Kruskal's Algorithm works. Also, you will find working examples of Kruskal's Algorithm in C, C++, Java and Python.

Kruskal's algorithm is a [minimum spanning tree](#) algorithm that takes a graph as input and finds the subset of the edges of that graph which form a tree that includes every vertex has the minimum sum of weights among all the trees that can be formed from the graph

---

## How Kruskal's algorithm works

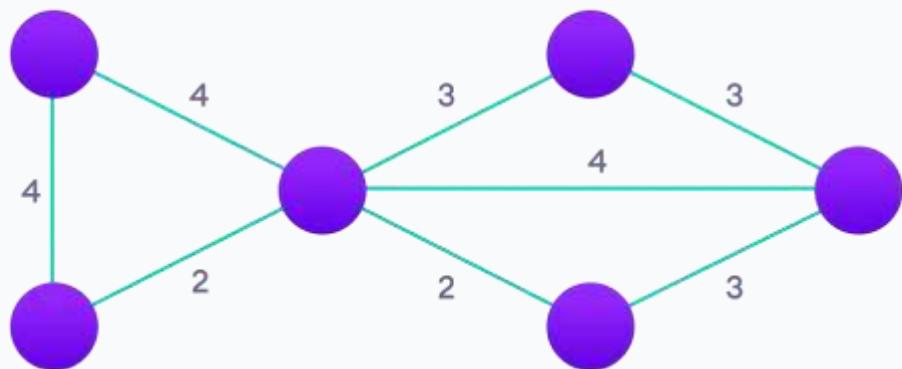
It falls under a class of algorithms called [greedy algorithms](#) that find the local optimum in the hopes of finding a global optimum.

We start from the edges with the lowest weight and keep adding edges until we reach our goal.

The steps for implementing Kruskal's algorithm are as follows:

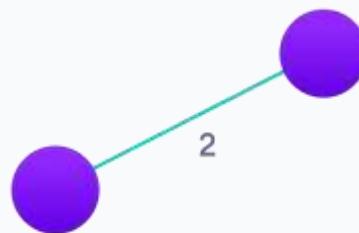
1. Sort all the edges from low weight to high
  2. Take the edge with the lowest weight and add it to the spanning tree. If adding the edge created a cycle, then reject this edge.
  3. Keep adding edges until we reach all vertices.
- 

## Example of Kruskal's algorithm



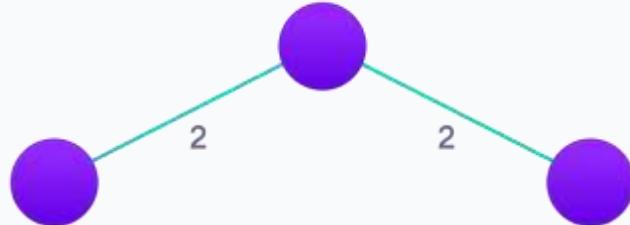
Step: 1

Start with a weighted graph



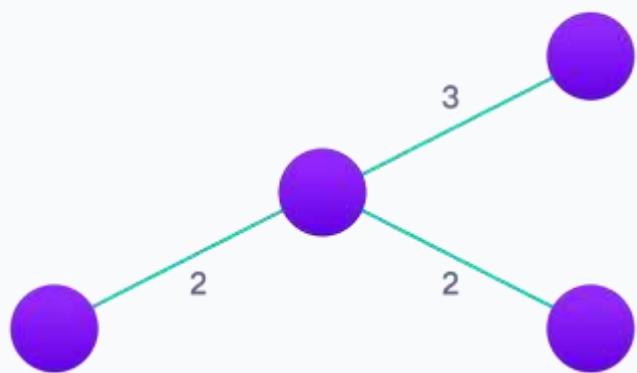
Step: 2

Choose the edge with the least weight, if there are more than 1, choose anyone



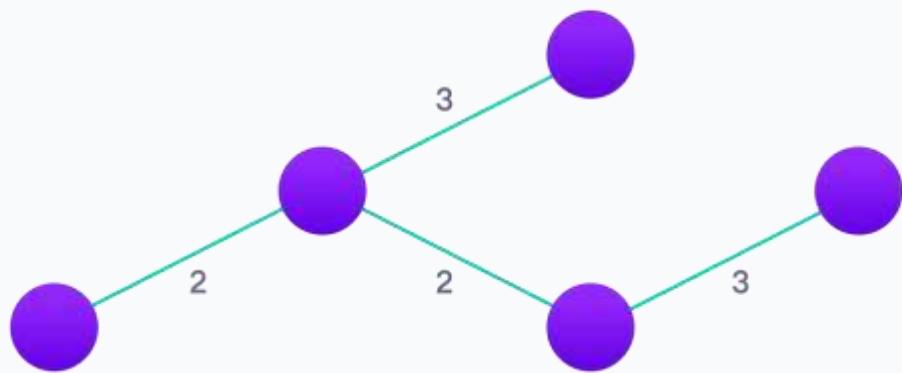
Step: 3

Choose the next shortest edge and add it



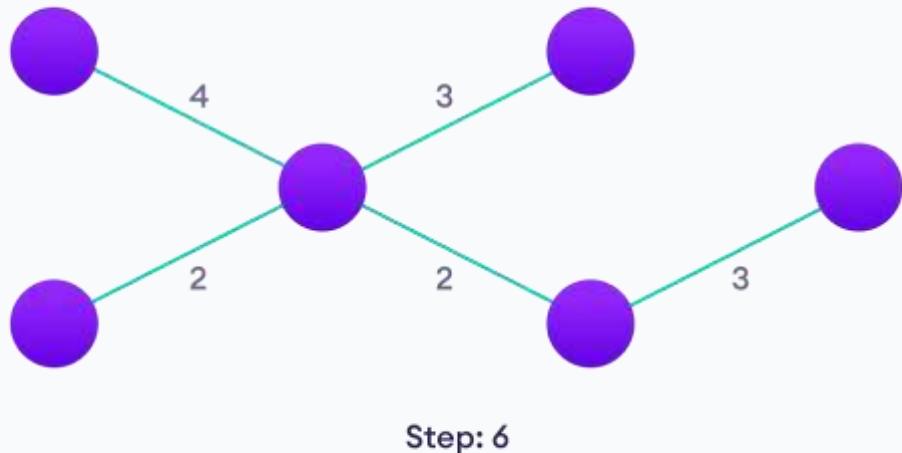
Step: 4

Choose the next shortest edge that doesn't create a cycle and add it



Step: 5

Choose the next shortest edge that doesn't create a cycle and add it



Repeat until you have a spanning tree

---

## Kruskal Algorithm Pseudocode

Any minimum spanning tree algorithm revolves around checking if adding an edge creates a loop or not.

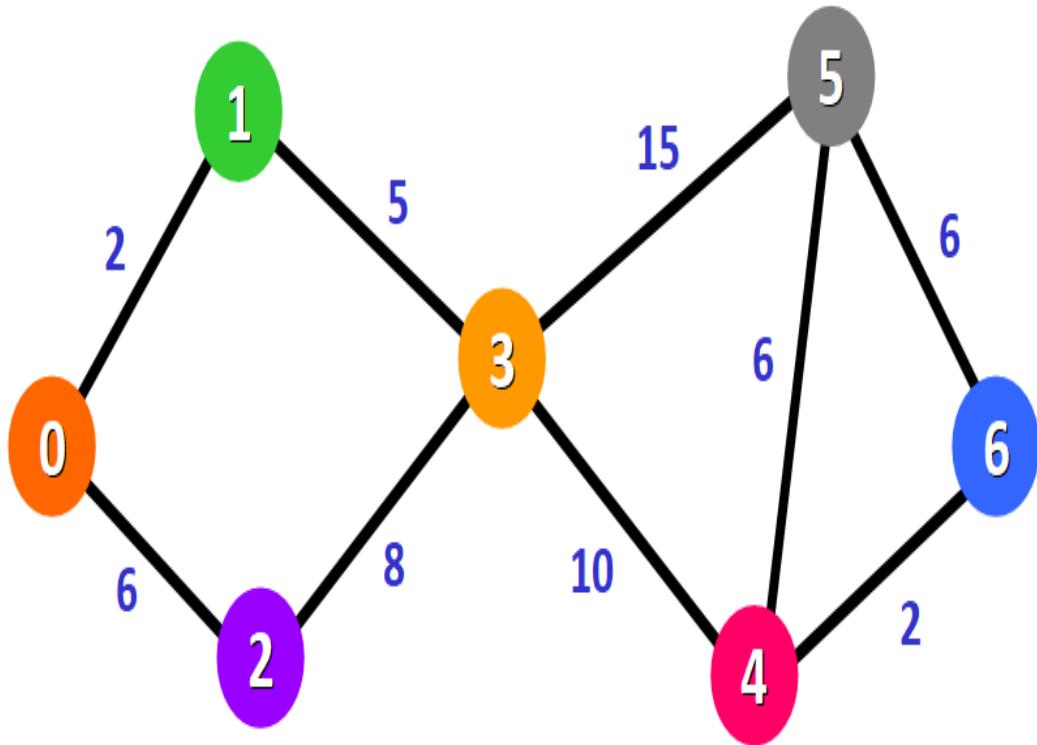
The most common way to find this out is an algorithm called [Union FInd](#). The Union-Find algorithm divides the vertices into clusters and allows us to check if two vertices belong to the same cluster or not and hence decide whether adding an edge creates a cycle.

```

KRUSKAL(G):
A = ∅
For each vertex v ∈ G.V:
    MAKE-SET(v)
For each edge (u, v) ∈ G.E ordered by increasing order by weight(u, v):
    if FIND-SET(u) ≠ FIND-SET(v):
        A = A ∪ {(u, v)}
        UNION(u, v)
return A

```

We have this graph:



The algorithm will generate the shortest path from node 0 to all the other nodes in the graph.

**Tip:** For this graph, we will assume that the weight of the edges represents the distance between two nodes.

We will have the shortest path from node 0 to node 1, from node 0 to node 2, from node 0 to node 3, and so on for every node in the graph.

Initially, we have this list of distances (please see the list below):

- The distance from the source node to itself is 0. For this example, the source node will be node 0 but it can be any node that you choose.
- The distance from the source node to all other nodes has not been determined yet, so we use the infinity symbol to represent this initially.

## Distance:

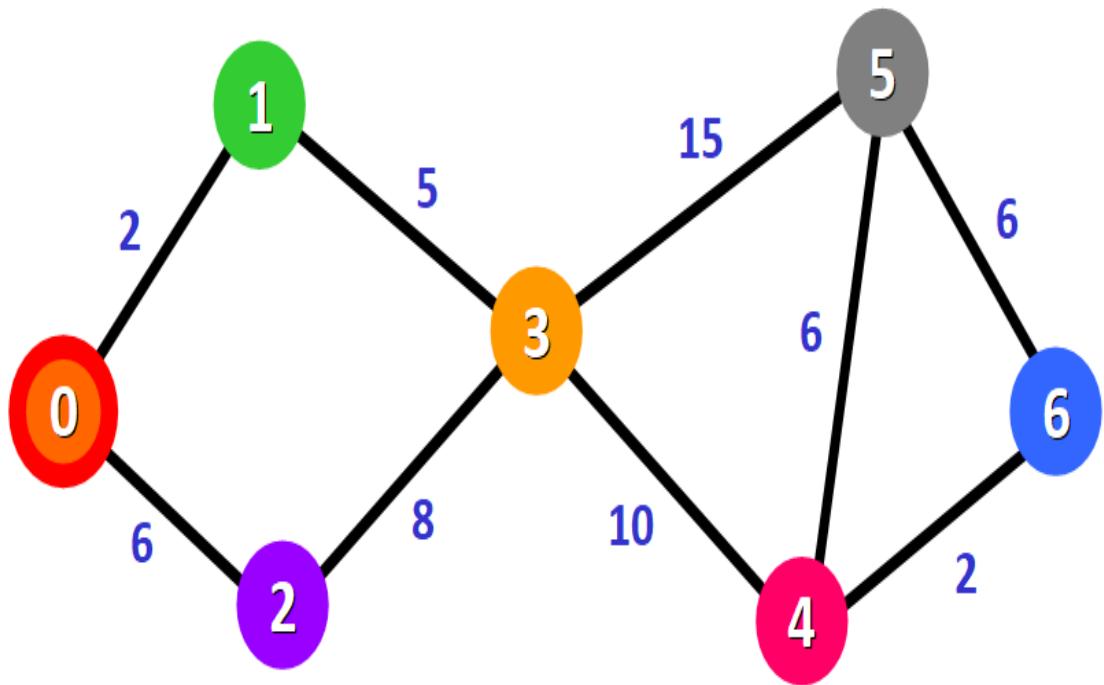
<b>0:</b>	0
<b>1:</b>	$\infty$
<b>2:</b>	$\infty$
<b>3:</b>	$\infty$
<b>4:</b>	$\infty$
<b>5:</b>	$\infty$
<b>6:</b>	$\infty$

We also have this list (see below) to keep track of the nodes that have not been visited yet (nodes that have not been included in the path):

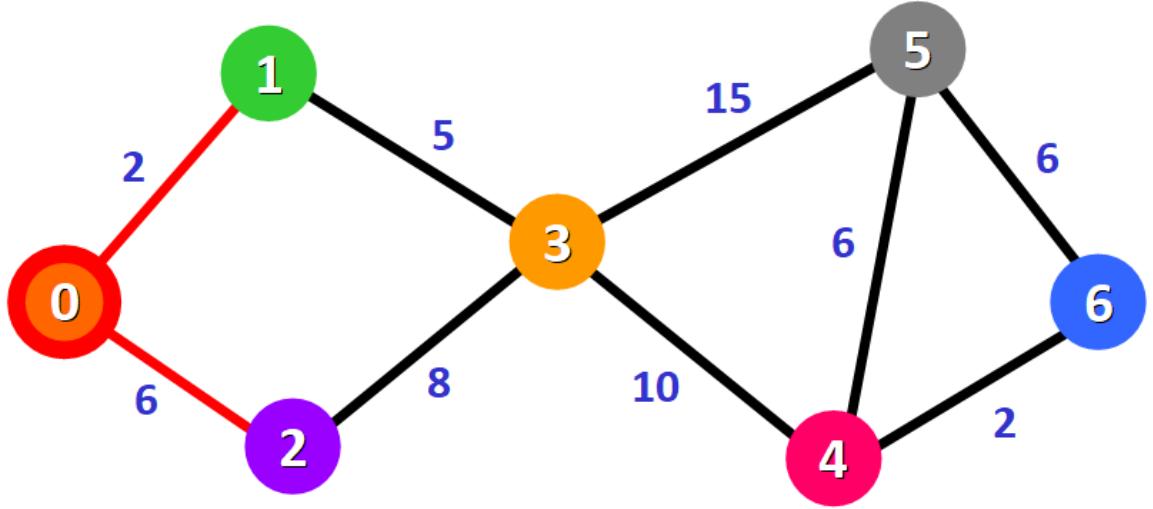
Unvisited Nodes: {0, 1, 2, 3, 4, 5, 6}

☞ **Tip:** Remember that the algorithm is completed once all nodes have been added to the path.

Since we are choosing to start at node 0, we can mark this node as visited. Equivalently, we cross it off from the list of unvisited nodes and add a red border to the corresponding node in diagram:



Now we need to start checking the distance from node 0 to its adjacent nodes. As you can see, these are nodes 1 and 2 (see the red edges):



💡 **Tip:** This doesn't mean that we are immediately adding the two adjacent nodes to the shortest path. Before adding a node to this path, we need to check if we have found the shortest path to reach it. We are simply making an initial examination process to see the options available.

We need to update the distances from node 0 to node 1 and node 2 with the weights of the edges that connect them to node 0 (the source node). These weights are 2 and 6, respectively:

## Distance:

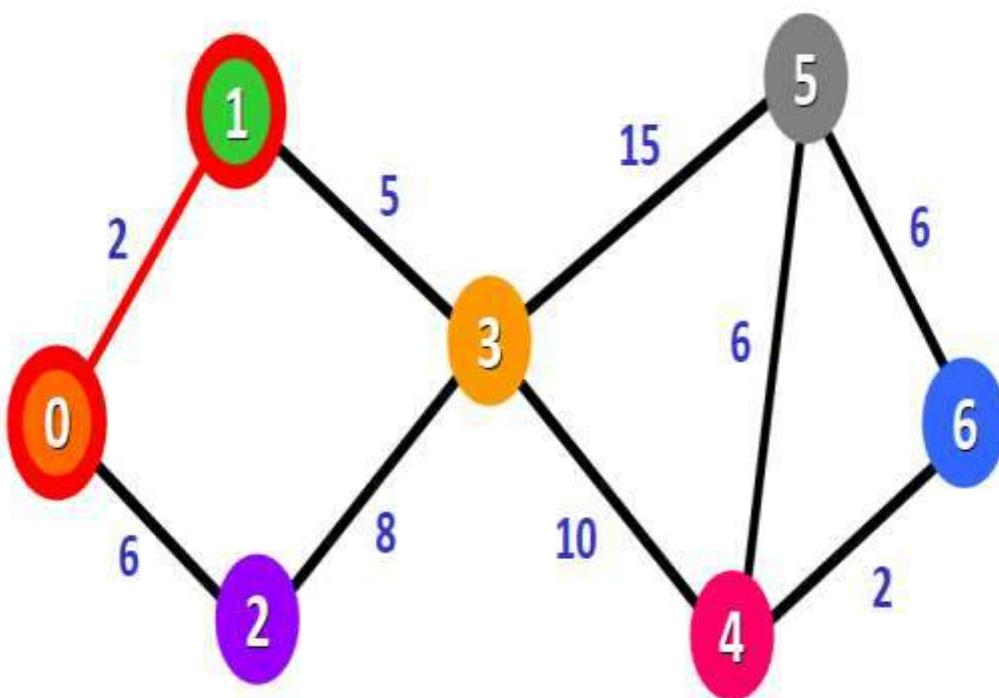
0:	0
1:	2
2:	6
3:	∞
4:	∞
5:	∞
6:	∞

After updating the distances of the adjacent nodes, we need to:

- Select the node that is closest to the source node based on the current known distances.
- Mark it as visited.
- Add it to the path.

If we check the list of distances, we can see that node 1 has the shortest distance to the source node (a distance of 2), so we add it to the path.

In the diagram, we can represent this with a red edge:

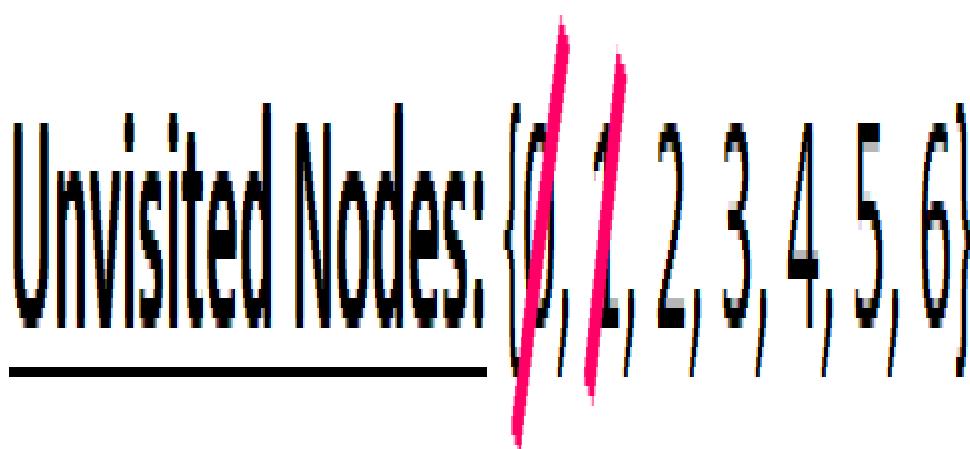


We mark it with a red square in the list to represent that it has been "visited" and that we have found the shortest path to this node:

## Distance:

<b>0:</b>	0	
<b>1:</b>	<del>∞</del>	2 -
<b>2:</b>	<del>∞</del>	6
<b>3:</b>	∞	
<b>4:</b>	∞	
<b>5:</b>	∞	
<b>6:</b>	∞	

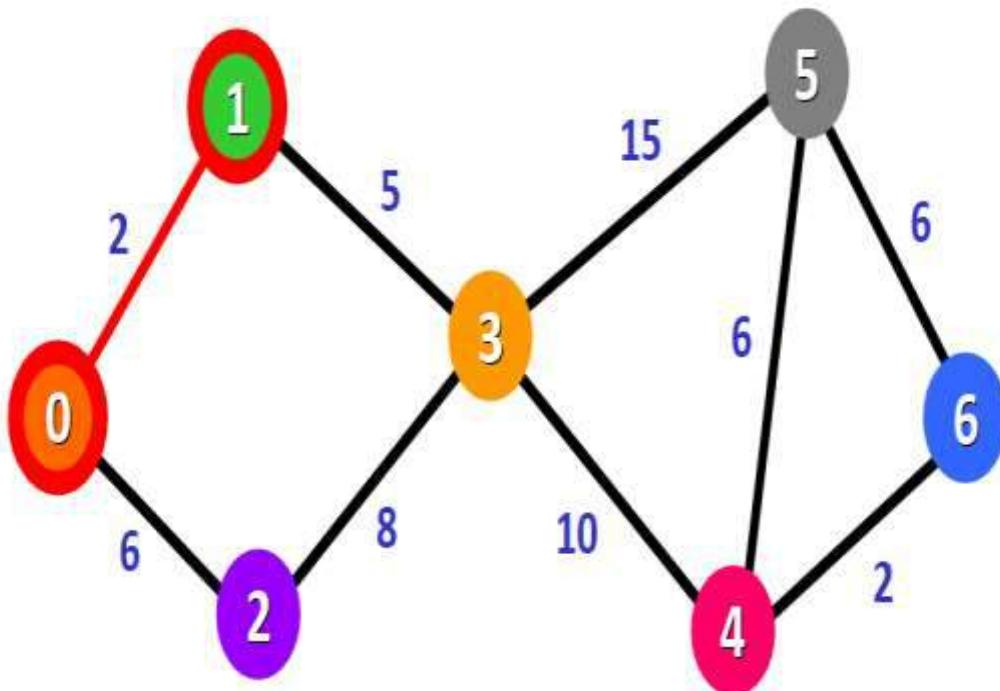
We cross it off from the list of unvisited nodes:



Now we need to analyze the new adjacent nodes to find the shortest path to reach them. We will only analyze the nodes that

are adjacent to the nodes that are already part of the shortest path (the path marked with red edges).

Node 3 and node 2 are both adjacent to nodes that are already in the path because they are directly connected to node 1 and node 0, respectively, as you can see below. These are the nodes that we will analyze in the next step.



Since we already have the distance from the source node to node 2 written down in our list, we don't need to update the distance this time. We only need to update the distance from the source node to the new adjacent node (node 3):

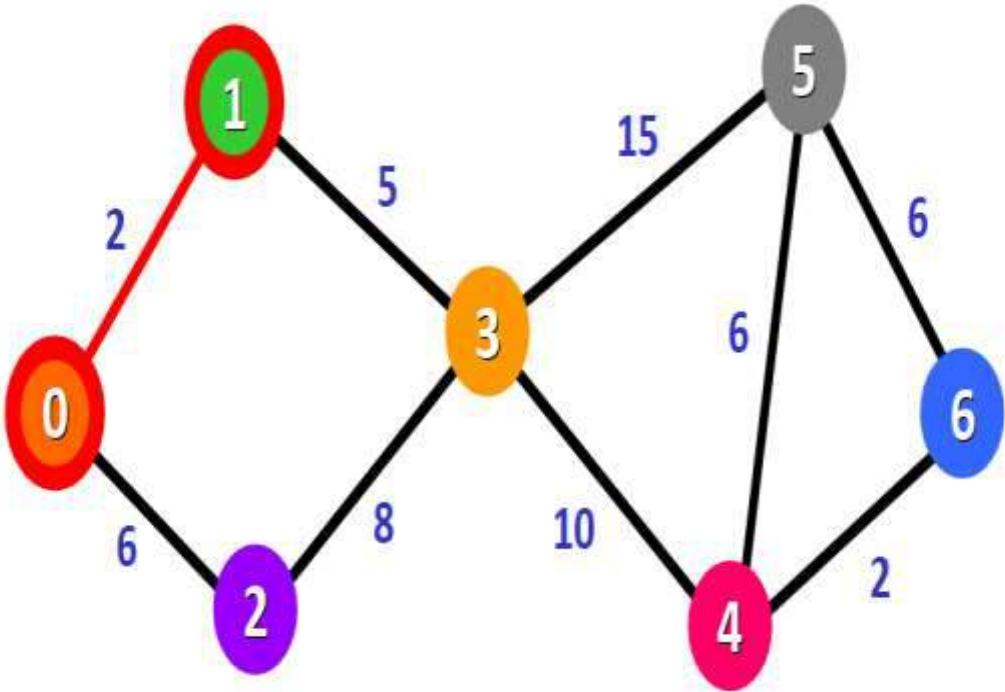
# Distance:

0:	0	
1:	<del>∞</del>	2 -
2:	<del>∞</del>	6
3:	<del>∞</del>	7
4:	∞	
5:	∞	
6:	∞	

This distance is 7. Let's see why.

To find the distance from the source node to another node (in this case, node 3), we add the weights of all the edges that form the shortest path to reach that node:

- **For node 3:** the total distance is 7 because we add the weights of the edges that form the path  $0 \rightarrow 1 \rightarrow 3$  (2 for the edge  $0 \rightarrow 1$  and 5 for the edge  $1 \rightarrow 3$ ).



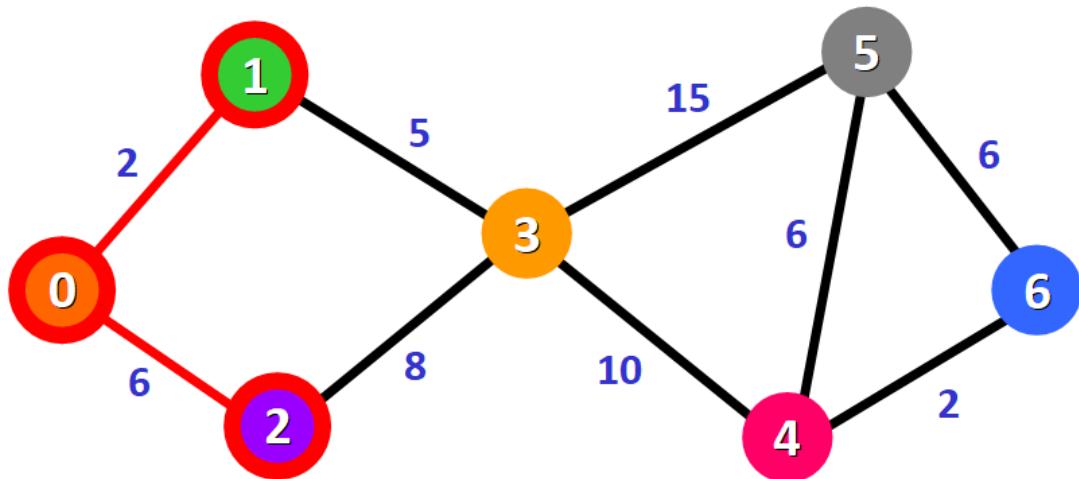
Now that we have the distance to the adjacent nodes, we have to choose which node will be added to the path. We must select the **unvisited** node with the shortest (currently known) distance to the source node.

From the list of distances, we can immediately detect that this is node 2 with distance **6**:

## Distance:

0:	0
1:	2.
2:	6
3:	7
4:	$\infty$
5:	$\infty$
6:	$\infty$

We add it to the path graphically with a red border around the node and a red edge:



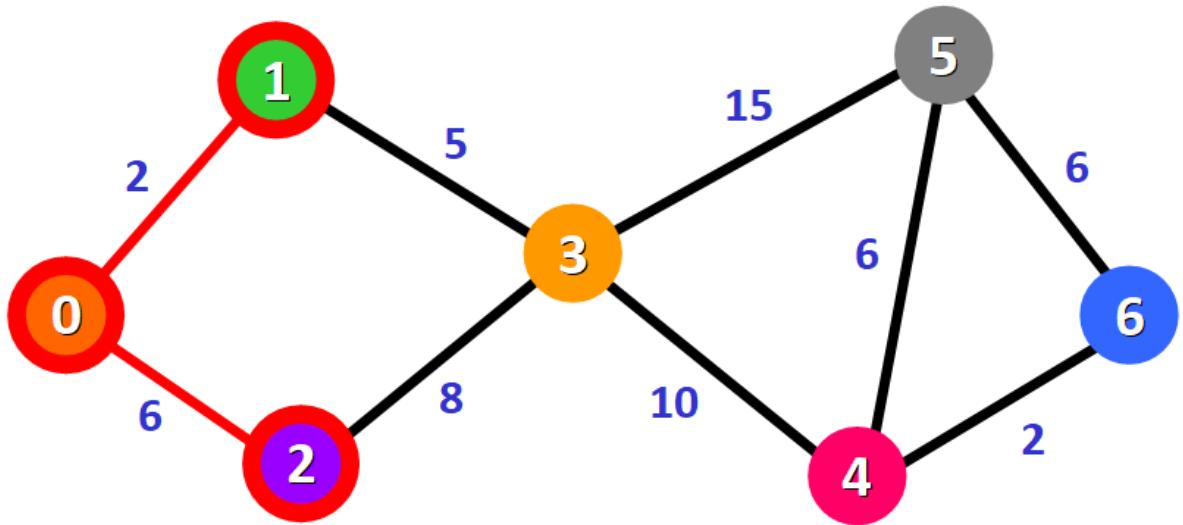
We also mark it as visited by adding a small red square in the list of distances and crossing it off from the list of unvisited nodes:

### Distance:

0:	0
1:	<del>∞</del> 2 ■
2:	<del>∞</del> 6 ■
3:	<del>∞</del> 7
4:	∞
5:	∞
6:	∞

Unvisited Nodes: {~~0~~, ~~1~~, ~~2~~, 3, 4, 5, 6}

Now we need to repeat the process to find the shortest path from the source node to the new adjacent node, which is node 3. You can see that we have two possible paths  $0 \rightarrow 1 \rightarrow 3$  or  $0 \rightarrow 2 \rightarrow 3$ . Let's see how we can decide which one is the shortest path.



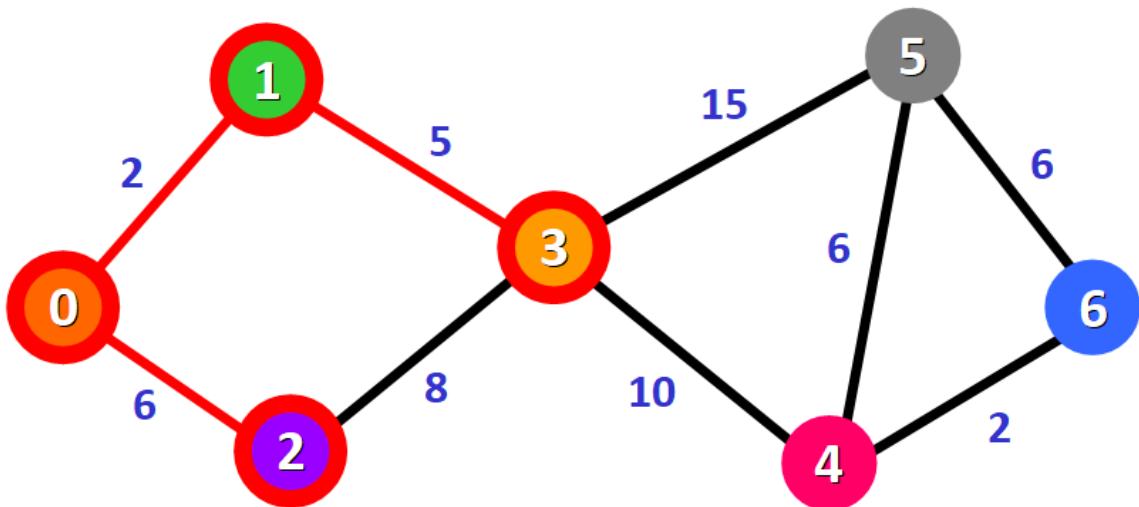
Node 3 already has a distance in the list that was recorded previously (7, see the list below). This distance was the result of a previous step, where we added the weights 5 and 2 of the two edges that we needed to cross to follow the path  $0 \rightarrow 1 \rightarrow 3$ . But now we have another alternative. If we choose to follow the path  $0 \rightarrow 2 \rightarrow 3$ , we would need to follow two edges  $0 \rightarrow 2$  and  $2 \rightarrow 3$  with weights 6 and 8, respectively, which represents a total distance of 14.

## Distance:

0:	0
1:	<del>∞</del> 2 ■
2:	<del>∞</del> 6 ■
3:	<del>∞</del> 7 from $(5 + 2)$ vs. 14 from $(6 + 8)$
4:	$\infty$
5:	$\infty$
6:	$\infty$

Clearly, the first (existing) distance is shorter (7 vs. 14), so we will choose to keep the original path  $0 \rightarrow 1 \rightarrow 3$ . **We only update the distance if the new path is shorter.**

Therefore, we add this node to the path using the first alternative:  $0 \rightarrow 1 \rightarrow 3$ .



We mark this node as visited and cross it off from the list of unvisited nodes:

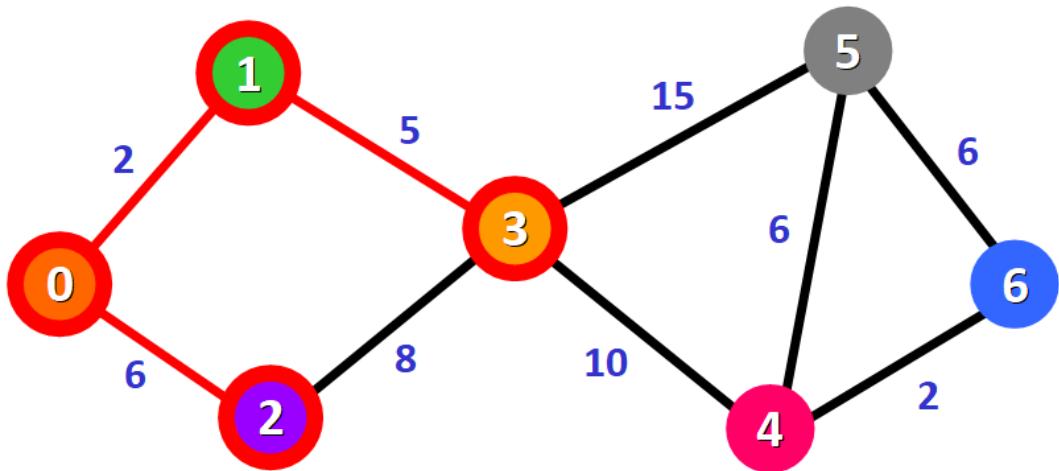
### Distance:

0:	0
1:	<del>∞</del> 2 ■
2:	<del>∞</del> 6 ■
3:	<del>∞</del> 7 ■
4:	$\infty$
5:	$\infty$
6:	$\infty$

Unvisited Nodes: {~~0, 1, 2, 3~~, 4, 5, 6}

Now we repeat the process again.

We need to check the new adjacent nodes that we have not visited so far. This time, these nodes are node 4 and node 5 since they are adjacent to node 3.



We update the distances of these nodes to the source node, always trying to find a shorter path, if possible:

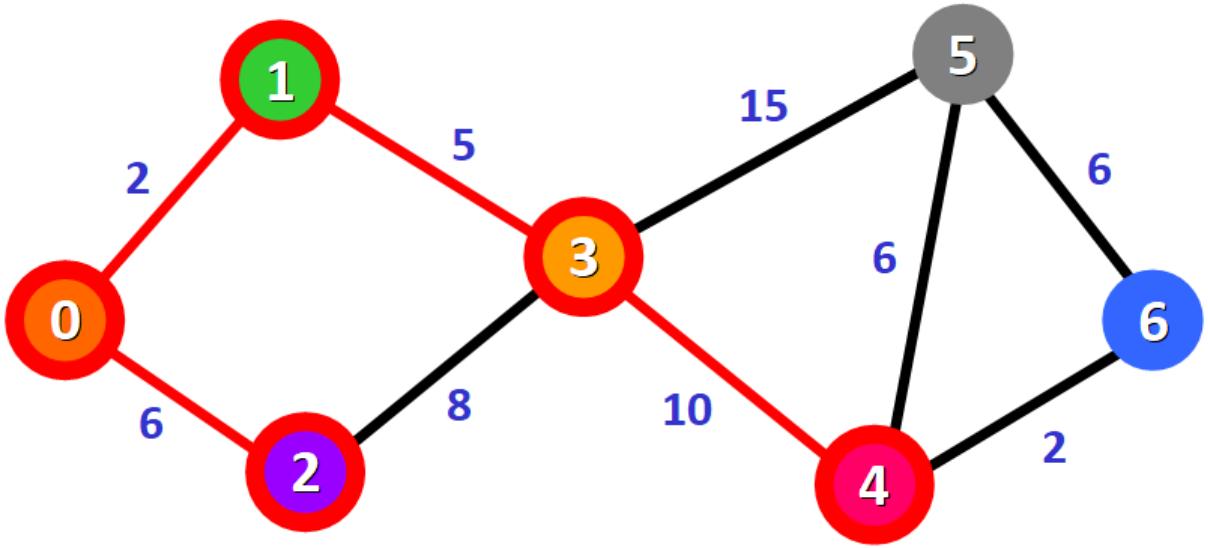
- **For node 4:** the distance is **17** from the path  $0 \rightarrow 1 \rightarrow 3 \rightarrow 4$ .
  - **For node 5:** the distance is **22** from the path  $0 \rightarrow 1 \rightarrow 3 \rightarrow 5$ .
- ⚠ **Tip:** Notice that we can only consider extending the shortest path (marked in red). We cannot consider paths that will take us through edges that have not been added to the shortest path (for example, we cannot form a path that goes through the edge  $2 \rightarrow 3$ ).

## Distance:

0:	0
1:	2.
2:	6.
3:	7.
4:	17 from $(2 + 5 + 10)$
5:	22 from $(2 + 5 + 15)$
6:	$\infty$

We need to choose which unvisited node will be marked as visited now. In this case, it's node 4 because it has the shortest

distance in the list of distances. We add it graphically in the diagram:



We also mark it as "visited" by adding a small red square in the list:

### Distance:

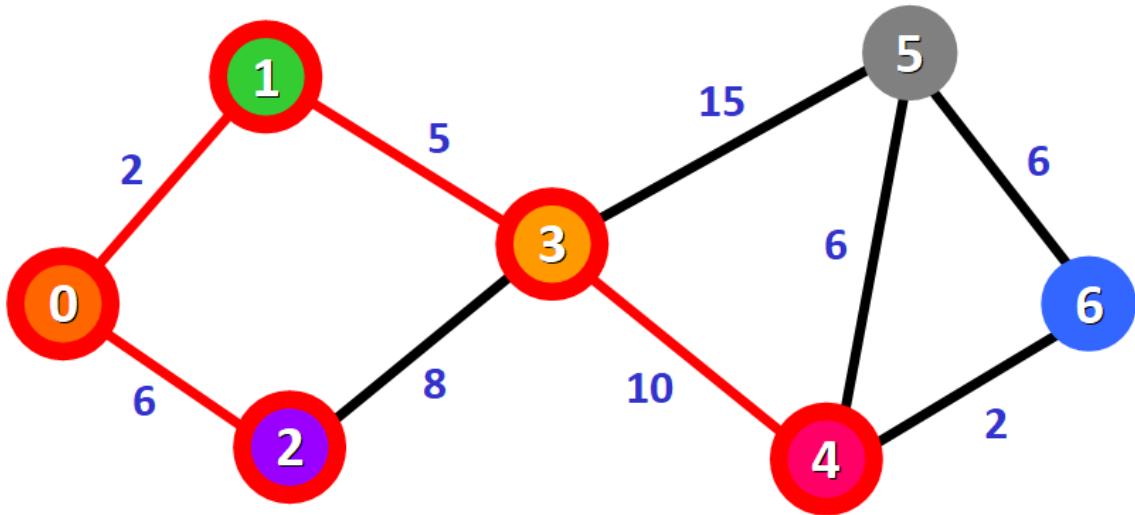
0:	0
1:	2 ■
2:	6 ■
3:	7 ■
4:	17 ■
5:	22 ■
6:	$\infty$

And we cross it off from the list of unvisited nodes:

Unvisited Nodes: {~~0, 1, 2, 3, 4, 5, 6~~}

And we repeat the process again. We check the adjacent nodes: node 5 and node 6. We need to analyze each possible path that

we can follow to reach them from nodes that have already been marked as visited and added to the path.



#### For node 5:

- The first option is to follow the path  $0 \rightarrow 1 \rightarrow 3 \rightarrow 5$ , which has a distance of **22** from the source node ( $2 + 5 + 15$ ). This distance was already recorded in the list of distances in a previous step.
- The second option would be to follow the path  $0 \rightarrow 1 \rightarrow 3 \rightarrow 4 \rightarrow 5$ , which has a distance of **23** from the source node ( $2 + 5 + 10 + 6$ ).

Clearly, the first path is shorter, so we choose it for node 5.

#### For node 6:

- The path available is  $0 \rightarrow 1 \rightarrow 3 \rightarrow 4 \rightarrow 6$ , which has a distance of **19** from the source node ( $2 + 5 + 10 + 2$ ).

### Distance:

0:	0
1:	2 ■
2:	6 ■
3:	7 ■
4:	17 ■
5:	22 vs. 23 ( $2 + 5 + 10 + 6$ )
6:	19 from ( $2 + 5 + 10 + 2$ )

We mark the node with the shortest (currently known) distance as visited. In this case, node 6.

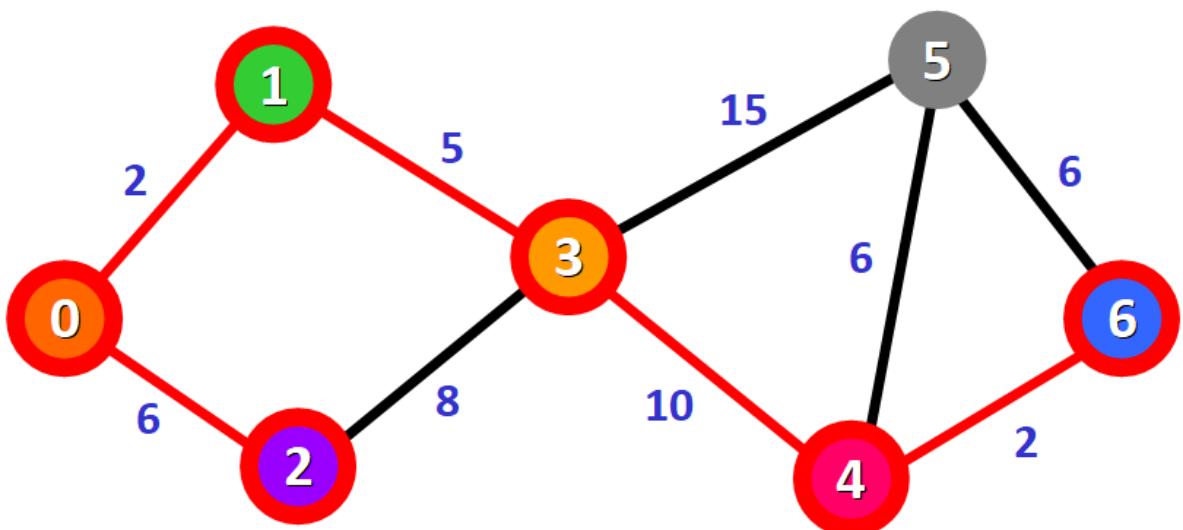
## Distance:

0:	0
1:	<del>2</del>
2:	<del>6</del>
3:	<del>7</del>
4:	<del>17</del>
5:	22
6:	<del>19</del>

And we cross it off from the list of unvisited nodes:

**Unvisited Nodes:** {~~0, 1, 2, 3, 4, 5, 6~~}

Now we have this path (marked in red):



Only one node has not been visited yet, node 5. Let's see how we can include it in the path.

There are three different paths that we can take to reach node 5 from the nodes that have been added to the path:

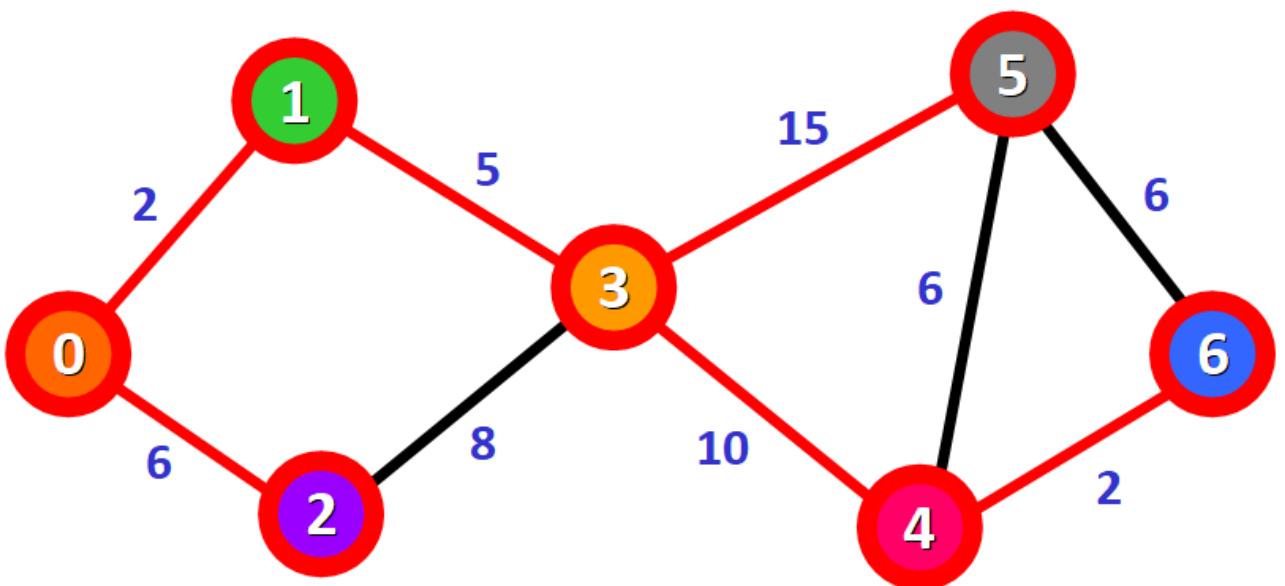
- **Option 1:**  $0 \rightarrow 1 \rightarrow 3 \rightarrow 5$  with a distance of 22 ( $2 + 5 + 15$ ).

- **Option 2:**  $0 \rightarrow 1 \rightarrow 3 \rightarrow 4 \rightarrow 5$  with a distance of **23** ( $2 + 5 + 10 + 6$ ).
- **Option 3:**  $0 \rightarrow 1 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 5$  with a distance of **25** ( $2 + 5 + 10 + 2 + 6$ ).

## Distance:

0:	0
1:	<del>2</del>
2:	<del>6</del>
3:	<del>7</del>
4:	<del>17</del>
5:	<b>22 from <math>(2 + 5 + 15)</math> vs. 23 from <math>(2 + 5 + 10 + 6)</math> vs. 25 from <math>(2 + 5 + 10 + 2 + 6)</math></b>
6:	<del>19</del>

We select the shortest path:  $0 \rightarrow 1 \rightarrow 3 \rightarrow 5$  with a distance of **22**.



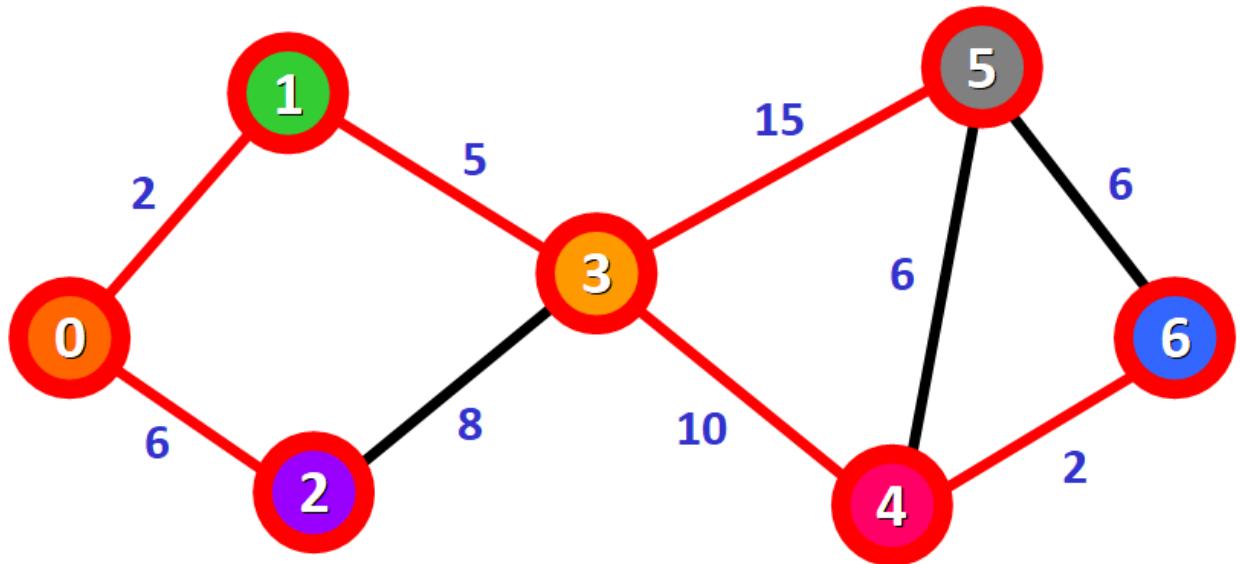
We mark the node as visited and cross it off from the list of unvisited nodes:

## Distance:

0:	0
1:	2.
2:	6.
3:	7.
4:	17.
5:	22.
6:	19.

Unvisited Nodes: {0, 1, 2, 3, 4, 5, 6}

**And voilà!** We have the final result with the shortest path from node 0 to each node in the graph.



In the diagram, the red lines mark the edges that belong to the shortest path. You need to follow these edges to follow the shortest path to reach a given node in the graph starting from node 0.

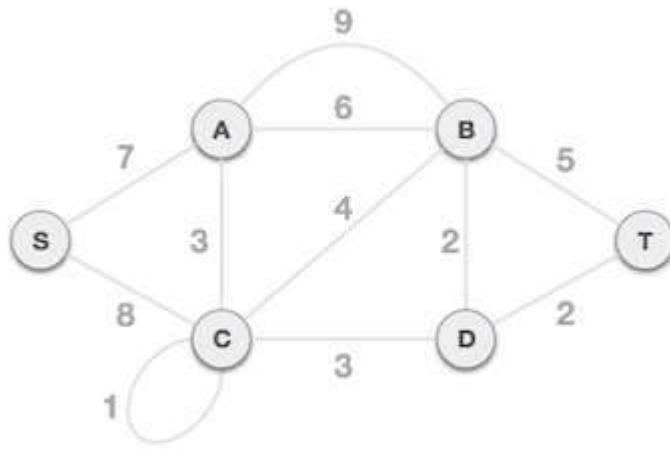
For example, if you want to reach node 6 starting from node 0, you just need to follow the red edges and you will be following the shortest path  $0 \rightarrow 1 \rightarrow 3 \rightarrow 4 \rightarrow 6$  automatically.

## Prim's Spanning Tree Algorithm

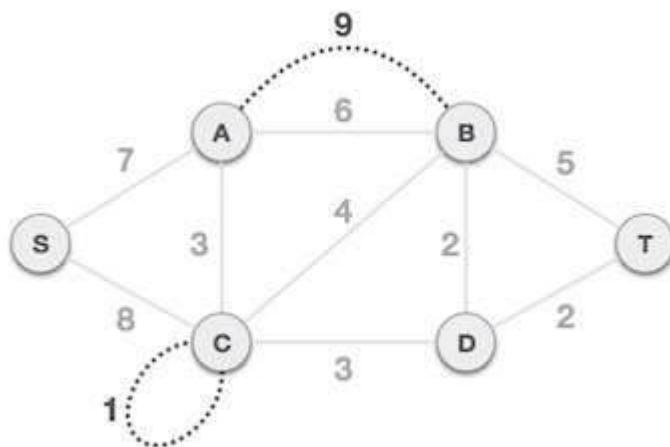
Prim's algorithm to find minimum cost spanning tree (as Kruskal's algorithm) uses the greedy approach. Prim's algorithm shares a similarity with the **shortest path first** algorithms.

Prim's algorithm, in contrast with Kruskal's algorithm, treats the nodes as a single tree and keeps on adding new nodes to the spanning tree from the given graph.

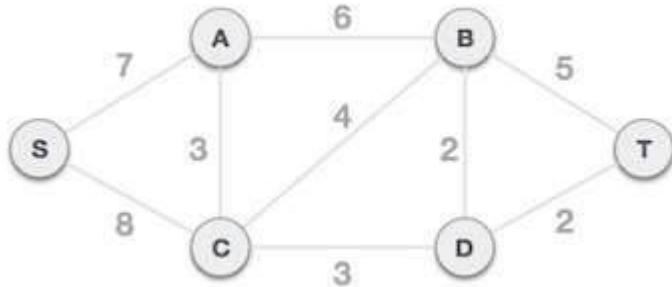
To contrast with Kruskal's algorithm and to understand Prim's algorithm better, we shall use the same example –



Step 1 - Remove all loops and parallel edges



Remove all loops and parallel edges from the given graph. In case of parallel edges, keep the one which has the least cost associated and remove all others.

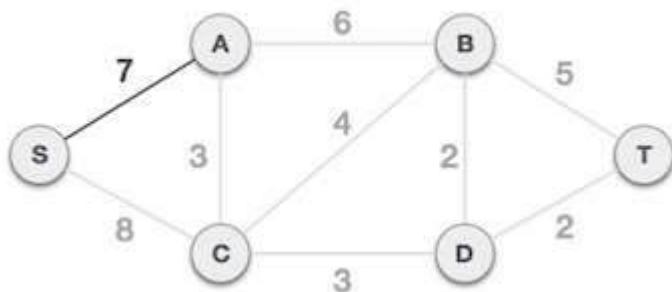


## Step 2 - Choose any arbitrary node as root node

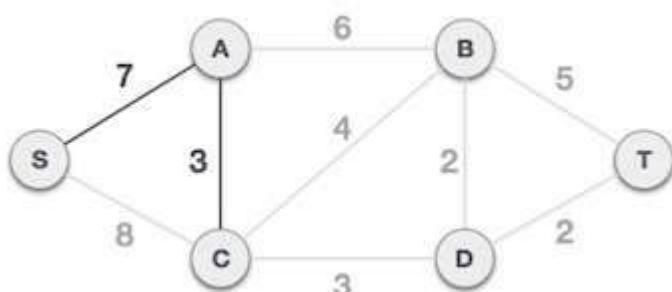
In this case, we choose **S** node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node. One may wonder why any video can be a root node. So the answer is, in the spanning tree all the nodes of a graph are included and because it is connected then there must be at least one edge, which will join it to the rest of the tree.

## Step 3 - Check outgoing edges and select the one with less cost

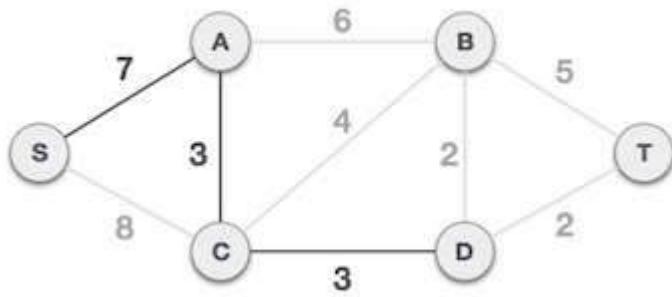
After choosing the root node **S**, we see that **S,A** and **S,C** are two edges with weight 7 and 8, respectively. We choose the edge **S,A** as it is lesser than the other.



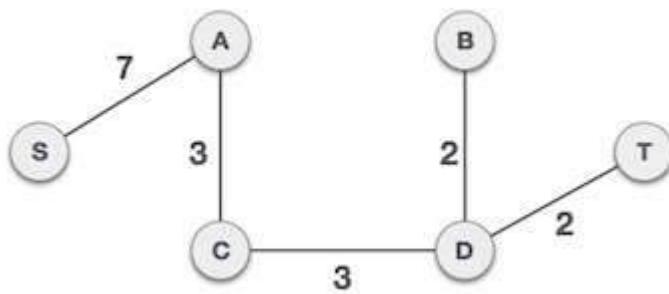
Now, the tree **S-7-A** is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.



After this step, **S-7-A-3-C** tree is formed. Now we'll again treat it as a node and will check all the edges again. However, we will choose only the least cost edge. In this case, **C-3-D** is the new edge, which is less than other edges' cost 8, 6, 4, etc.



After adding node **D** to the spanning tree, we now have two edges going out of it having the same cost, i.e. D-2-T and D-2-B. Thus, we can add either one. But the next step will again yield edge 2 as the least cost. Hence, we are showing a spanning tree with both edges included.



We may find that the output spanning tree of the same graph using two different algorithms is same.

# Hashing

# **Index**

- Introduction
- Advantages
- Hash Function
- Hash Table
- Collision Resolution Techniques
- Separate Chaining
- Linear Chaining
- Quadratic Probing
- Double Hashing
- Application
- Reference

# Introduction

- Hashing is a technique that is *used* to *uniquely identify a specific object* from a *group of similar objects*.
- Some examples of how hashing is used in our lives include:
  - In universities, each student is assigned a unique roll number that can be used to retrieve information about them.
  - In libraries, each book is assigned a unique number that can be used to determine information about the book, such as its exact position in the library or the users it has been issued to etc.
- In both these examples the students and books were hashed to a *unique number*.

## Continue...

- Assume that you have an *object* and you want to *assign a key* to it to make *searching easy*. To store the key/value pair, you can use a *simple array* like a data structure where keys (integers) can be used *directly as an index* to store values. However, in cases where the *keys are large* and cannot be used directly as an index, you should use *hashing*.

## Continue...

- In **hashing**, *large keys* are **converted** into *small keys* by using **hash functions**.
- The values are then *stored in a data structure* called **hash table**.
- The *idea* of hashing is to **distribute entries** (key/value pairs) *uniformly across an array*.
- Each element is assigned a key (converted key).
- By using that key you can access the element in **O(1)** time. Using the key, the *algorithm* (hash function) computes an index that suggests where an entry can be *found* or *inserted*.

## Continue...

- Hashing is implemented in two steps:
  - An *element is converted into an integer* by using a *hash function*. This element can be used as an *index to store the original element*, which falls into the hash table.
  - The element is stored in the hash table where it can be quickly retrieved using *hashed key*.
  - $\text{hash} = \text{hashfunc(key)}$   
 $\text{index} = \text{hash \% array\_size}$

## Advantages

- The main advantage of hash tables over other table *data structures is speed.*
- This advantage is more *apparent* when the number of entries is large (thousands or more).
- Hash tables are particularly efficient when the maximum number of *entries can be predicted in advance*, so that the bucket array can be allocated once with the optimum size and never resized.

# Hash function

- A *hash function* is any function that can be *used* to *map a data set of an arbitrary size to a data set of a fixed size*, which falls into the hash table.
- The values returned by a hash function are called hash values, hash codes, hash sums, or simply hashes.

## Continue...

- A hash function is any function that can be *used to map a data* set of an *arbitrary size* to a data set of a fixed size, which falls into the hash table.
- The values returned by a hash function are called *hash values, hash codes, hash sums, or simply hashes*.
- To achieve a good hashing mechanism, It is important to have a good hash function with the following basic requirements:
  - Easy to compute
  - Uniform distribution
  - Less Collision

# *Hash Table*

- A hash table is a data structure that is used to store keys/value pairs. It uses a hash function to compute an index into an array in which an element will be inserted or searched.
- By using a good hash function, hashing can work well. Under reasonable assumptions, the average time required to search for an element in a hash table is  $O(1)$ .

- Let us understand **hashing in a data structure** with an example. Imagine you need to store some items (arranged in a key-value pair) inside a hash table with 30 cells.
- The values are: (3,21) (1,72) (40,36) (5,30) (11,44) (15,33) (18,12) (16,80) (38,99)
- The hash table will look like the following:

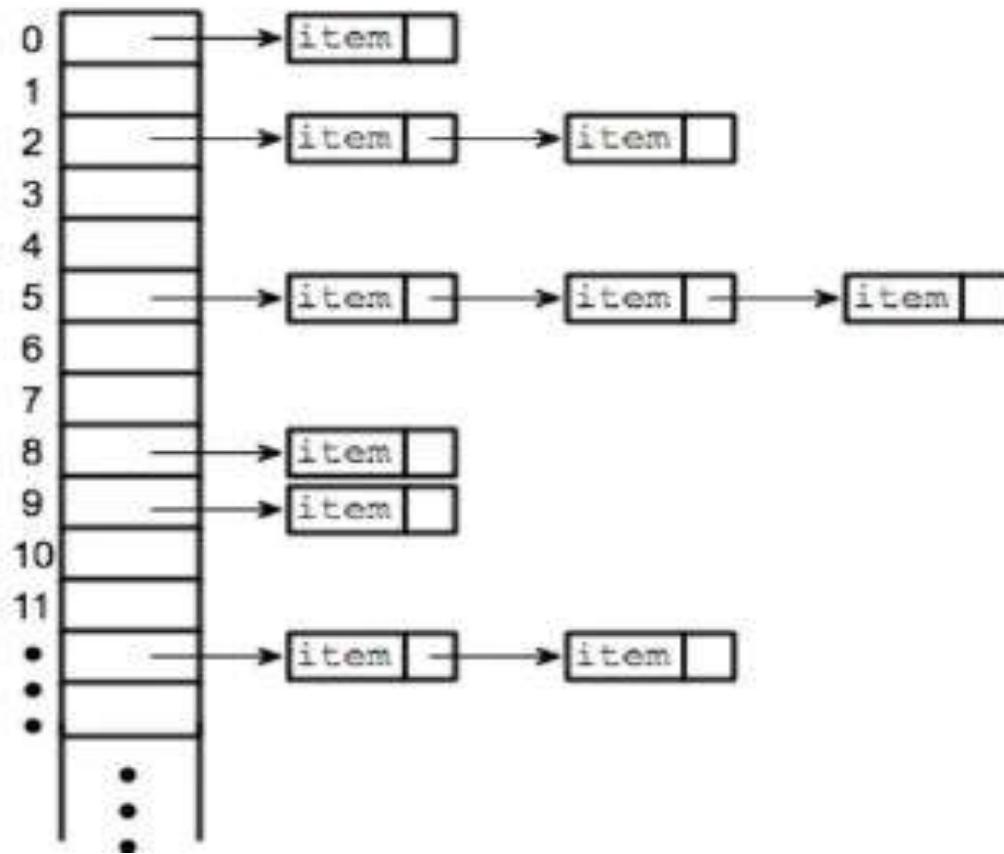
Serial Number	Key	Hash	Array Index
1	3	$3 \% 30 = 3$	3
2	1	$1 \% 30 = 1$	1
3	40	$40 \% 30 = 10$	10
4	5	$5 \% 30 = 5$	5
5	11	$11 \% 30 = 11$	11
6	15	$15 \% 30 = 15$	15
7	18	$18 \% 30 = 18$	18
8	16	$16 \% 30 = 16$	16
9	38	$38 \% 30 = 8$	8

# Collision resolution techniques

- If  $x_1$  and  $x_2$  are two different keys, it is possible that  $h(x_1) = h(x_2)$ . This is called a collision. Collision resolution is the most important issue in hash table implementations.
- Choosing a hash function that minimizes the number of collisions and also hashes uniformly is another critical issue.
  - *Separate chaining (open hashing)*
  - *Linear probing (open addressing or closed hashing)*
  - *Quadratic Probing*
  - *Double hashing*

## **Separate Chaining (Open Hashing)**

- Separate chaining is one of the most commonly used collision resolution techniques.
- It is usually implemented using linked lists. In separate chaining, each element of the hash table is a linked list.
- To store an element in the hash table you must insert it into a specific linked list.
- If there is any collision (i.e. two different elements have same hash value) then store both the elements in the same linked list.



# Linear Probing

- Imagine you have been asked to store some items inside a hash table of size 30. The items are already sorted in a key-value pair format. The values given are: (3,21) (1,72) (63,36) (5,30) (11,44) (15,33) (18,12) (16,80) (46,99).
- The  $\text{hash}(n)$  is the index computed using a hash function and  $T$  is the table size. If slot index =  $(\text{hash}(n) \% T)$  is full, then we look for the next slot index by adding 1  $((\text{hash}(n) + 1) \% T)$ . If  $(\text{hash}(n) + 1) \% T$  is also full, then we try  $(\text{hash}(n) + 2) \% T$ . If  $(\text{hash}(n) + 2) \% T$  is also full, then we try  $(\text{hash}(n) + 3) \% T$ .

Serial Number	Key	Hash	Array Index	Array Index after Linear Probing
1	3	$3\%30 = 3$	3	3
2	1	$1\%30 = 1$	1	1
3	63	$63\%30 = 3$	3	4
4	5	$5\%30 = 5$	5	5
5	11	$11\%30 = 11$	11	11
6	15	$15\%30 = 15$	15	15
7	18	$18\%30 = 18$	18	18
8	16	$16\%30 = 16$	16	16
9	46	$46\%30 = 8$	16	17

## Quadratic Probing

- Quadratic probing is similar to linear probing and the only difference is the interval between successive probes or entry slots.
- Here, when the slot at a hashed index for an entry record is already occupied, you must start traversing until you find an unoccupied slot.
- The interval between slots is computed by adding the successive value of an arbitrary polynomial in the original hashed index.

## Continue...

- Let us assume that the hashed index for an entry is **index** and at **index** there is an occupied slot. The probe sequence will be as follows:

$$\text{index} = \text{index \% hashTableSize}$$
$$\text{index} = (\text{index} + 1^2) \% \text{hashTableSize}$$
$$\text{index} = (\text{index} + 2^2) \% \text{hashTableSize}$$
$$\text{index} = (\text{index} + 3^2) \% \text{hashTableSize}$$

# Double Hashing

- The double hashing technique uses two hash functions. The second hash function comes into use when the first function causes a collision. It provides an offset index to store the value.
- The formula for the double hashing technique is as follows:
- **(firstHash(key) + i \* secondHash(key)) % sizeOfTable**
- Where i is the offset value. This offset value keeps incremented until it finds an empty slot.

- Insert the keys 79, 69, 98, 72, 14, 50 into the **Hash Table of size 13**. Resolve all collisions using Double Hashing where first hash-function is  $h_1(k) = k \bmod 13$  and second hash-function is  $h_2(k) = 1 + (k \bmod 11)$

Initially, all the hash table locations are empty. We pick our **first key = 79** and apply  $h_1(k)$  over it,

$h_1(79) = 79 \% 13 = 1$ , hence key **79** hashes to  $1^{st}$  location of the hash table. But before placing the key, we need to make sure the  $1^{st}$  location of the hash table is empty. In our case it is empty and we can easily place key **79** in there.

**Second key = 69**, we again apply  $h_1(k)$  over it,  $h_1(69) = 69 \% 13 = 4$ , since the  $4^{th}$  location is empty we can place **69** there.

**Third key = 98**, we apply  $h_1(k)$  over it,  $h_1(98) = 98 \% 13 = 7$ ,  $7^{th}$  location is empty so **98** placed there.

Fourth key = 72,  $h_1(72) = 72 \% 13 = 7$ , now this is a collision because the 7<sup>th</sup> location is already occupied, we need to resolve this collision using double hashing.

$$h_{new} = [ h_1(72) + i * h_2(72) ] \% 13$$

$$= [ 7 + 1 * ( 1 + 72 \% 11 ) ] \% 13$$

$$= 1$$

Location 1<sup>st</sup> is already occupied in the hash-table, hence we again had a collision.

Now we again recalculate with  $i = 2$

$$h_{new} = [ h_1(72) + i * h_2(72) ] \% 13$$

$$= [ 7 + 2 * ( 1 + 72 \% 11 ) ] \% 13$$

---

$$= 8$$

Location  $8^{th}$  is empty in hash-table and now we can place key **72** in there.

**Fifth key = 14**,  $h_1(14) = 14 \% 13 = 1$ , now this is again a collision because 1st location is already occupied, we now need to resolve this collision using double hashing.

$$h_{new} = [ h_1(14) + i * h_2(14) ] \% 13$$

$$= [ 1 + 1 * ( 1 + 14 \% 11 ) ] \% 13$$

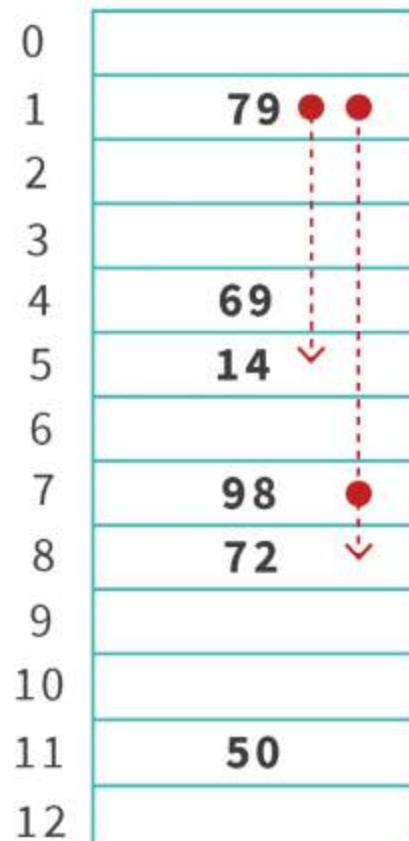
$$= 5$$

Location  $5^{th}$  is empty and we can easily place key **14** there.

**Sixth key = 50**,  $h_1(50) = 50 \% 13 = 11$ ,  $11^{th}$  location is already empty and we can place our key **50** there.

**KEYS : 79, 69, 98, 72, 14, 50**

---

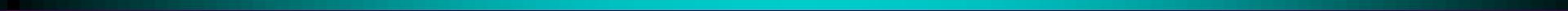


Hash table

# Applications

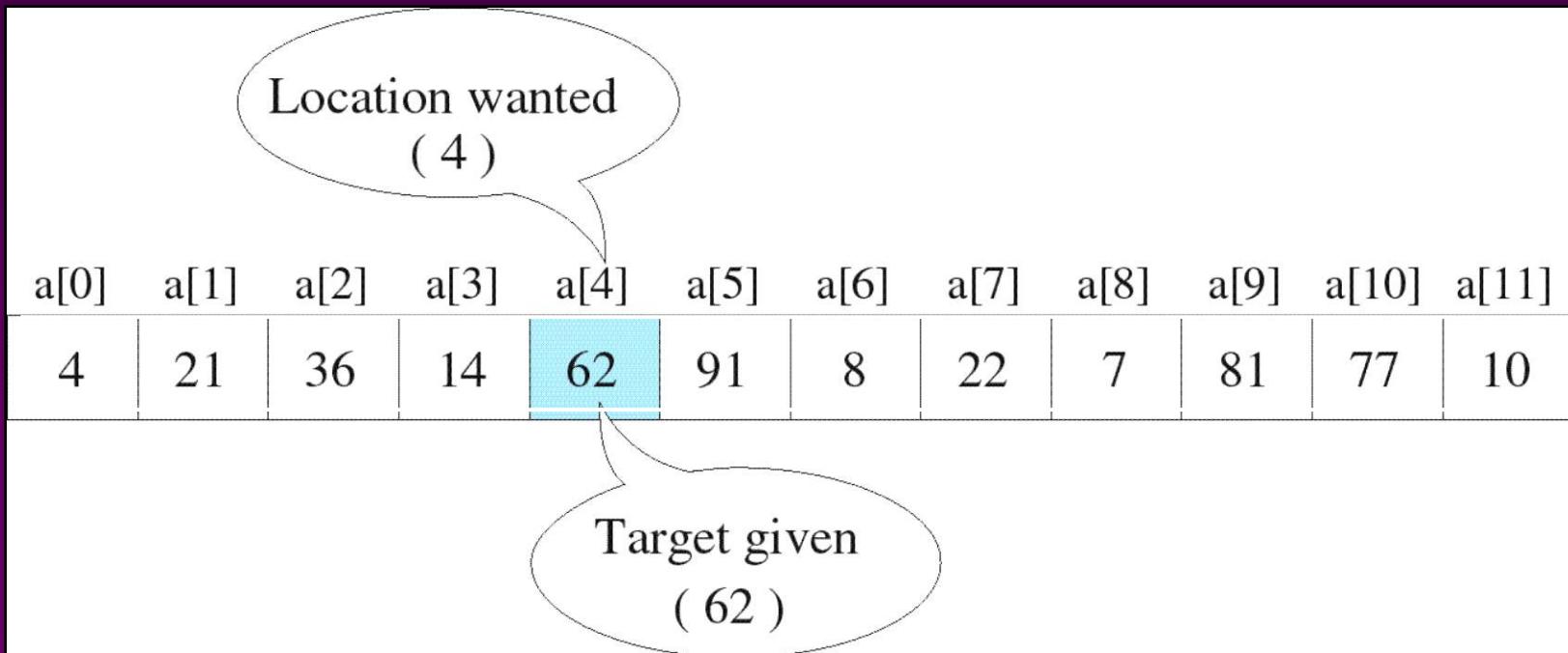
- *Associative arrays*: Hash tables are commonly used to implement many types of in-memory tables. They are used to implement associative arrays (arrays whose indices are arbitrary strings or other complicated objects).
- *Database indexing*: Hash tables may also be used as disk-based data structures and database indices (such as in dbm).
- *Caches*: Hash tables can be used to implement caches i.e. auxiliary data tables that are used to speed up the access to data, which is primarily stored in slower media.
- *Object representation*: Several dynamic languages, such as Perl, Python, JavaScript, and Ruby use hash tables to implement objects.
- Hash Functions are used in various algorithms to make their computing faster.

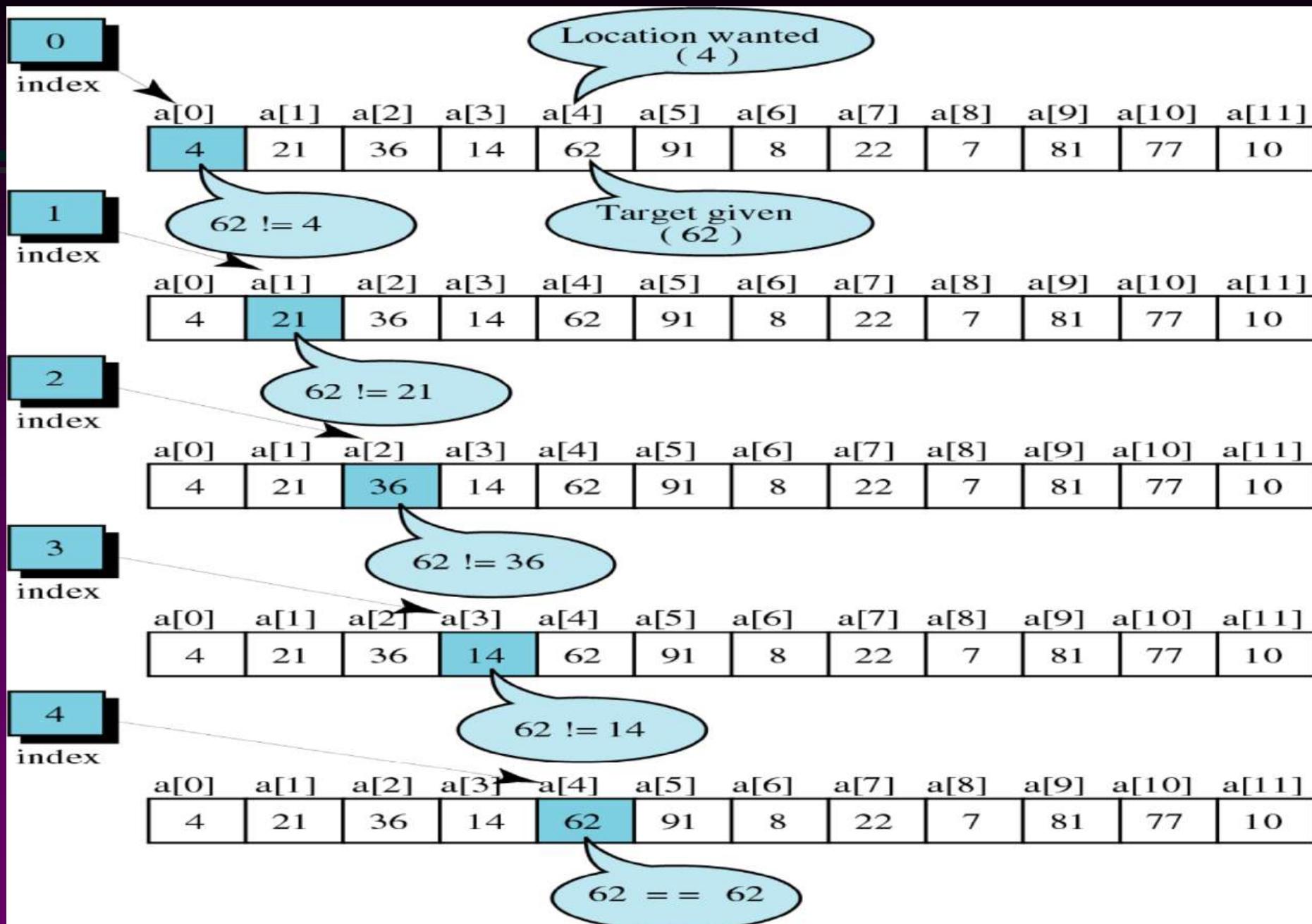
# Searching Arrays

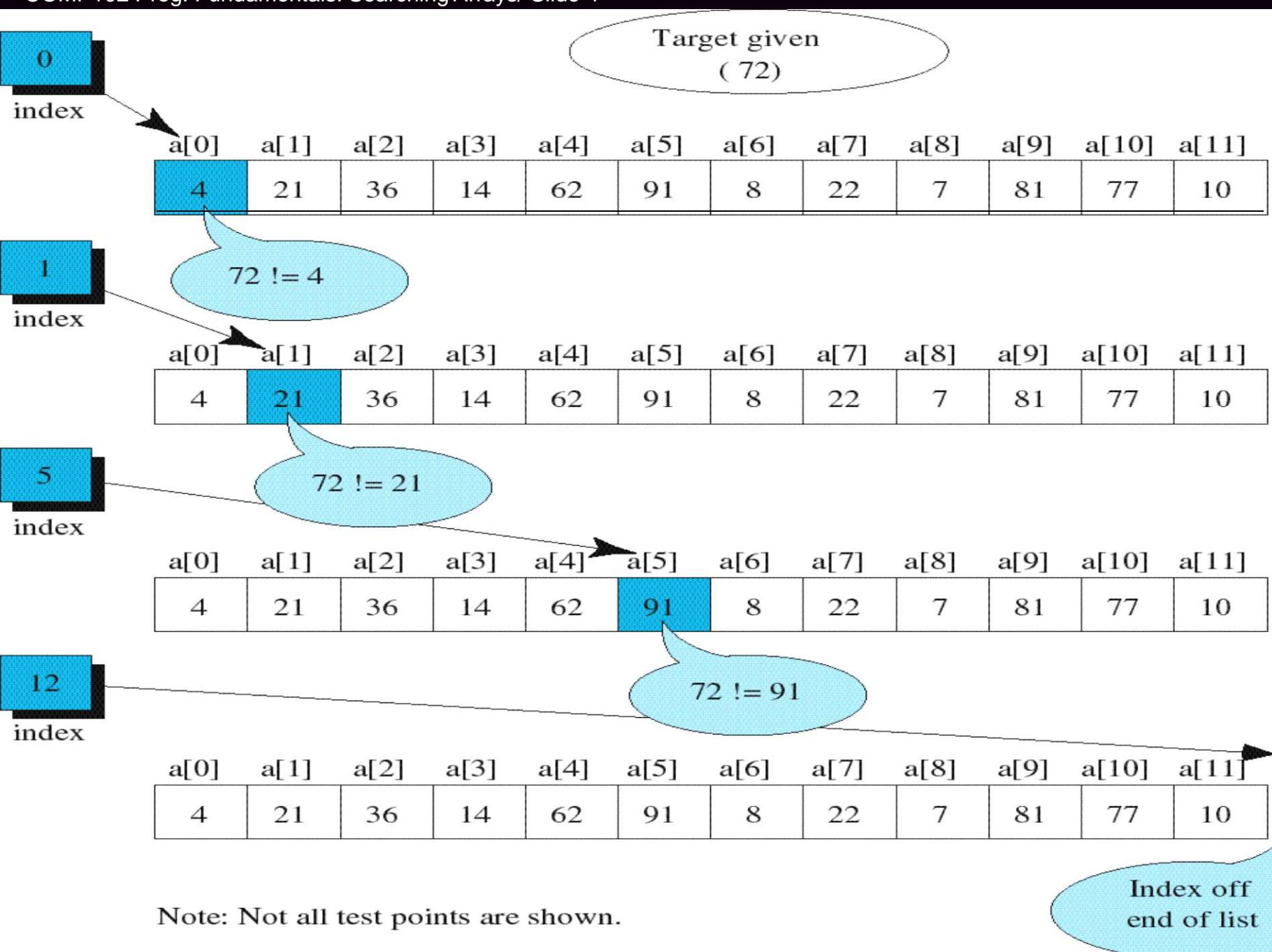


# Searching

- The process used to find the location of a target among a list of objects
  - Searching an array finds the index of first element in an array containing that value







# Unordered Linear Search

- Search an unordered array of integers for a value and return its index if the value is found. Otherwise, return -1.

A[0] A[1] A[2] A[3] A[4] A[5] A[6] A[7]

14	2	10	5	1	3	17	2
----	---	----	---	---	---	----	---

- Algorithm:

```
Start with the first array element (index 0)
while(more elements in array) {
    if value found at current index, return index;
    Try next element (increment index);
}
Value not found, return -1;
```

# Unordered Linear Search

```
// Searches an unordered array of integers
int search(int data[],    //input: array
           int size,      //input: array size
           int value){   //input: search value
    // output: if found, return index;
    //          otherwise, return -1.
    for(int index = 0; index < size; index++) {
        if(data[index] == value)
            return index;
    }
    return -1;
}
```

# Unordered Linear Search

```
#include <iostream>
Using namespace std;
int main() {
    const int array_size = 8;
    int list[array_size]={14,2,10,5,1,3,17,2};
    int search_value;

    cout << "Enter search value: ";
    cin >> search_value;

    cout << search(list,array_size,search_value)
        << endl;
    return 0;
}
```

# Ordered Linear Search

- Search an ordered array of integers for a value and return its index if the value is found; Otherwise, return -1.

A[0] A[1] A[2] A[3] A[4] A[5] A[6] A[7]

1	2	3	5	7	10	14	17
---	---	---	---	---	----	----	----

- Linear search can stop immediately when it has passed the possible position of the search value.

# Ordered Linear Search

## □ Algorithm:

```
Start with the first array element (index 0)
while(more elements in the array){
    if value at current index is greater than value,
        value not found, return -1;
    if value found at current index, return index;
    Try next element (increment index);
}
value not found, return -1;
```

# Ordered Linear Search

```
// Searches an ordered array of integers
int lsearch(int data[], // input: array
            int size, // input: array size
            int value // input: value to find
            ) { // output: index if found
    for(int index=0; index<size; index++) {
        if(data[index] > value)
            return -1;
        else if(data[index] == value)
            return index;
    }
    return -1;
}
```

# Ordered Linear Search

```
#include <iostream>
using namespace std;
int main() {
    const int array_size = 8;
    int list[array_size]={1,2,3,5,7,10,14,17};
    int search_value;

    cout << "Enter search value: ";
    cin >> search_value;
    cout << lsearch(list,array_size,search_value)
        << endl;

    return 0;
}
```

# Definition

- Recursion lends itself to a general problem-solving technique (algorithm design) called **divide & conquer**
  - *Divide* the problem into 1 or more similar sub-problems
  - *Conquer* each sub-problem, usually using a recursive call
  - *Combine* the results from each sub-problem to form a solution to the original problem
- Algorithmic Pattern:

```
DC( problem )
    solution = ∅
    if ( problem is small enough )
        solution = problem.solve()
    else
        children = problem.divide()
        for each c in children
            solution = solution + c.solve()
    return solution
```

Divide

Conquer

Combine

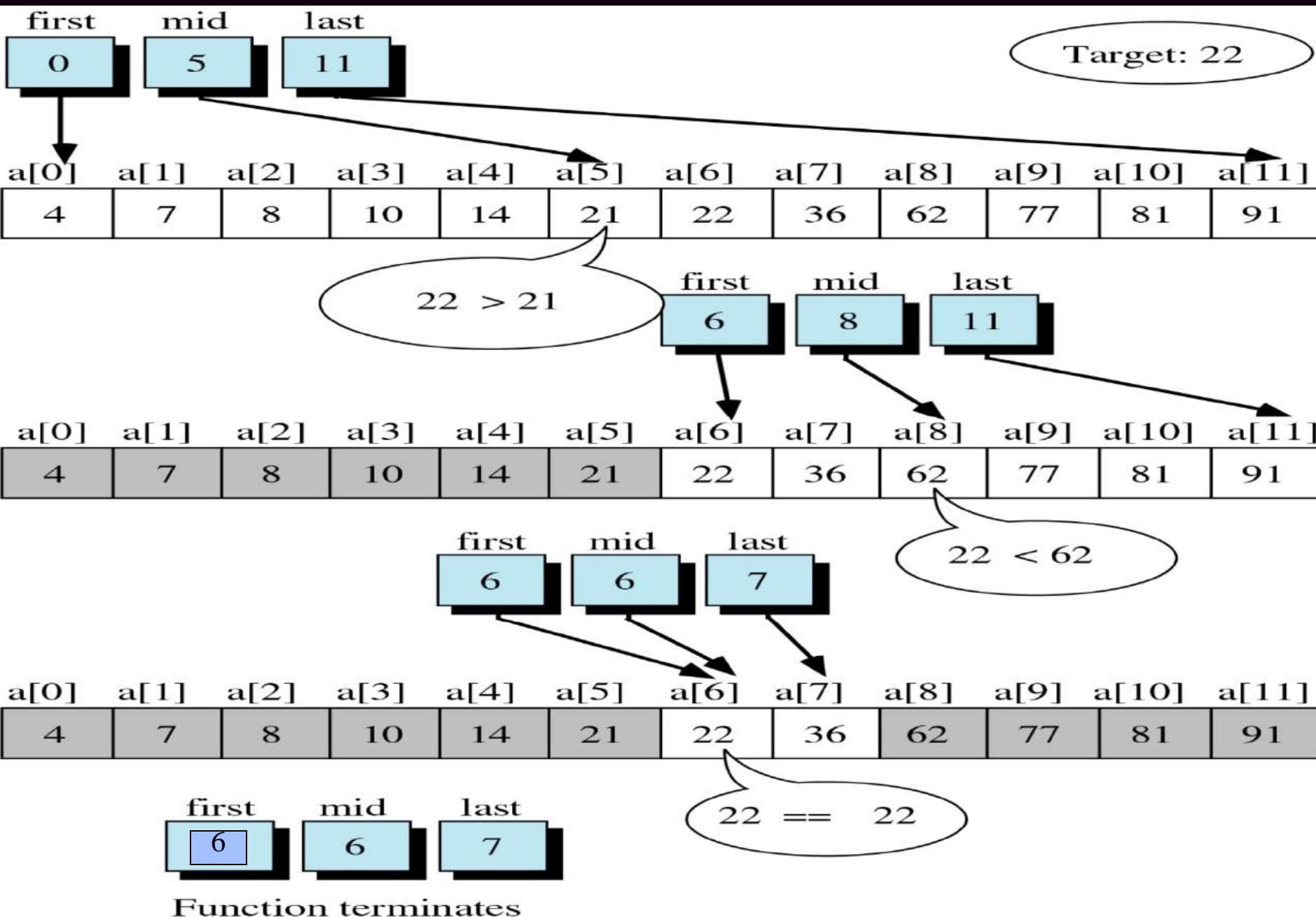
# Binary Search

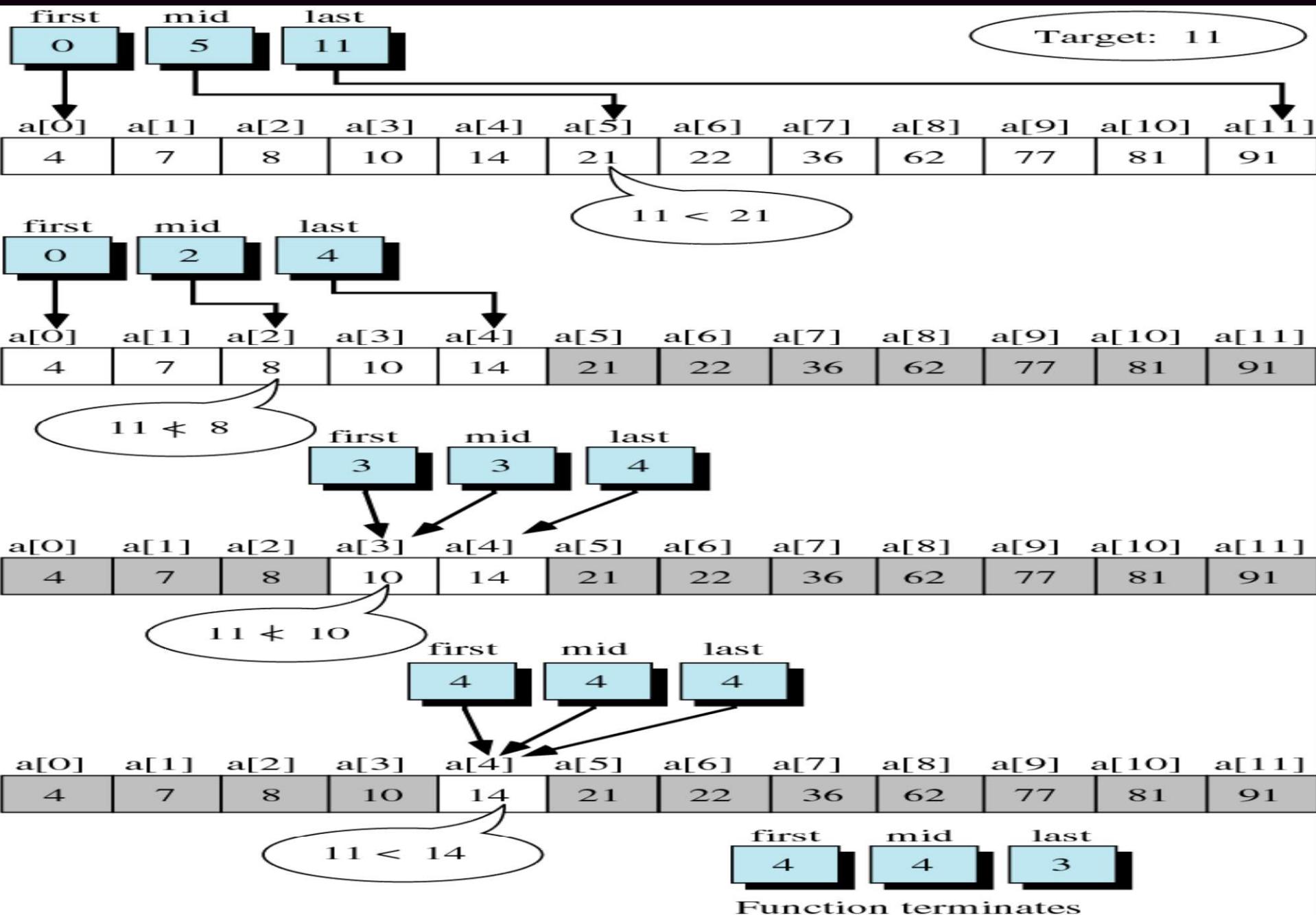
- Search an ordered array of integers for a value and return its index if the value is found. Otherwise, return -1.

A[0] A[1] A[2] A[3] A[4] A[5] A[6] A[7]

1	2	3	5	7	10	14	17
---	---	---	---	---	----	----	----

- Binary search skips over parts of the array if the search value cannot possibly be there.





# Binary Search

- Binary search is based on the “divide-and-conquer” strategy which works as follows:
  - Start by looking at the middle element of the array
    - 1. If the value it holds is lower than the search element, eliminate the first half of the array from further consideration.
    - 2. If the value it holds is higher than the search element, eliminate the second half of the array from further consideration.
  - Repeat this process until the element is found, or until the entire array has been eliminated.

# Binary Search

## □ Algorithm:

Set **first** and **last** boundary of array to be searched

Repeat the following:

```
Find middle element between first and last boundaries;  
if (middle element contains the search value)  
    return middle_element position;  
else if (first >= last )  
    return -1;  
else if (value < the value of middle_element)  
    set last to middle_element position - 1;  
else  
    set first to middle_element position + 1;
```

# Binary Search

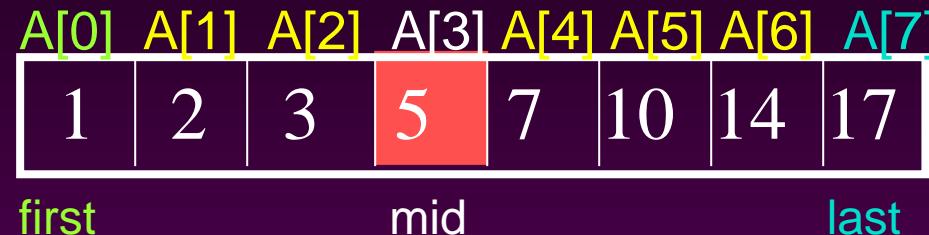
```
// Searches an ordered array of integers
int bsearch(int data[], // input: array
            int size,    // input: array size
            int value    // input: value to find
            )           // output: if found, return index
                        //                  otherwise, return -1
{
    int first, middle, last;
    first = 0;
    last = size - 1;
    while (true) {
        middle = (first + last) / 2;
        if (data[middle] == value)
            return middle;
        else if (first >= last)
            return -1;
        else if (value < data[middle])
            last = middle - 1;
        else
            first = middle + 1;
    }
}
```

# Binary Search

```
int main() {  
    const int array_size = 8;  
    int list[array_size]={1,2,3,5,7,10,14,17};  
    int search_value;  
  
    cout << "Enter search value: ";  
    cin >> search_value;  
    cout << bsearch(list,array_size,search_value)  
        << endl;  
  
    return 0;  
}
```

# Example: binary search

□ 14 ?



A[6] A[7]

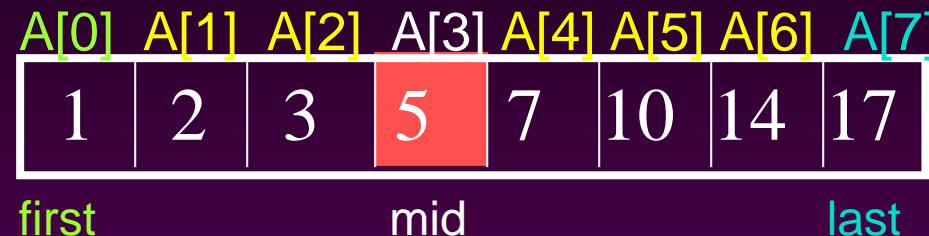


f mid last

In this case,  
`(data[middle] == value)`  
`return middle;`

# Example: binary search

8?



```
In this case, (first  
    == last)  
    return -1;
```

A[4]

7

f m l

# Example: binary search

□ 4 ?

A[0] A[1] A[2] A[3] A[4] A[5] A[6] A[7]

1	2	3	5	7	10	14	17
---	---	---	---	---	----	----	----

first mid last

A[0] A[1] A[2]

1	2	3
---	---	---

first mid last

A[2]

3
---

f m l

In this case, (first  
== last)  
return -1;

- At Iteration 1, Length of array =  $n$
- At Iteration 2, Length of array =  $\frac{n}{2}$
- At Iteration 3, Length of array =  $\frac{n}{2}/2 = \frac{n}{2^2}$
- Therefore, after Iteration  $k$ , Length of array =  $\frac{n}{2^k}$
- Also, we know that after  $k$  divisions, the length of array becomes 1
- Therefore Length of array =  $\frac{n}{2^k} = 1 \Rightarrow n = 2^k$
- Applying log function on both sides:  $\Rightarrow \log_2(n) = \log_2(2^k) \Rightarrow \log_2(n) = k \log_2(2)$
- As  $(\log_a(a) = 1)$   
Therefore,  $\Rightarrow k = \log_2(n)$

- Best case: $O(1)$
- Worst case:  $O(\log n)$
- Average case: $O(\log n)$

# Bubble Sort

# Sorting

- Sorting takes an unordered collection and makes it an ordered one.

1	2	3	4	5
6 77	42	35	12	101 5



1	2	3	4	5
6 5	12	35	42	77 101

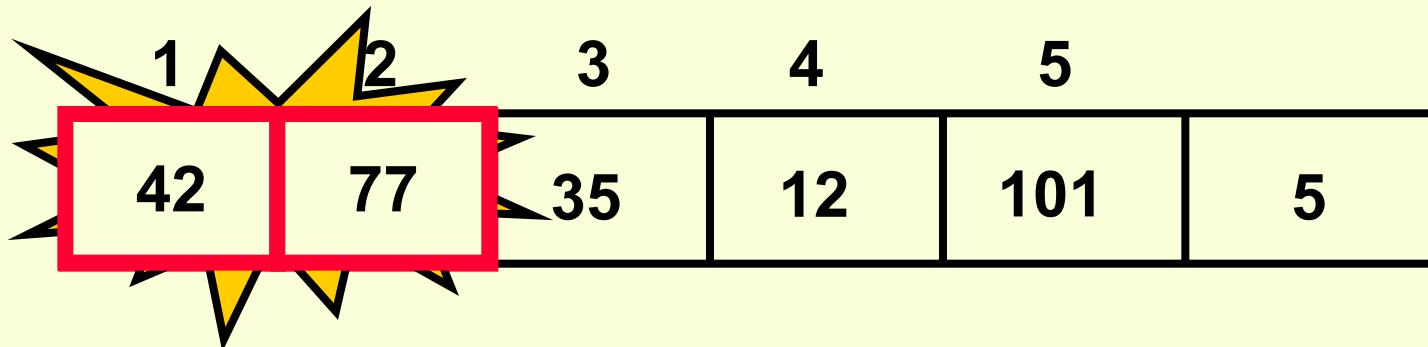
# "Bubbling Up" the Largest Element

- Traverse a collection of elements
  - Move from the front to the end
  - “Bubble” the largest value to the end using pair-wise comparisons and swapping

1	2	3	4	5	
6 77	42	35	12	101	5

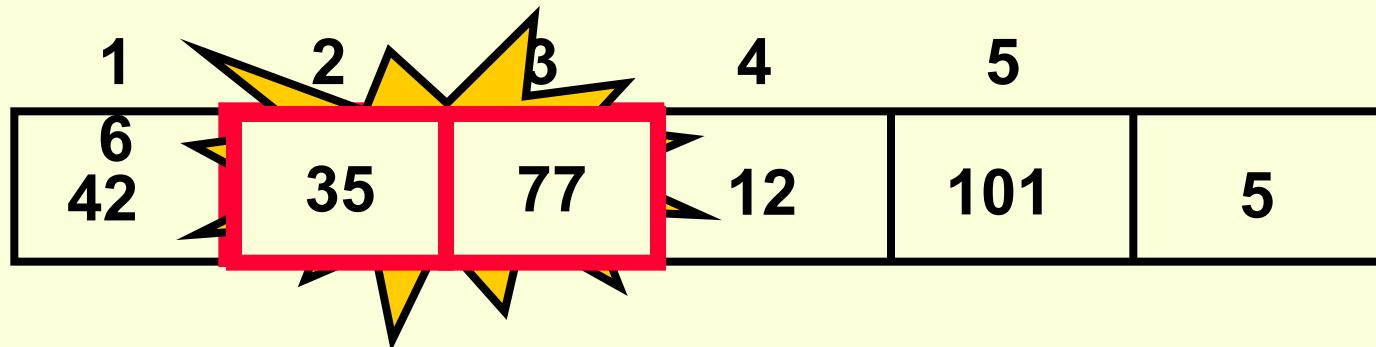
# "Bubbling Up" the Largest Element

- Traverse a collection of elements
  - Move from the front to the end
  - “Bubble” the largest value to the end using pair-wise comparisons and swapping



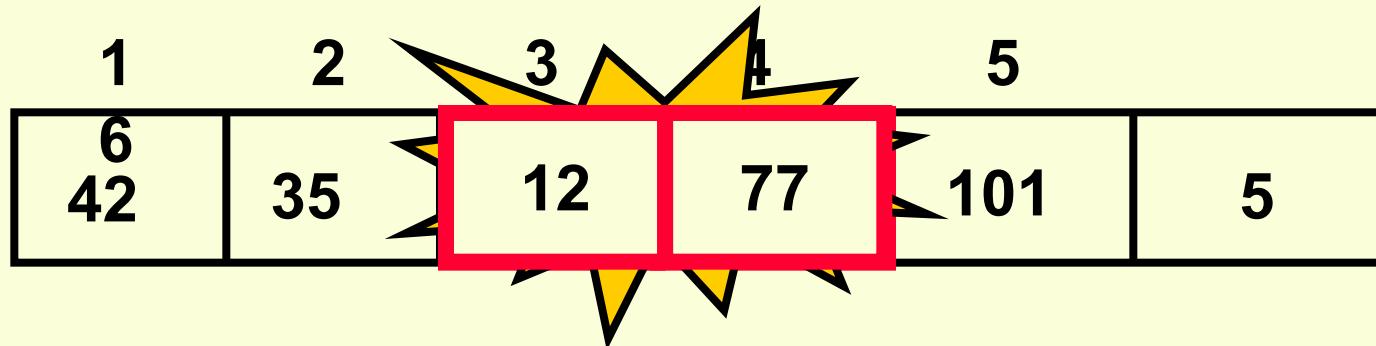
# "Bubbling Up" the Largest Element

- Traverse a collection of elements
  - Move from the front to the end
  - “Bubble” the largest value to the end using pair-wise comparisons and swapping



# "Bubbling Up" the Largest Element

- Traverse a collection of elements
  - Move from the front to the end
  - “Bubble” the largest value to the end using pair-wise comparisons and swapping



# "Bubbling Up" the Largest Element

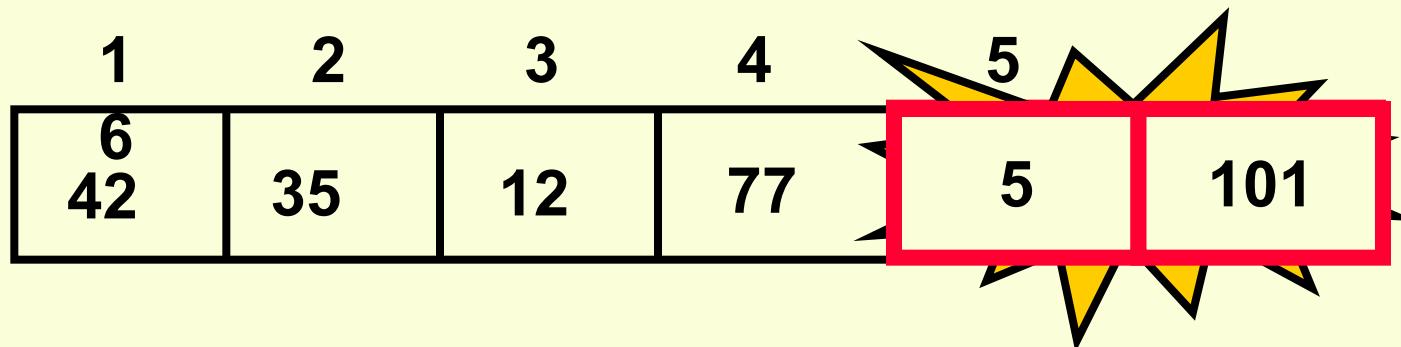
- Traverse a collection of elements
  - Move from the front to the end
  - “Bubble” the largest value to the end using pair-wise comparisons and swapping

1	2	3	4	5	
6 42	35	12	77	101	5

No need to swap

# "Bubbling Up" the Largest Element

- Traverse a collection of elements
  - Move from the front to the end
  - “Bubble” the largest value to the end using pair-wise comparisons and swapping



# "Bubbling Up" the Largest Element

- Traverse a collection of elements
  - Move from the front to the end
  - “Bubble” the largest value to the end using pair-wise comparisons and swapping

1	2	3	4	5	
6 42	35	12	77	5	101

Largest value correctly placed

```
// below we have a simple C program for bubble sort
#include <stdio.h>

void bubbleSort(int arr[], int n)
{
    int i, j, temp;
    for(i = 0; i < n; i++)
    {
        for(j = 0; j < n-i-1; j++)
        {
            if( arr[j] > arr[j+1])
            {
                // swap the elements
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}
```

## Items of Interest

- Notice that only the largest value is correctly placed
- All other values are still out of order
- So we need to repeat this process

1	2	3	4	5	
6 42	35	12	77	5	101

Largest value correctly placed

# **Repeat “Bubble Up” How Many Times?**

- If we have N elements...
- And if each time we bubble an element, we place it in its correct location...
- Then we repeat the “bubble up” process  $N - 1$  times.
- This guarantees we’ll correctly place all N elements.

# “Bubbling” All the Elements

1	2	3	4	5	
42	35	12	77	5	101
1	2	3	4	5	
35	12	42	5	77	101
1	2	3	4	5	
12	35	5	42	77	101
1	2	3	4	5	
12	5	35	42	77	101
1	2	3	4	5	
5	12	35	42	77	101

↓  
n

# Reducing the Number of Comparisons

1	2	3	4	5	
67	42	35	12	101	5
1	2	3	4	5	
42	35	12	77	5	101
1	2	3	4	5	
35	12	42	5	77	101
1	2	3	4	5	
12	35	5	42	77	101
1	2	3	4	5	
12	5	35	42	77	101

# **Putting It All Together**

# Already Sorted Collections?

- What if the collection was already sorted?
- What if only a few elements were out of place and after a couple of “bubble ups,” the collection was sorted?
- We want to be able to detect this and “stop early”!

1	2	3	4	5
5	12	35	42	77

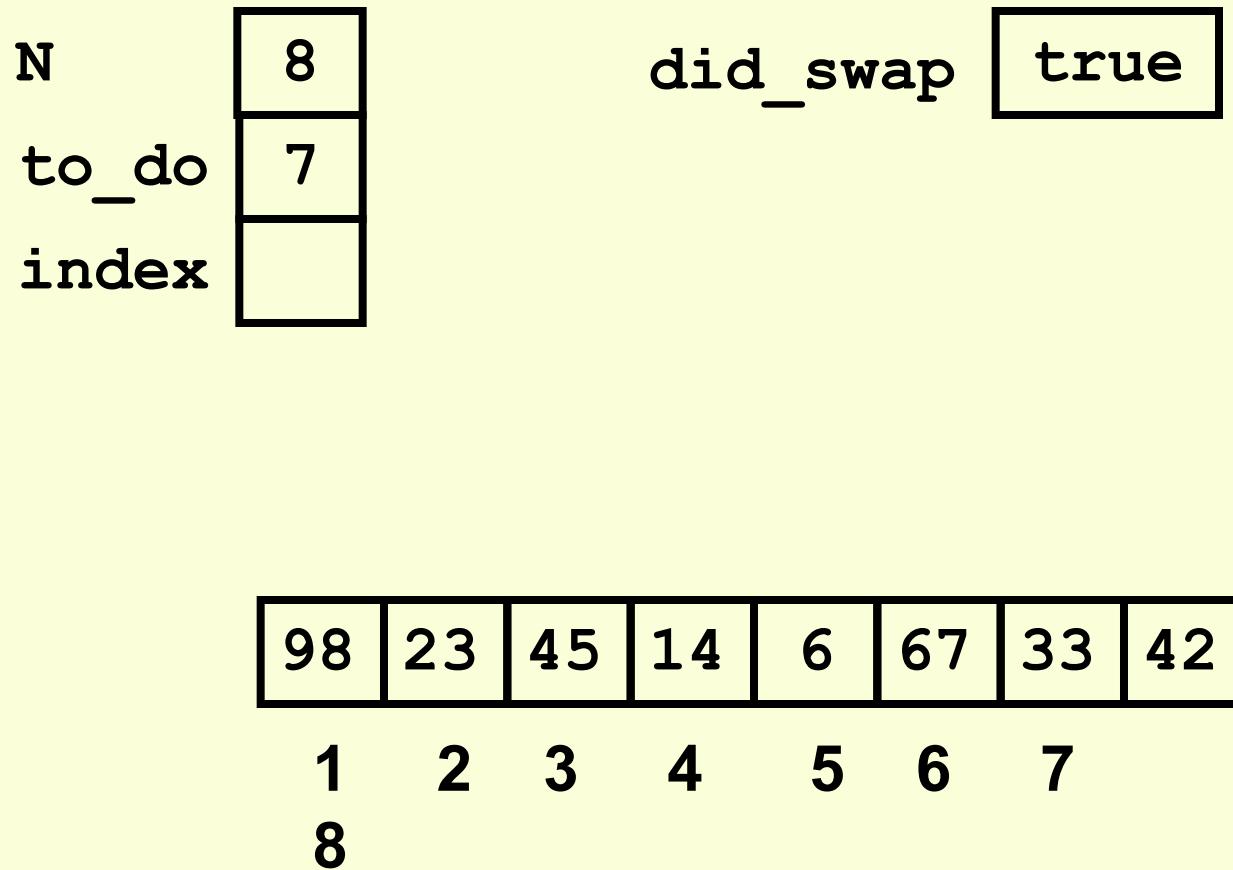
# Using a Boolean “Flag”

- We can use a boolean variable to determine if any swapping occurred during the “bubble up.”
- If no swapping occurred, then we know that the collection is already sorted!
- This boolean “flag” needs to be reset after each “bubble up.”

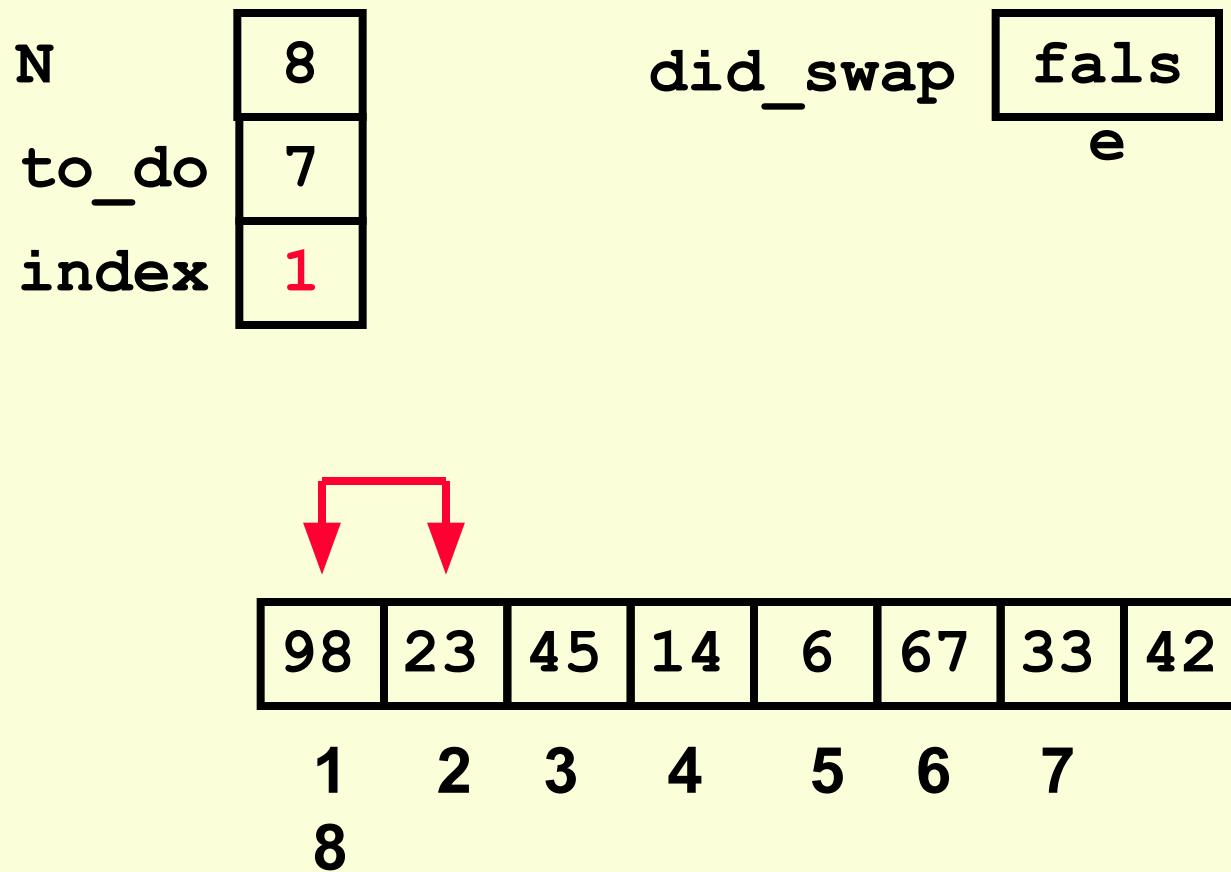
```
void bubbleSort(int arr[], int n)
{
    int i, j, temp, flag=0;
    for(i = 0; i < n; i++)
    {
        for(j = 0; j < n-i-1; j++)
        {
            // introducing a flag to monitor swapping
            if( arr[j] > arr[j+1])
            {
                // swap the elements
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
                // if swapping happens update flag to 1
                flag = 1;
            }
        }
    }
}
```

```
// WHEN BREAK OUT  
if(flag==0)  
{  
    break;  
}  
}
```

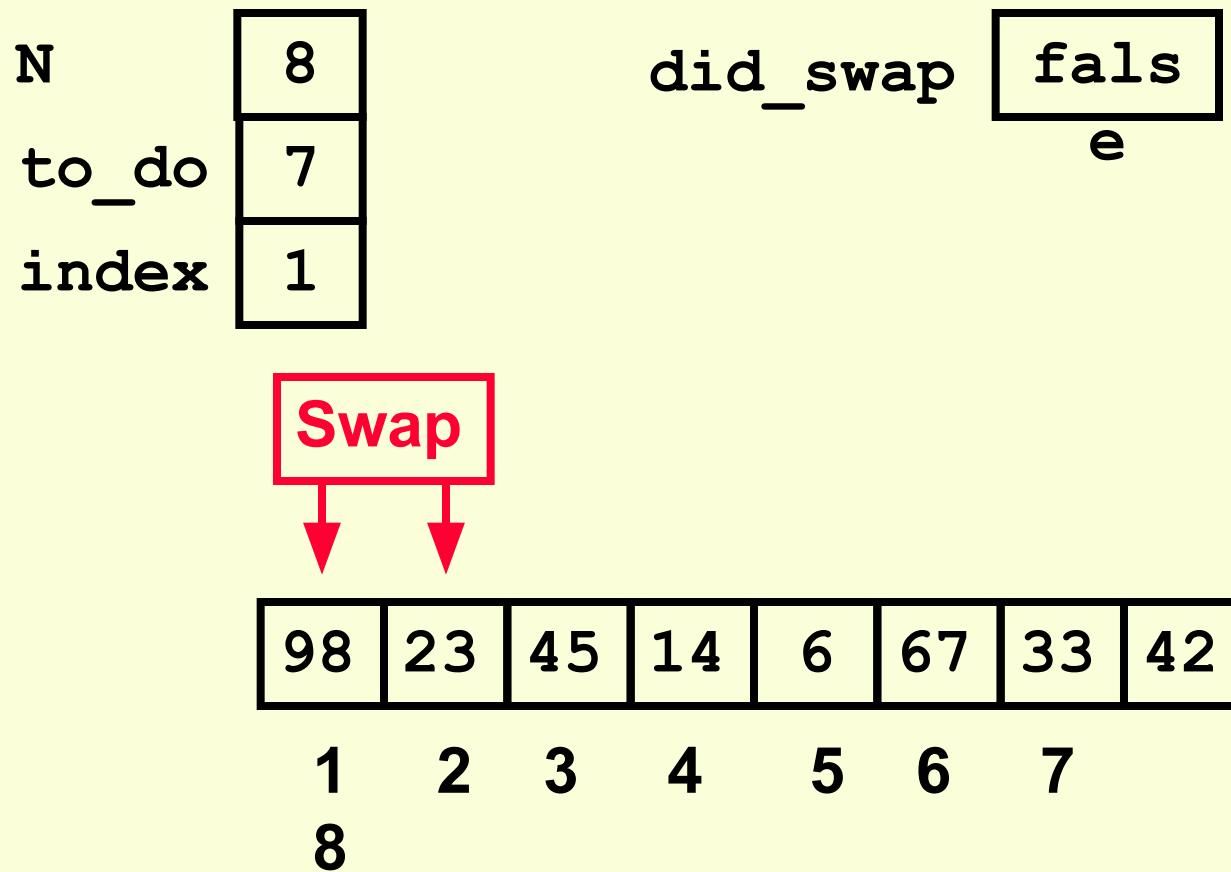
# An Animated Example



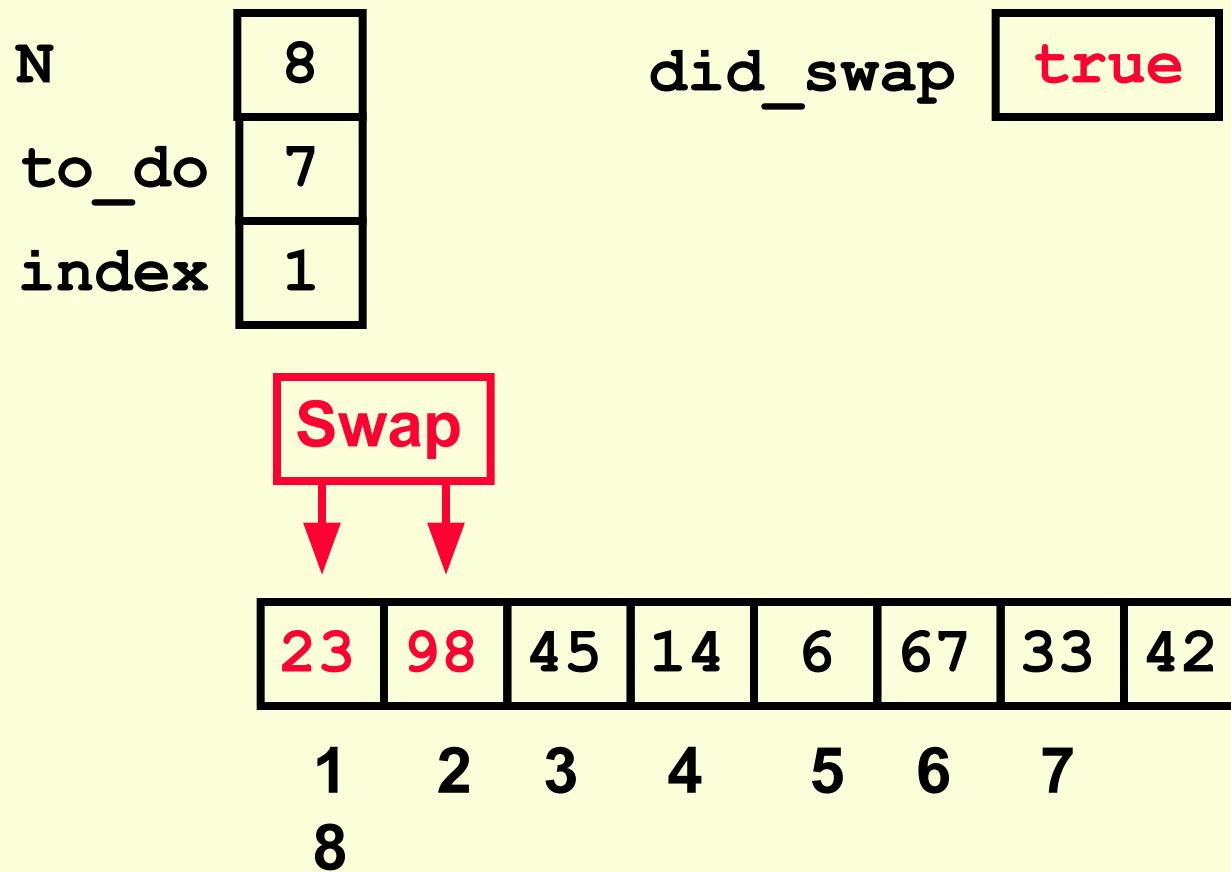
# An Animated Example



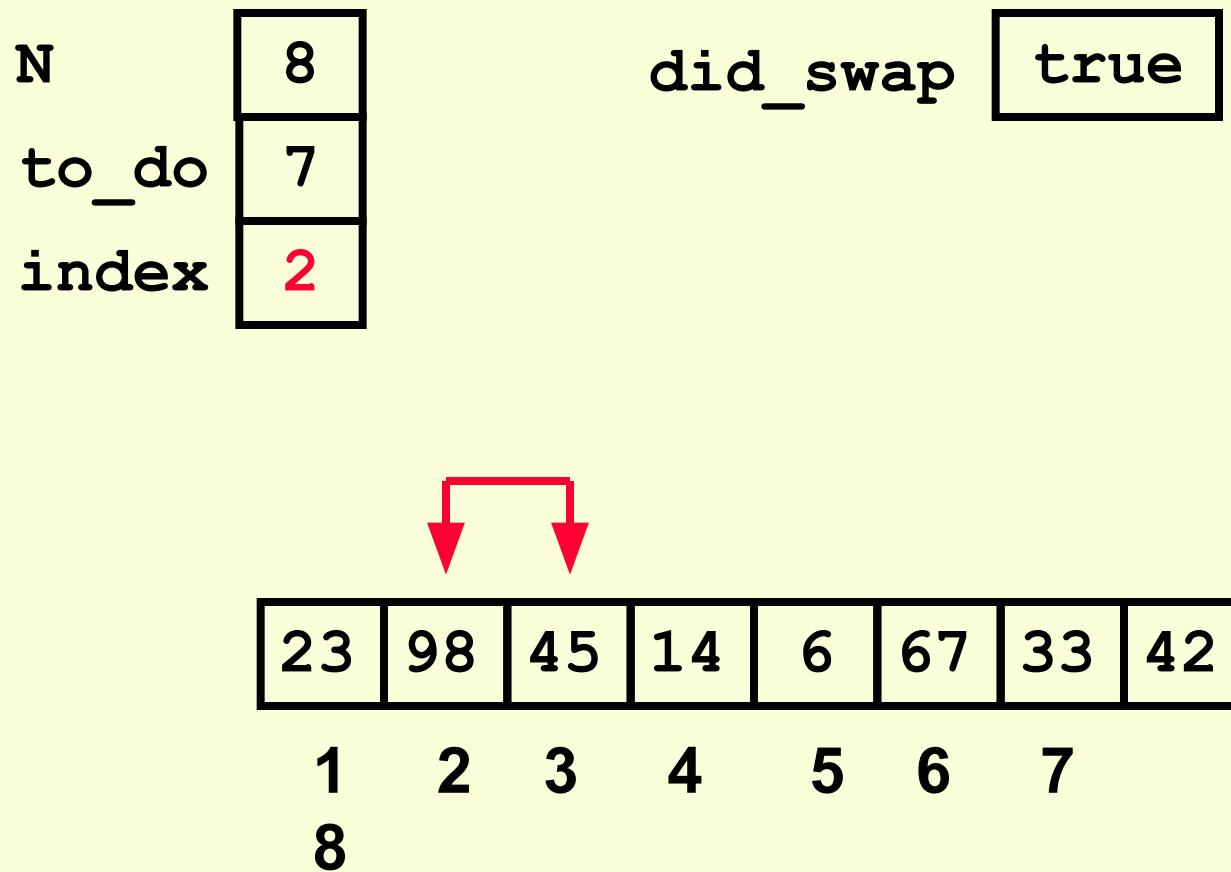
# An Animated Example



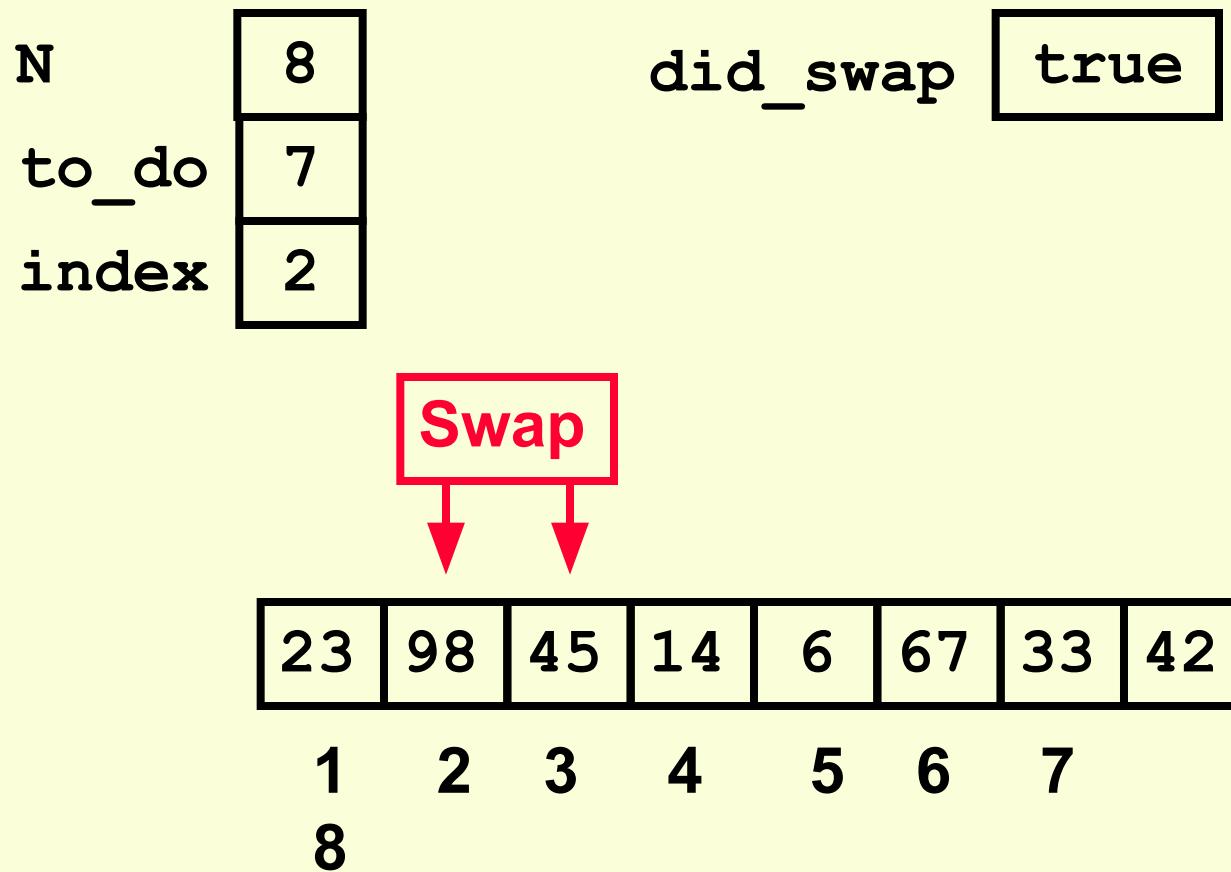
# An Animated Example



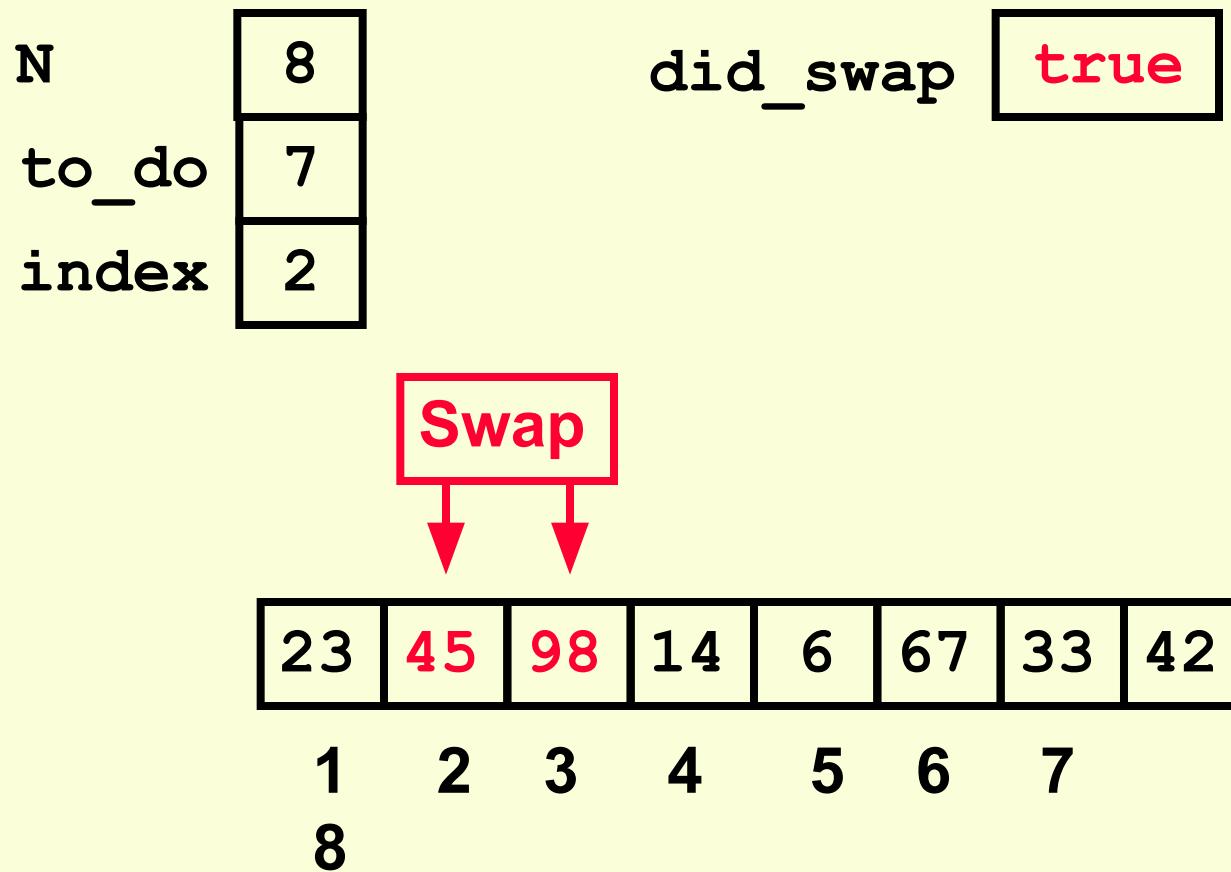
# An Animated Example



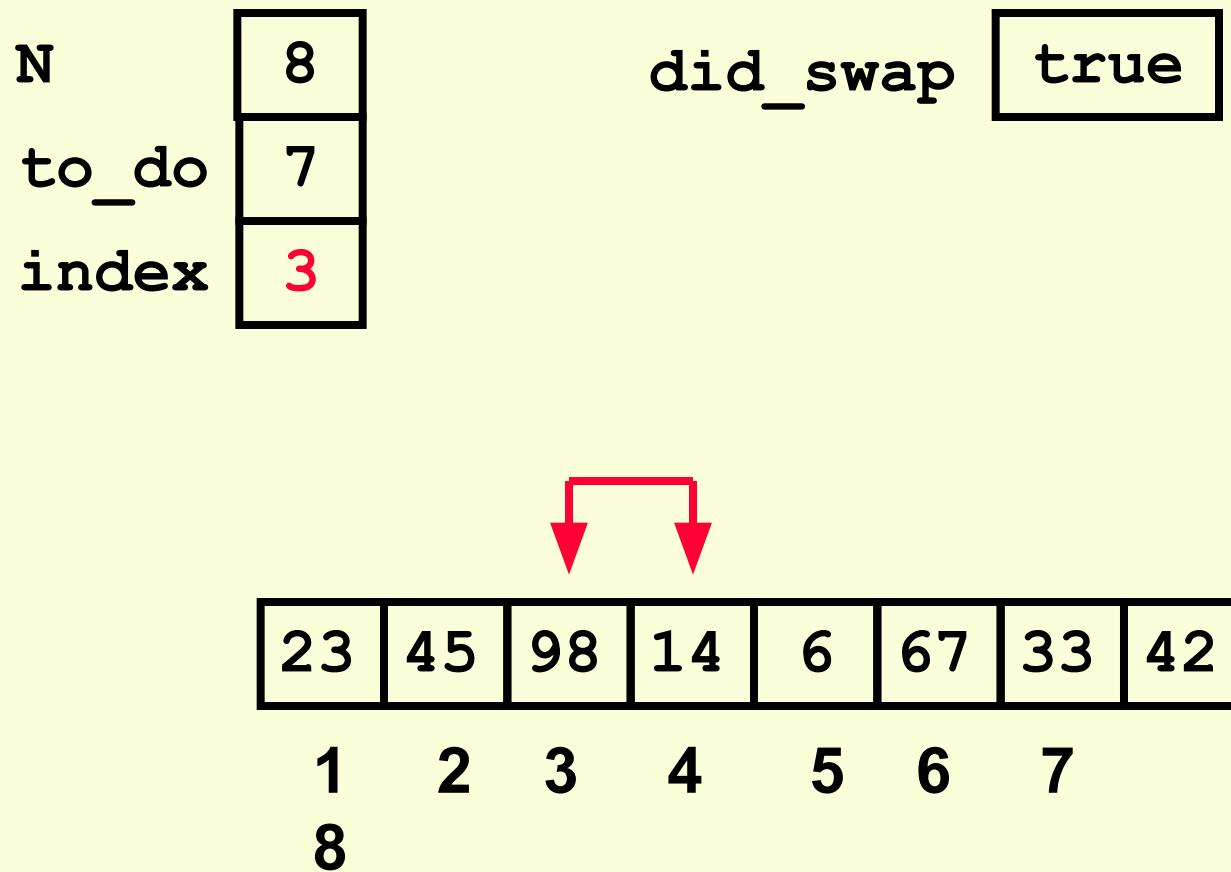
# An Animated Example



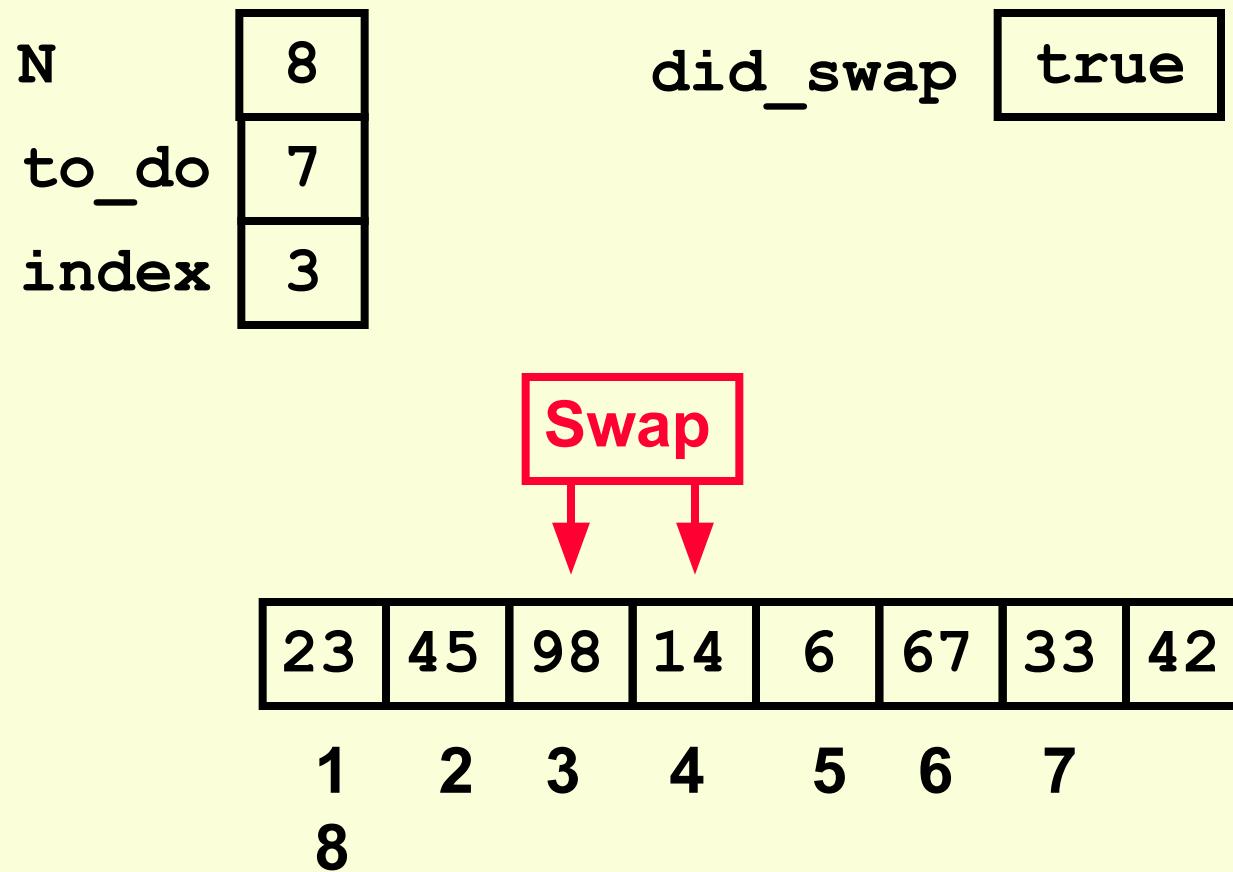
# An Animated Example



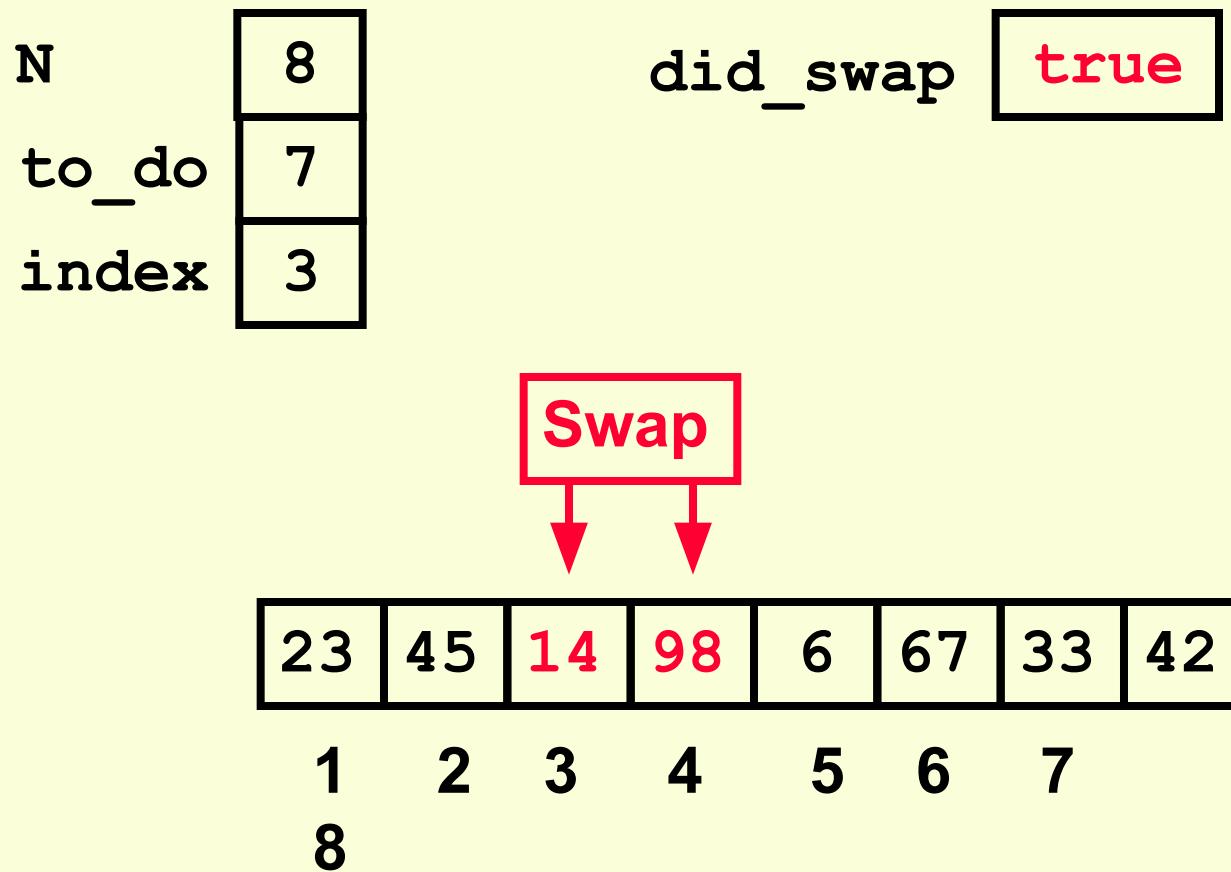
# An Animated Example



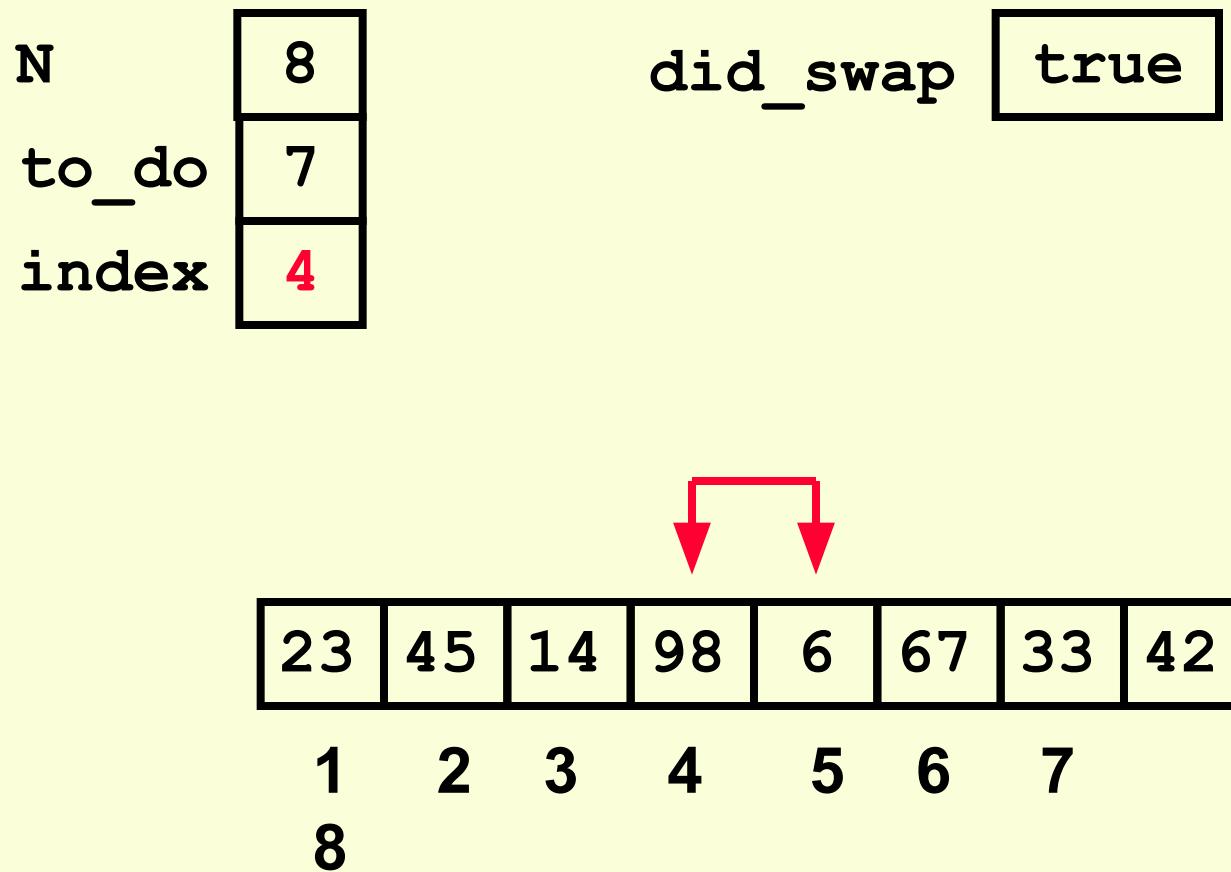
# An Animated Example



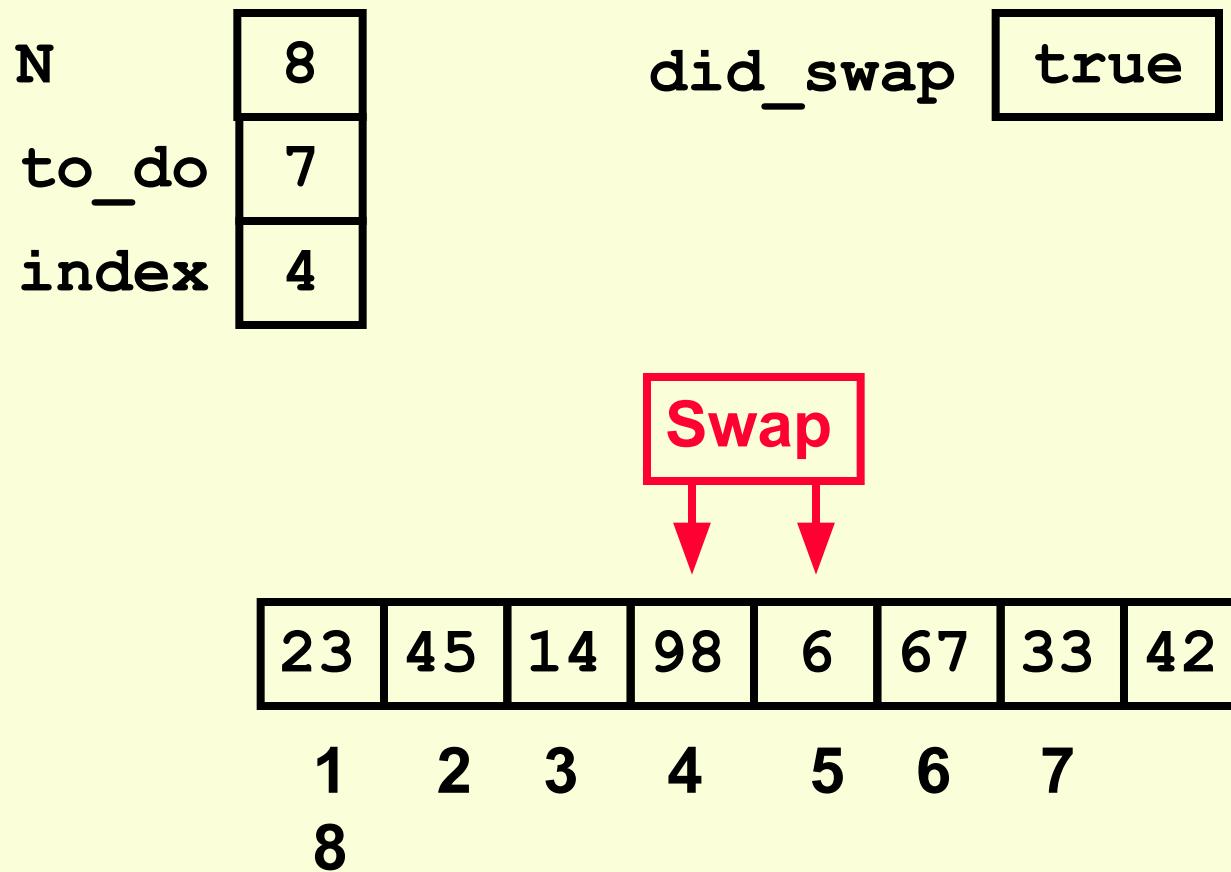
# An Animated Example



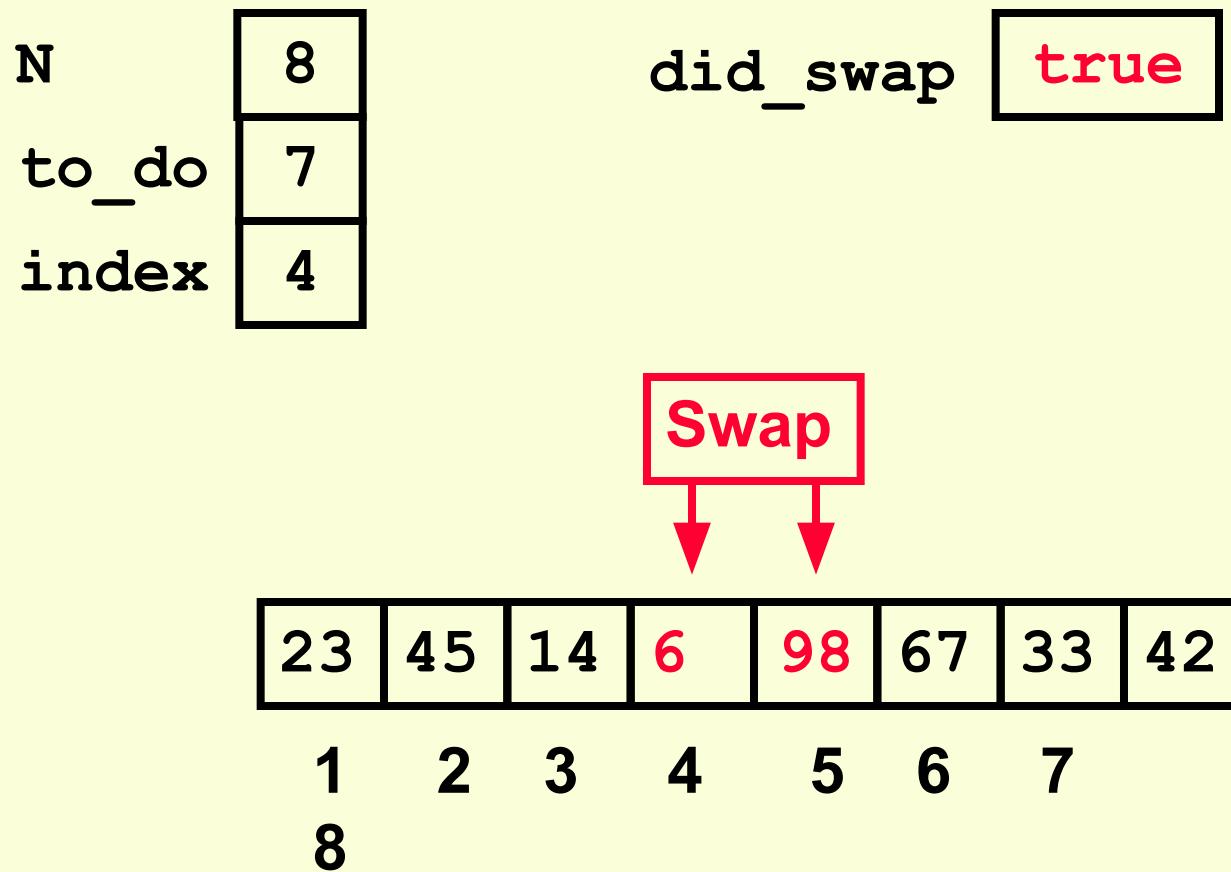
# An Animated Example



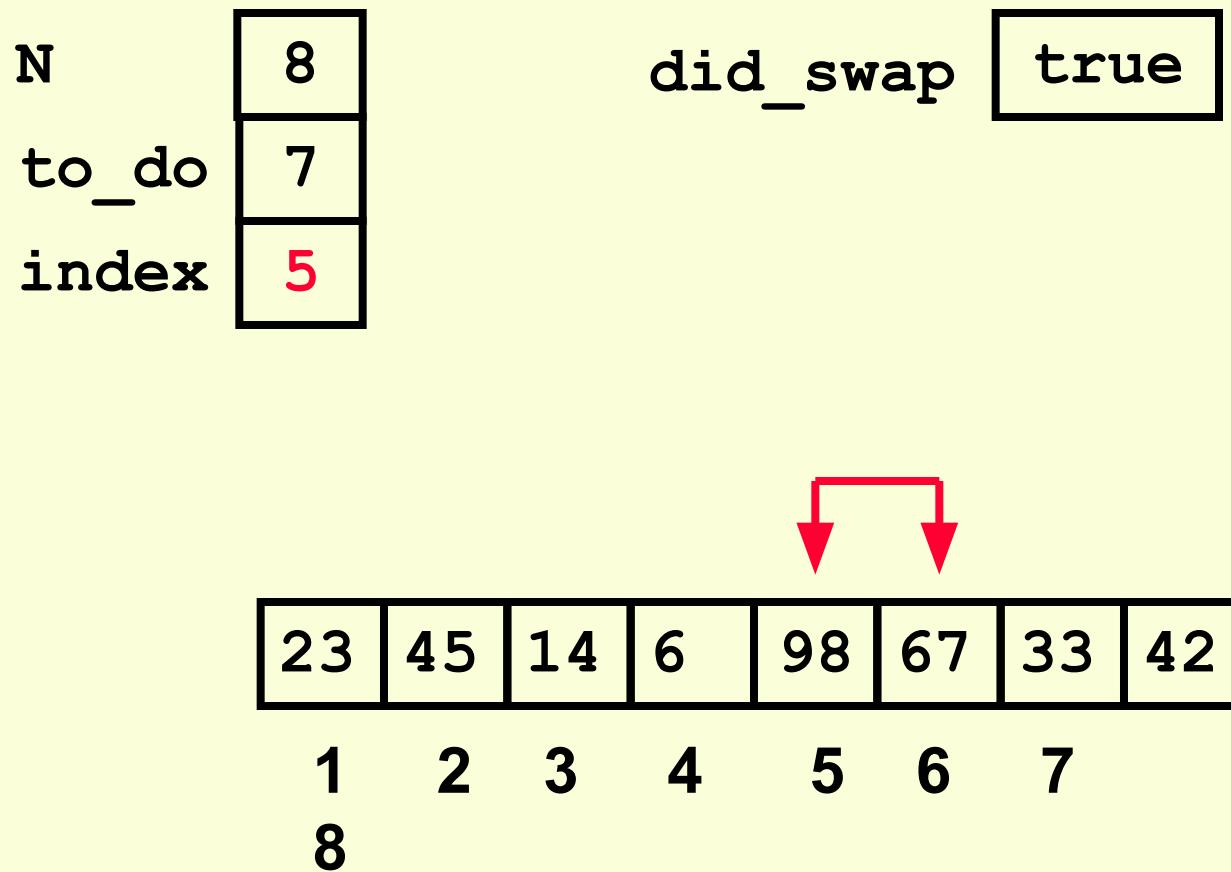
# An Animated Example



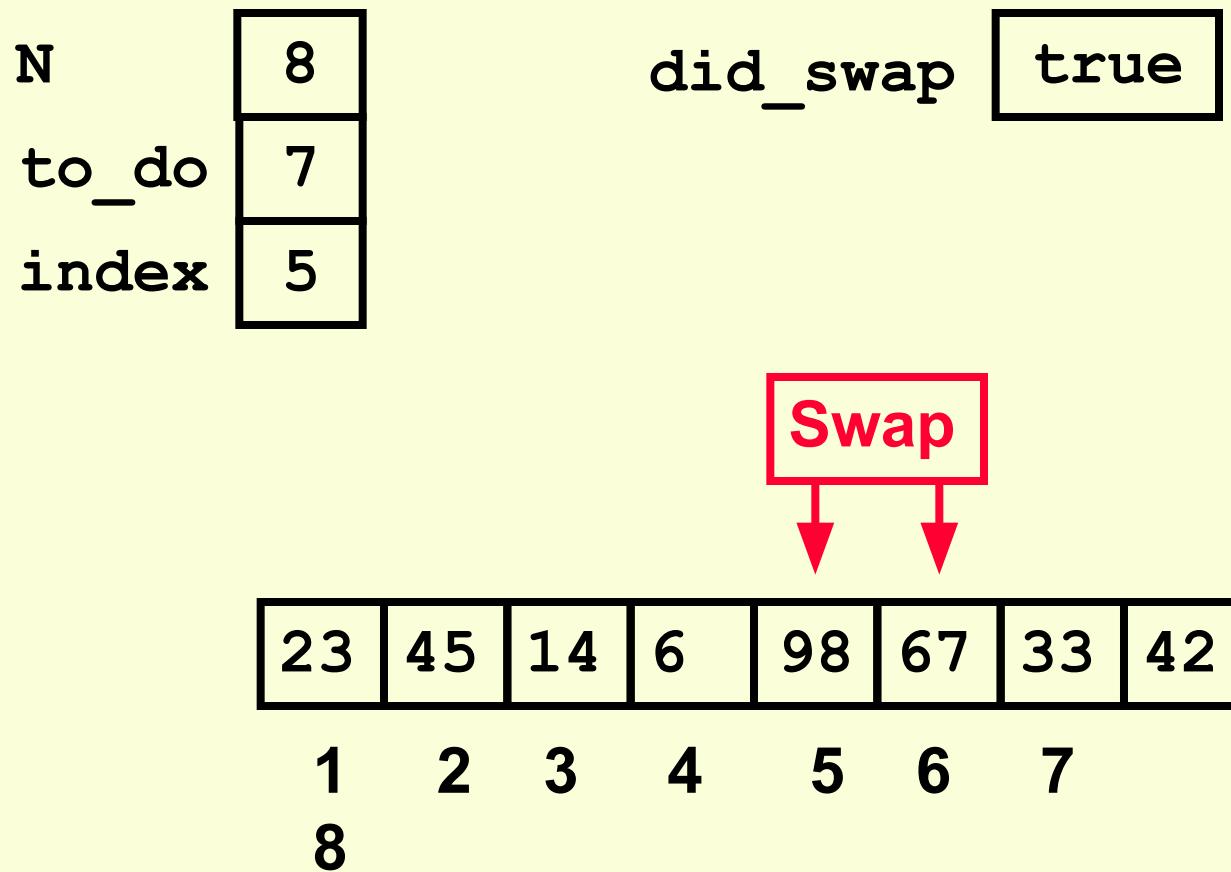
# An Animated Example



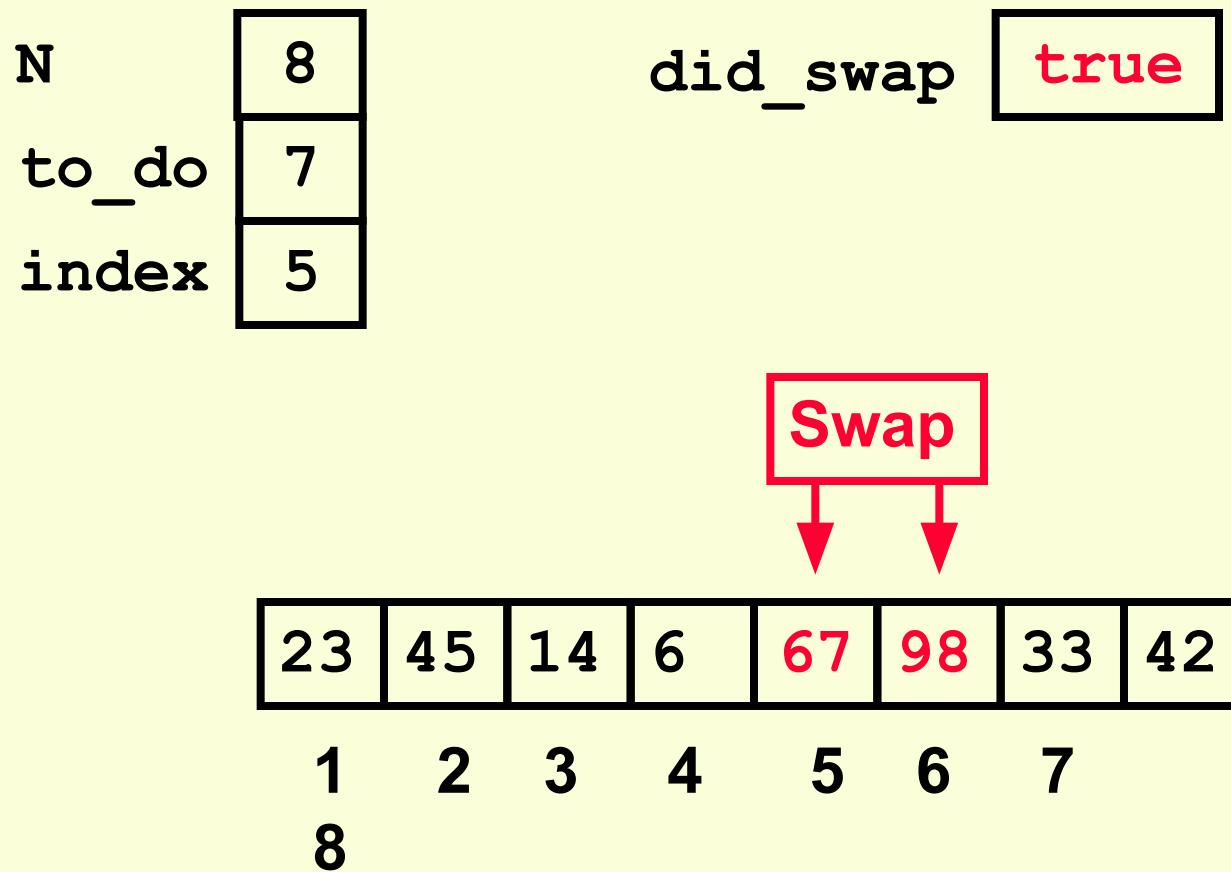
# An Animated Example



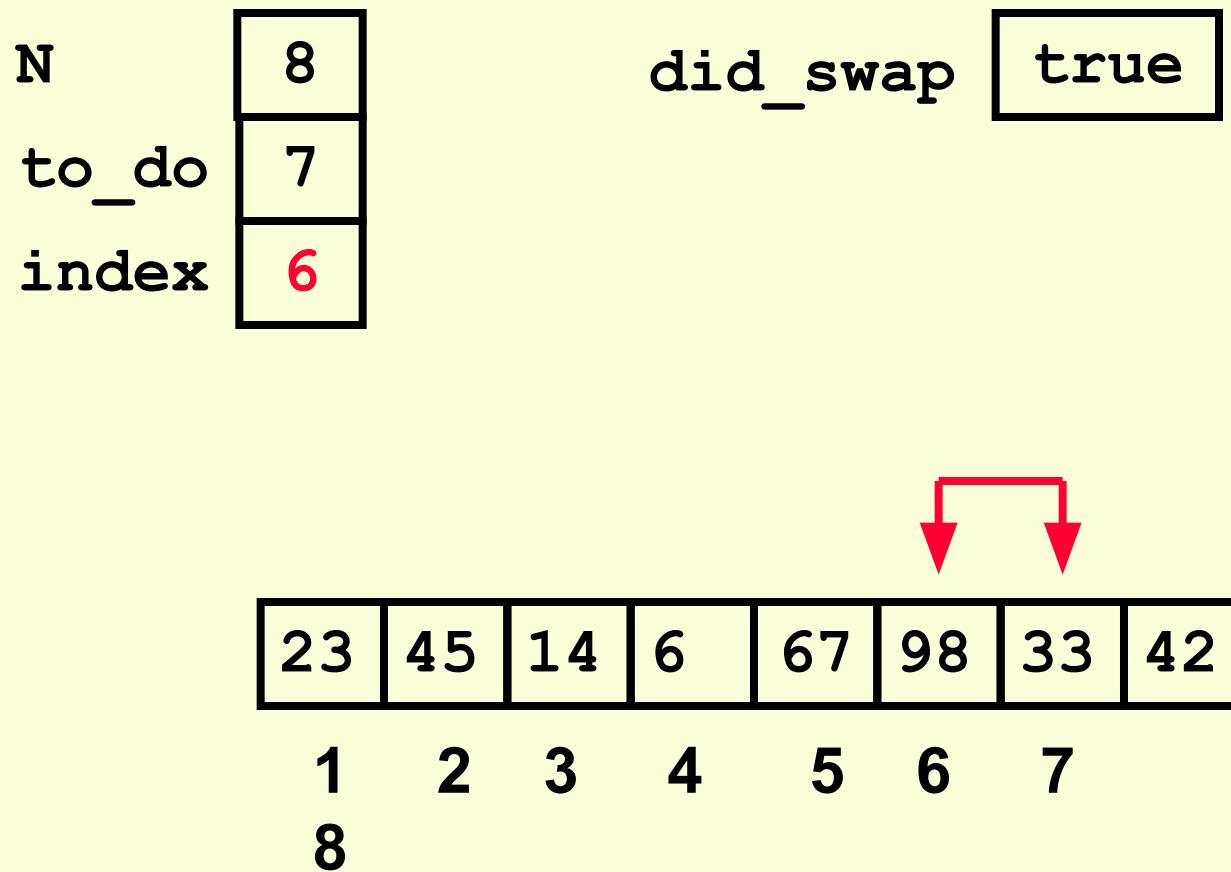
# An Animated Example



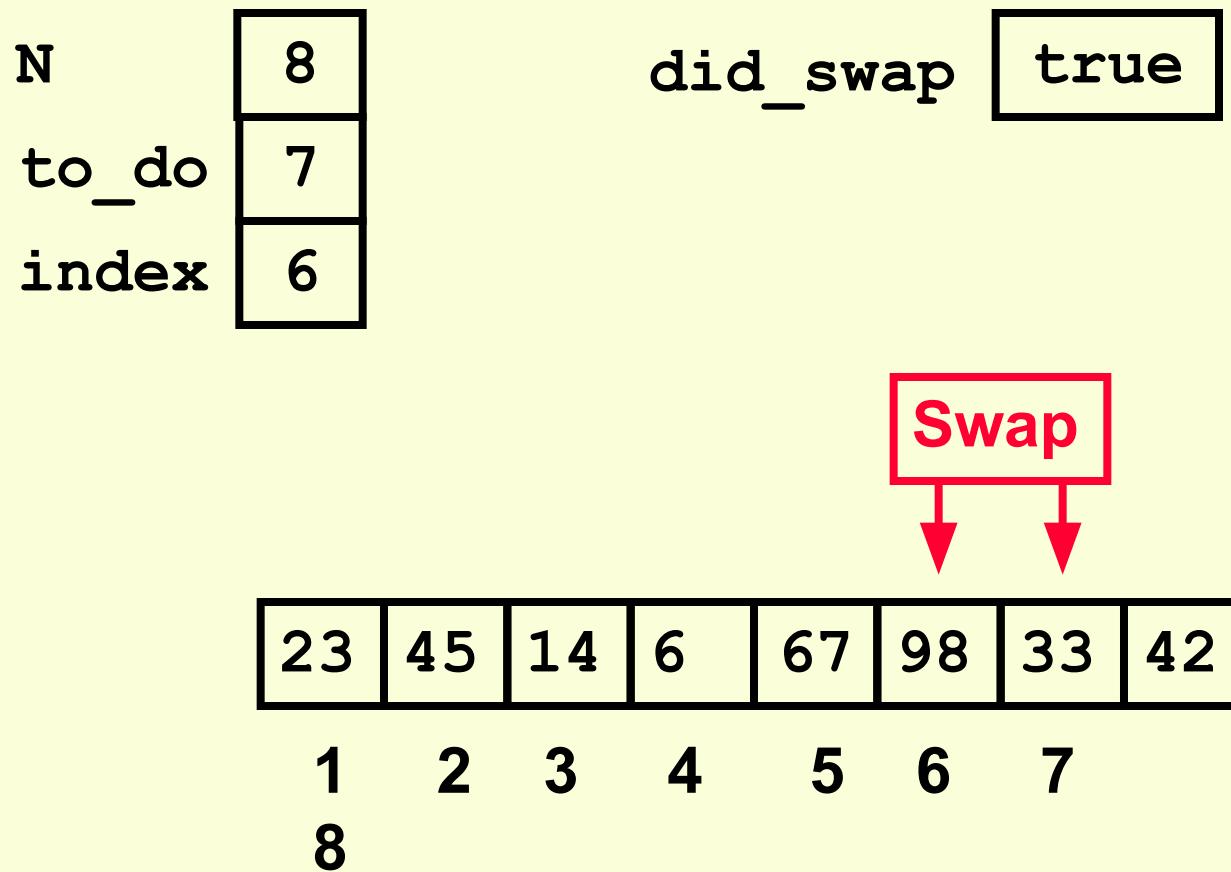
# An Animated Example



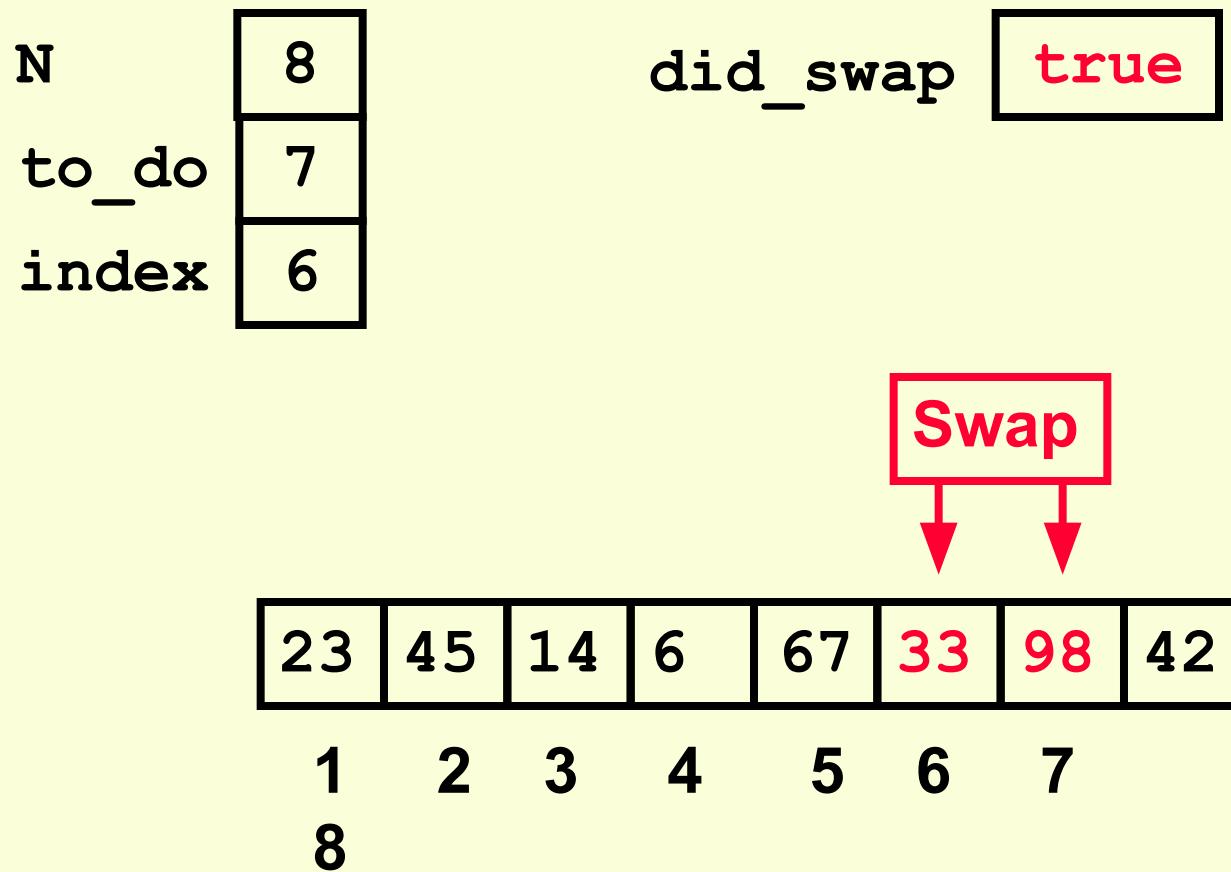
# An Animated Example



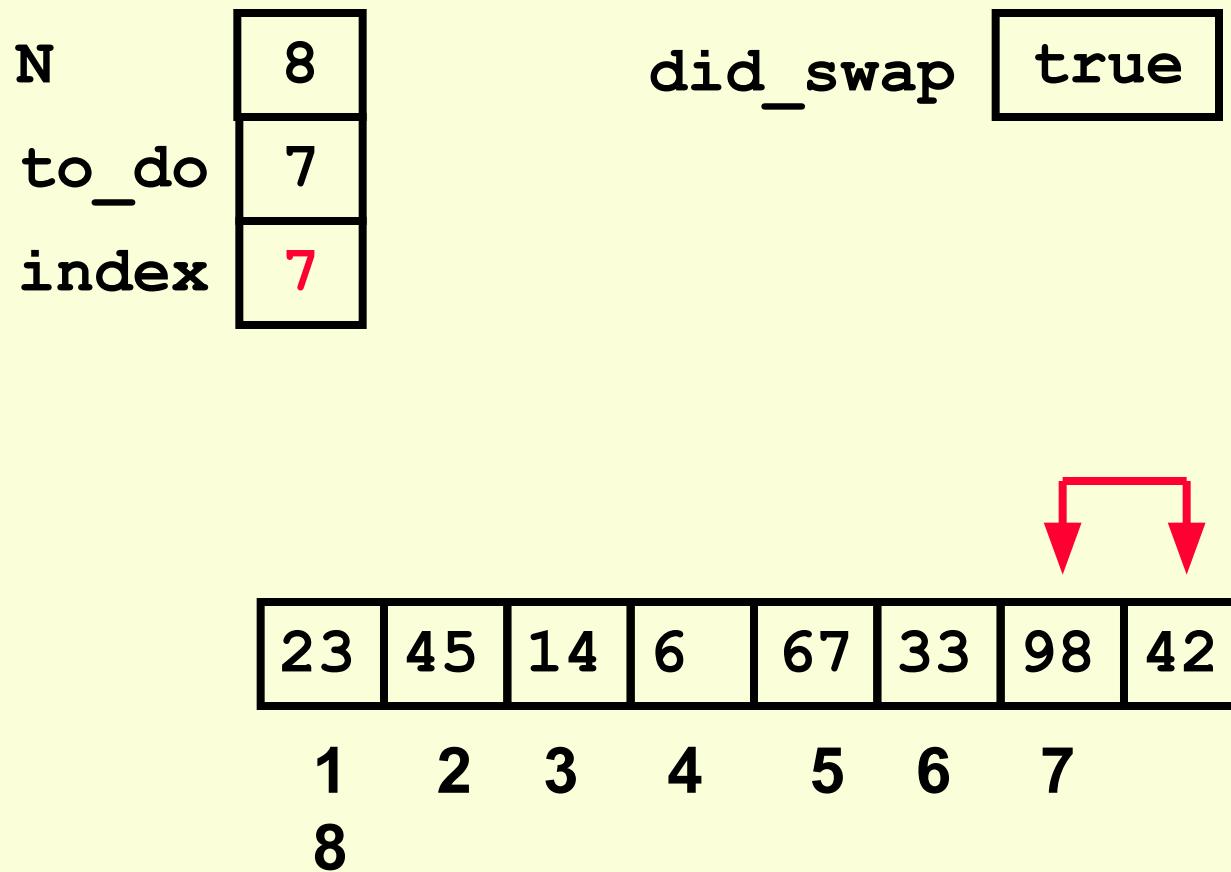
# An Animated Example



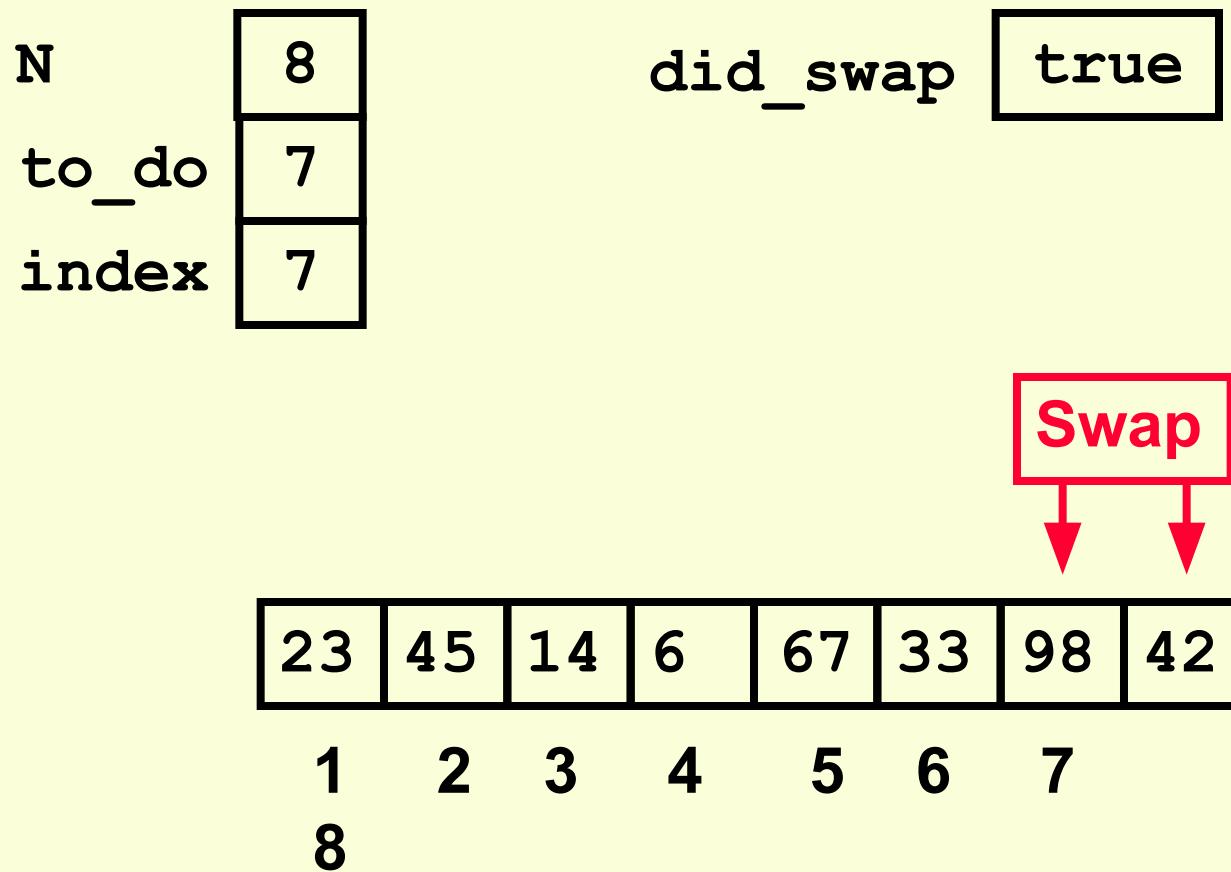
# An Animated Example



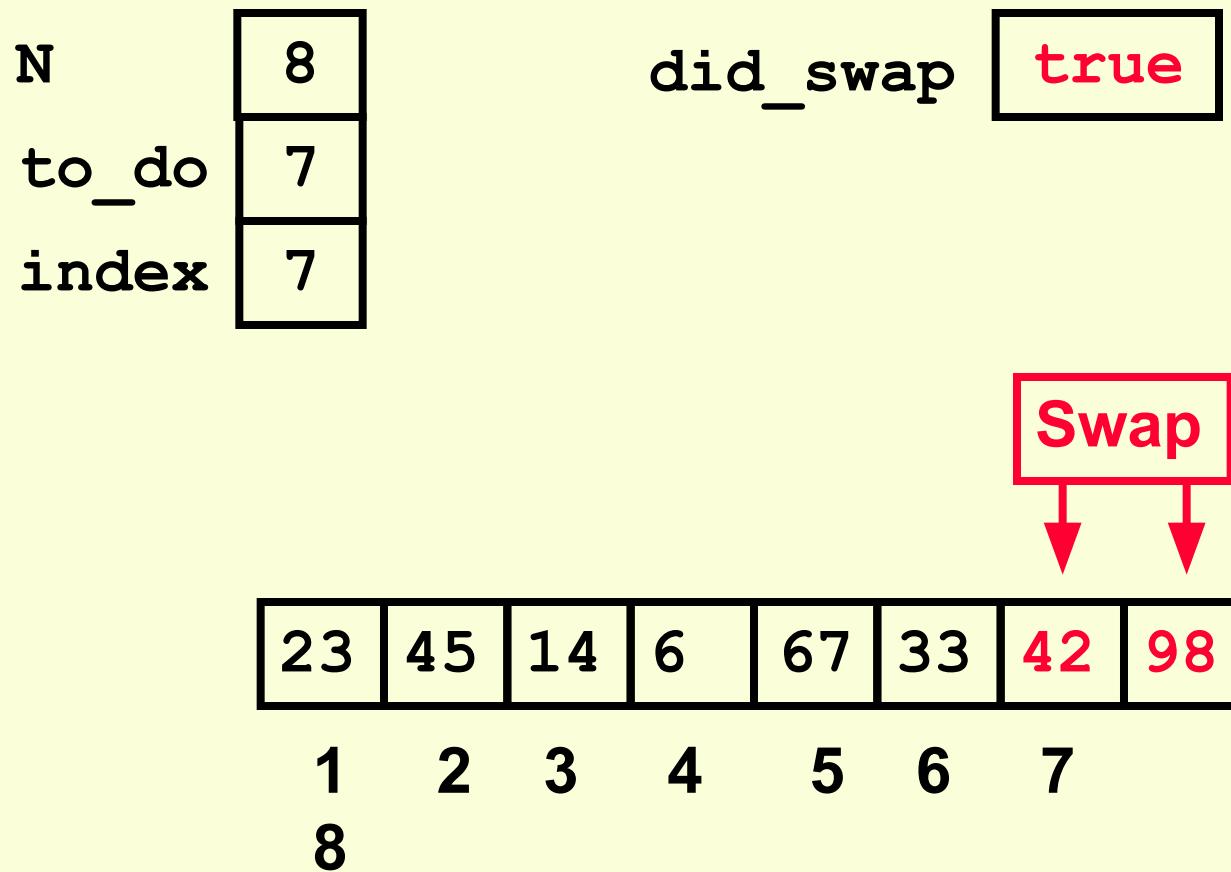
# An Animated Example



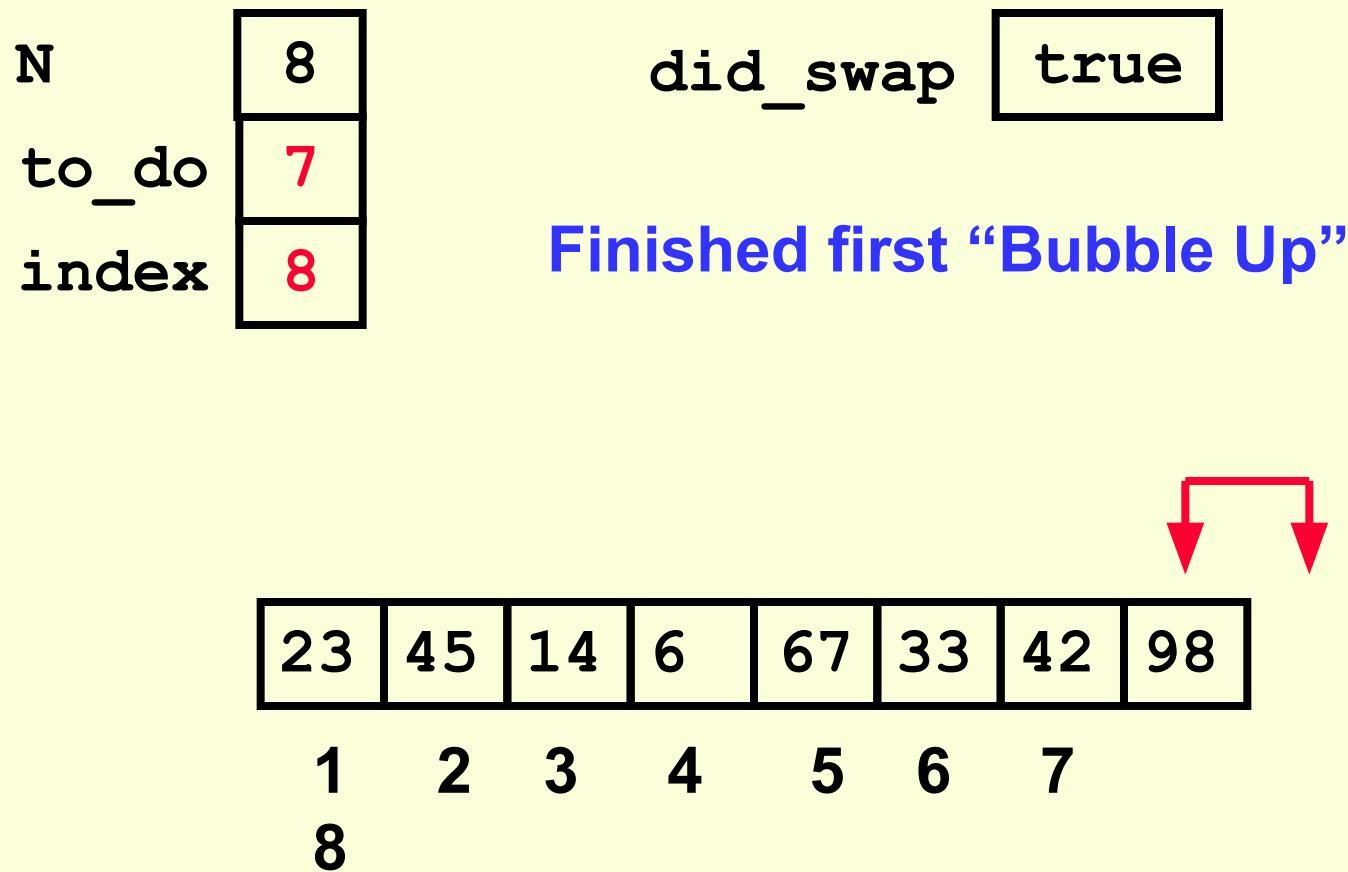
# An Animated Example



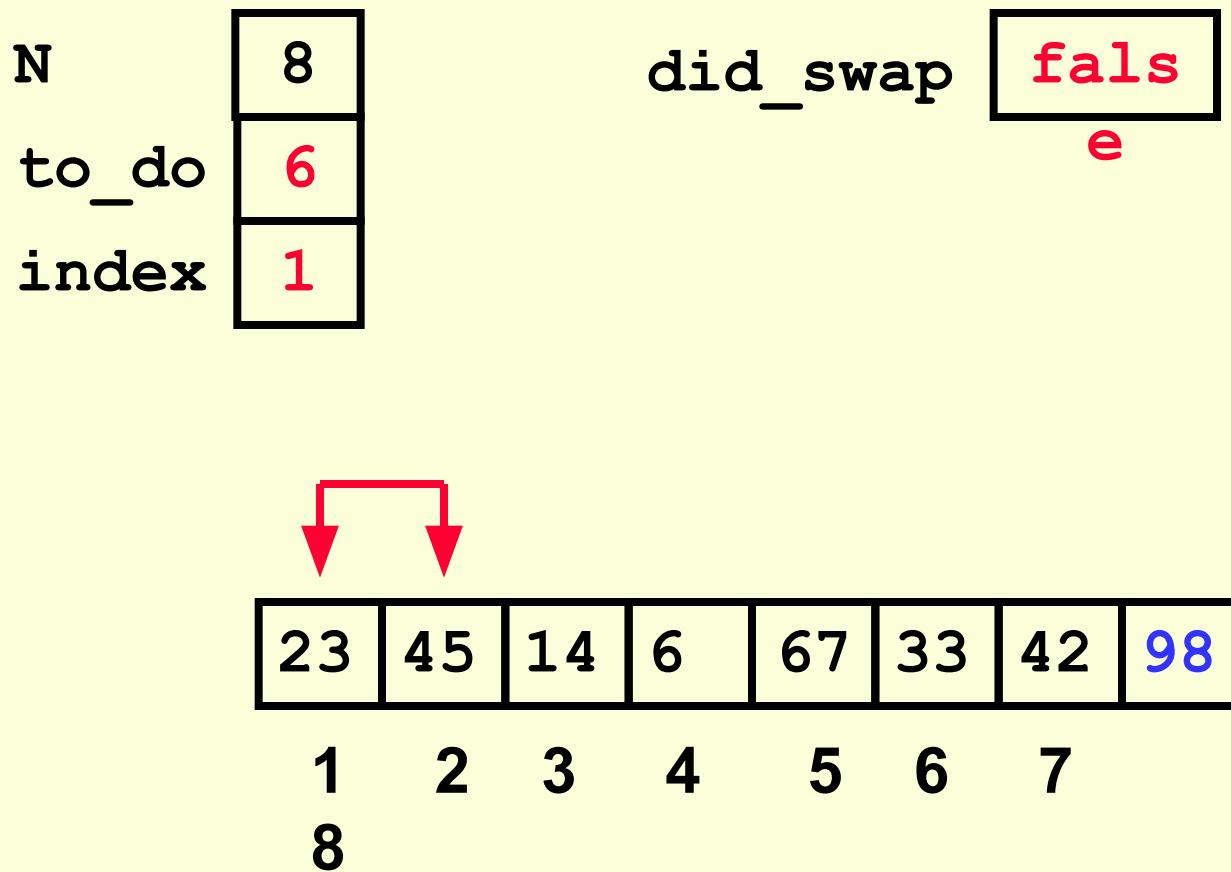
# An Animated Example



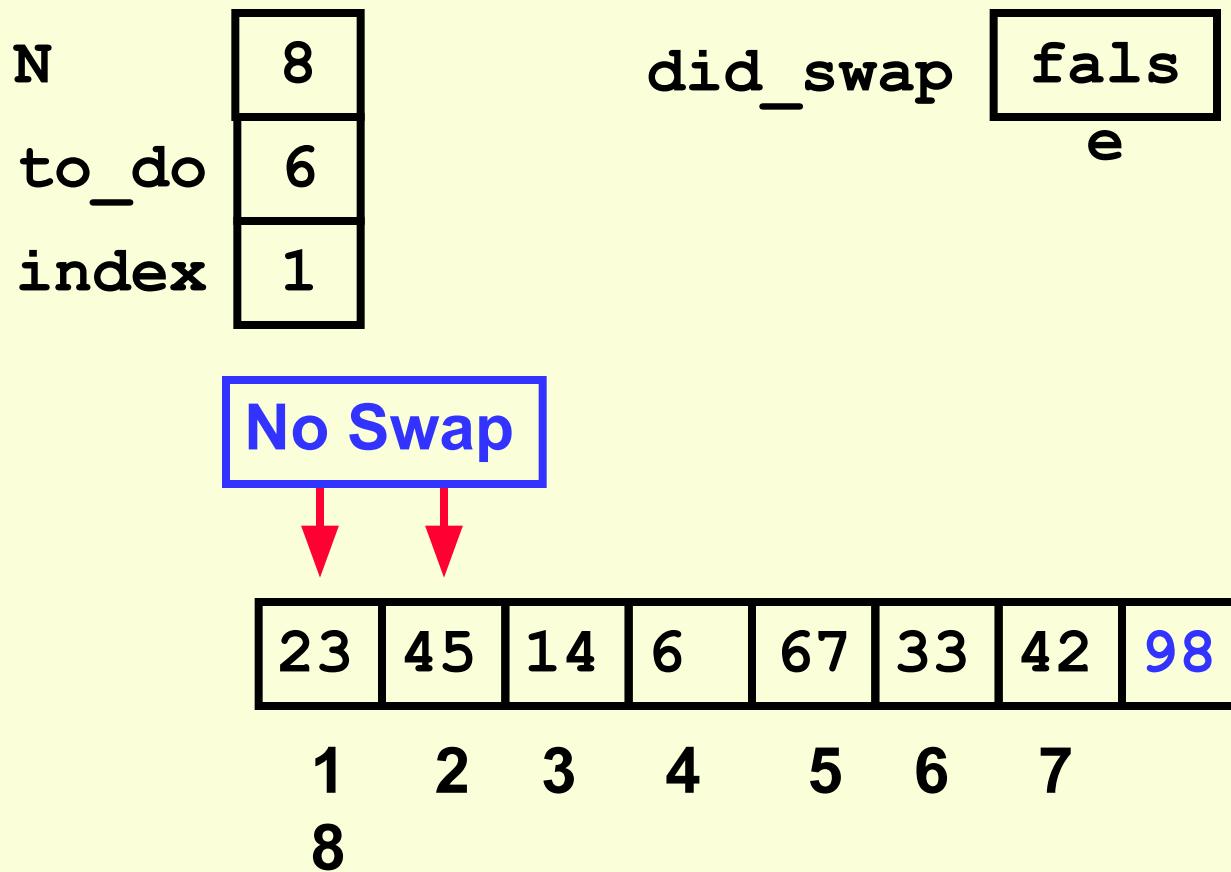
# After First Pass of Outer Loop



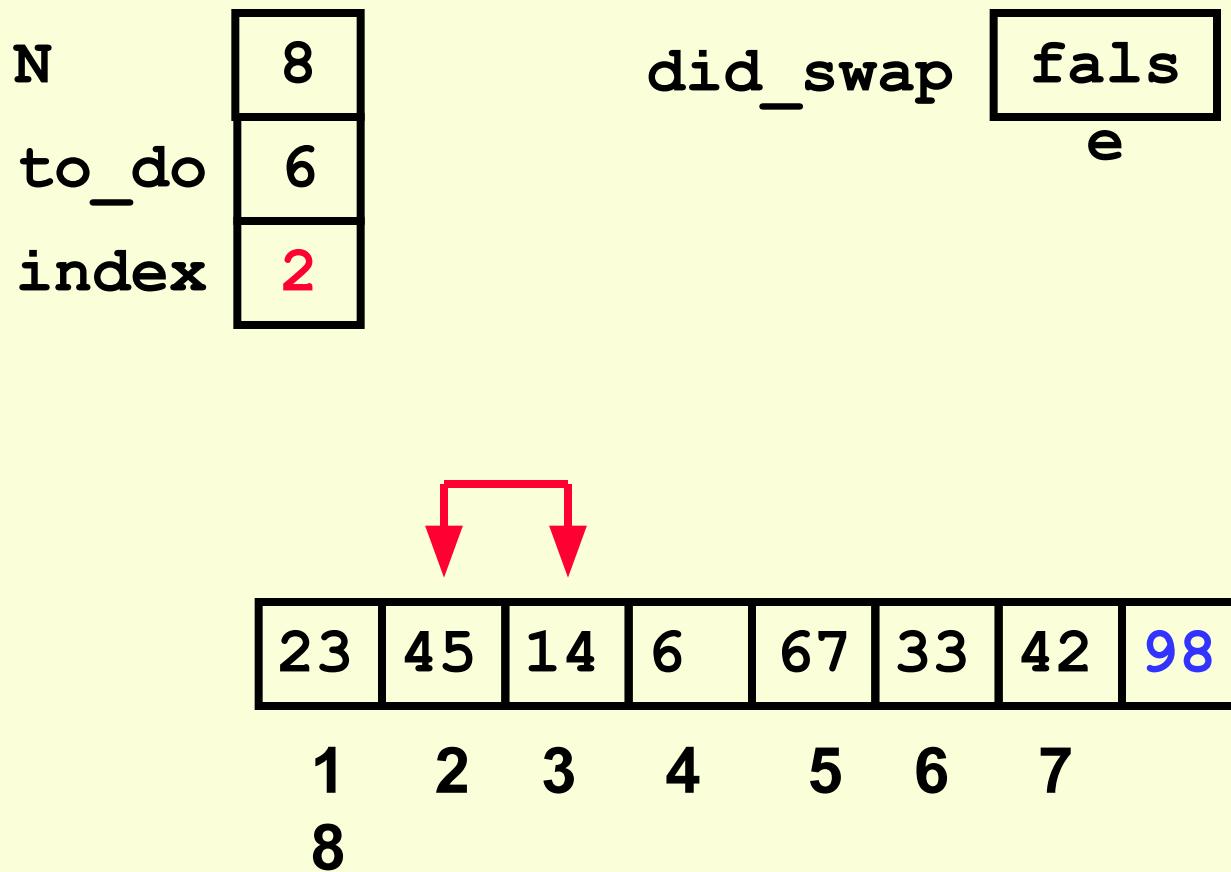
## The Second “Bubble Up”



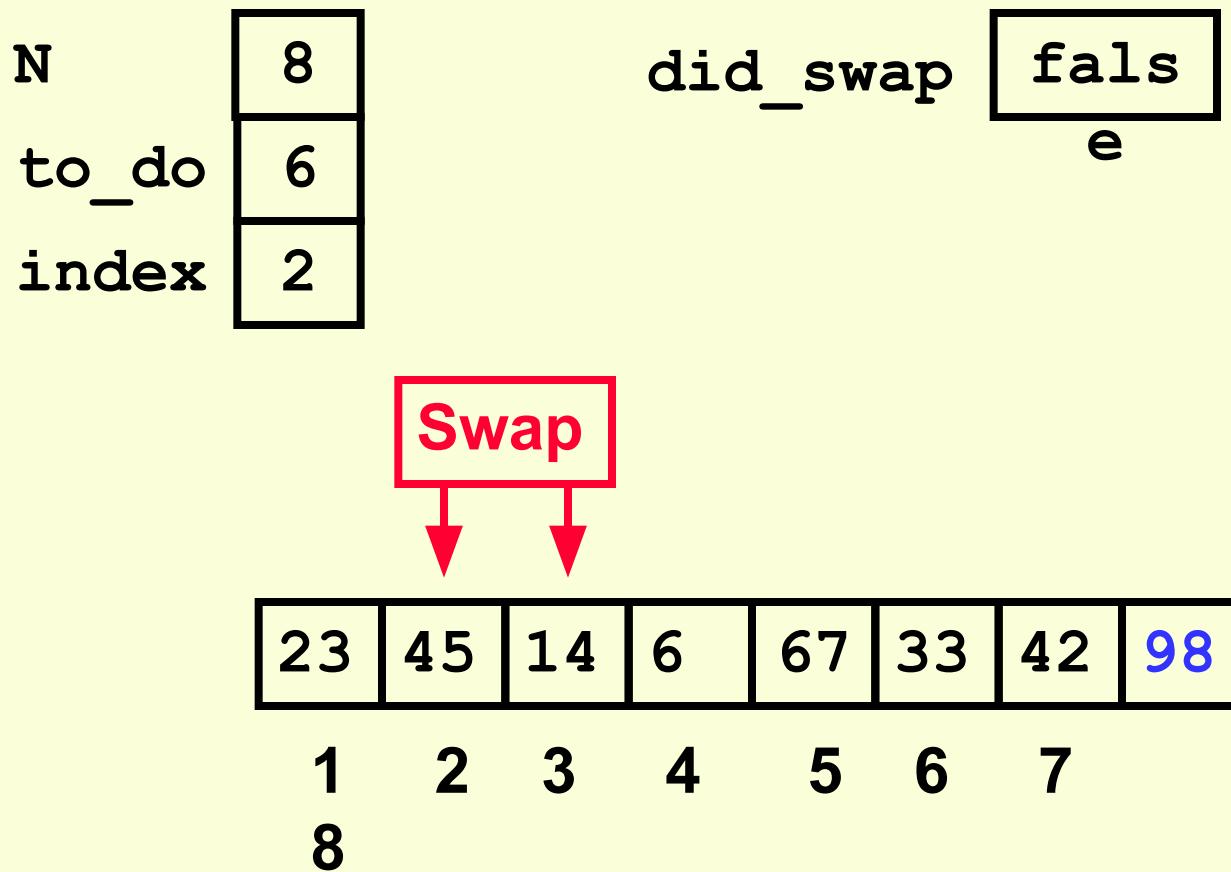
## The Second “Bubble Up”



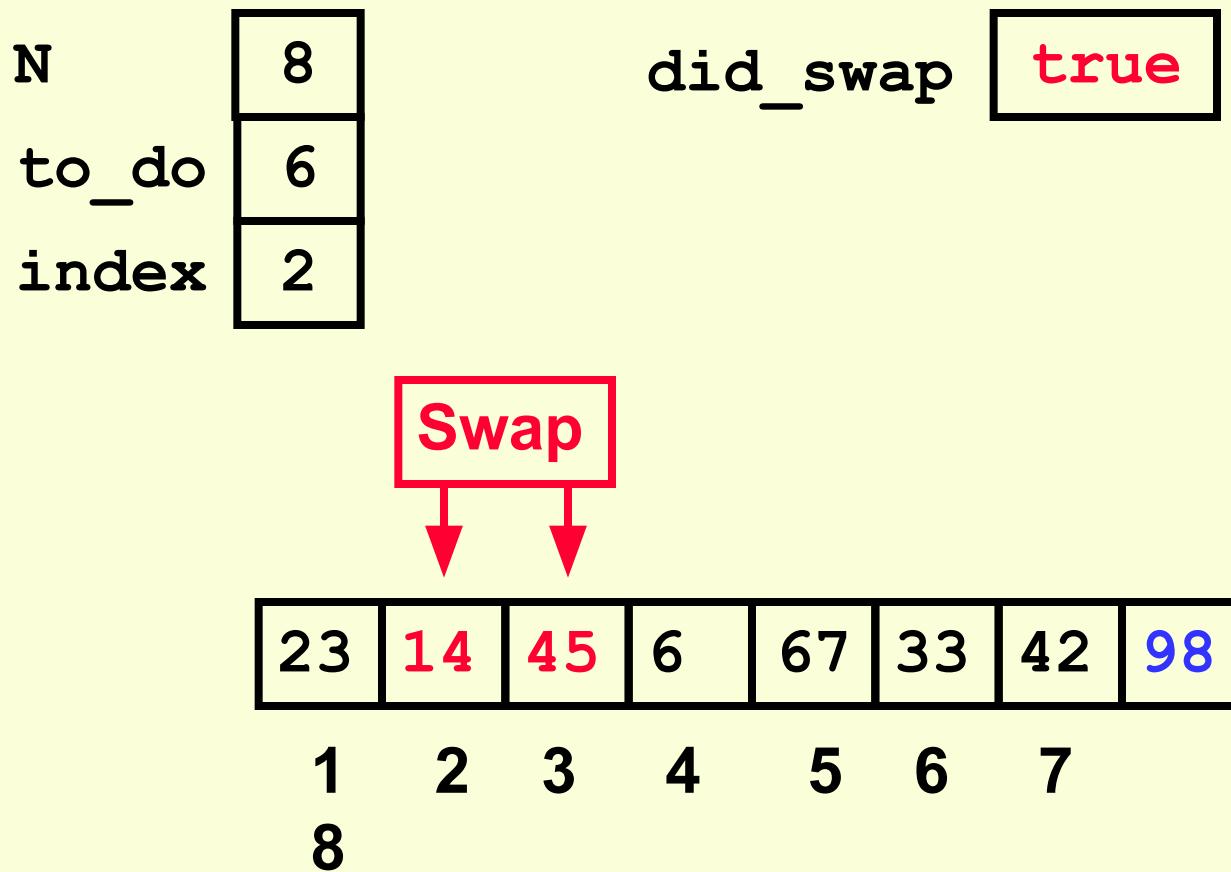
## The Second “Bubble Up”



## The Second “Bubble Up”



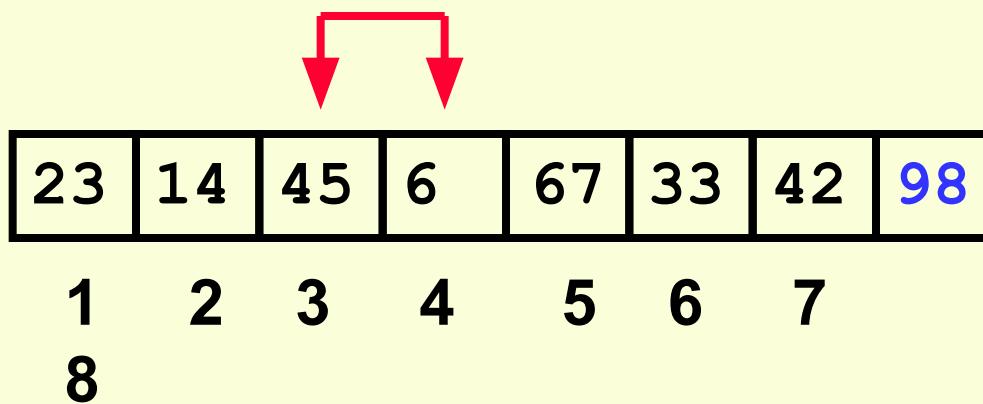
## The Second “Bubble Up”



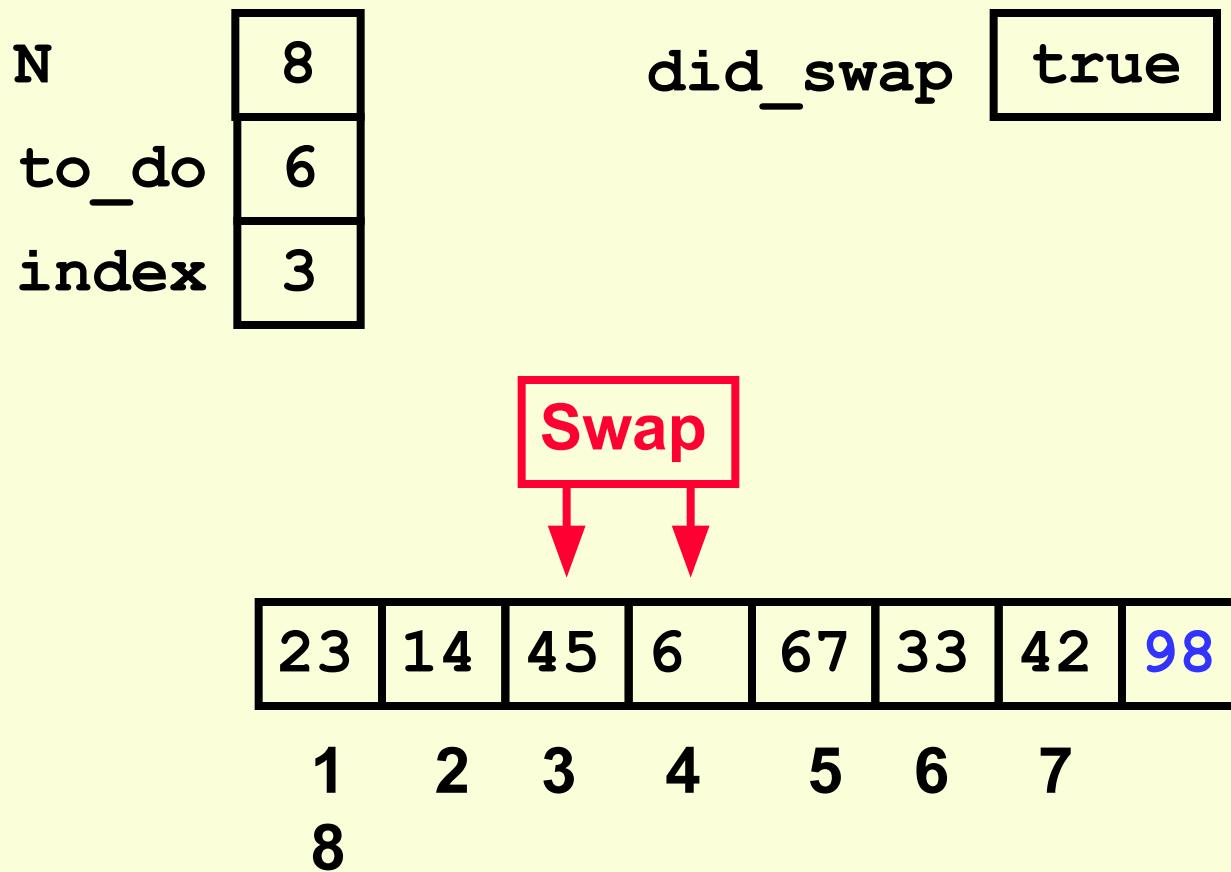
## The Second “Bubble Up”

N	8
to_do	6
index	3

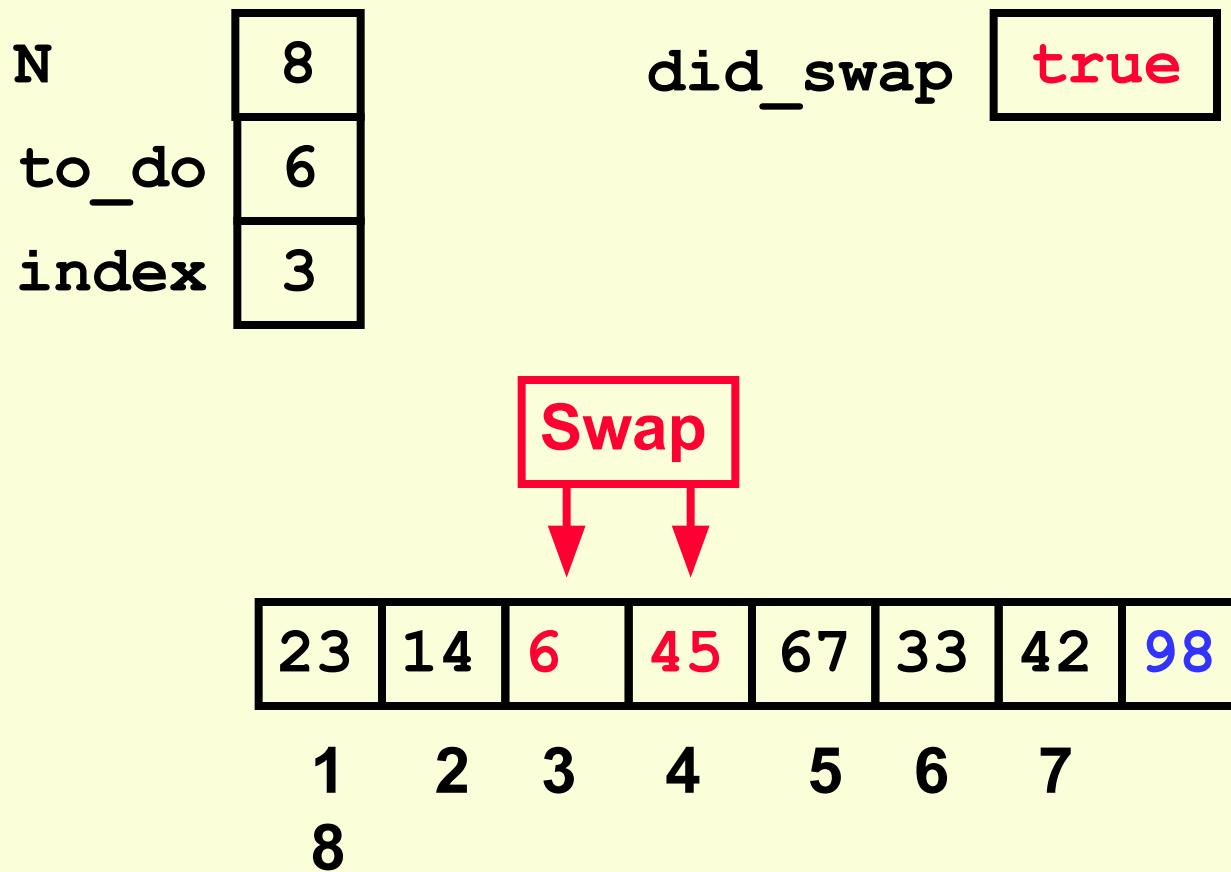
did\_swap      true



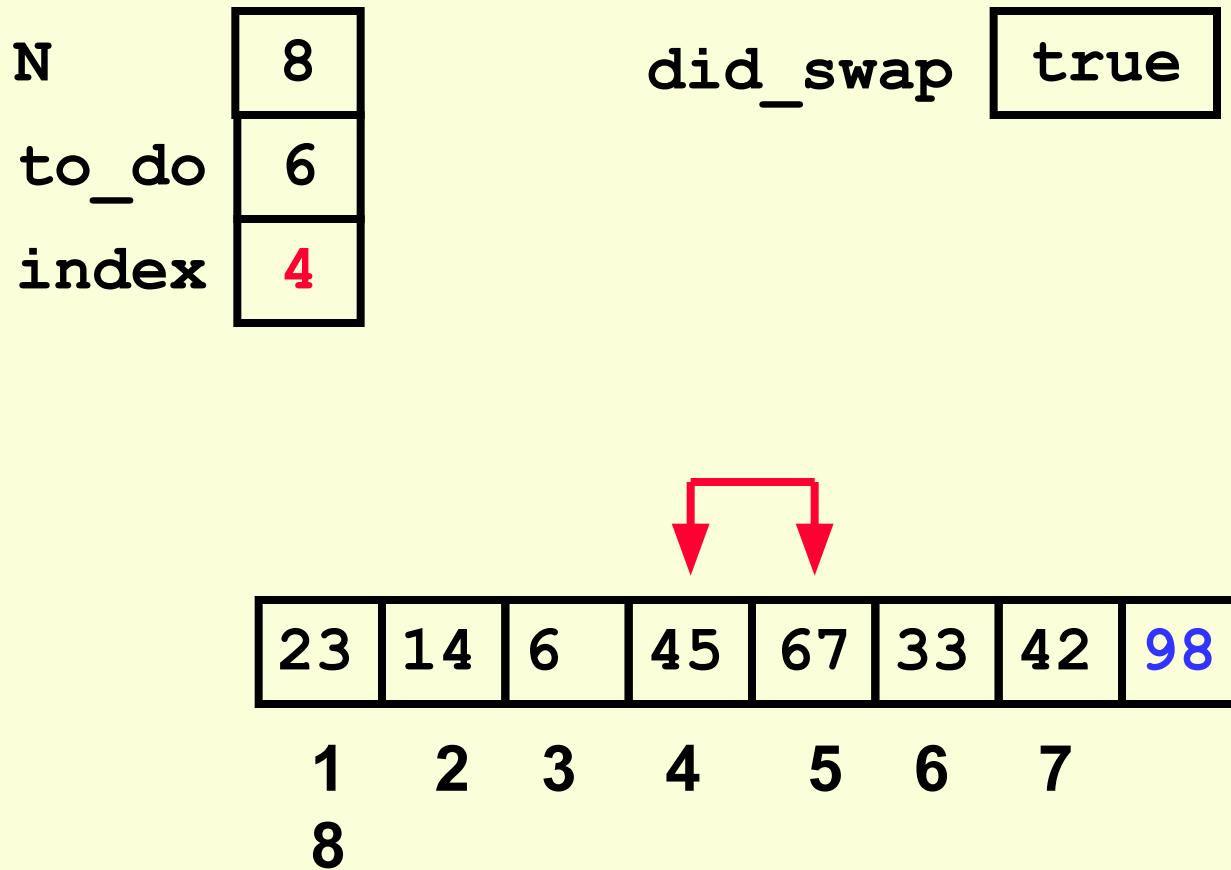
## The Second “Bubble Up”



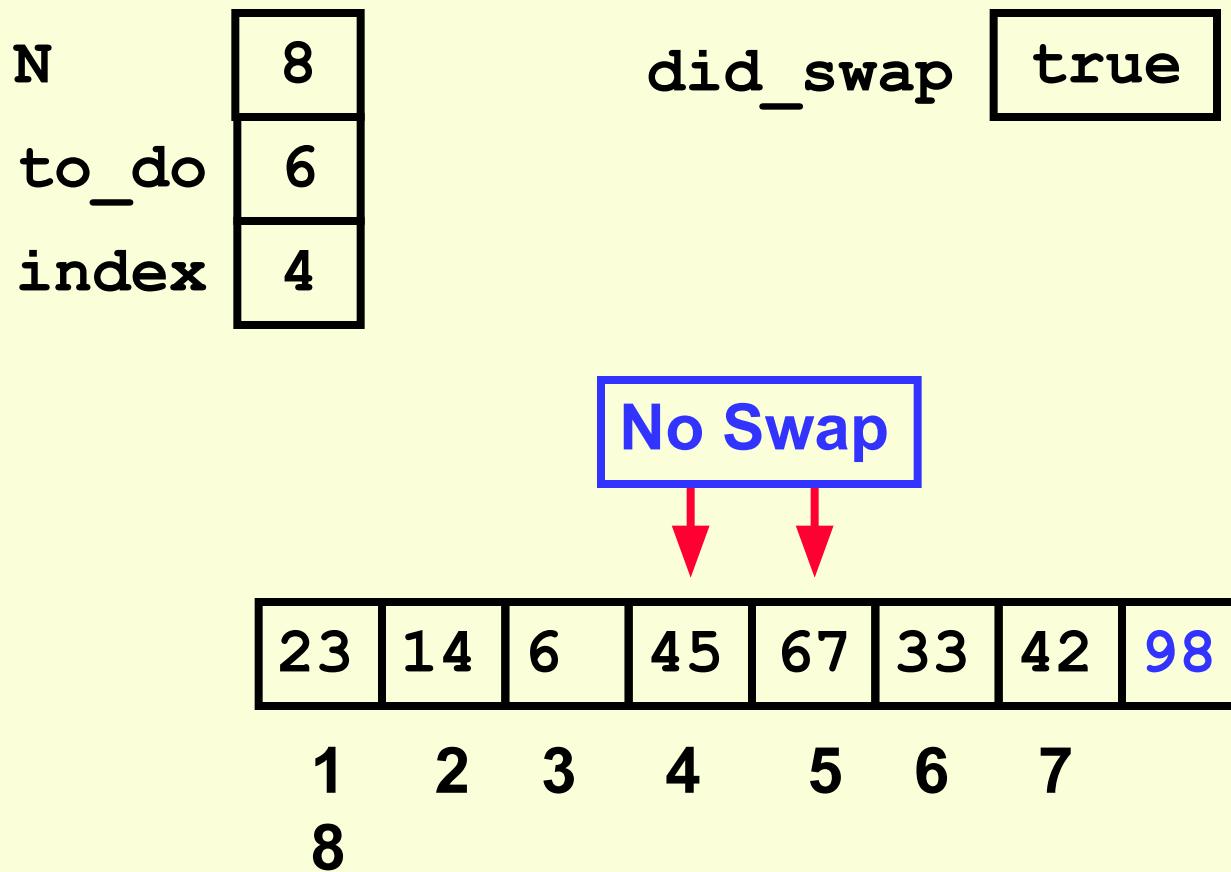
## The Second “Bubble Up”



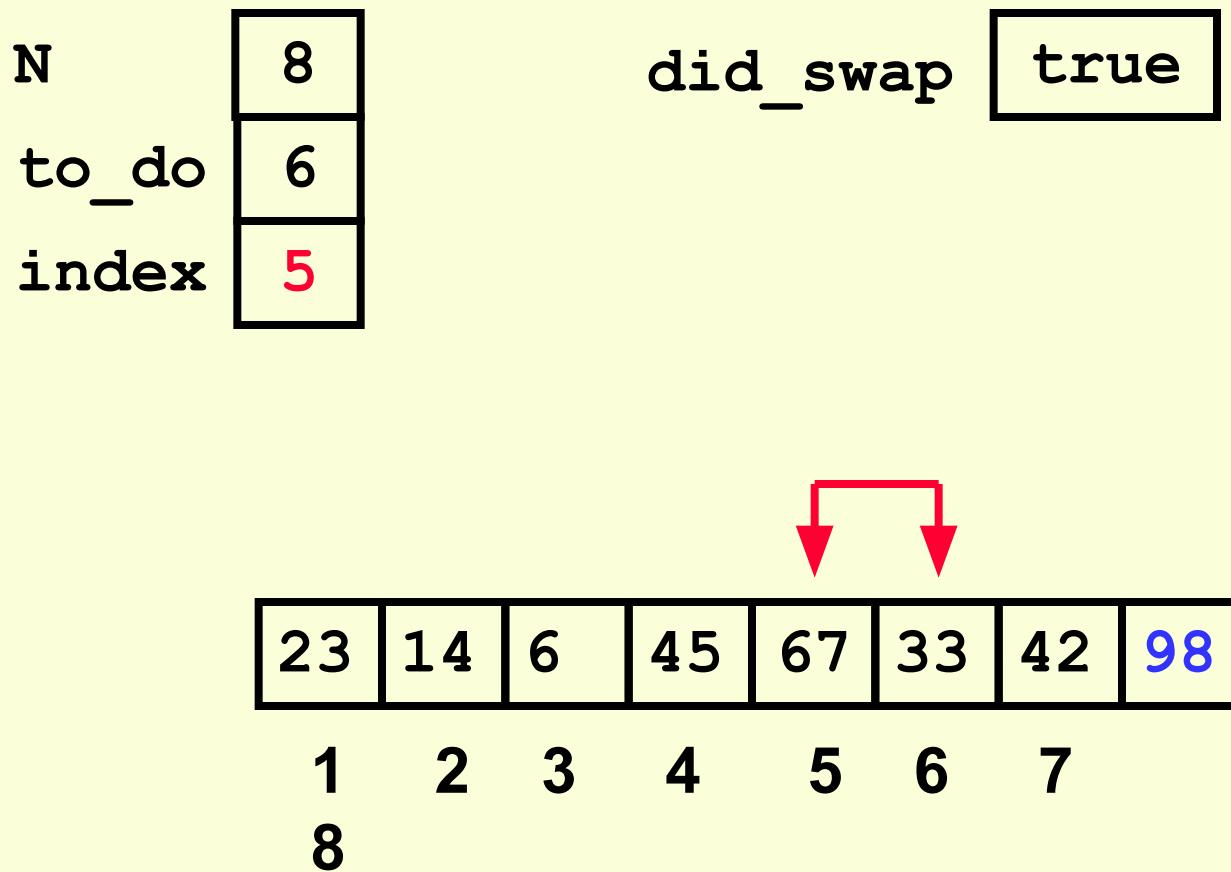
## The Second “Bubble Up”



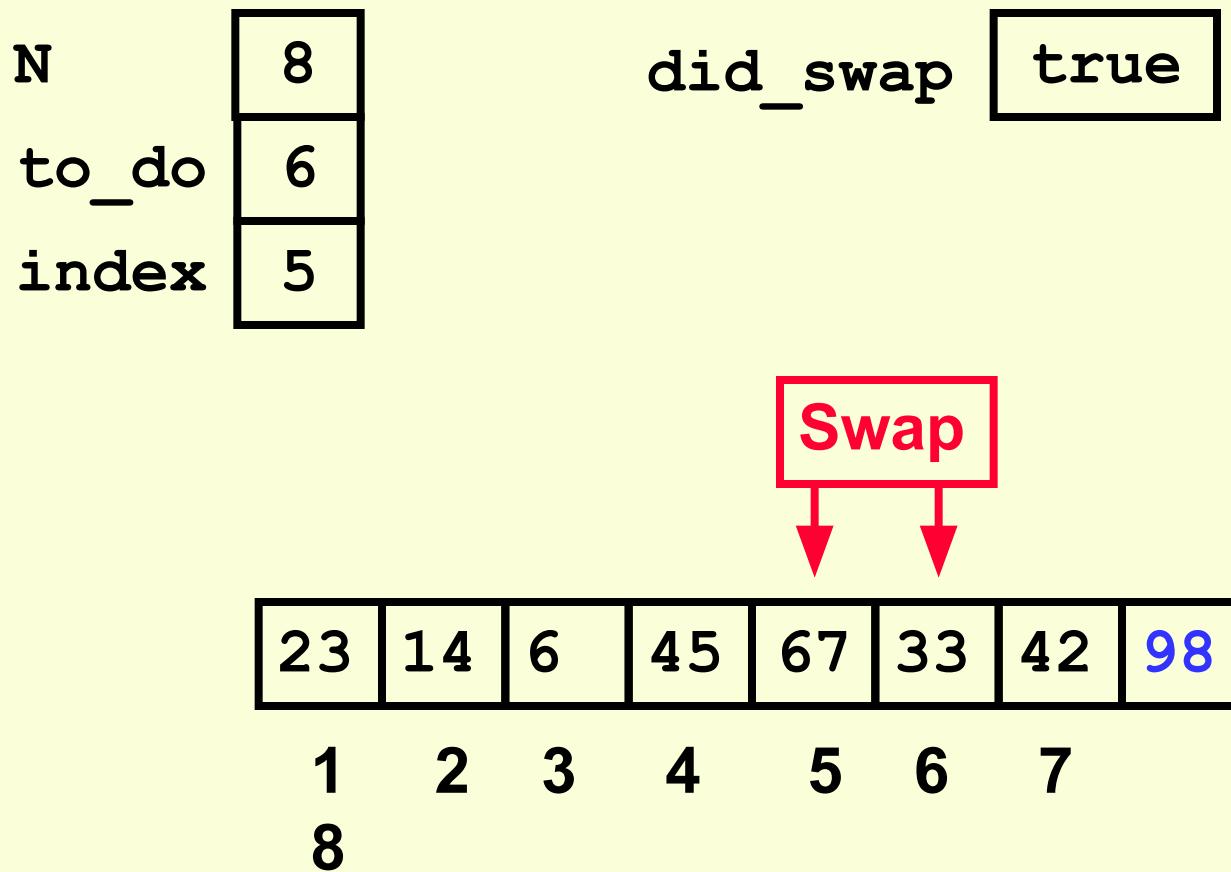
## The Second “Bubble Up”



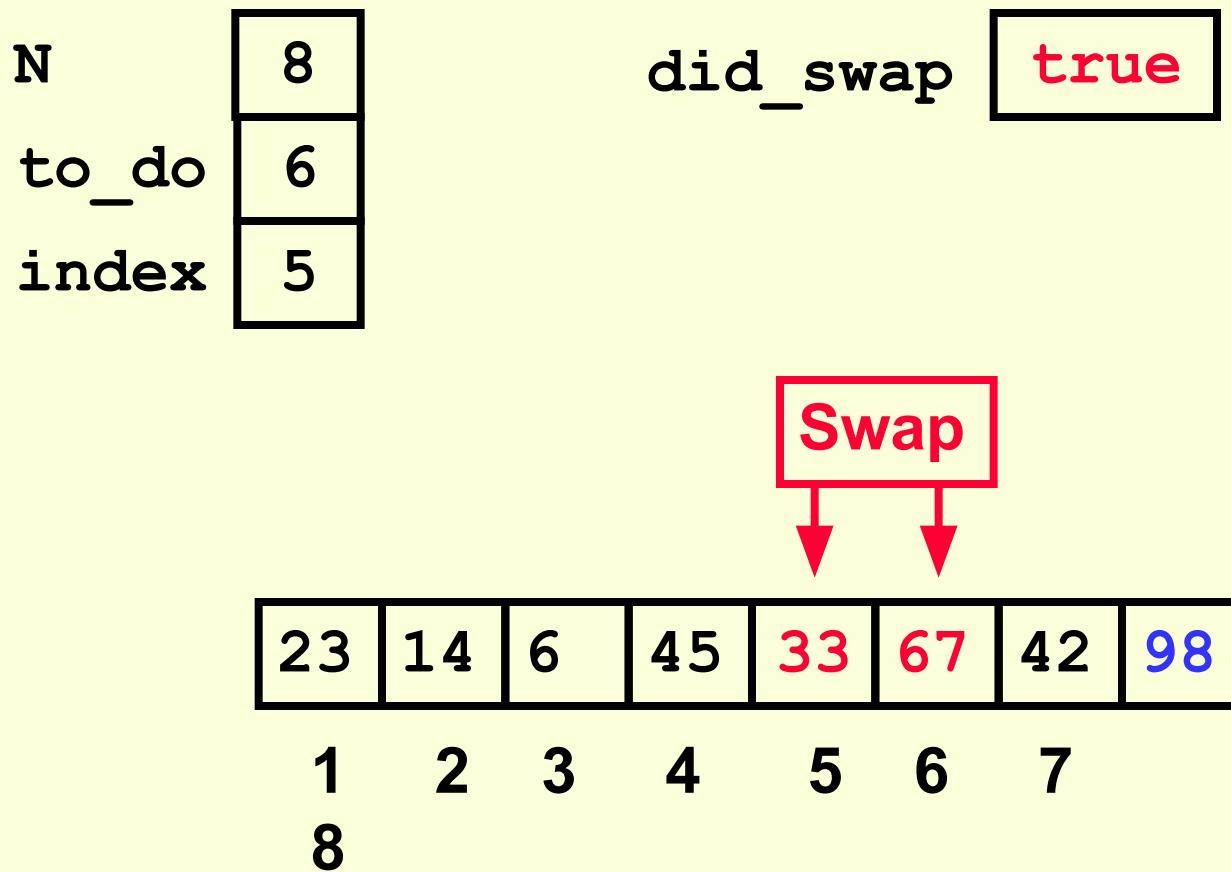
## The Second “Bubble Up”



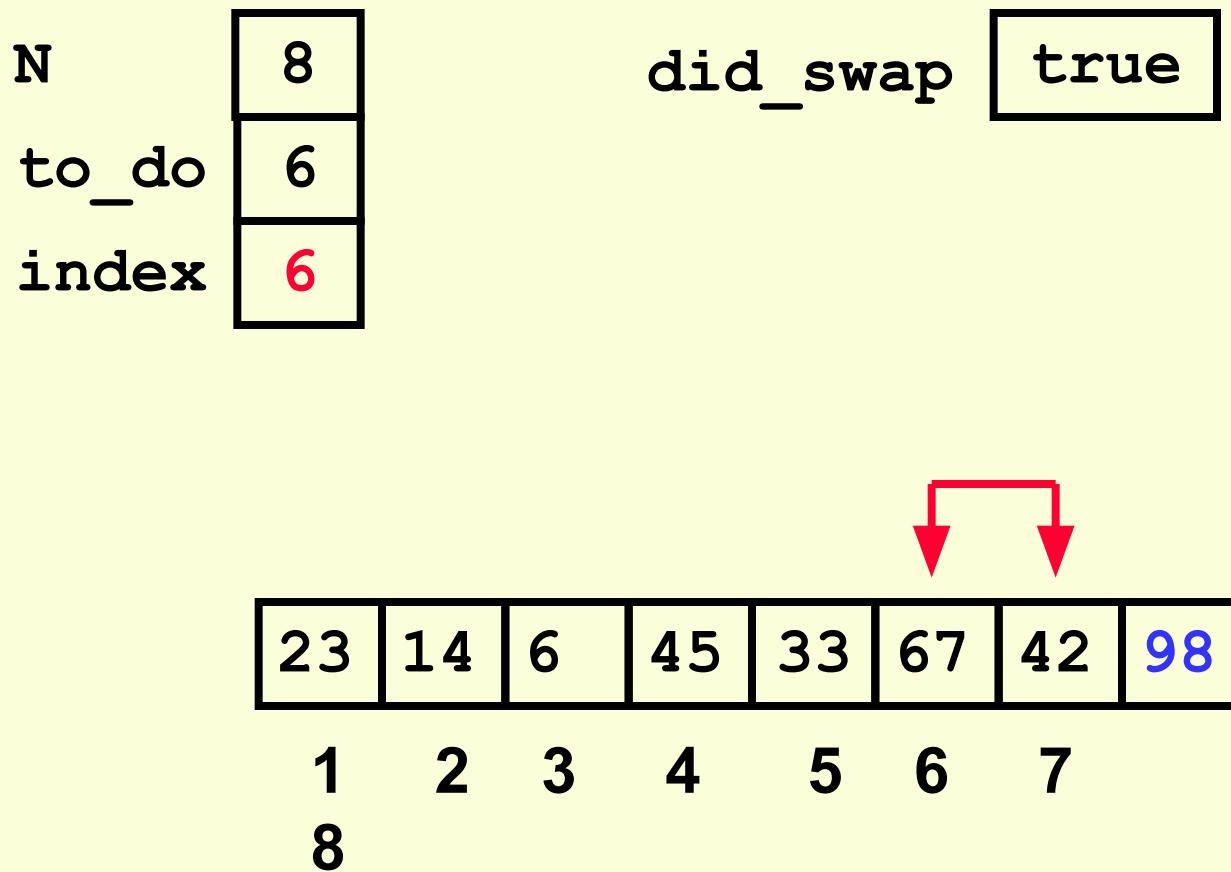
## The Second “Bubble Up”



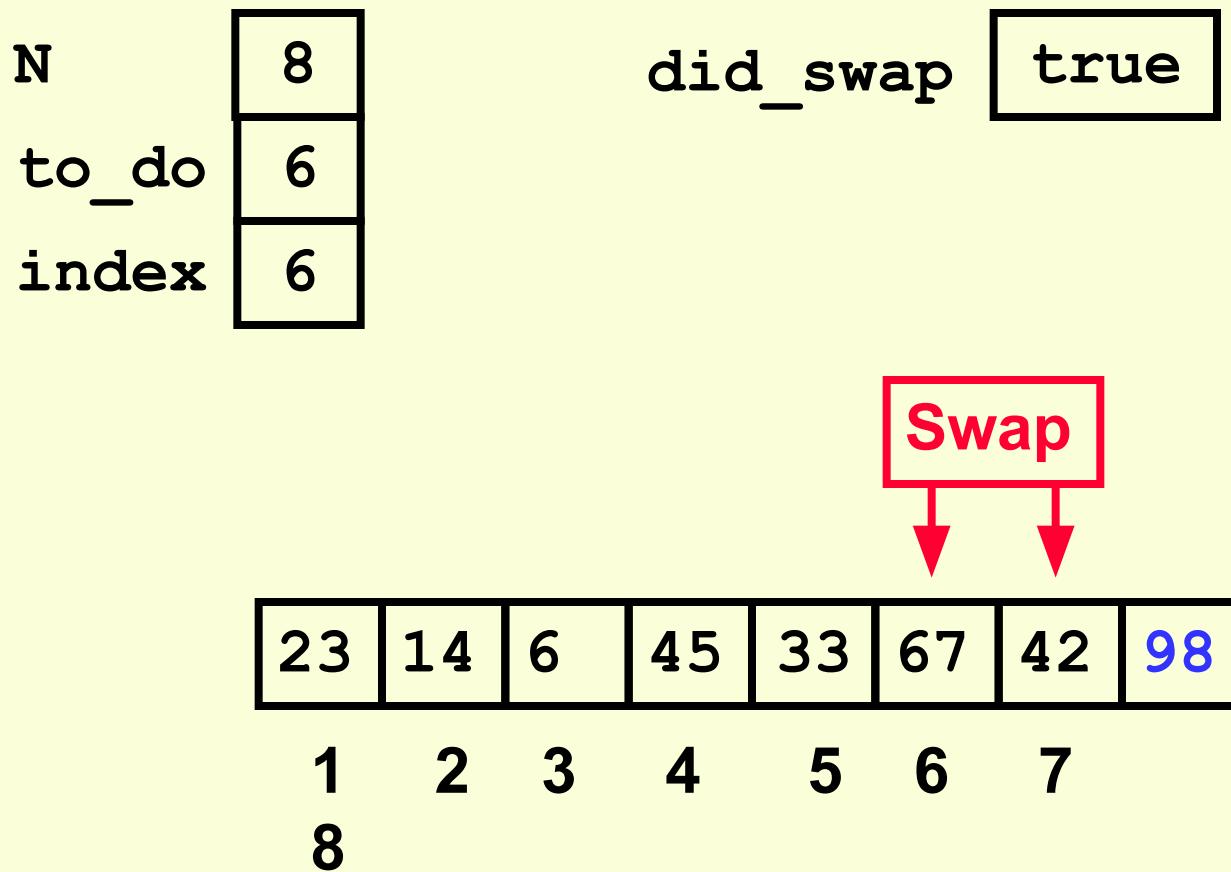
## The Second “Bubble Up”



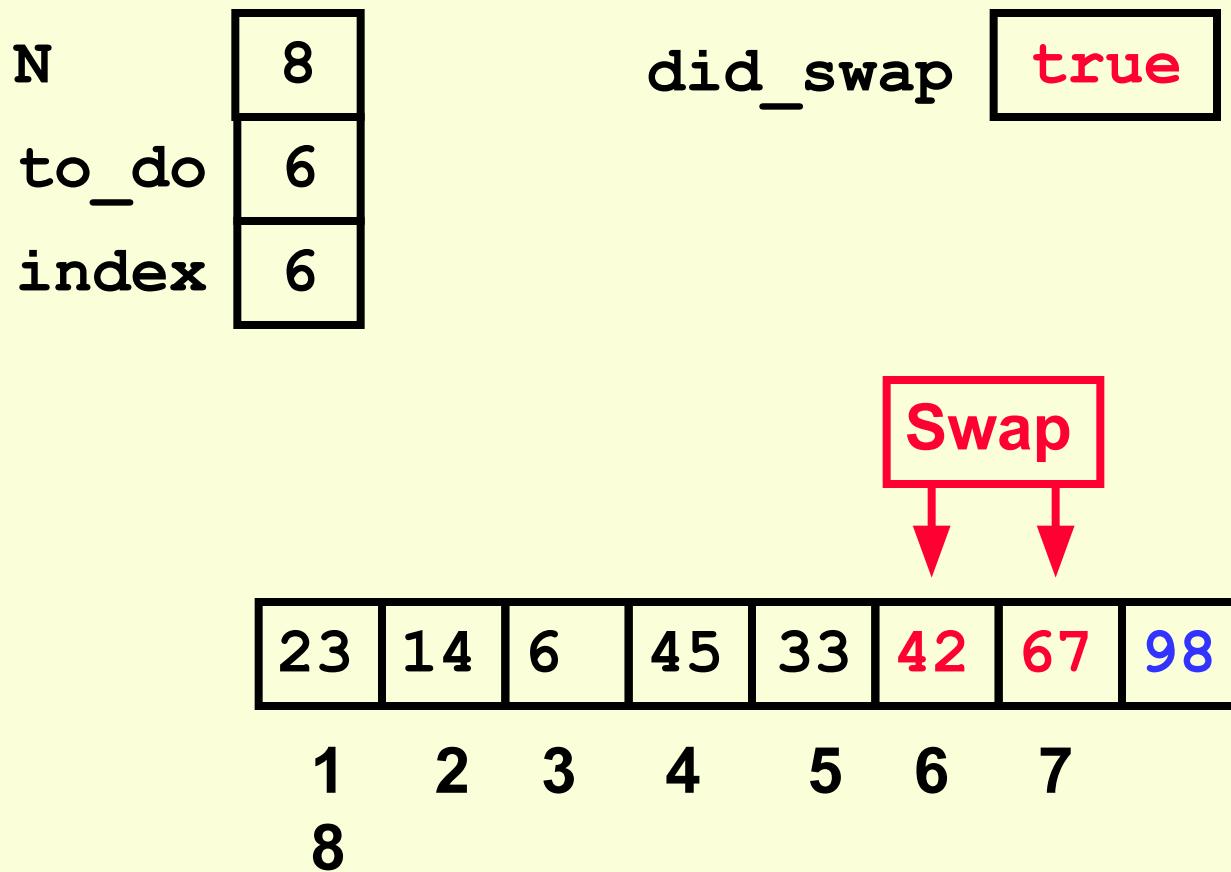
## The Second “Bubble Up”



## The Second “Bubble Up”



## The Second “Bubble Up”

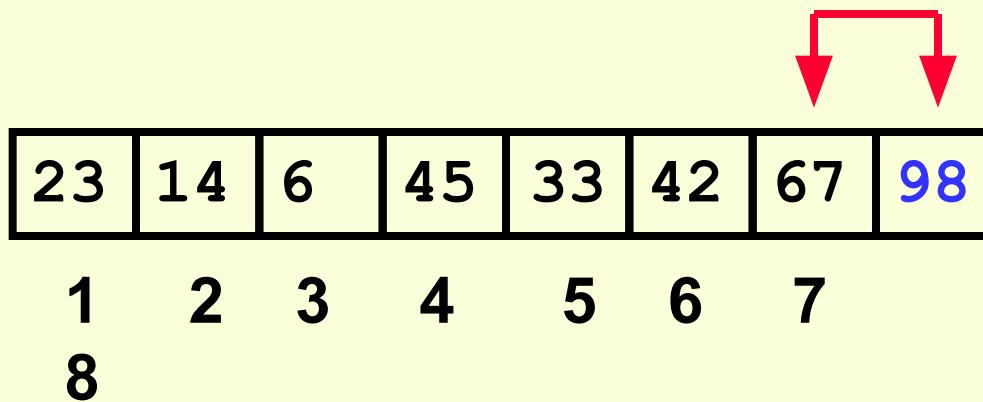


## After Second Pass of Outer Loop

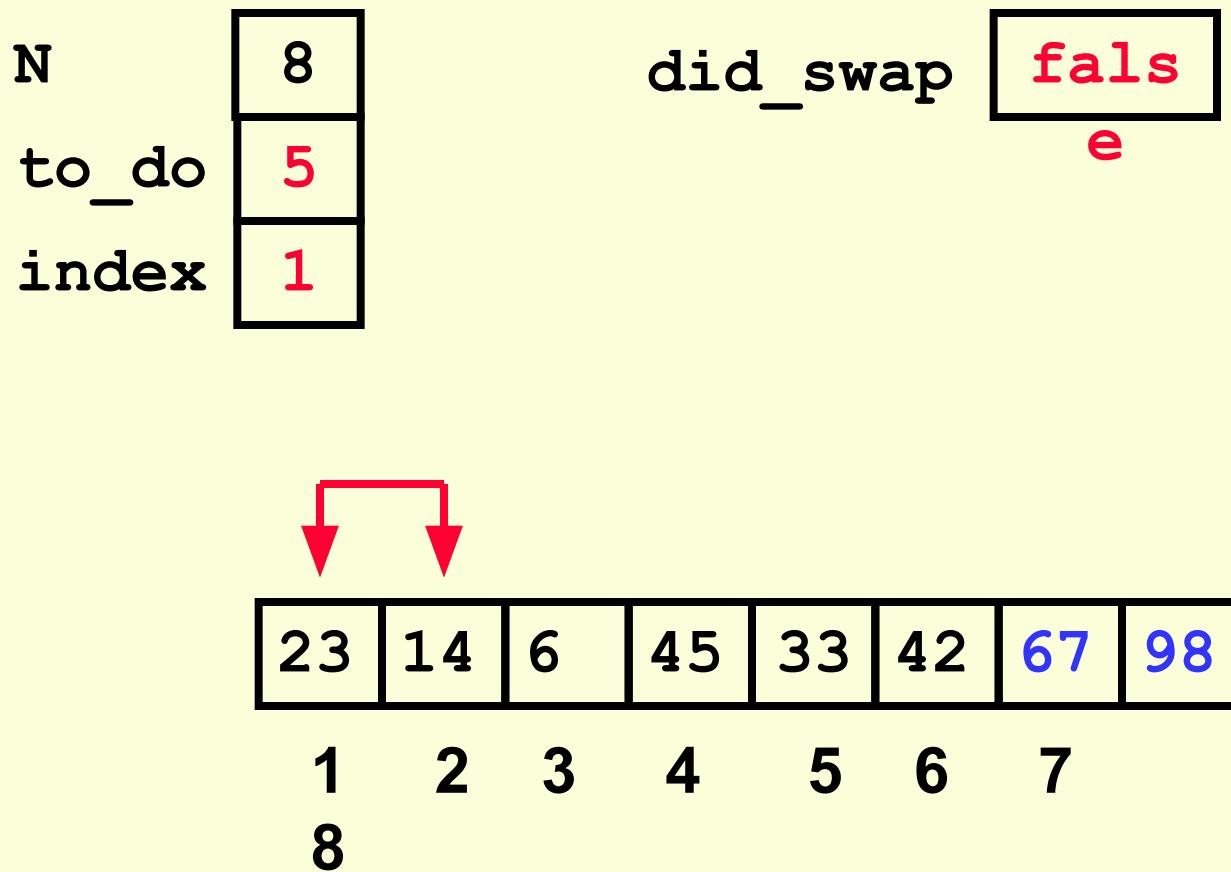
N	8
to_do	6
index	7

did\_swap      true

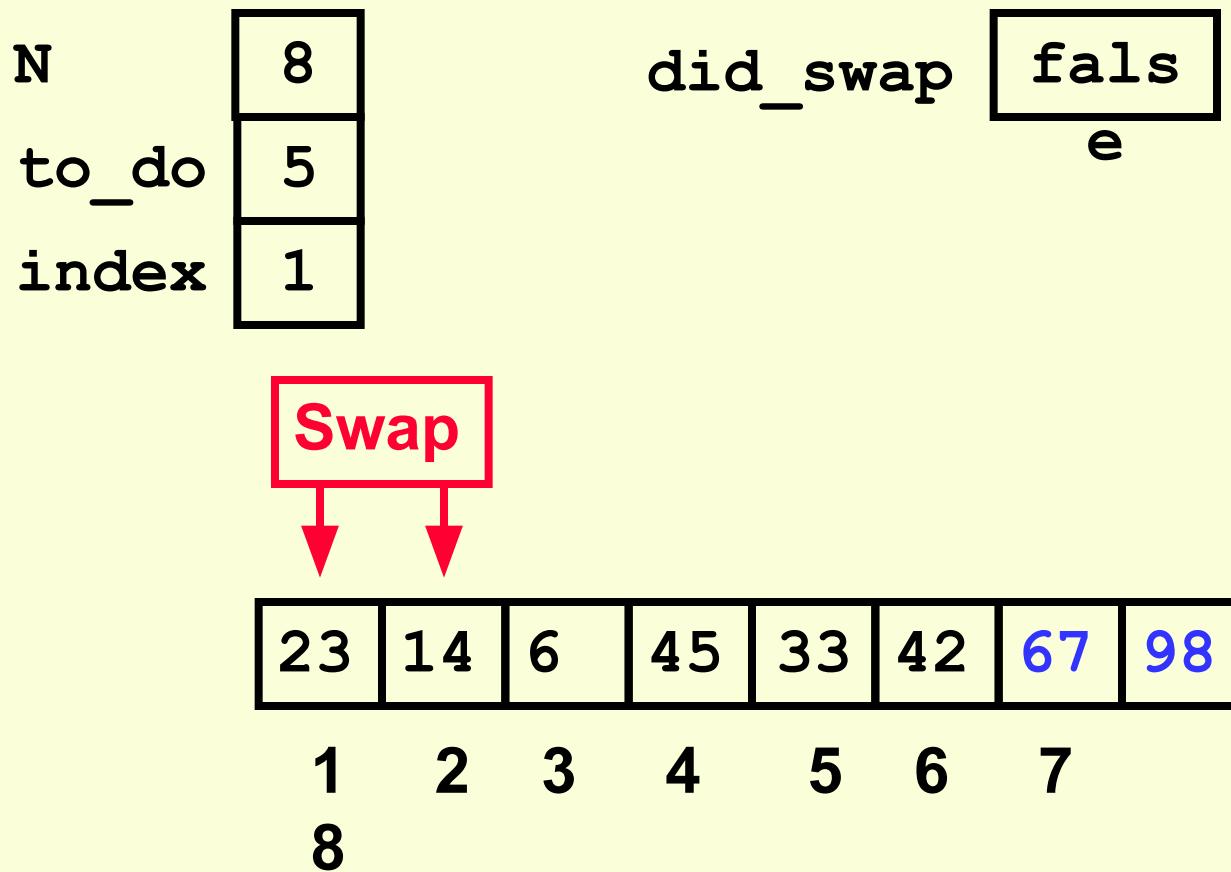
Finished second “Bubble Up”



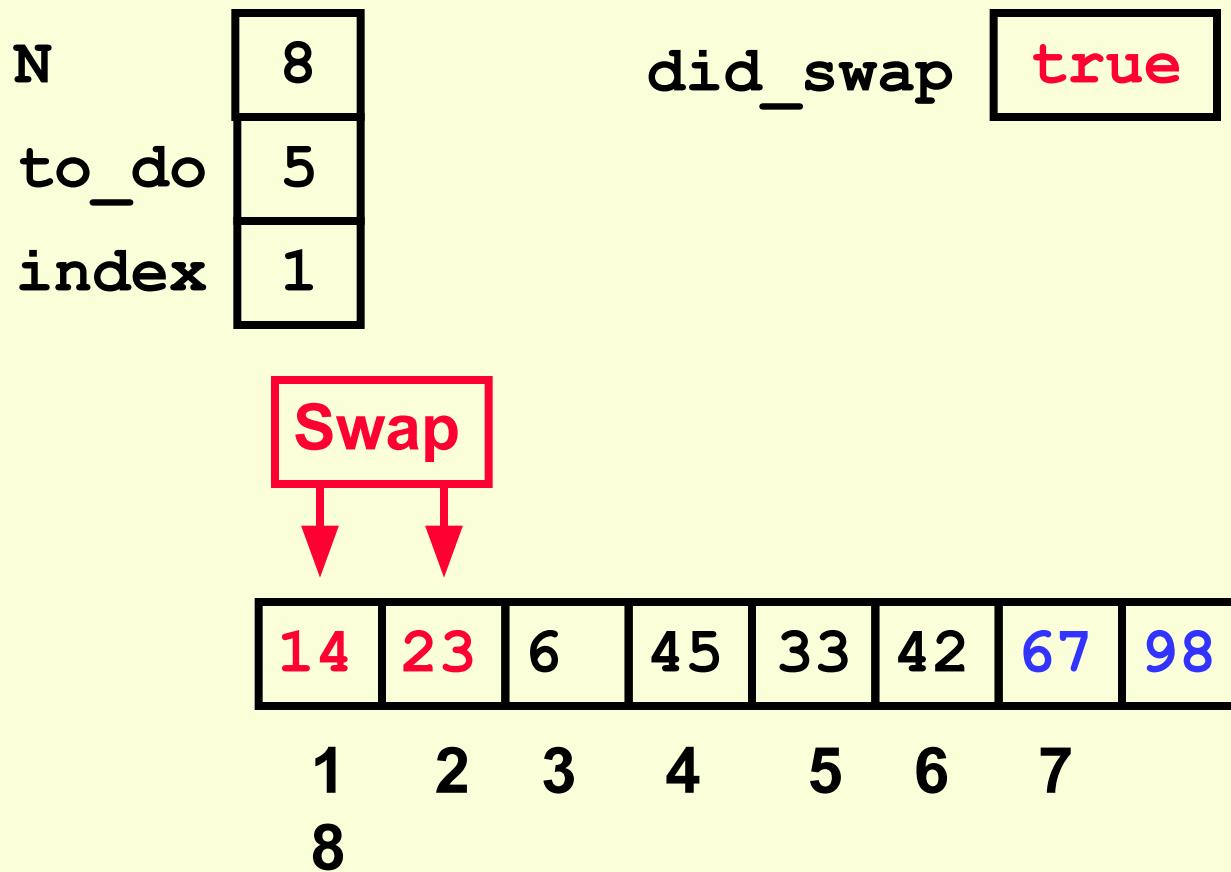
# The Third “Bubble Up”



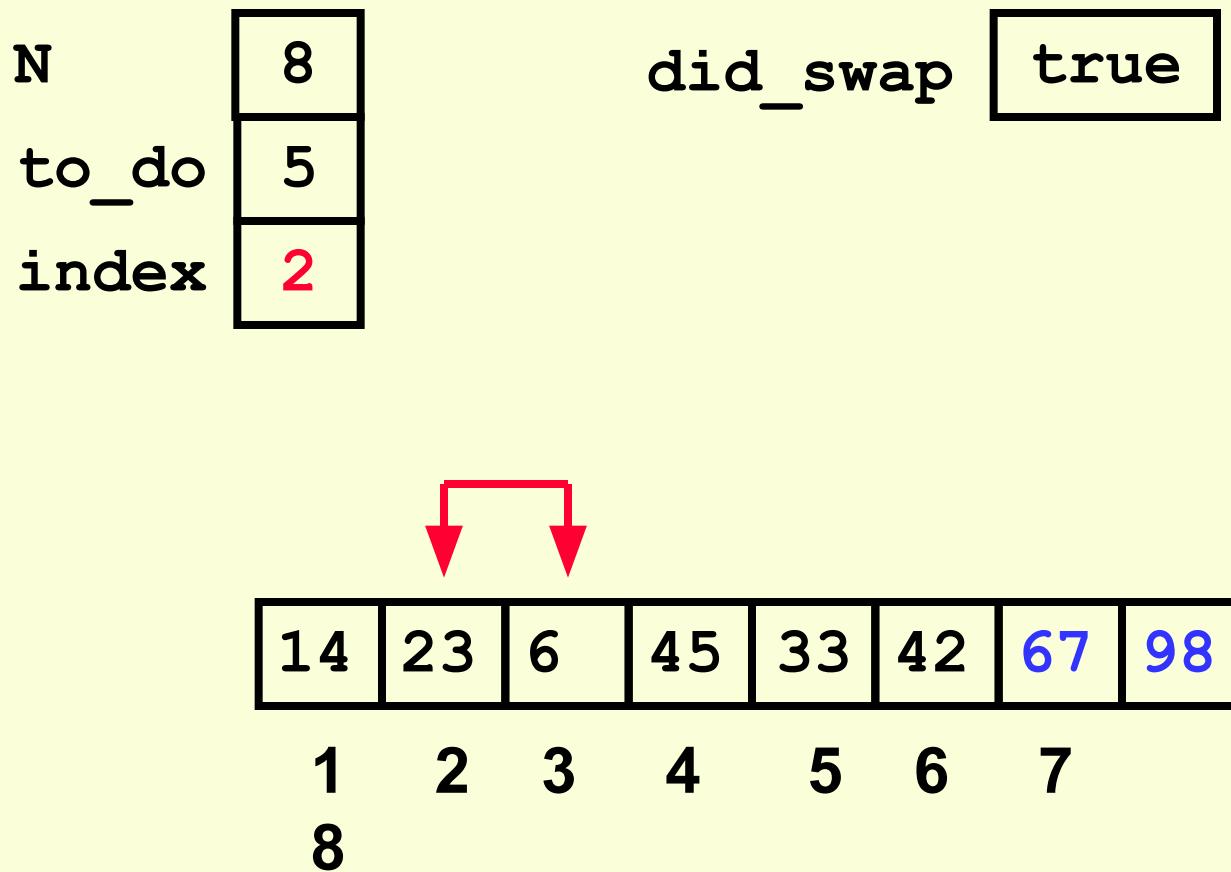
## The Third “Bubble Up”



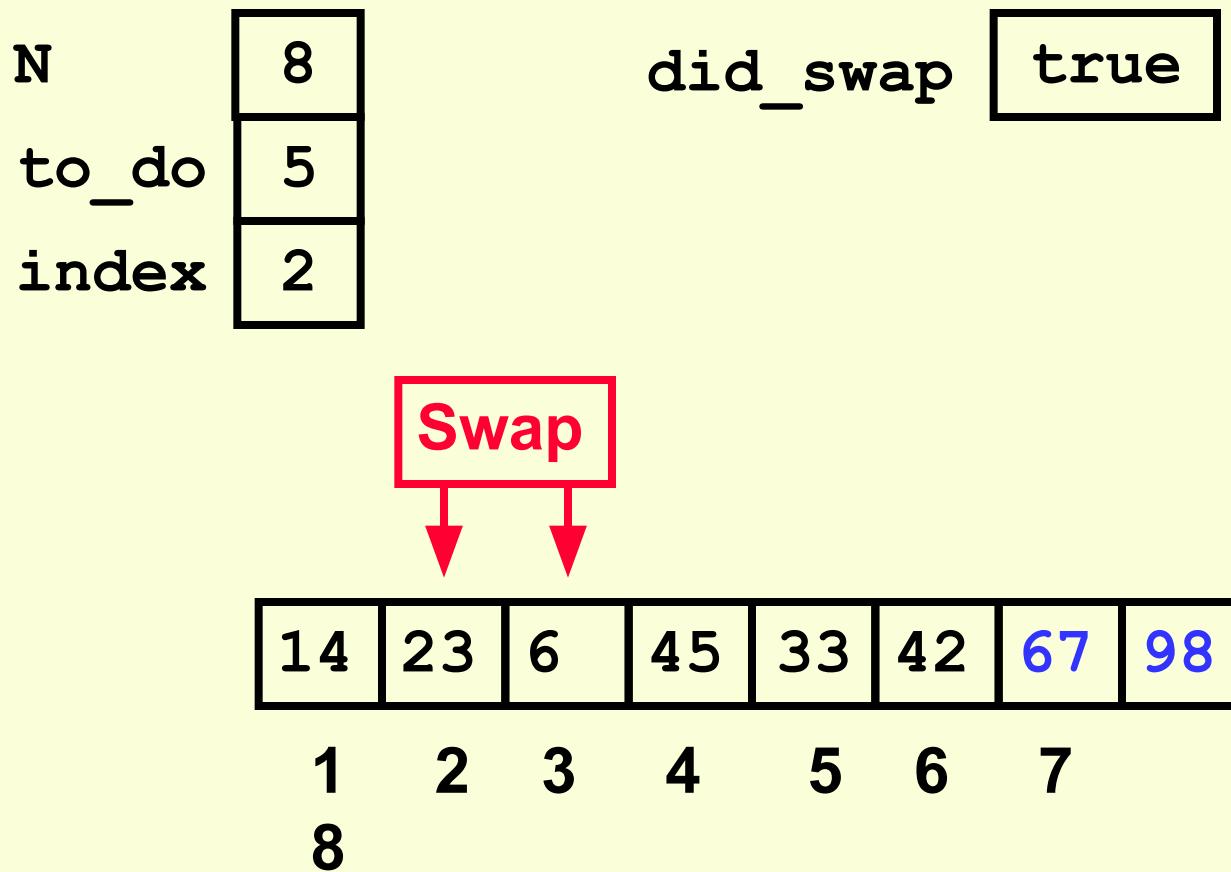
# The Third “Bubble Up”



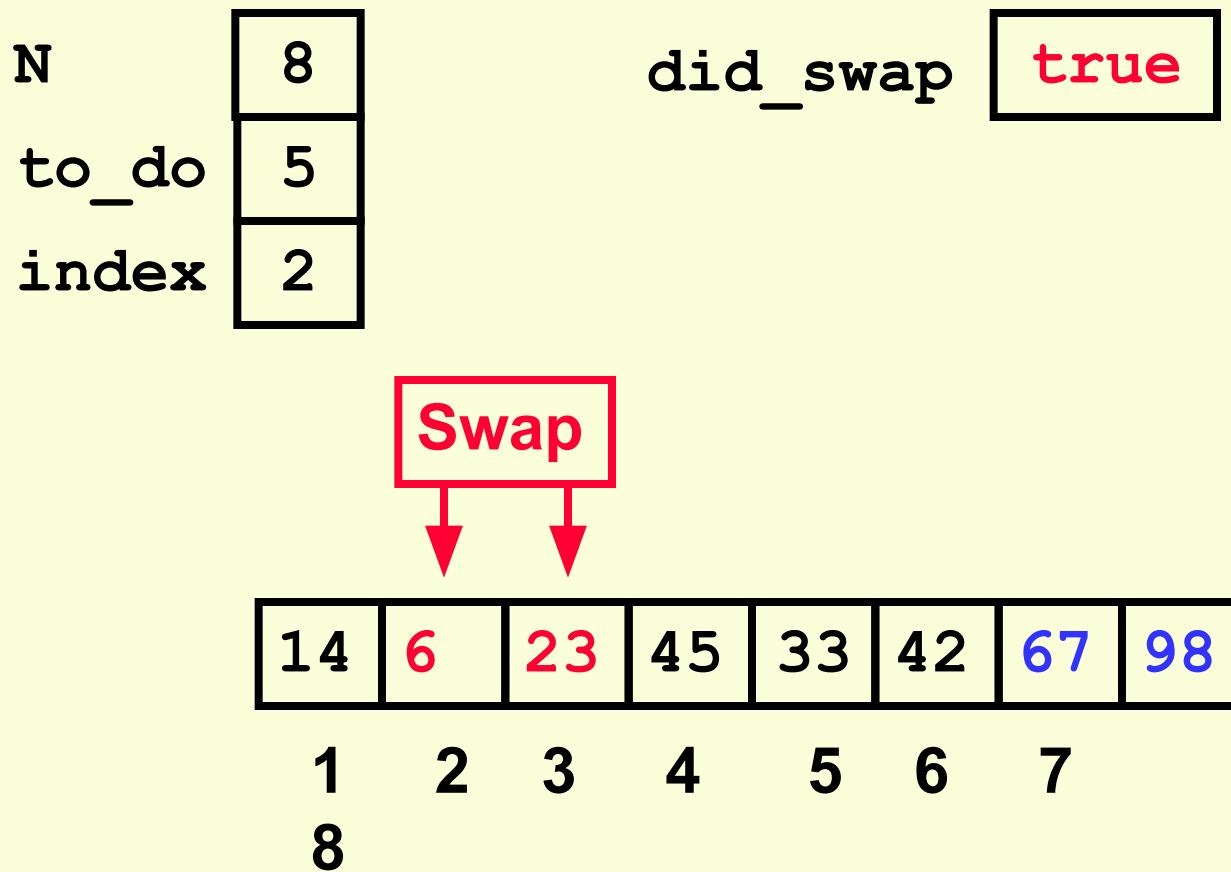
## The Third “Bubble Up”



## The Third “Bubble Up”



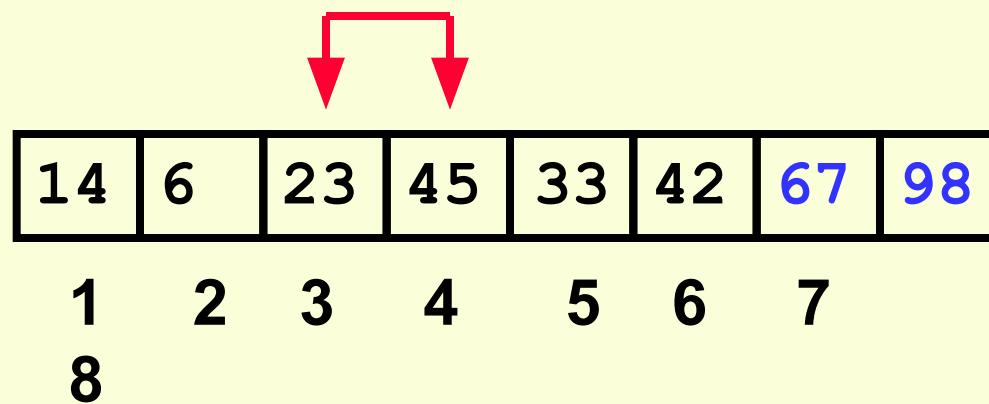
# The Third “Bubble Up”



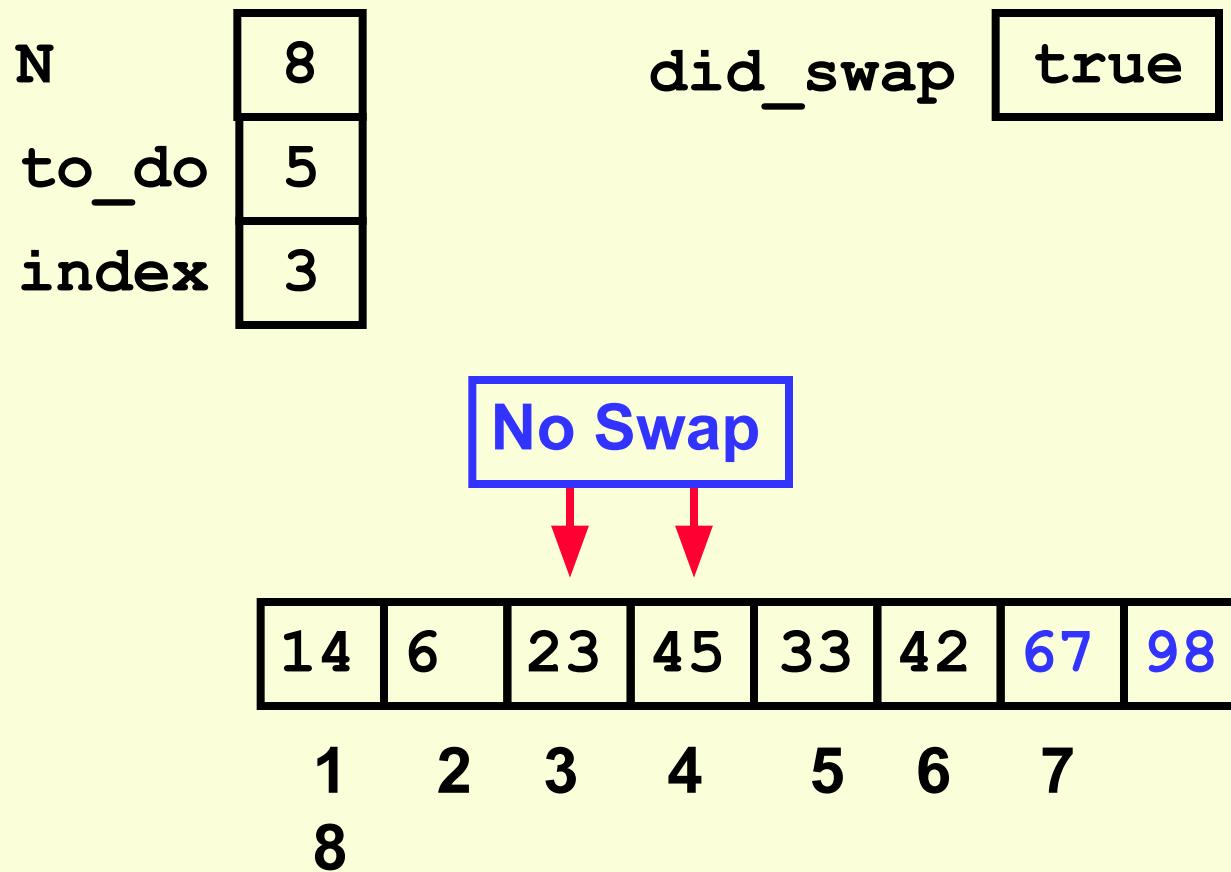
# The Third “Bubble Up”

N	8
to_do	5
index	3

did\_swap      true



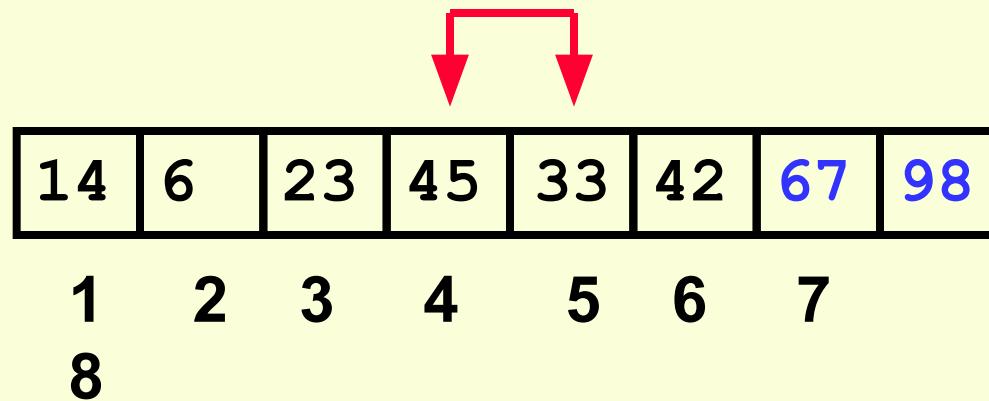
# The Third “Bubble Up”



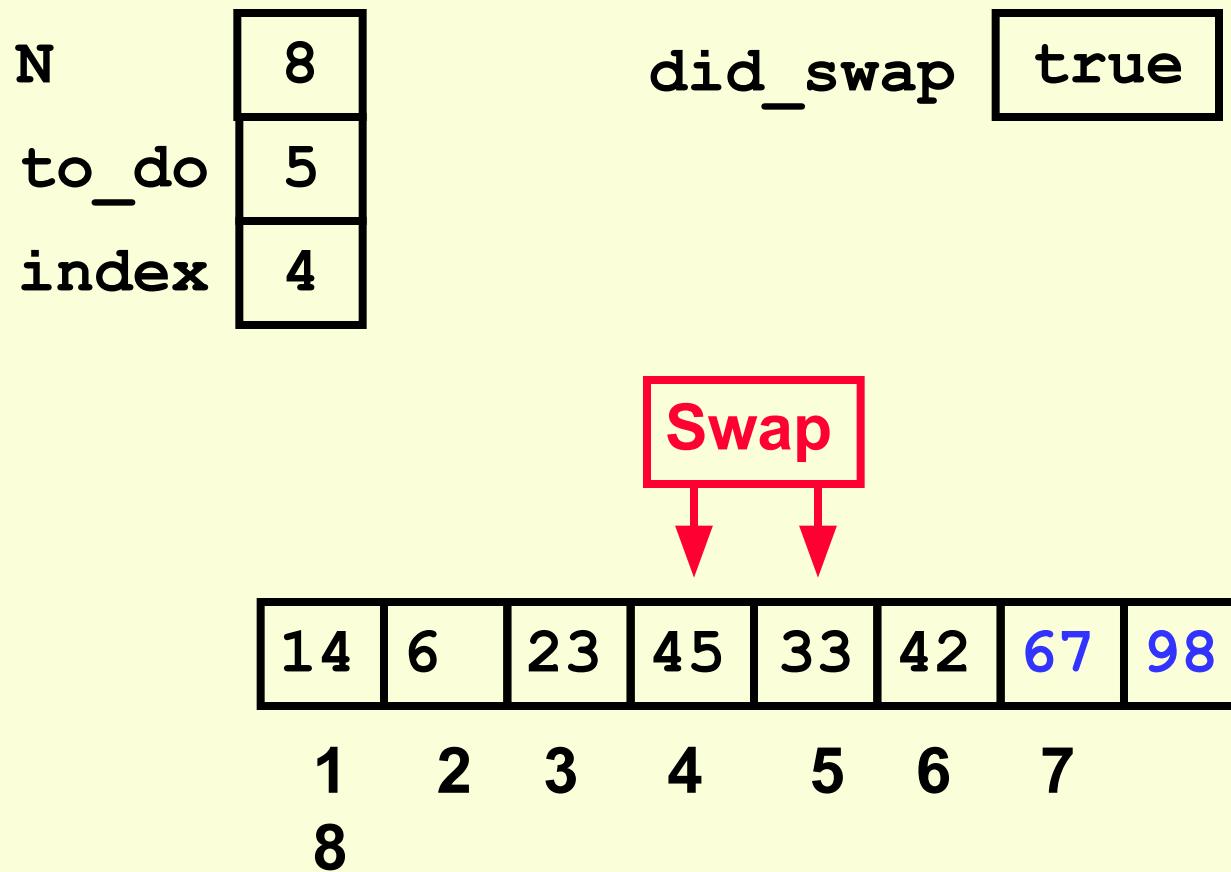
# The Third “Bubble Up”

N	8
to_do	5
index	4

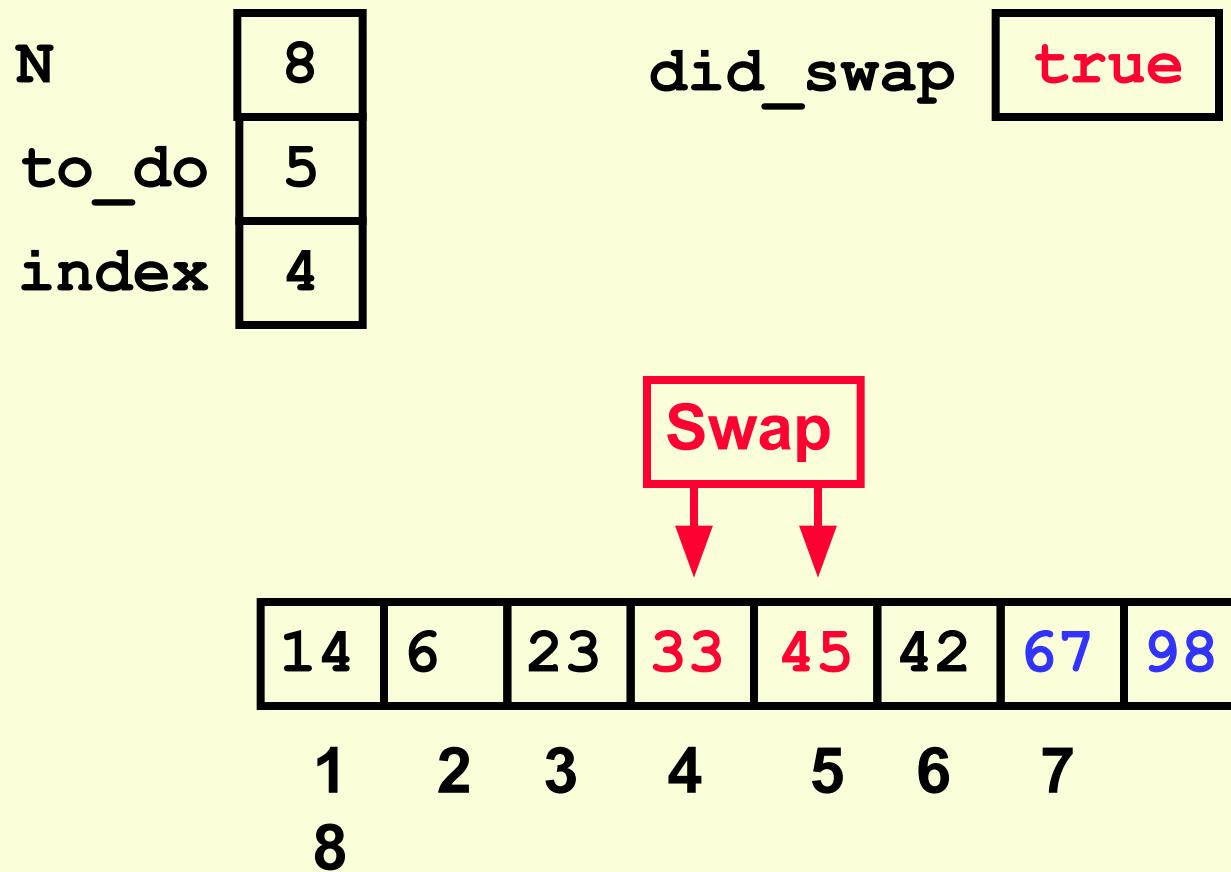
did\_swap      true



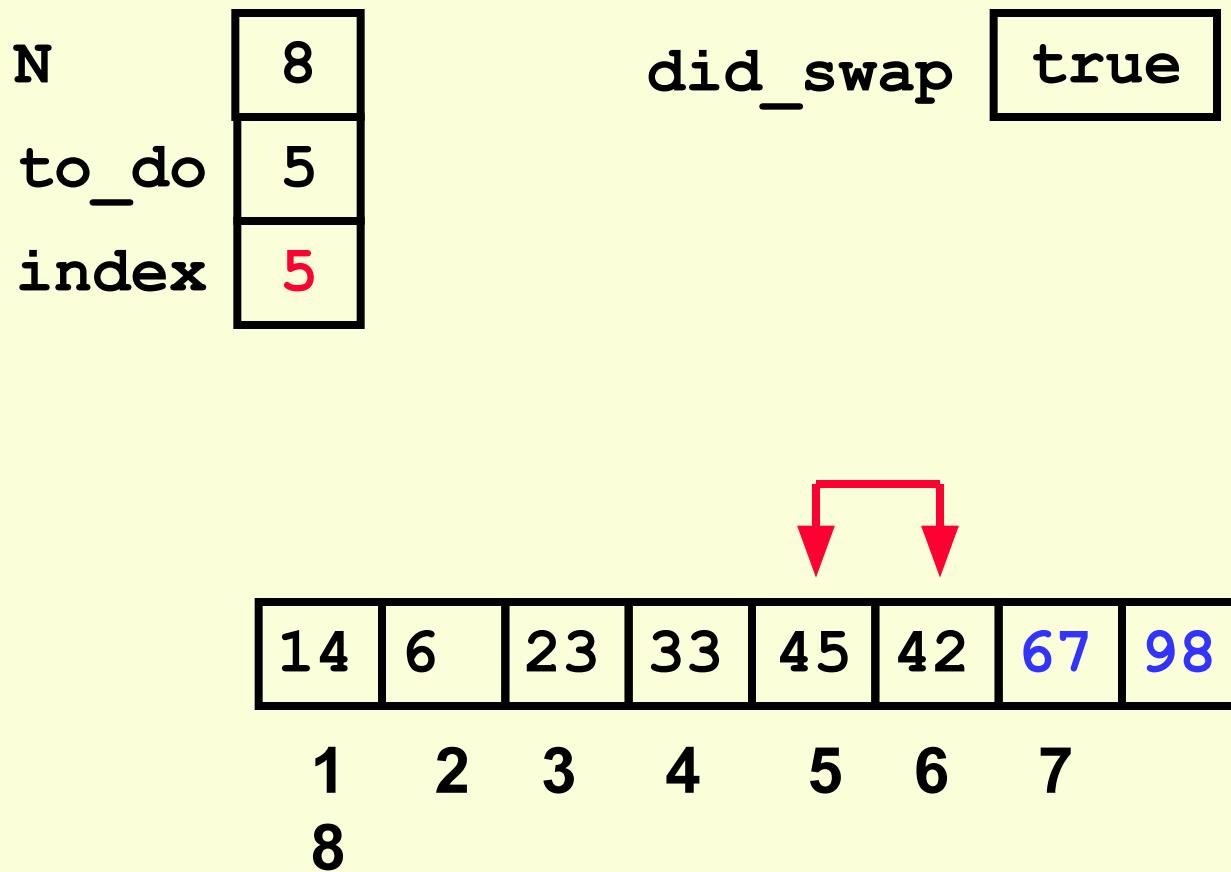
# The Third “Bubble Up”



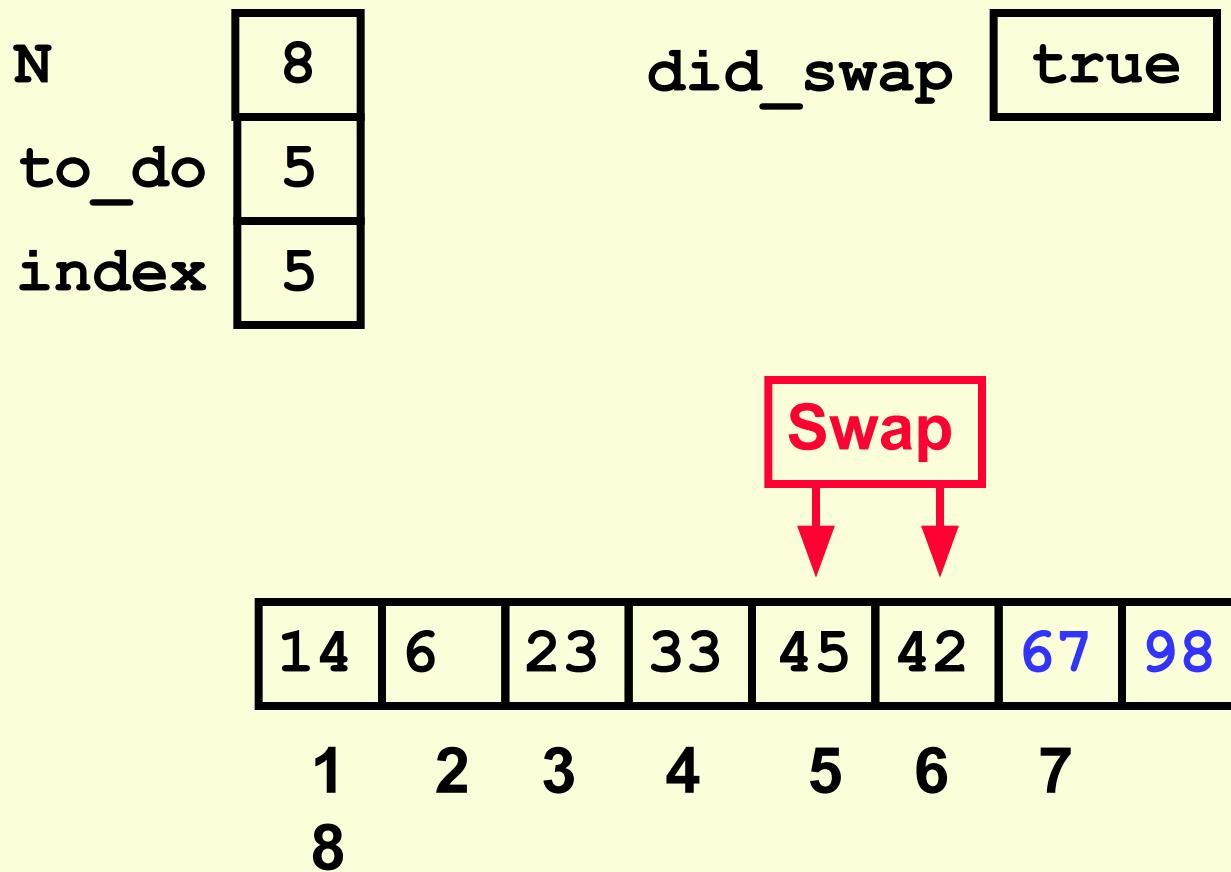
# The Third “Bubble Up”



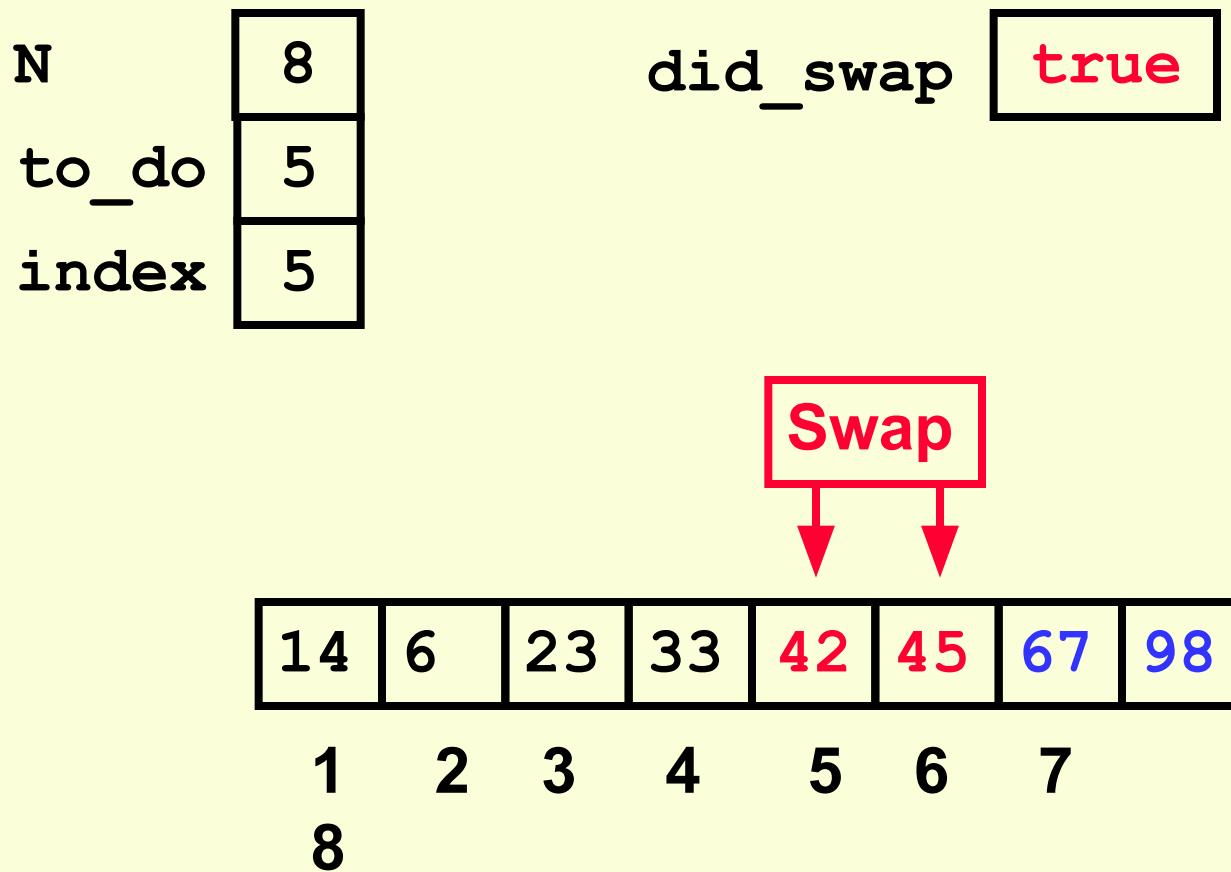
# The Third “Bubble Up”



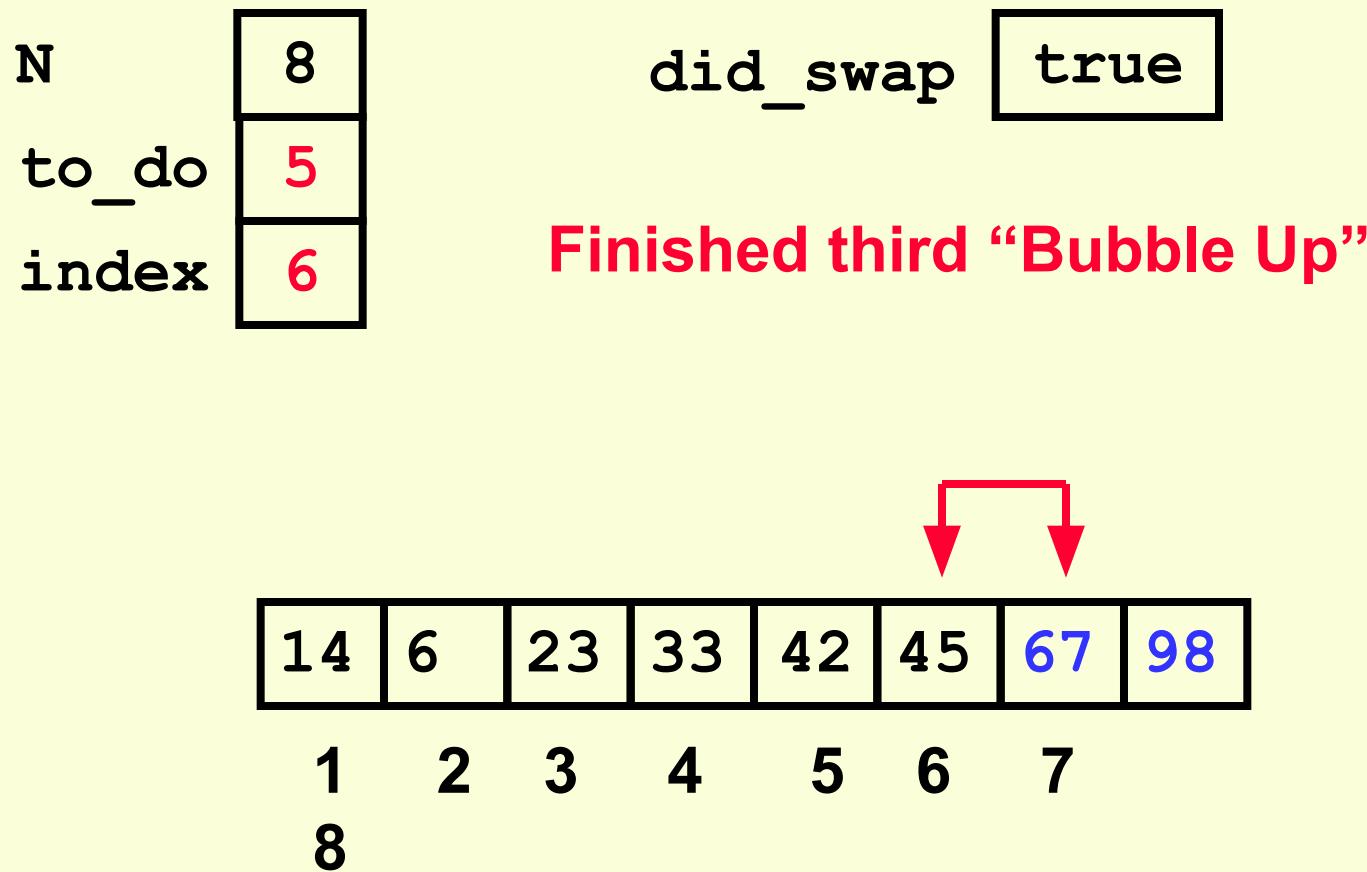
# The Third “Bubble Up”



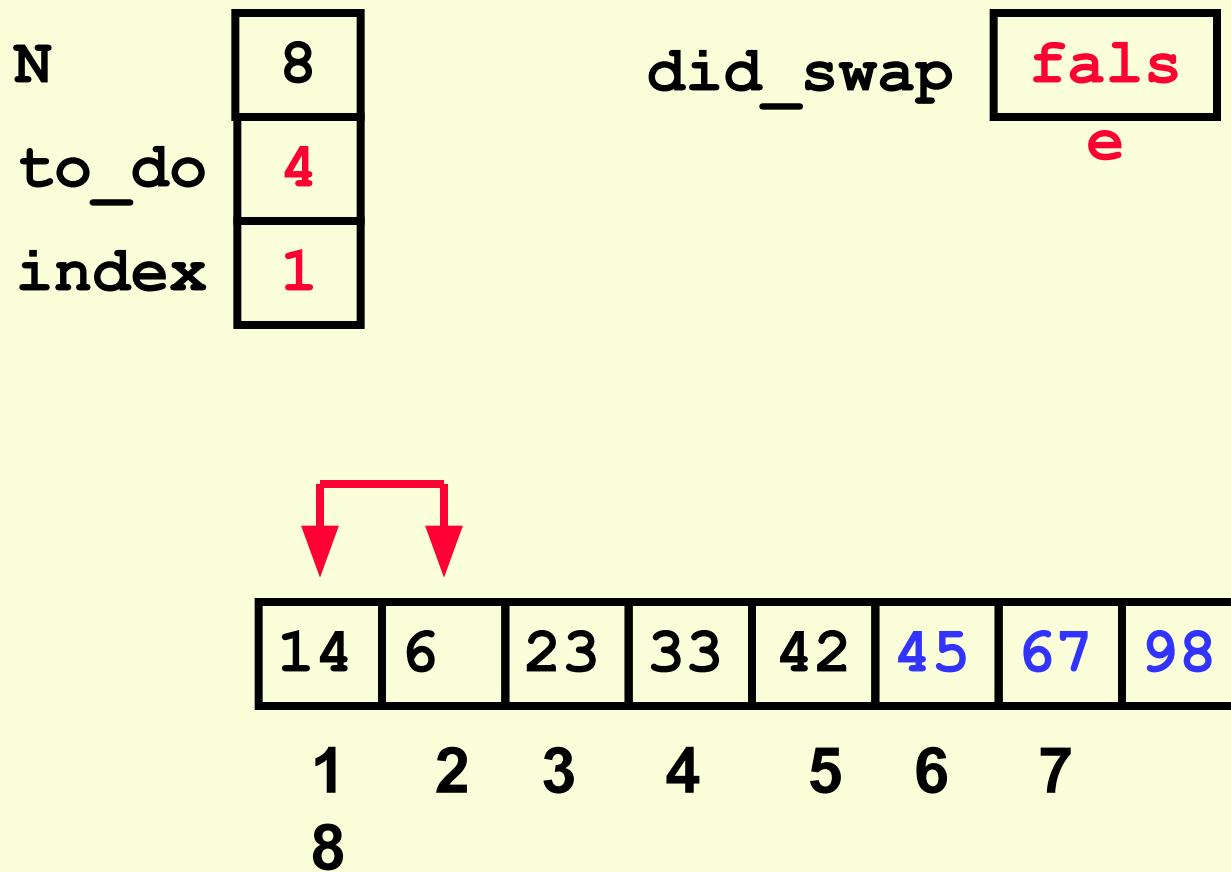
# The Third “Bubble Up”



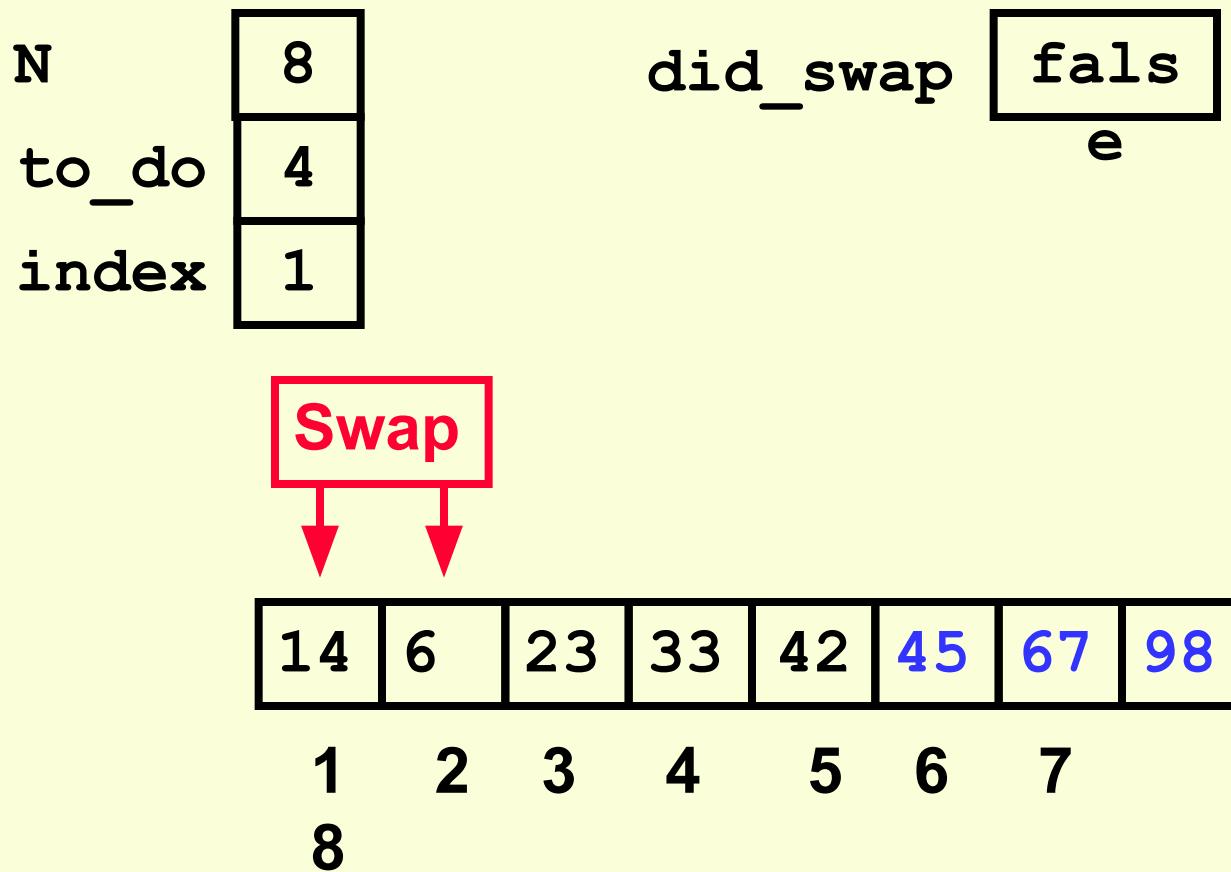
## After Third Pass of Outer Loop



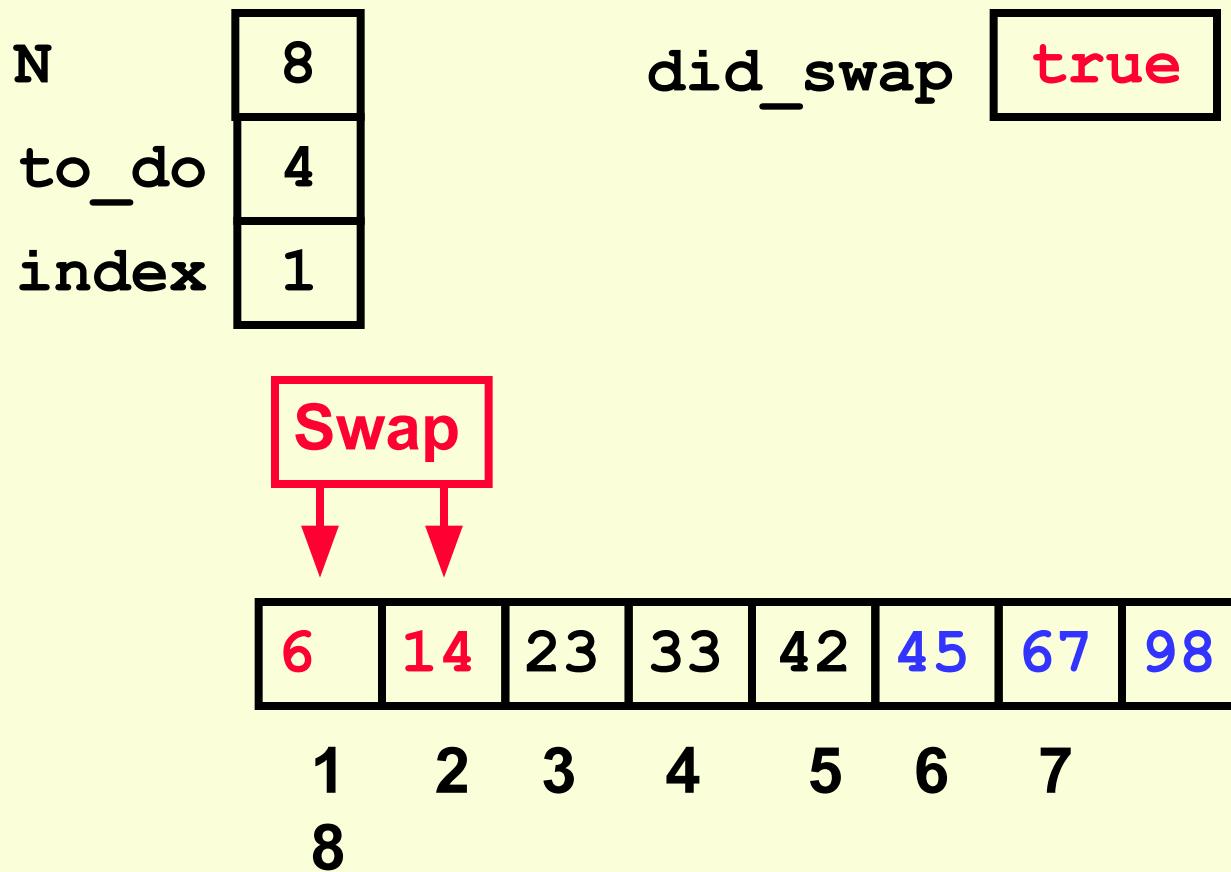
# The Fourth “Bubble Up”



# The Fourth “Bubble Up”



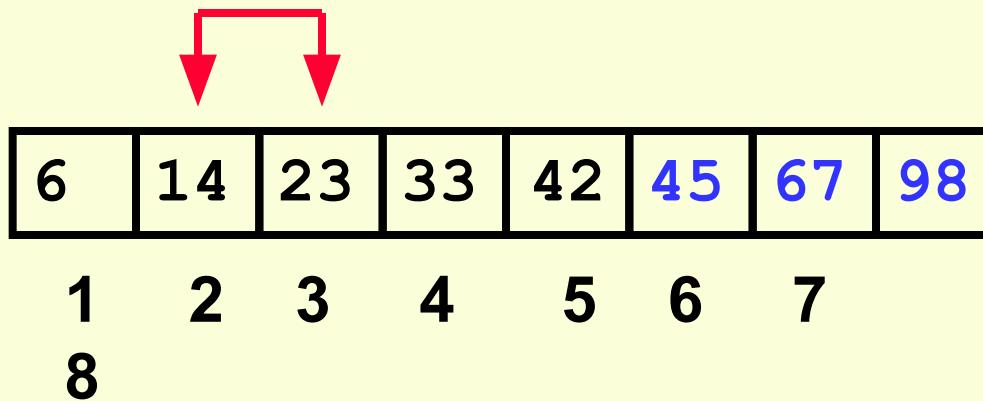
# The Fourth “Bubble Up”



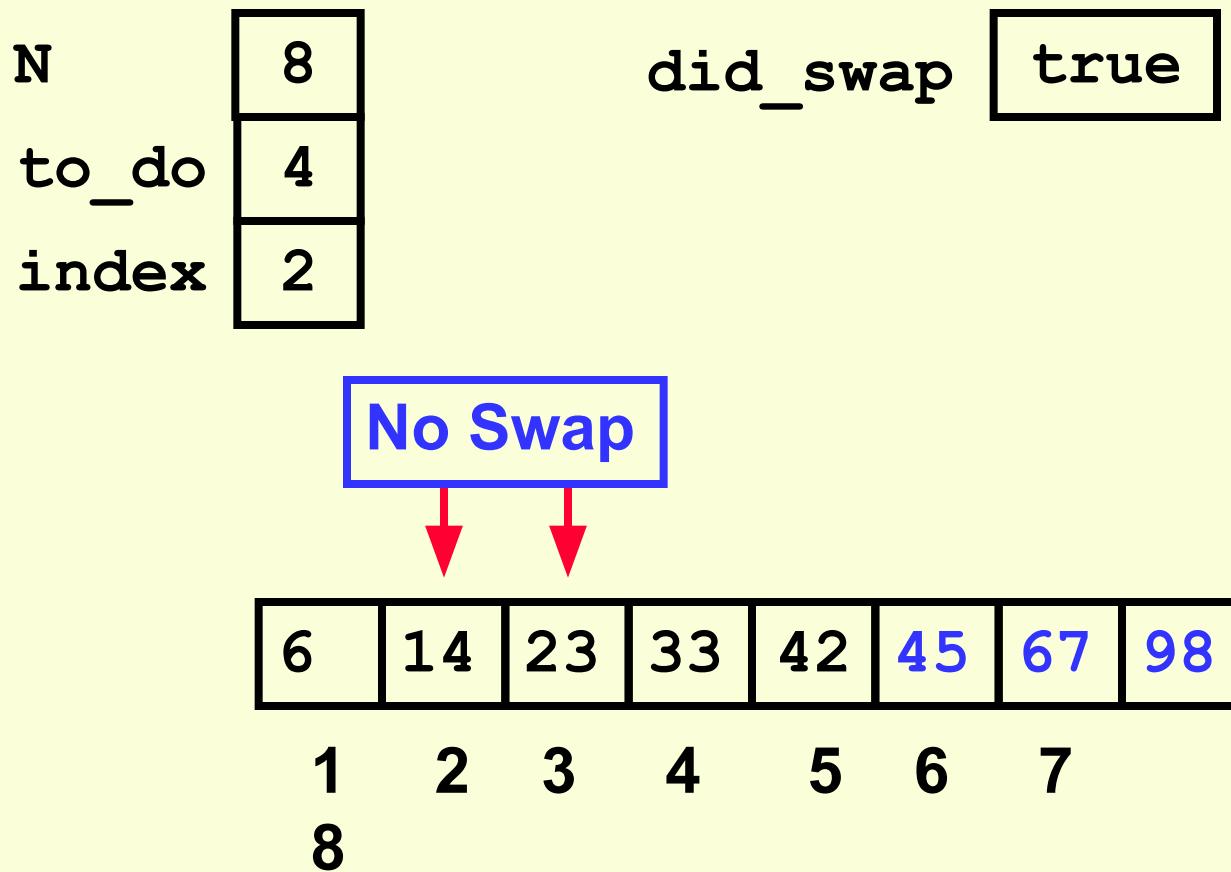
# The Fourth “Bubble Up”

N	8
to_do	4
index	2

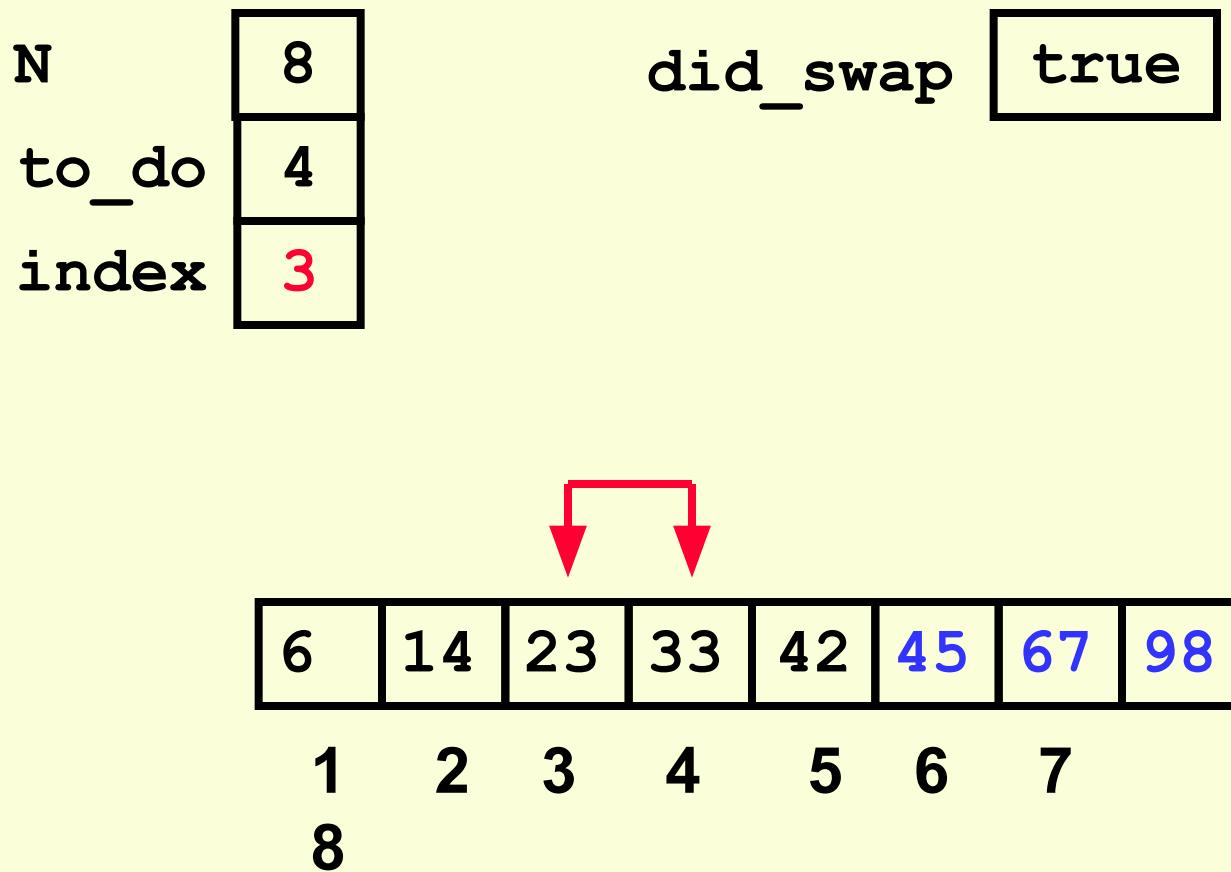
did\_swap      true



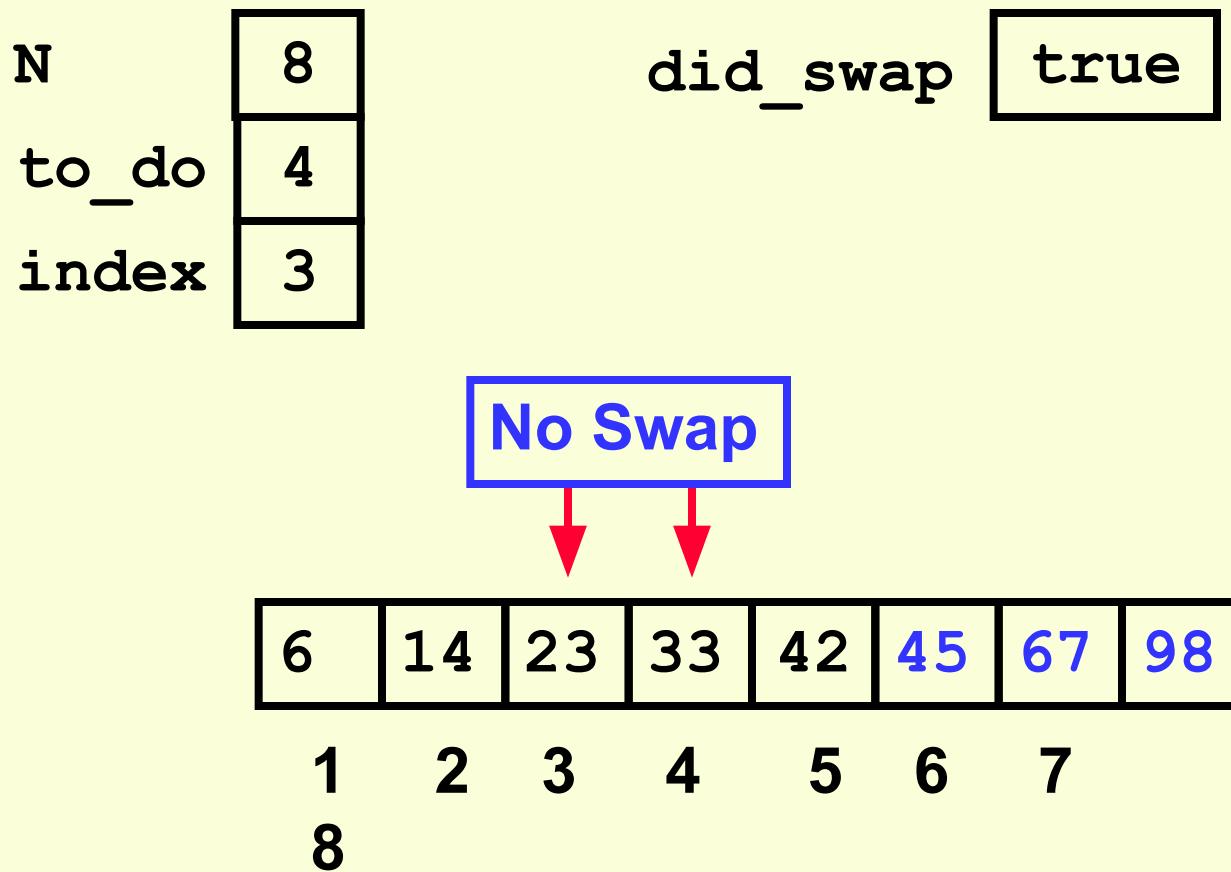
# The Fourth “Bubble Up”



# The Fourth “Bubble Up”



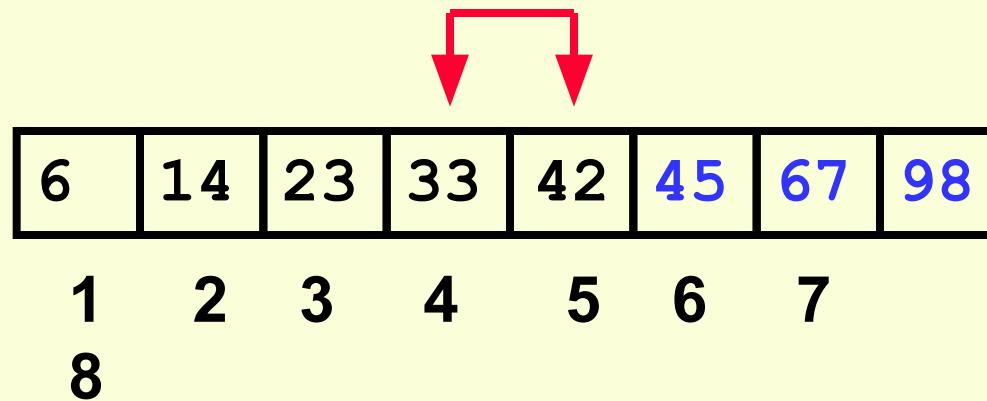
# The Fourth “Bubble Up”



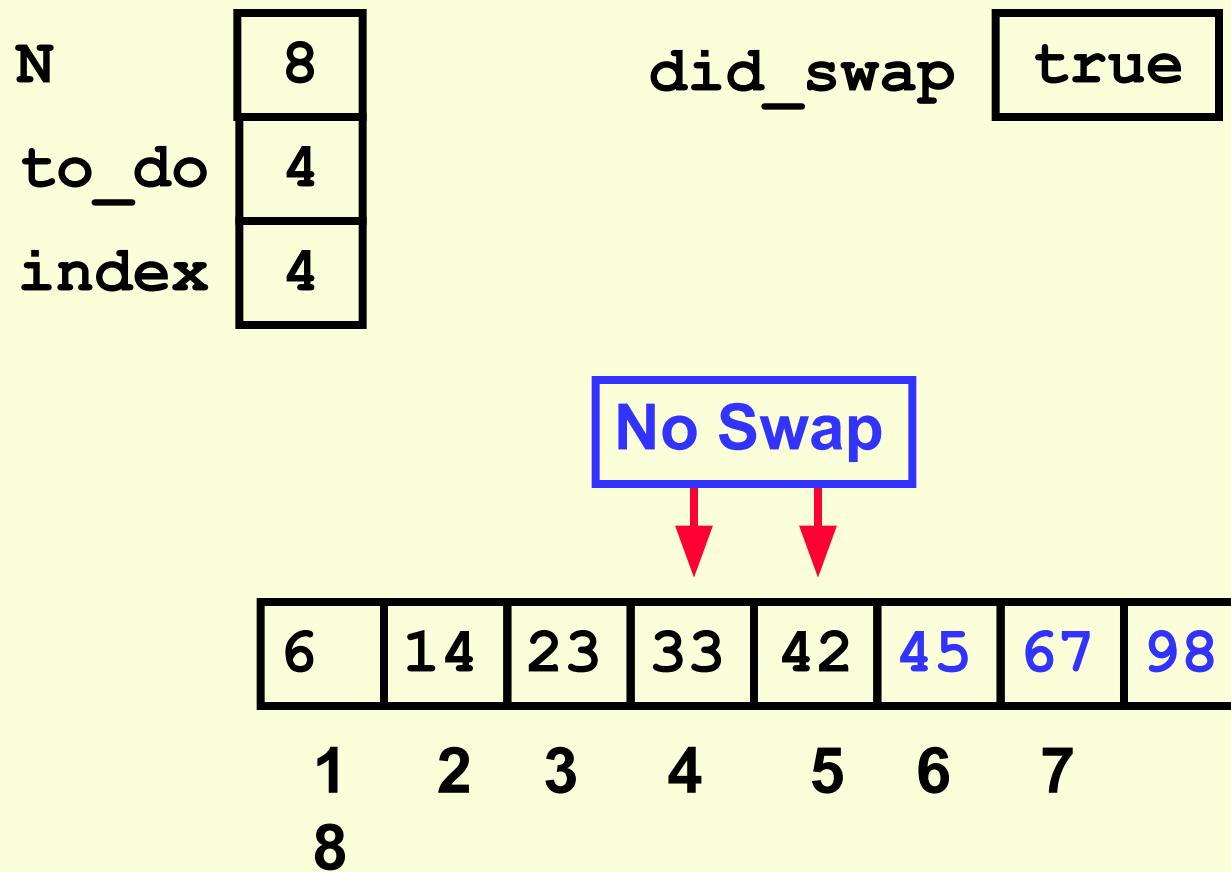
# The Fourth “Bubble Up”

N	8
to_do	4
index	4

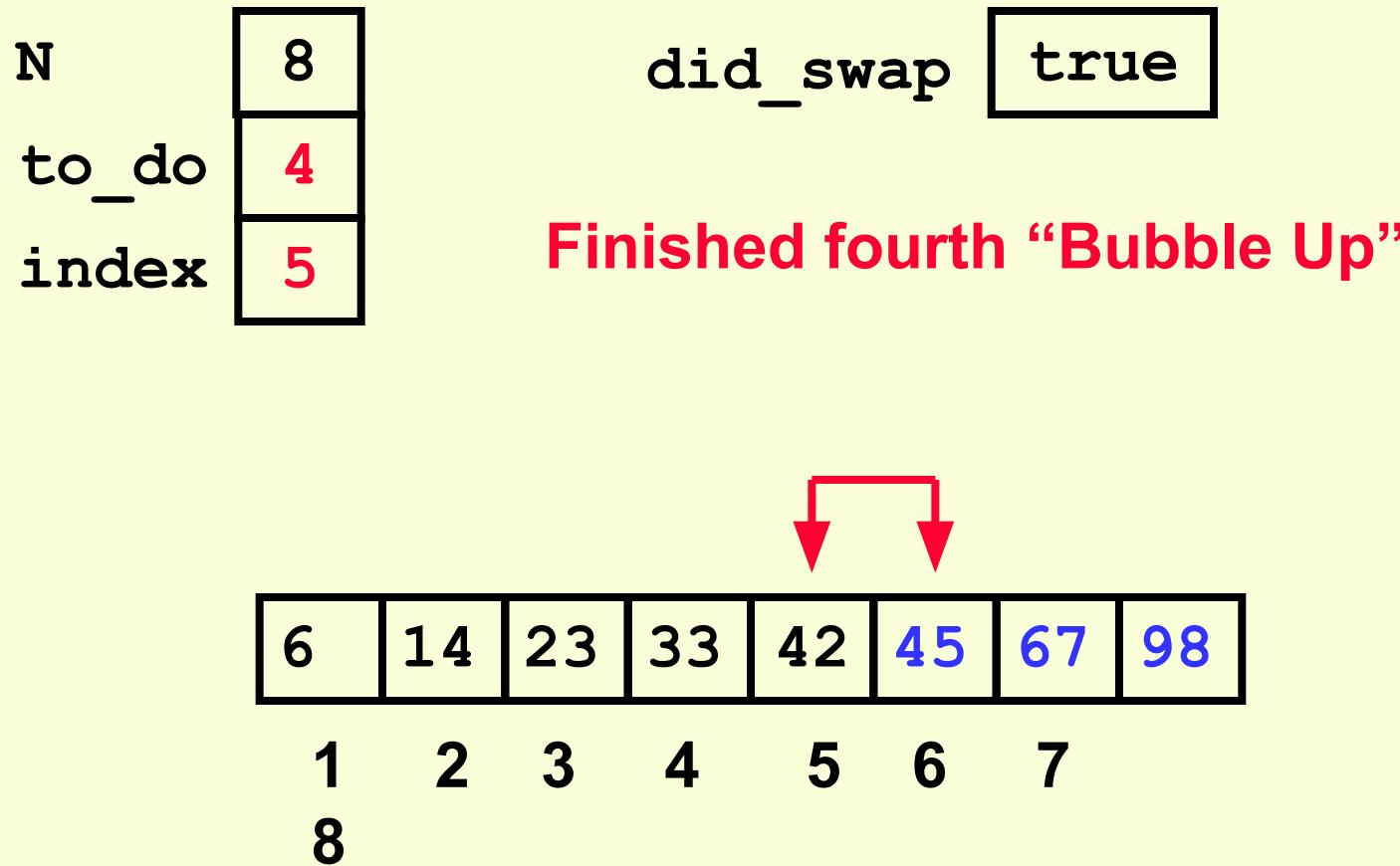
did\_swap      true



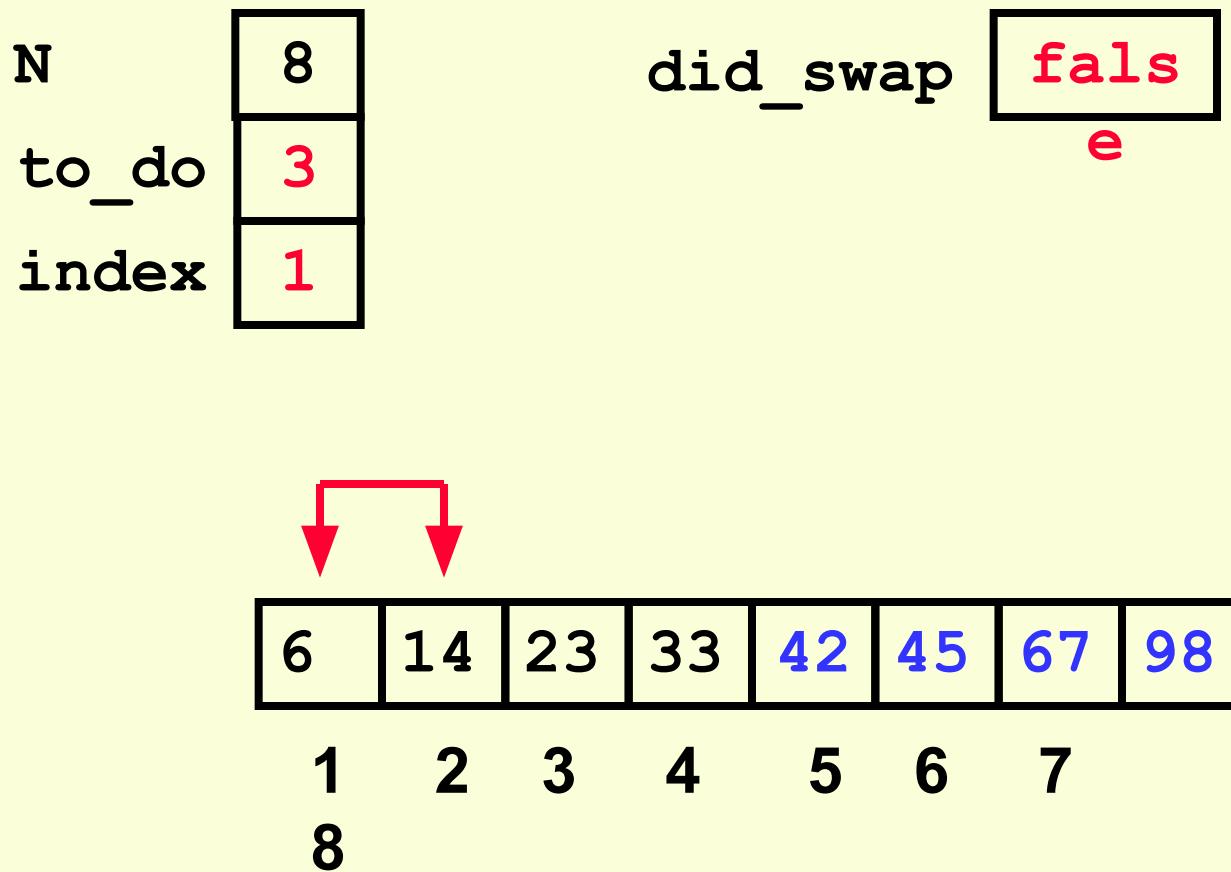
# The Fourth “Bubble Up”



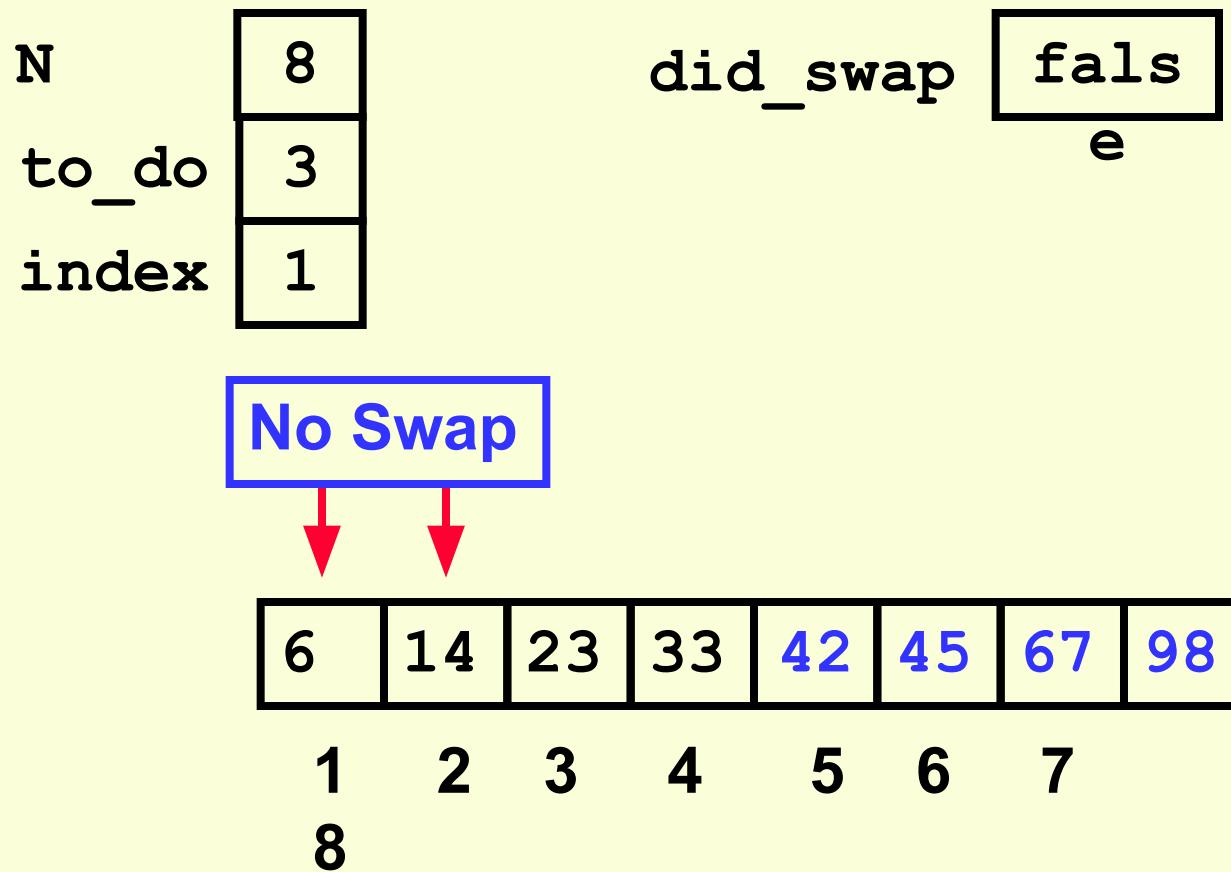
## After Fourth Pass of Outer Loop



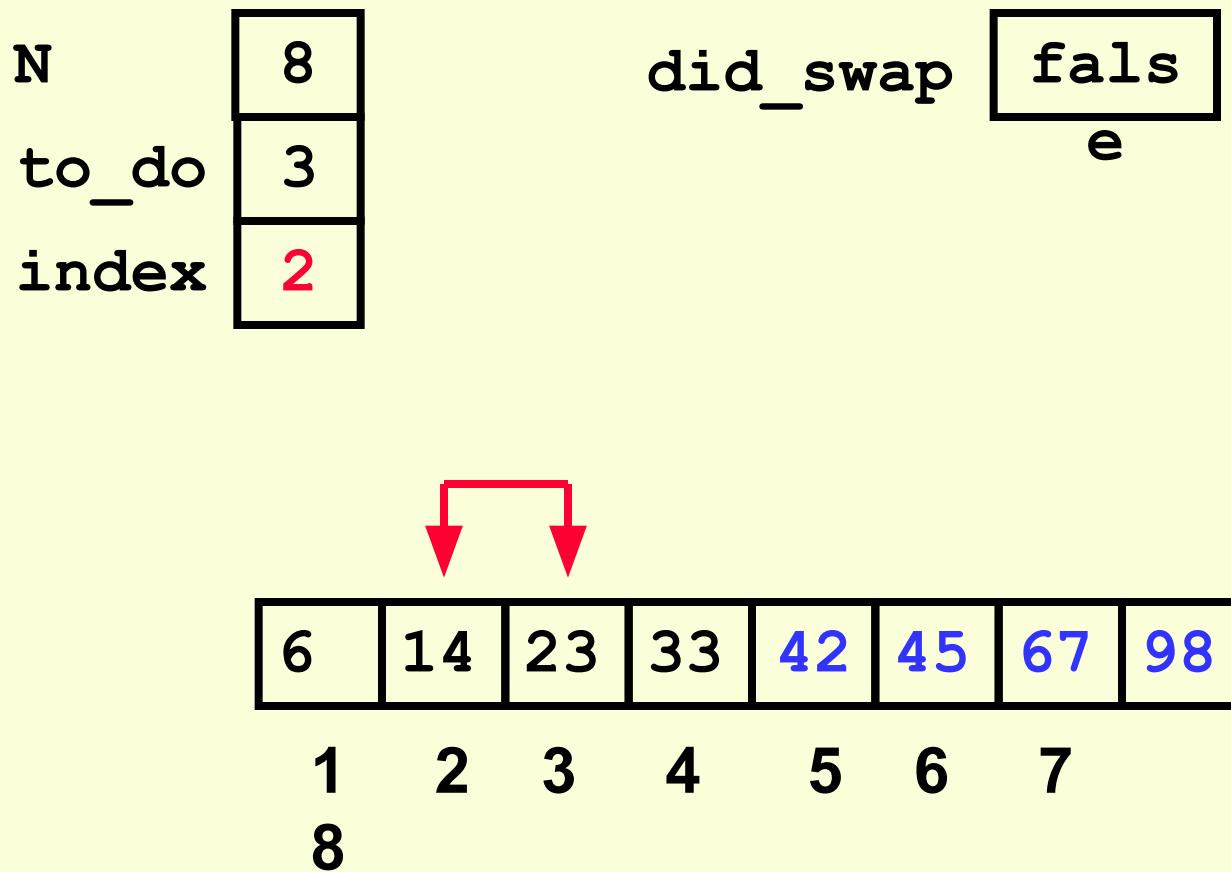
# The Fifth “Bubble Up”



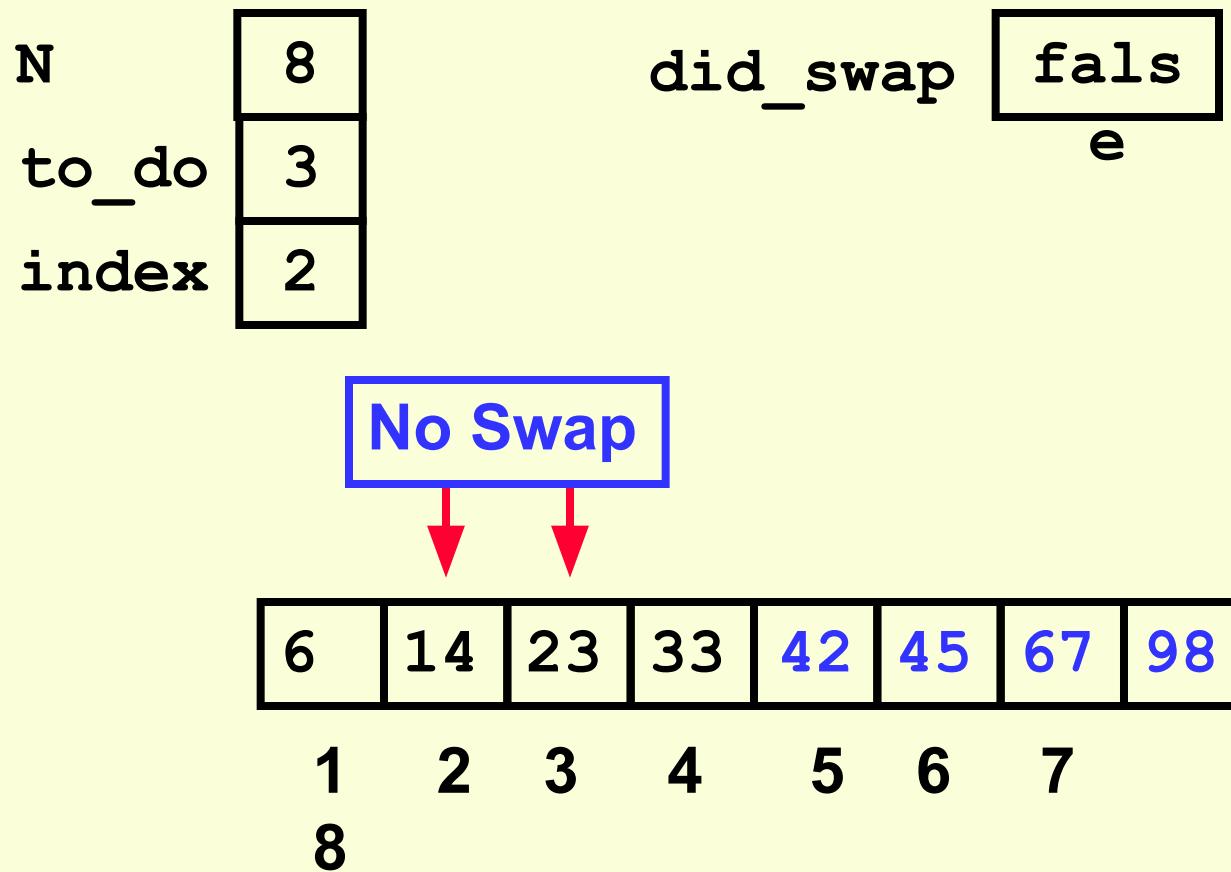
# The Fifth “Bubble Up”



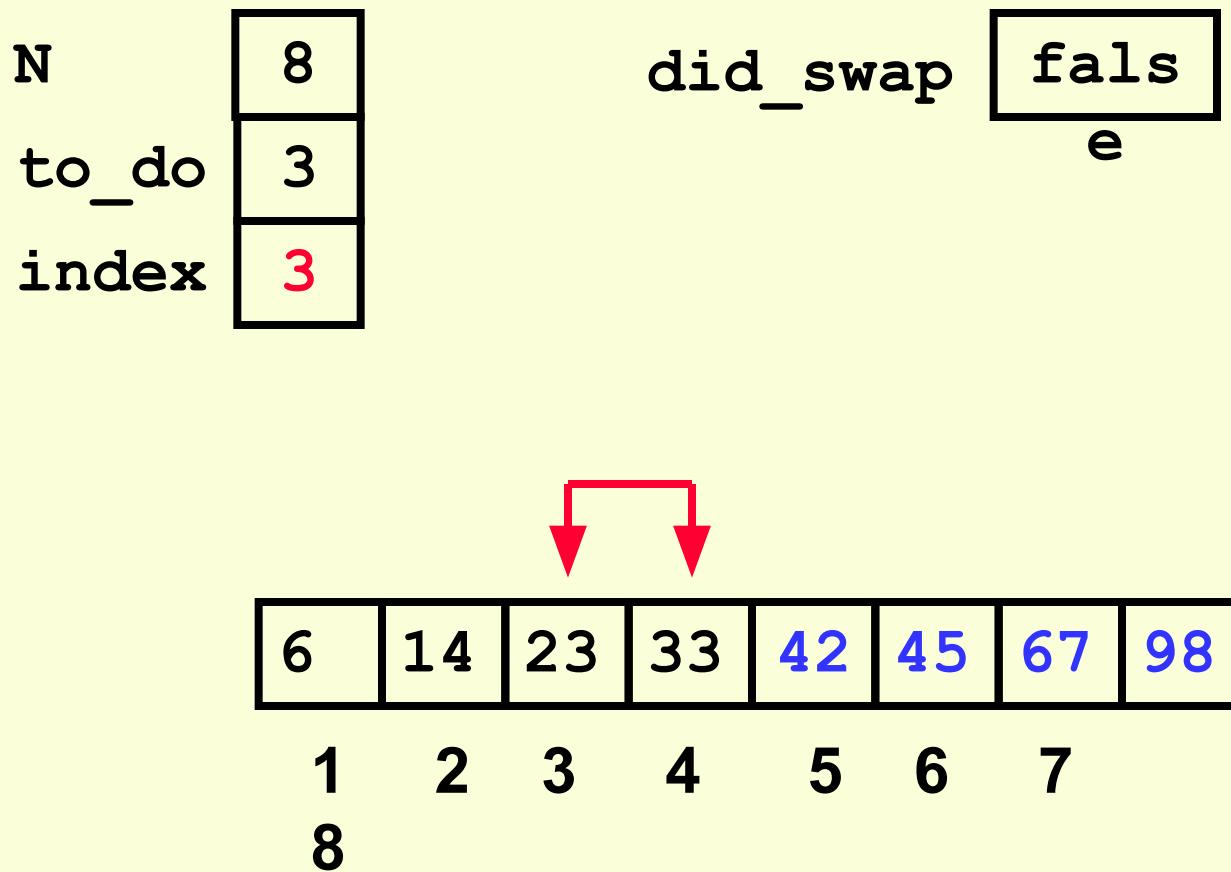
# The Fifth “Bubble Up”



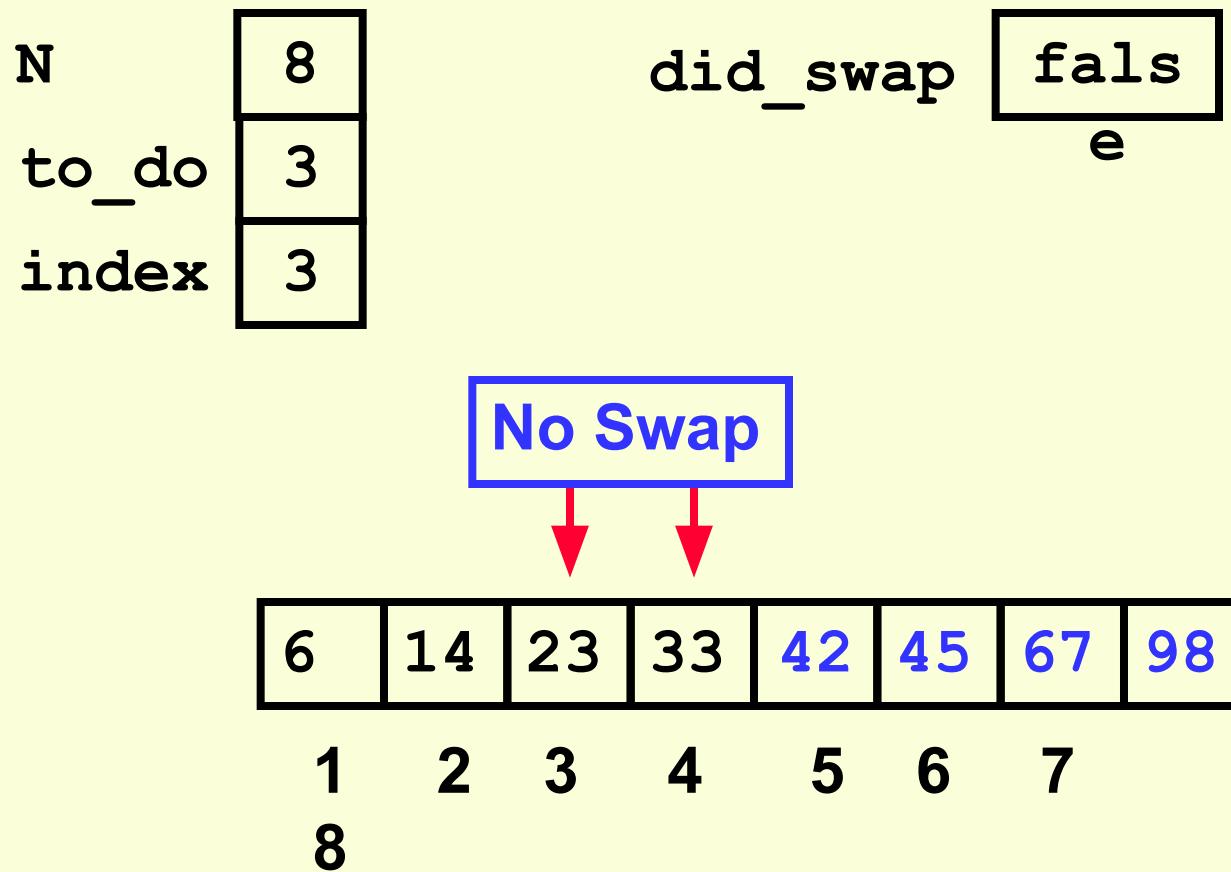
# The Fifth “Bubble Up”



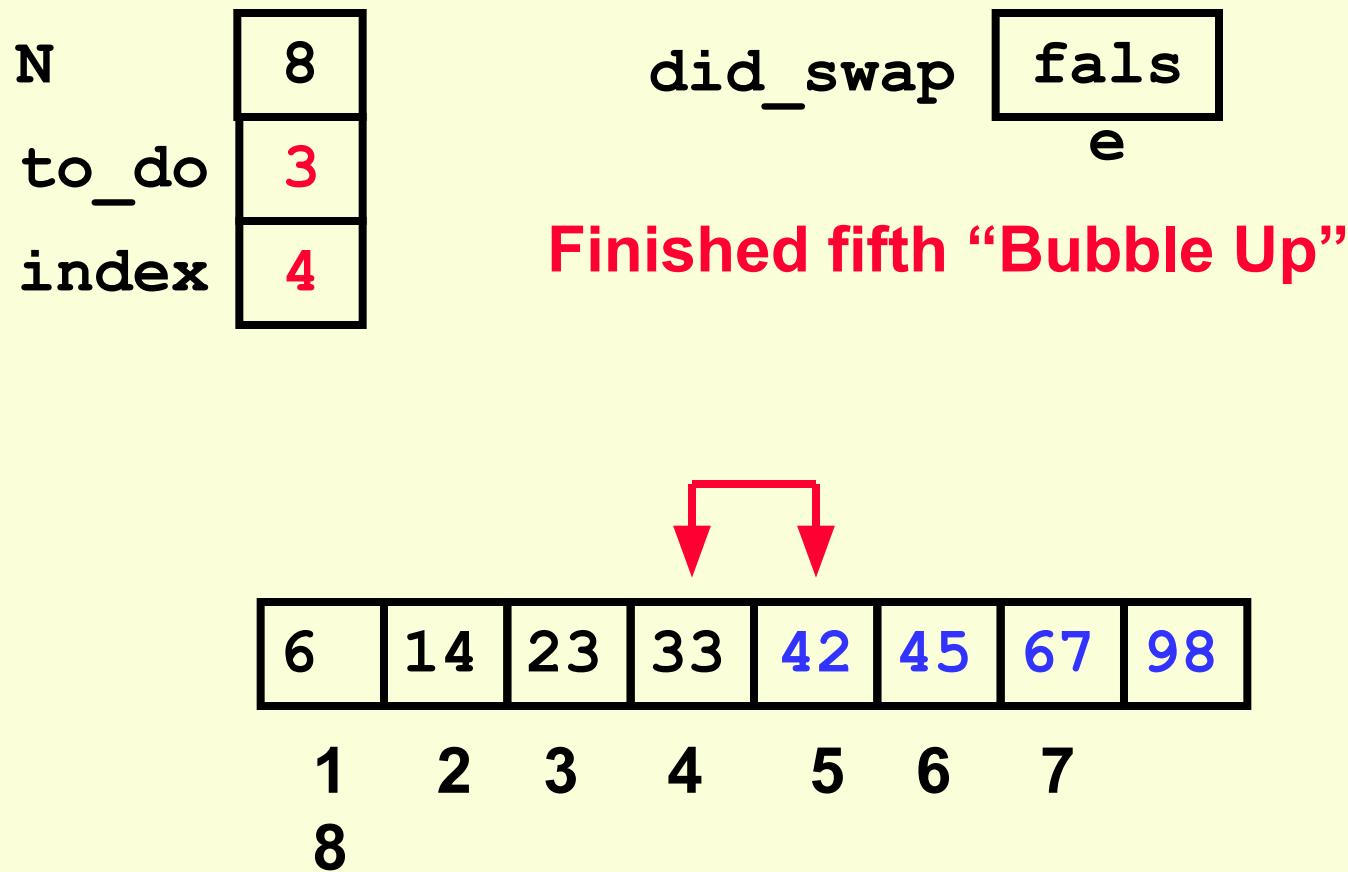
# The Fifth “Bubble Up”



# The Fifth “Bubble Up”



## After Fifth Pass of Outer Loop



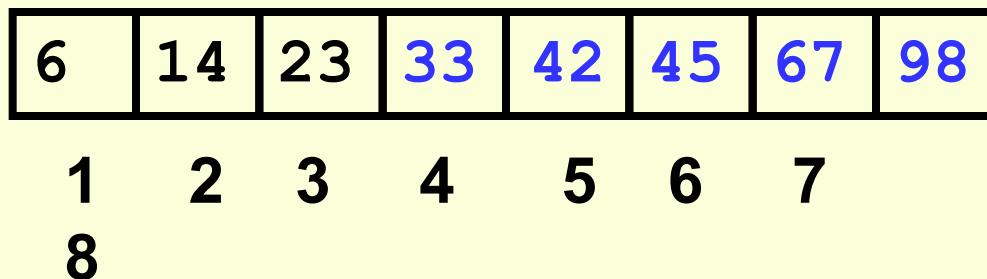
# Finished “Early”

N	8
to_do	3
index	4

did\_swap false  
e

We didn't do any swapping,  
so all of the other elements  
must be correctly placed.

We can “skip” the last two  
passes of the outer loop.



## Radix Sort Algorithm

Radix sort is one of the sorting algorithms used to sort a list of integer numbers in order. In radix sort algorithm, a list of integer numbers will be sorted based on the digits of individual numbers. Sorting is performed from **least significant digit to the most significant digit**.

Radix sort algorithm requires the number of passes which are equal to the number of digits present in the largest number among the list of numbers. For example, if the largest number is a 3 digit number then that list is sorted with 3 passes.

### Step by Step Process

The Radix sort algorithm is performed using the following steps...

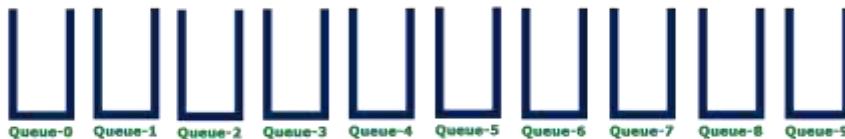
- **Step 1** - Define 10 queues each representing a bucket for each digit from 0 to 9.
- **Step 2** - Consider the least significant digit of each number in the list which is to be sorted.
- **Step 3** - Insert each number into their respective queue based on the least significant digit.
- **Step 4** - Group all the numbers from queue 0 to queue 9 in the order they have inserted into their respective queues.
- **Step 5** - Repeat from step 3 based on the next least significant digit.
- **Step 6** - Repeat from step 2 until all the numbers are grouped based on the most significant digit.

### Example

Consider the following list of unsorted integer numbers

**82, 901, 100, 12, 150, 77, 55 & 23**

**Step 1 -** Define 10 queues each represents a bucket for digits from 0 to 9.



**Step 2 -** Insert all the numbers of the list into respective queue based on the Least significant digit (once placed digit) of every number.

**82, 901, 100, 12, 150, 77, 55 & 23**



Group all the numbers from queue-0 to queue-9 in the order they have inserted & consider the list for next step as input list.

**100, 150, 901, 82, 12, 23, 55 & 77**

**Step 3 -** Insert all the numbers of the list into respective queue based on the next Least significant digit (Tens placed digit) of every number.

**100, 150, 901, 82, 12, 23, 55 & 77**



Group all the numbers from queue-0 to queue-9 in the order they have inserted & consider the list for next step as input list.

**100, 901, 12, 23, 150, 55, 77 & 82**

**Step 4 -** Insert all the numbers of the list into respective queue based on the next Least significant digit (Hundreds placed digit) of every number.

**100, 901, 12, 23, 150, 55, 77 & 82**



Group all the numbers from queue-0 to queue-9 in the order they have inserted & consider the list for next step as input list.

**12, 23, 55, 77, 82, 100, 150, 901**

List got sorted in the increasing order.

# Selection Sort

Suppose an array  $A$  with  $N$  elements is in memory. Selection sort works as follows

First find the smallest element in the list and put it in the first position. Then, find the second smallest element in the list and put it in the second position and so on.

# Selection Sort

Pass	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
K=1 LOC=4	77	33	44	11	88	22	66	55
K=2 LOC=6	11	33	44	77	88	22	66	55
K=3 LOC=6	11	22	44	77	88	33	66	55
K=4 LOC=6	11	22	33	77	88	44	66	55
K=5 LOC=8	11	22	33	44	88	77	66	55
K=6 LOC=7	11	22	33	44	55	77	66	88
K=7 LOC=4	11	22	33	44	55	66	77	88

Sorted	11	22	33	44	55	66	77	88
--------	----	----	----	----	----	----	----	----

Pass 1: Find the location LOC of the smallest element in the list  $A[1], A[2], \dots, A[N]$ . Then interchange  $A[LOC]$  and  $A[1]$ . Then:  $A[1]$  is sorted

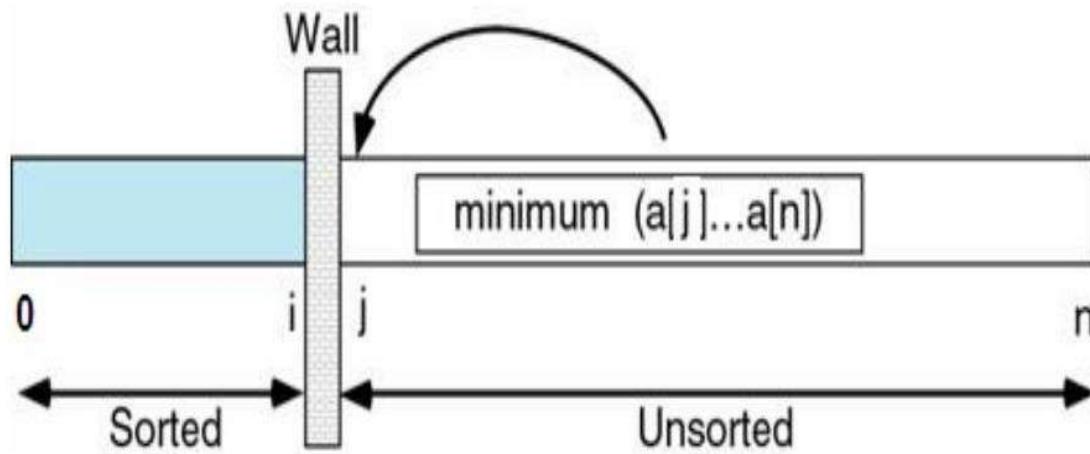
Pass 2: Find the location LOC of the smallest element in the sublist  $A[2], A[3], \dots, A[N]$ . Then interchange  $A[LOC]$  and  $A[2]$ . Then:  $A[1], A[2]$  is sorted since  $A[1] \leq A[2]$ .

Pass 3: Find the location LOC of the smallest element in the sublist  $A[3], A[4], \dots, A[N]$ . Then interchange  $A[LOC]$  and  $A[3]$ . Then:  $A[1], A[2], A[3]$  is sorted, since  $A[2] \leq A[3]$ .

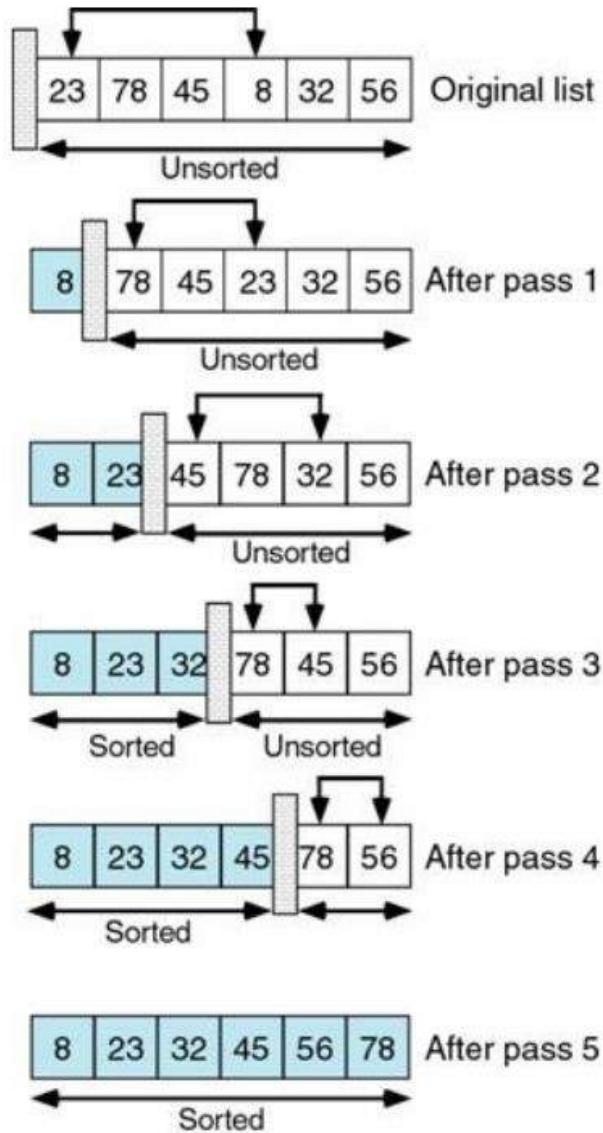
Pass N-1: Find the location  $LOC$  of the smallest element in the sublist  $A[N-1], A[N]$ . Then interchange  $A[LOC]$  and  $A[N-1]$ . Then:  $A[1], A[2], \dots, A[N]$  is sorted, since  $A[N-1] \leq A[N]$ .

**A is sorted after N-1 pass.**

# Selection Sort



# Example: Selection Sort



---

```
for (i = 0 ; i < n-1 ; i++)
{
    index = i;
    for(j = i+1 ; j < n ; j++)
    {
        if(A[j] < A[index])
            index = j;
    }
    temp = A[i];
    A[i] = A[index];
    A[index] = temp;
```

# Example

- Unsorted list:

5	2	1	4	3
---	---	---	---	---

- 1st iteration:

- Smallest = 5
- $2 < 5$ , smallest = 2
- $1 < 2$ , smallest = 1
- $4 > 1$ , smallest = 1
- $3 > 1$ , smallest = 1

Swap 5 and 1

1	2	5	4	3
---	---	---	---	---

## **Example contd.**

- 2nd iteration:
  - Smallest = 2
  - $2 < 5$ , smallest = 2
  - $2 < 4$ , smallest = 2
  - $2 < 3$ , smallest = 2
- No swap

1	2	5	4	3
---	---	---	---	---

## Example contd.

- 3rd iteration:
  - Smallest = 5
  - $4 < 5$ , smallest = 4
  - $3 < 4$ , smallest = 3
- Swap 5 and 3

1	2	3	4	5
---	---	---	---	---

## Example contd.

- **4th iteration:**

- Smallest = 4
- $4 < 5$ , smallest = 4

- **No swap**

1	2	3	4	5
---	---	---	---	---

- **Finally, the sorted list is**

1	2	3	4	5
---	---	---	---	---

# Selection Sort: Analysis

## Complexity:

- Best case performance  $O(n^2)$
- Average case performance  $O(n^2)$
- Worst case performance  $O(n^2)$

# **Advantages & Disadvantages**

## **Advantages**

- The main advantage of the selection sort is that it performs well on a small list.
- no additional temporary storage is required.

## **Disadvantages**

- The primary disadvantage of the selection sort is its poor efficiency when dealing with a huge list of items.
- the selection sort requires n-squared number of steps for sorting n elements.



# **Insertion Sort**

# *Introduction*

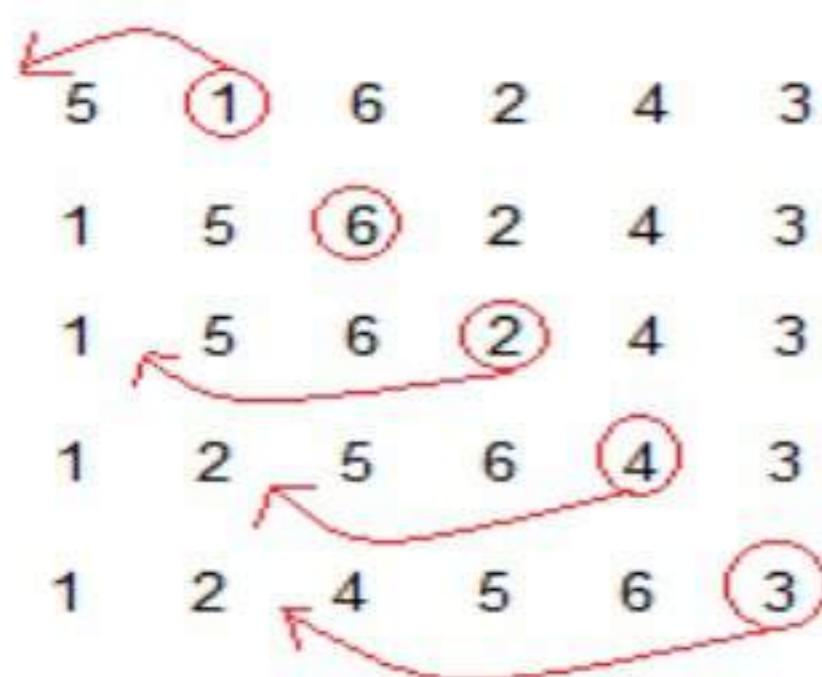
The Idea of the insertion sort is similar to the Idea of sorting the Playing cards .



# Insertion Sorting

- It is a simple Sorting algorithm which sorts the array by shifting elements one by one. Following are some of the important characteristics of Insertion Sort.
- It has one of the simplest implementation
- It is efficient for smaller data sets, but very inefficient for larger lists.
- Insertion Sort is adaptive, that means it reduces its total number of steps if given a partially sorted list, hence it increases its efficiency.
- It is better than Selection Sort and Bubble Sort algorithms.
- Its space complexity is less, like Bubble Sorting, insertion sort also requires a single additional memory space.
- It is Stable, as it does not change the relative order of elements with equal keys

5	1	6	2	4	3
---	---	---	---	---	---



Lets take this Array.

As we can see here, in insertion sort, we pick up a key, and compares it with elements ahead of it, and puts the key in the right place

5 has nothing before it.

1 is compared to 5 and is inserted before 5.

6 is greater than 5 and 1.

2 is smaller than 6 and 5, but greater than 1, so its is inserted after 1.

( Always we start with the second element as key.)

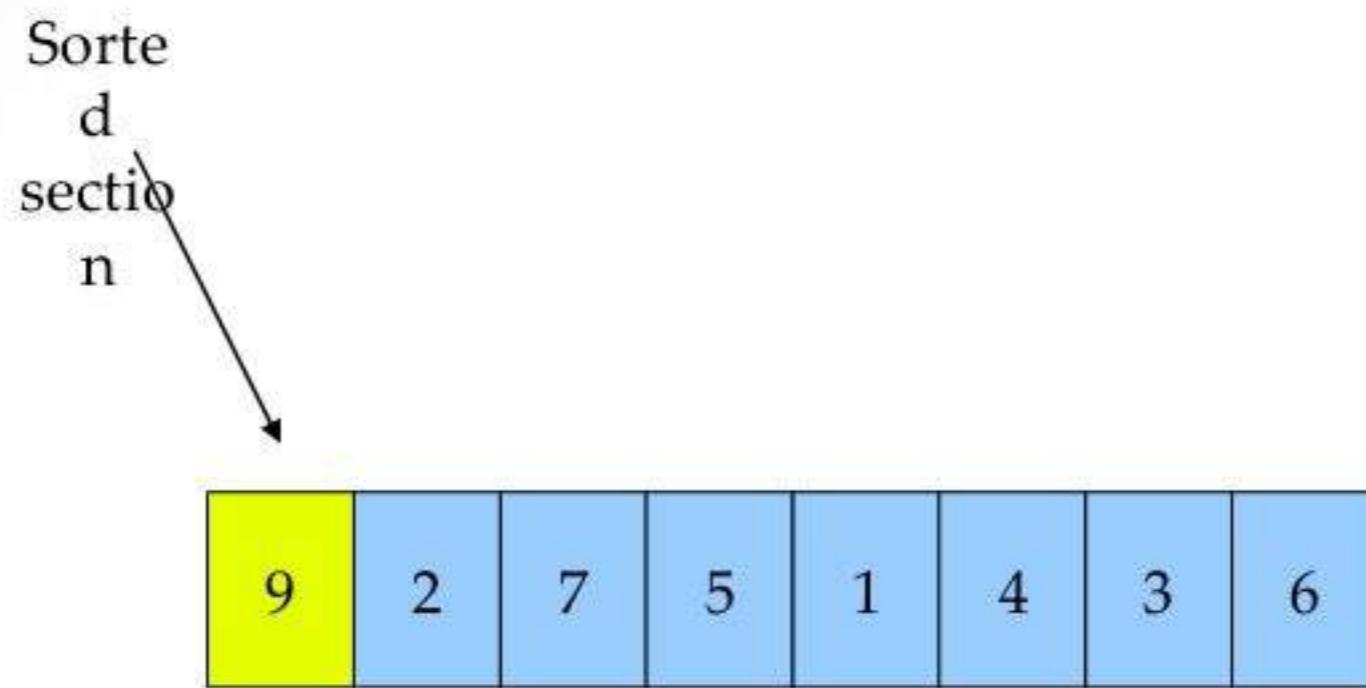
And this goes on...

# Insertion Sort

**Example :**

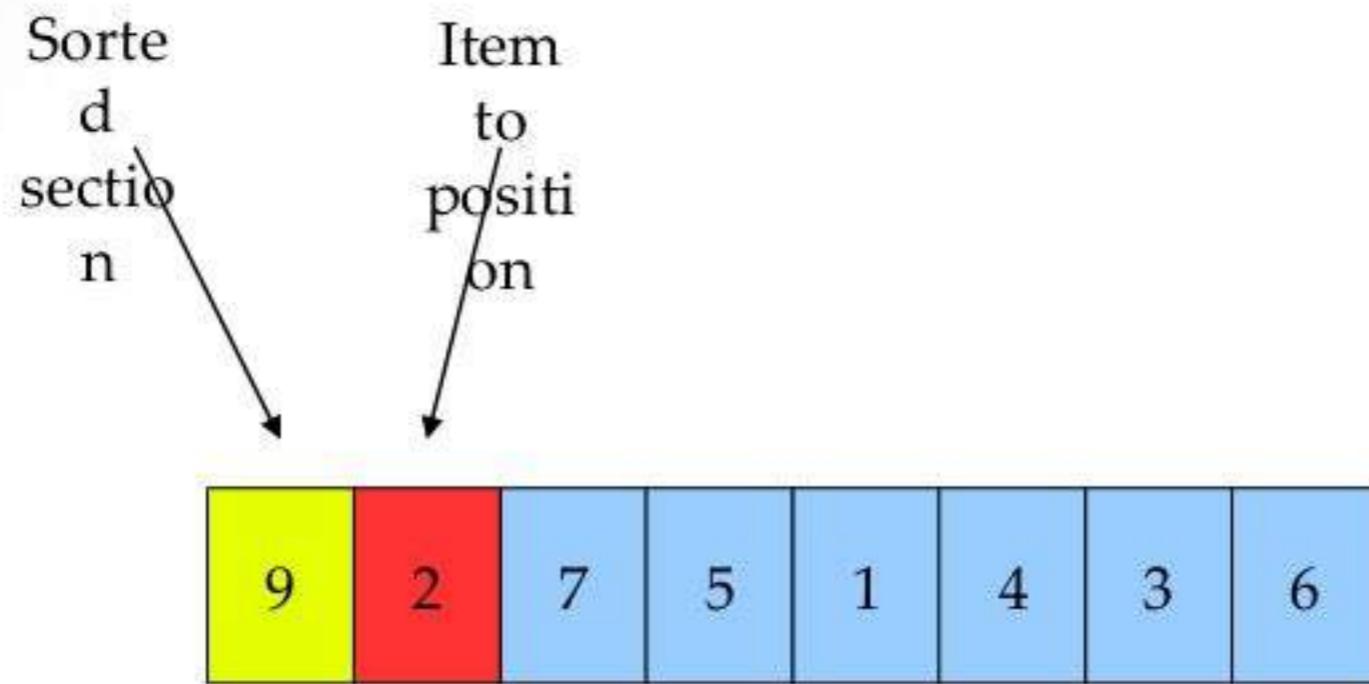
9	2	7	5	1	4	3	6
---	---	---	---	---	---	---	---

# Insertion Sort



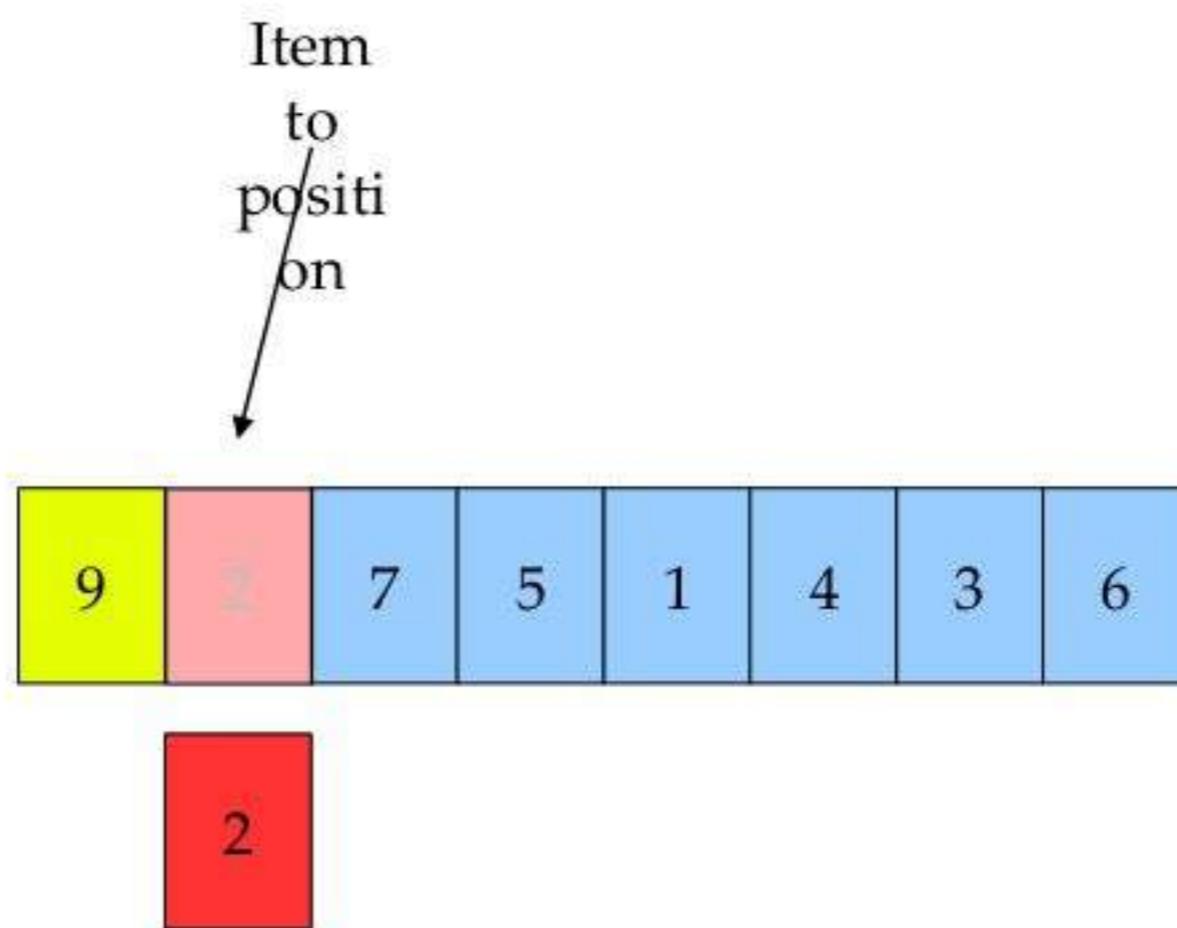
We start by dividing the array in a sorted section and an unsorted section. We put the first element as the only element in the sorted section, and the rest of the array is the unsorted section.

# Insertion Sort



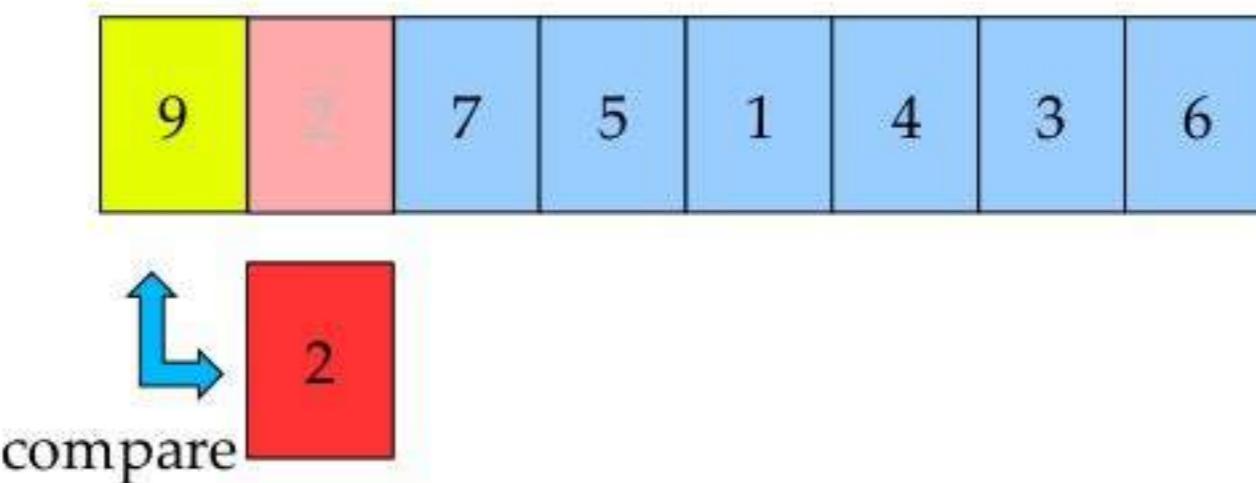
**The first element in the unsorted section is the next element to be put into the correct position.**

# Insertion Sort



We copy the element to be placed into another variable so it doesn't get overwritten.

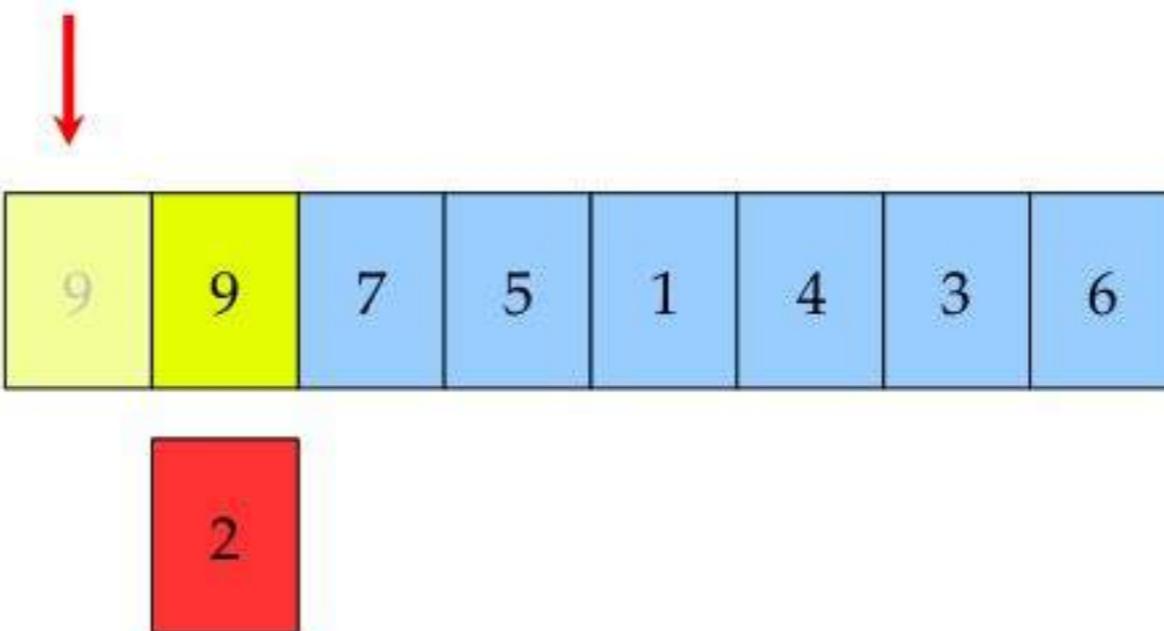
# Insertion Sort



If the previous position is more than the item being placed, copy the value into the next position

# Insertion Sort

belongs here



If there are no more items in the sorted section to compare with, the item to be placed must go at the front.

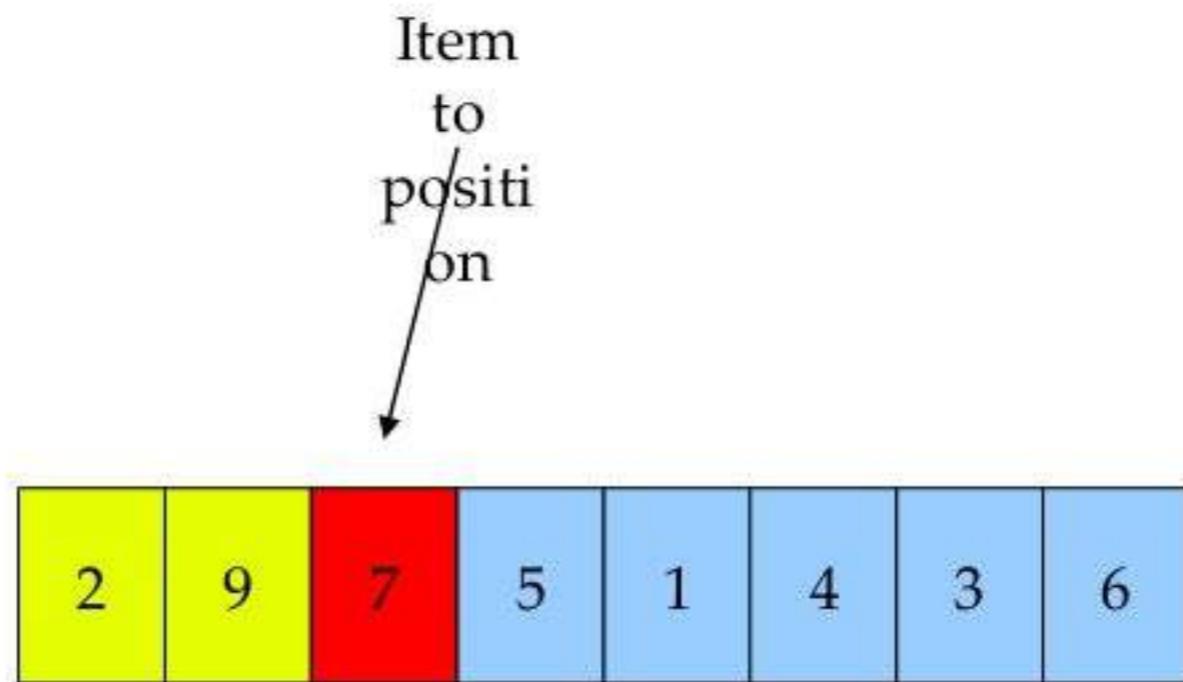
# Insertion Sort



# Insertion Sort

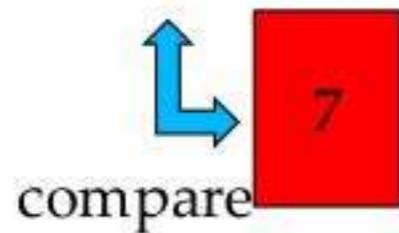
2	9	7	5	1	4	3	6
---	---	---	---	---	---	---	---

# Insertion Sort

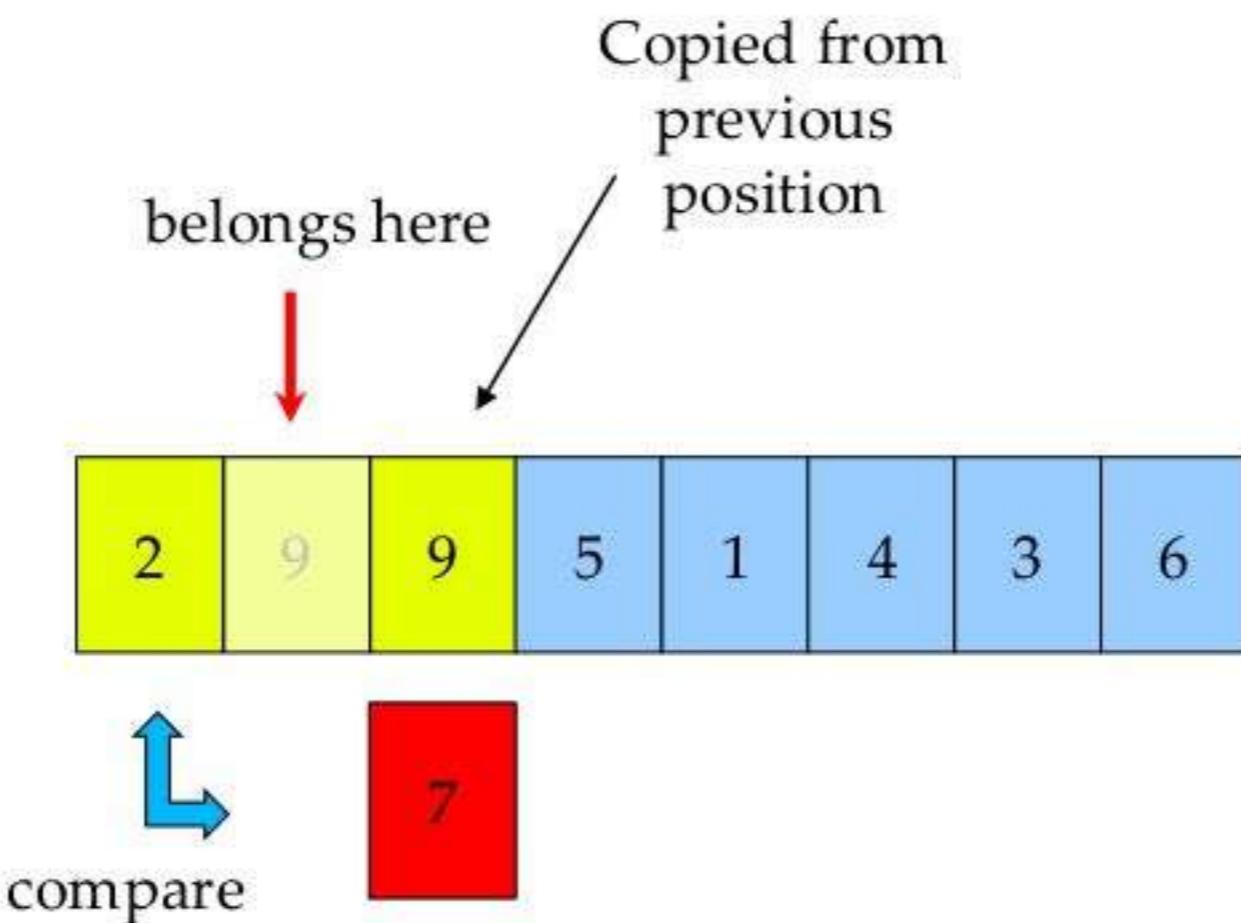


# Insertion Sort

2	9	7	5	1	4	3	6
---	---	---	---	---	---	---	---



# Insertion Sort



If the item in the sorted section is less than the item to place, the item to place goes *after* it in the array.

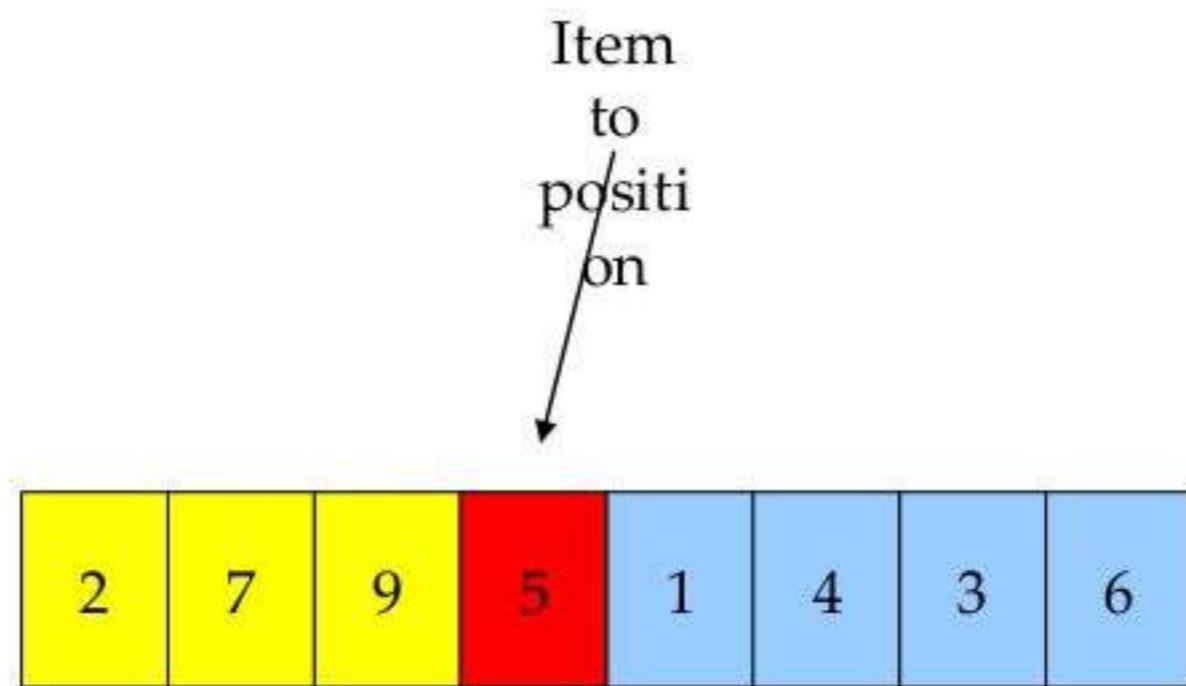
# Insertion Sort



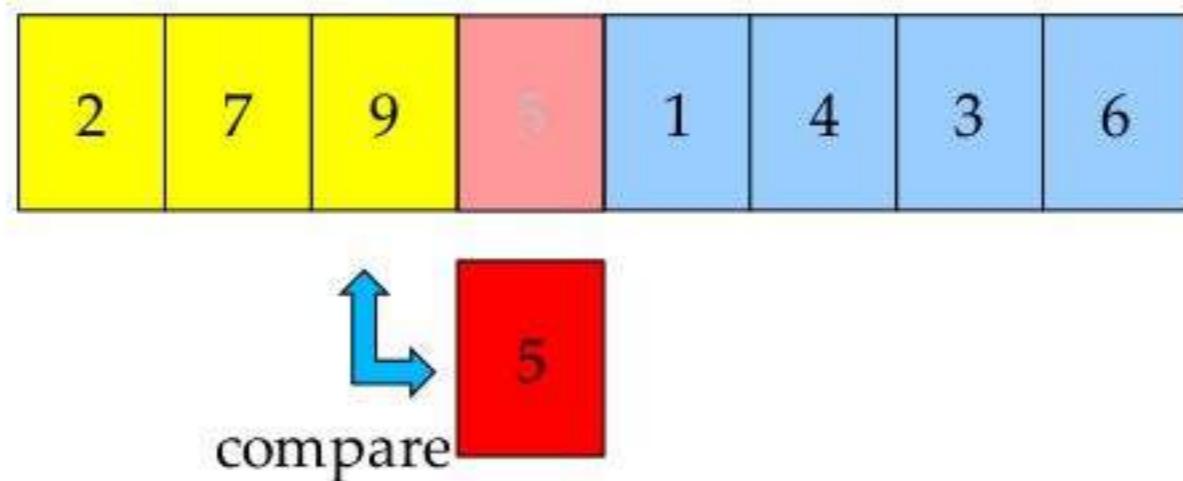
# Insertion Sort

2	7	9	5	1	4	3	6
---	---	---	---	---	---	---	---

# Insertion Sort

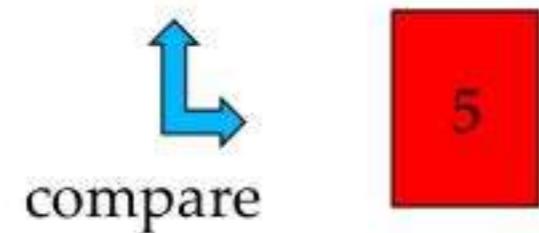


# Insertion Sort



# Insertion Sort

2	7	9	9	1	4	3	6
---	---	---	---	---	---	---	---



# Insertion Sort

belongs here



2	7	7	9	1	4	3	6
---	---	---	---	---	---	---	---



compare



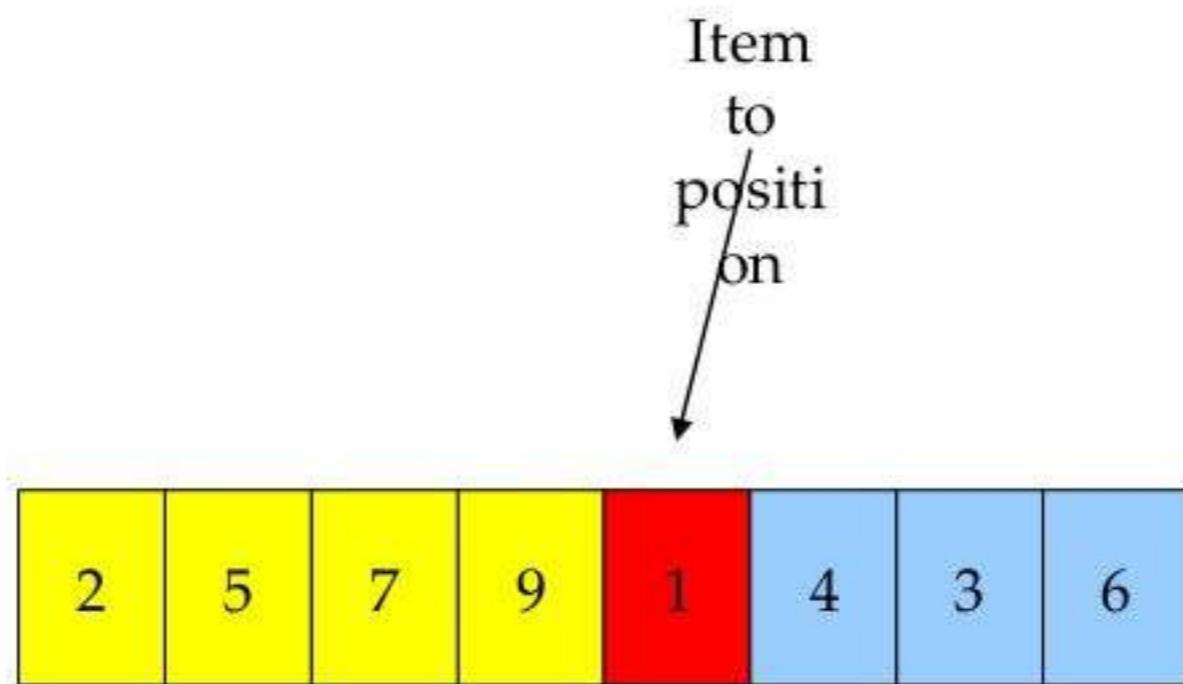
# Insertion Sort



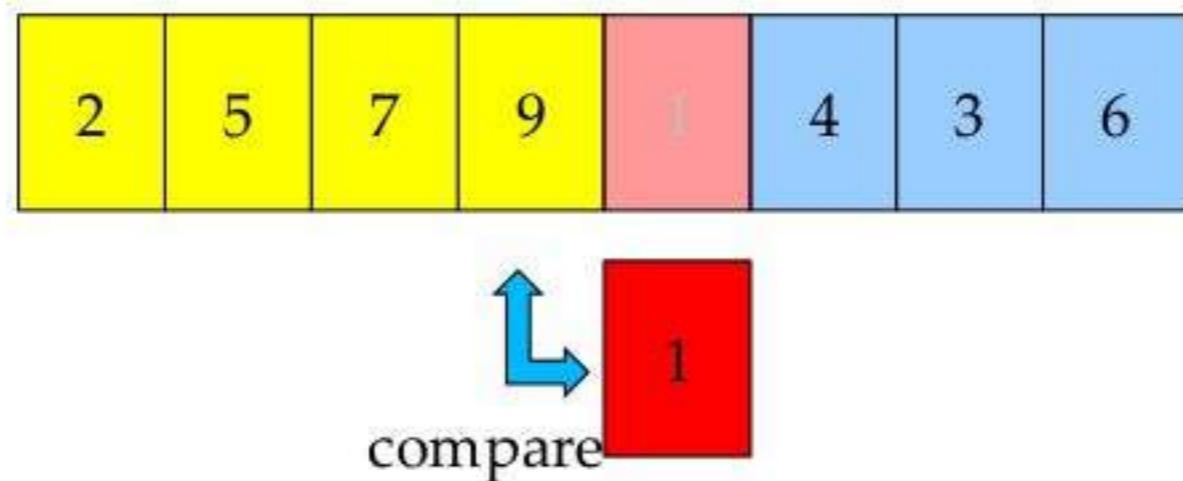
# Insertion Sort

2	5	7	9	1	4	3	6
---	---	---	---	---	---	---	---

# Insertion Sort

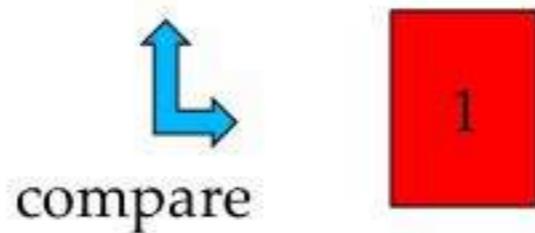


# Insertion Sort

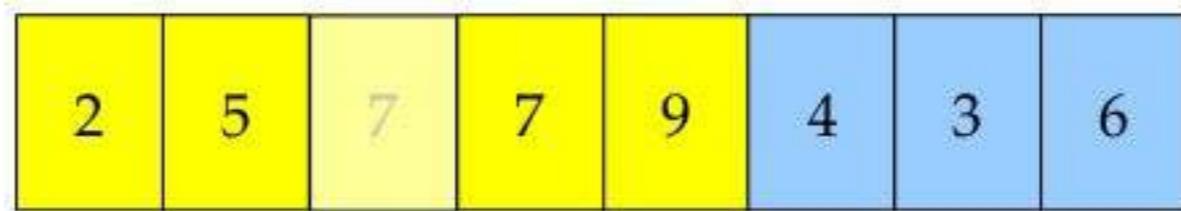


# Insertion Sort

2	5	7	9	9	4	3	6
---	---	---	---	---	---	---	---



# Insertion Sort



# Insertion Sort



compare



# Insertion Sort

belongs here



1

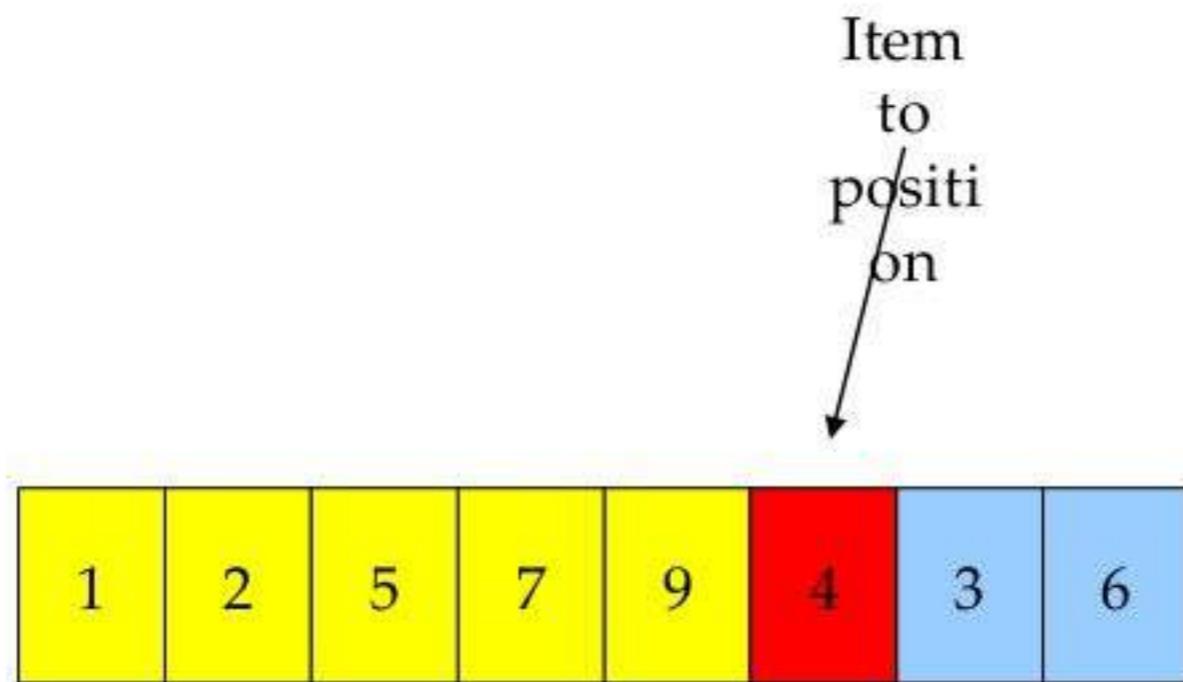
# Insertion Sort



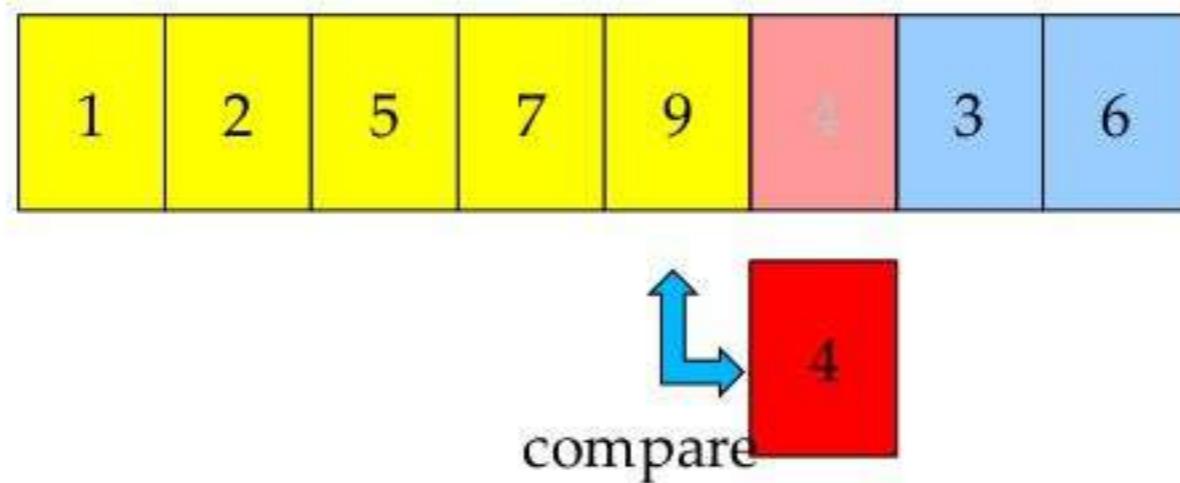
# Insertion Sort

1	2	5	7	9	4	3	6
---	---	---	---	---	---	---	---

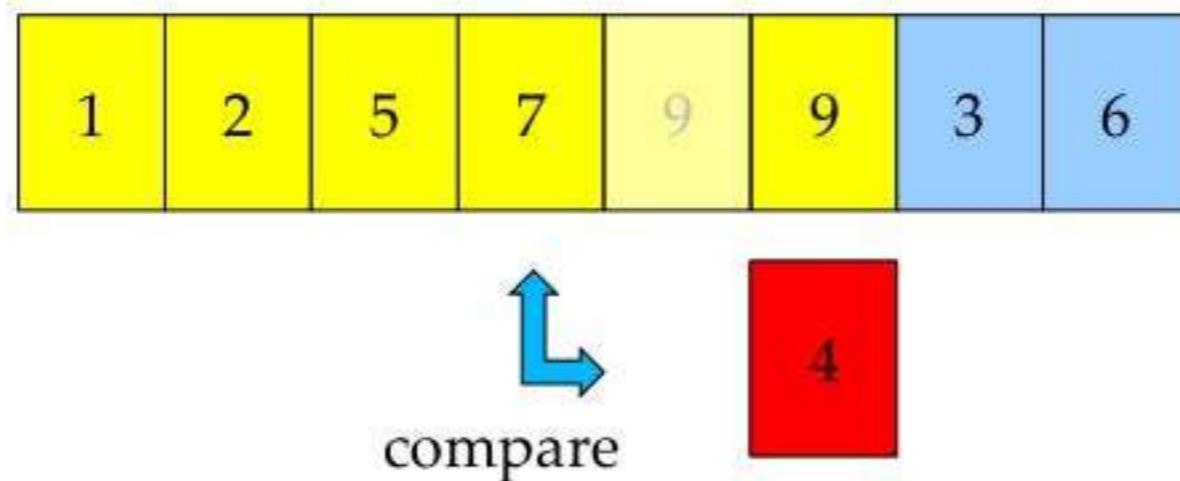
# Insertion Sort



# Insertion Sort



# Insertion Sort



# Insertion Sort

1	2	5	7	7	9	3	6
---	---	---	---	---	---	---	---



compare



# Insertion Sort

belongs here



1	2	5	5	7	9	3	6
---	---	---	---	---	---	---	---



compare



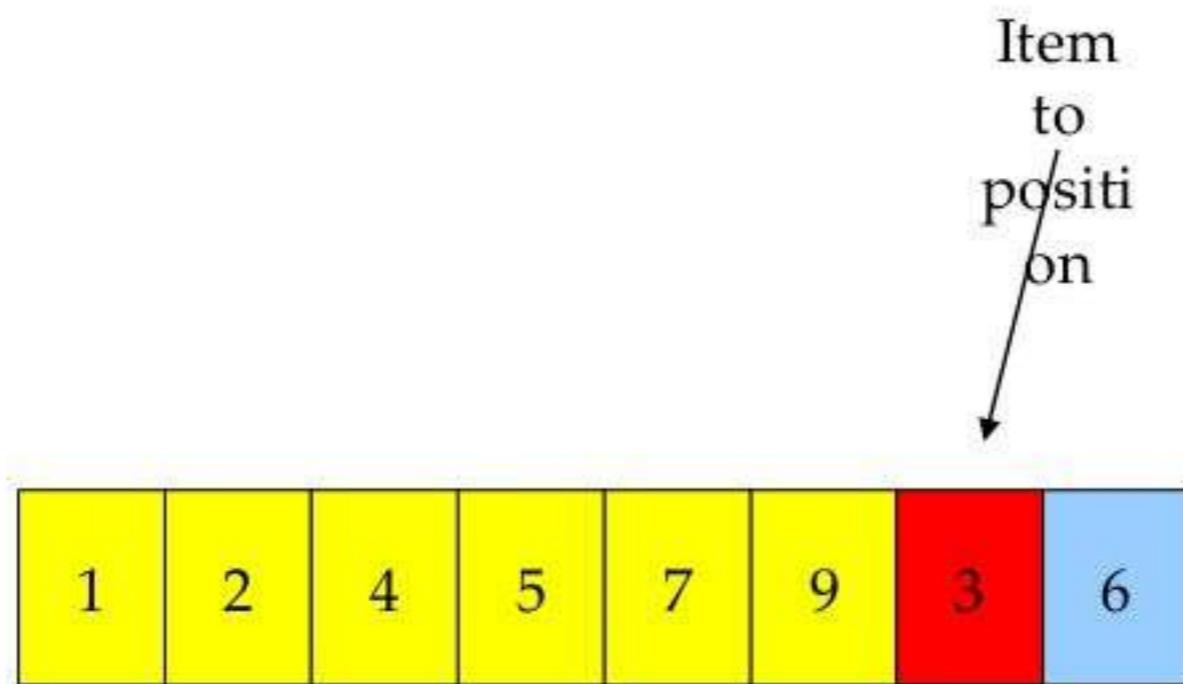
# Insertion Sort



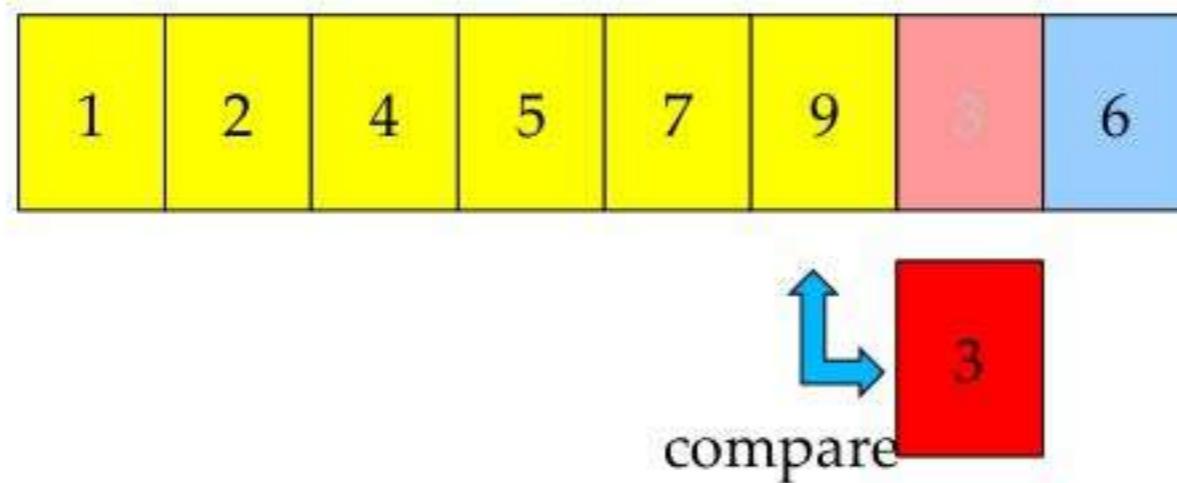
# Insertion Sort

1	2	4	5	7	9	3	6
---	---	---	---	---	---	---	---

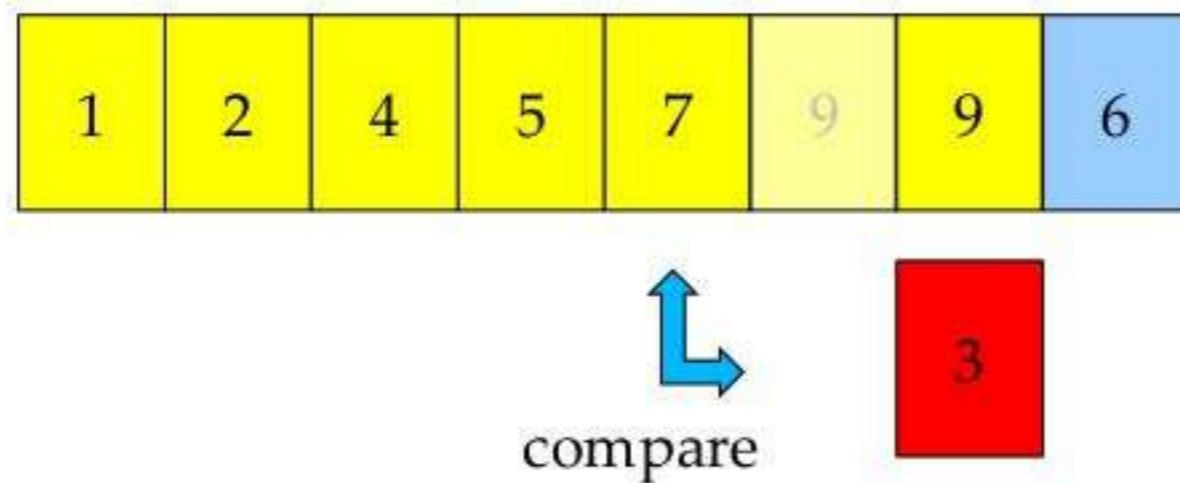
# Insertion Sort



# Insertion Sort



# Insertion Sort



# Insertion Sort

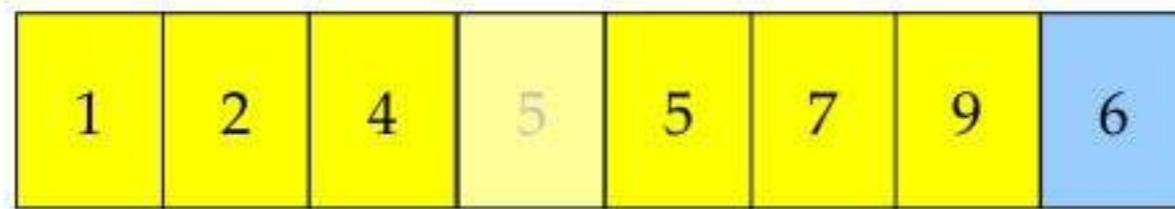
1	2	4	5	7	7	9	6
---	---	---	---	---	---	---	---



compare



# Insertion Sort



# Insertion Sort

belongs here



compare



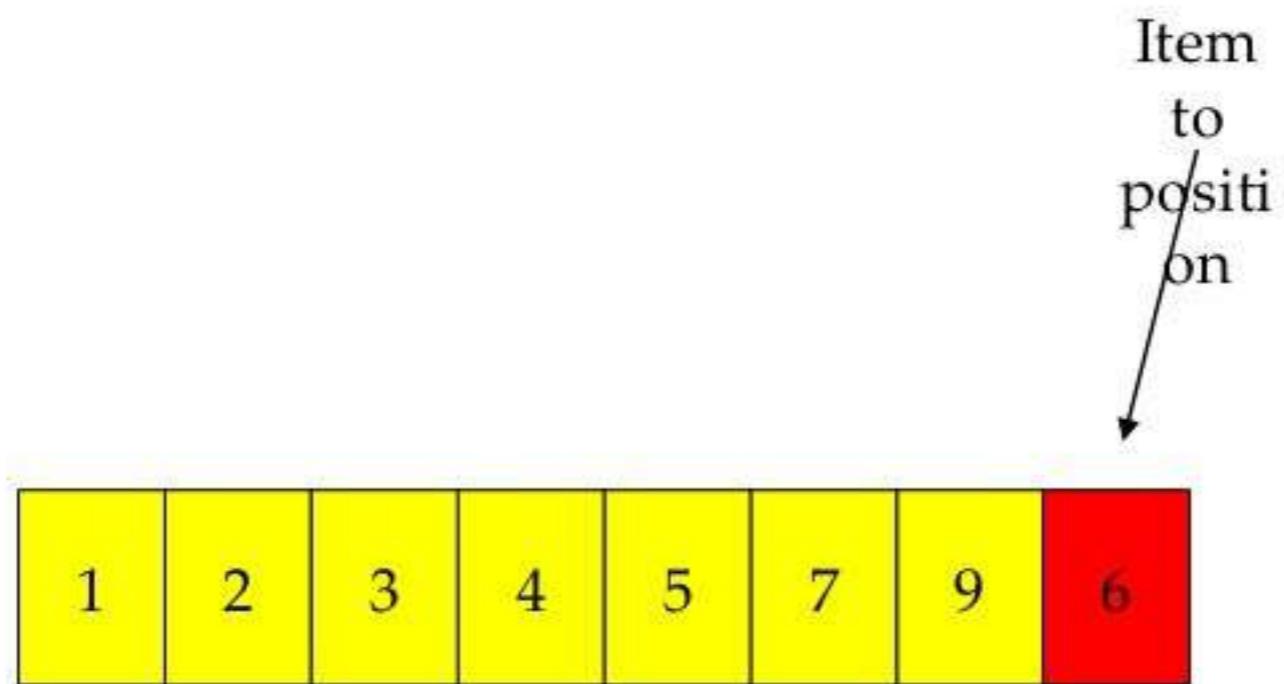
# Insertion Sort



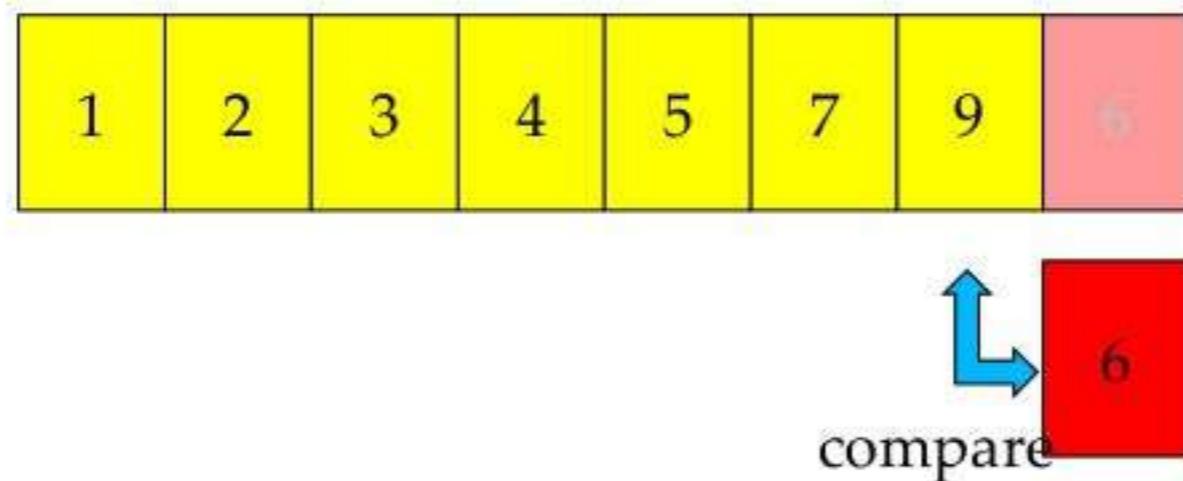
# Insertion Sort



# Insertion Sort

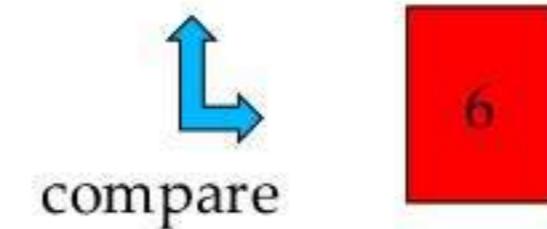


# Insertion Sort

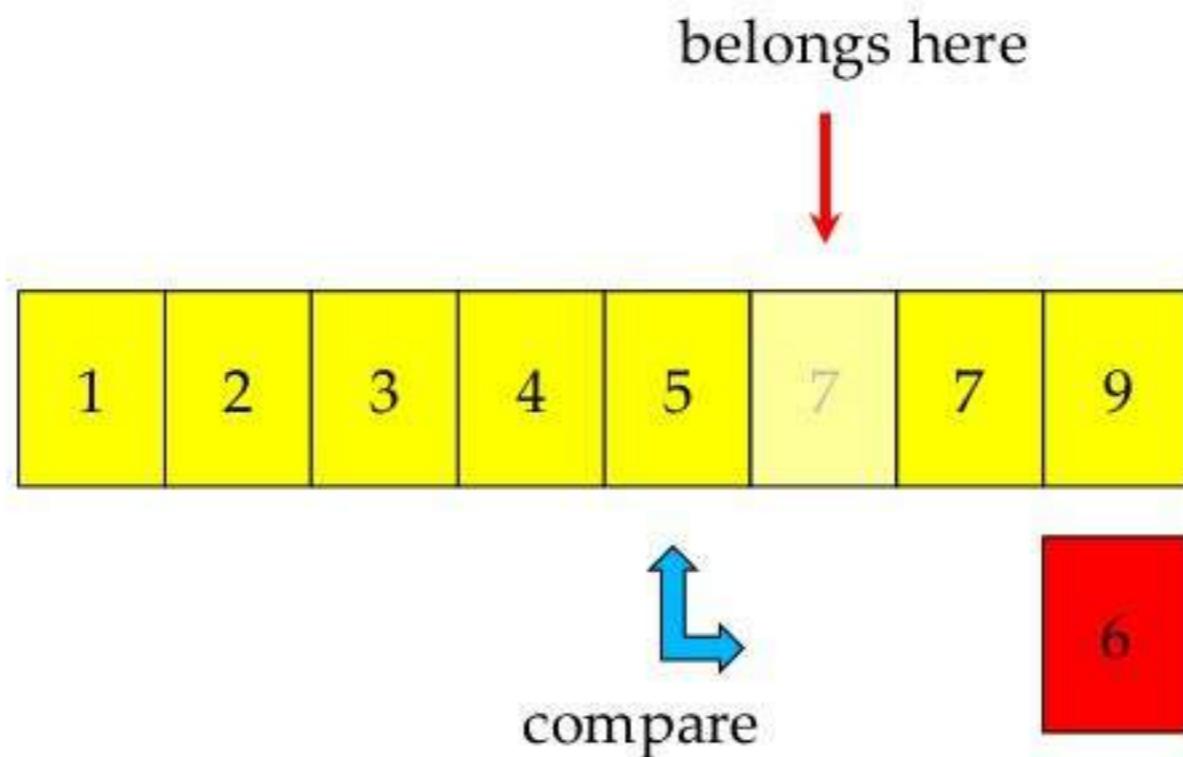


# *Insertion Sort*

1	2	3	4	5	7	9	9
---	---	---	---	---	---	---	---



# Insertion Sort



# Insertion Sort

1	2	3	4	5	6	7	9
---	---	---	---	---	---	---	---

# Insertion Sort

1	2	3	4	5	6	7	9
---	---	---	---	---	---	---	---

SORTED!

\*-\*-\* (Way of working) \*-\*-\*

Select – Compare – Shift - Insert

$j$	1	2	3	4	5	6
	5	2	4	6	1	3

$j$	1	2	3	4	5	6
	2	4	5	6	1	3

$j$	1	2	3	4	5	6
	2	5	4	6	1	3

$j$	1	2	3	4	5	6
	1	2	4	5	6	3

$j$	1	2	3	4	5	6
	2	4	5	6	1	3

$j$	1	2	3	4	5	6
	1	2	3	4	5	6



# Pseudo Code

- **for**  $i = 0$  to  $n - 1$ 
  - $j = 1$
  - while**  $j > 0$  and  $A[j] < A[j - 1]$ 
    - swap**( $A[j]$ ,  $A[j-1]$ )
    - $j = j - 1$

# CODE OF INSERTION SORT

```
void insertion_sort (int arr[], int length)
{
    int j,temp;

    for (int i = 0; i < length; i++)
    {
        j = i;

        while (j > 0 && arr[j] < arr[j-1])
        {
            temp = arr[j];
            arr[j] = arr[j-1];
            arr[j-1] = temp;
            j--;
        }
    }
}
```

# Sorting

- Insertion sort
  - Design approach: incremental
  - Sorts in place: Yes
  - Best case:  $\Theta(n)$
  - Worst case:  $\Theta(n^2)$
- Bubble Sort
  - Design approach: incremental
  - Sorts in place: Yes
  - Best case:  $\Theta(n)$
  - Running time:  $\Theta(n^2)$

# Sorting

- Selection sort
  - Design approach: incremental
  - Sorts in place: Yes
  - Running time:  $\Theta(n^2)$
- Merge Sort
  - Design approach: divide and conquer
  - Sorts in place: No
  - Running time: Let's see!!

# Divide-and-Conquer

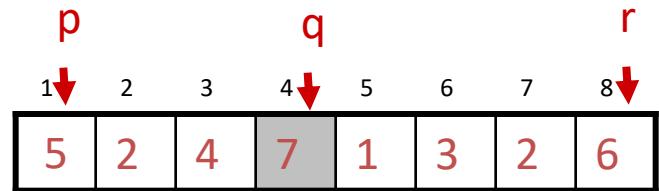
- **Divide** the problem into a number of sub-problems
  - Similar sub-problems of smaller size
- **Conquer** the sub-problems
  - Solve the sub-problems recursively
  - Sub-problem size small enough  $\Rightarrow$  solve the problems in straightforward manner
- **Combine** the solutions of the sub-problems
  - Obtain the solution for the original problem

# Merge Sort Approach

- To sort an array  $A[p \dots r]$ :
- **Divide**
  - Divide the  $n$ -element sequence to be sorted into two subsequences of  $n/2$  elements each
- **Conquer**
  - Sort the subsequences recursively using merge sort
  - When the size of the sequences is 1 there is nothing more to do
- **Combine**
  - Merge the two sorted subsequences

# Merge Sort

*Alg.:* MERGE-SORT( $A, p, r$ )



if  $p < r$  ▷ Check for base case

then  $q \leftarrow \lfloor (p + r)/2 \rfloor$  ▷ Divide

MERGE-SORT( $A, p, q$ ) Conquer

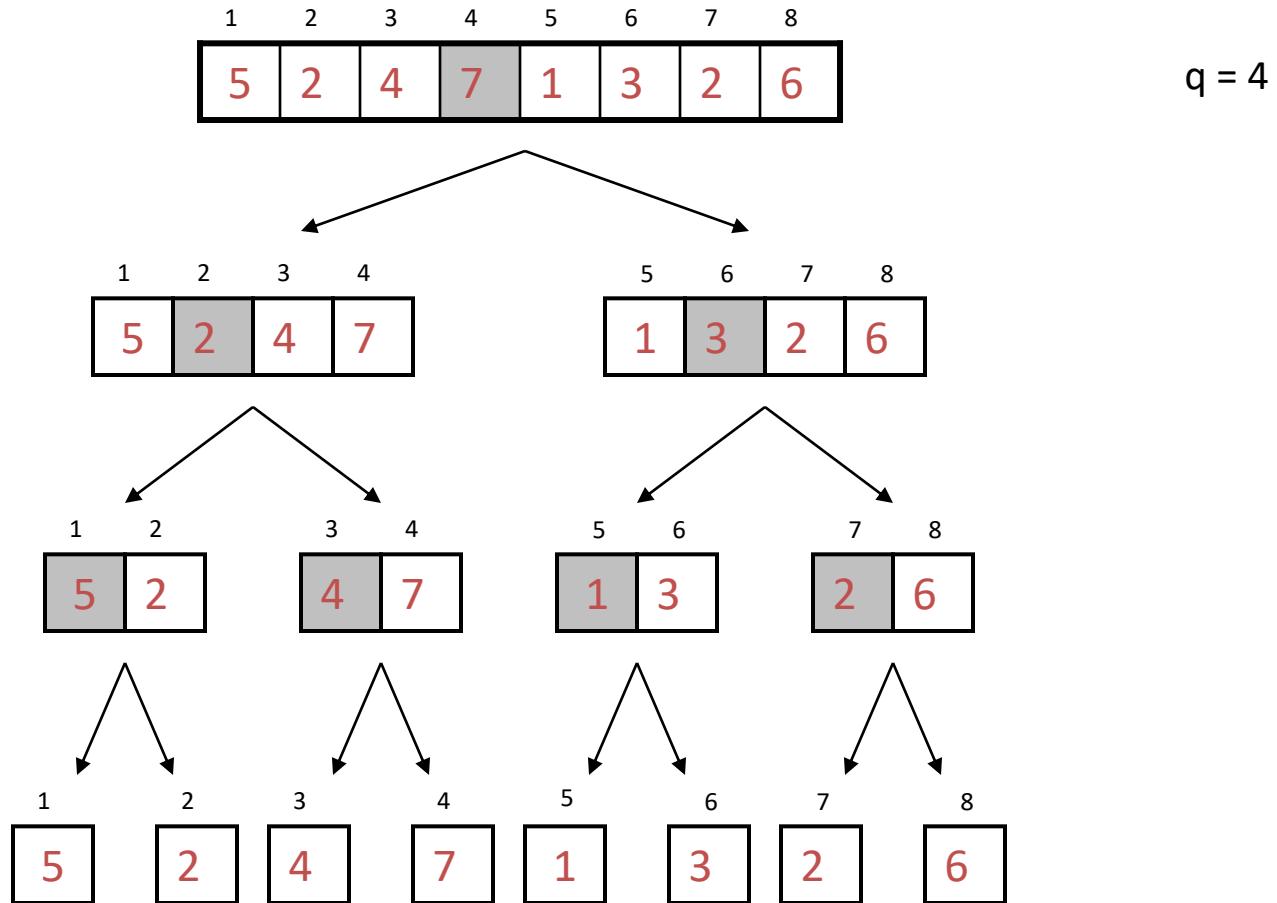
MERGE-SORT( $A, q + 1, r$ ) ▷ Conquer

MERGE( $A, p, q, r$ ) ▷ Combine

- Initial call: MERGE-SORT( $A, 1, n$ )

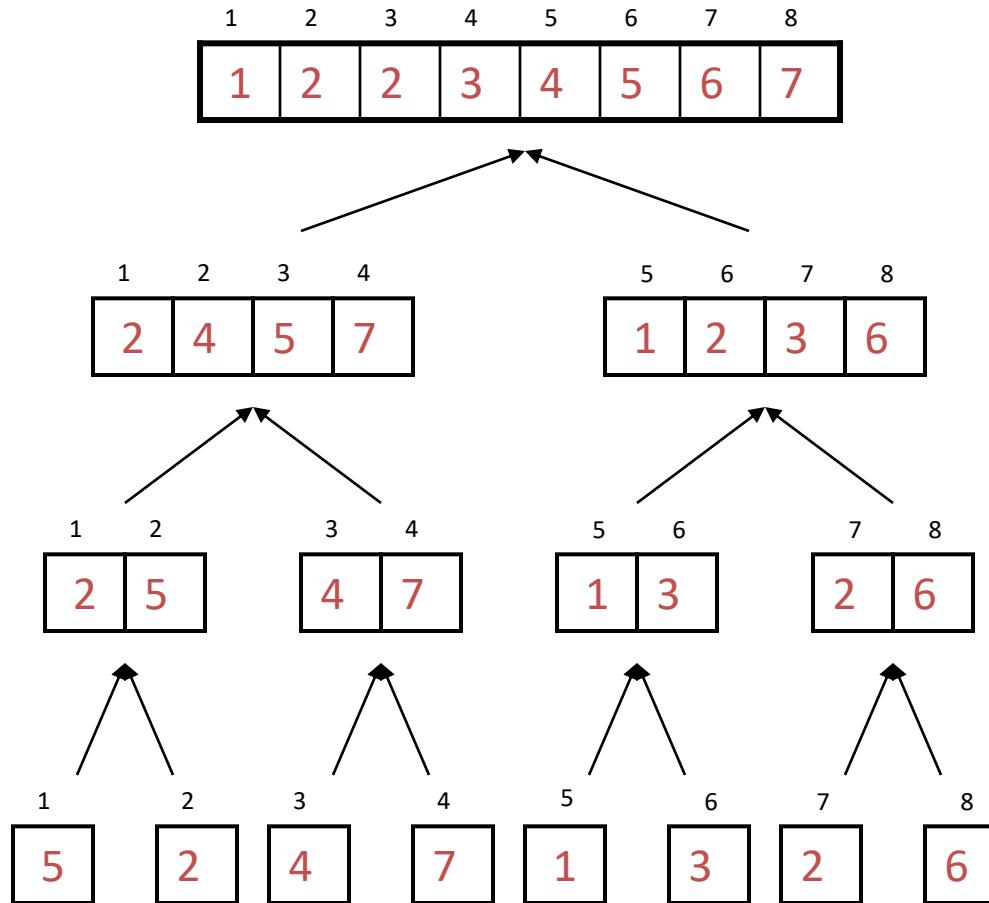
# Example – $n$ Power of 2

Divide



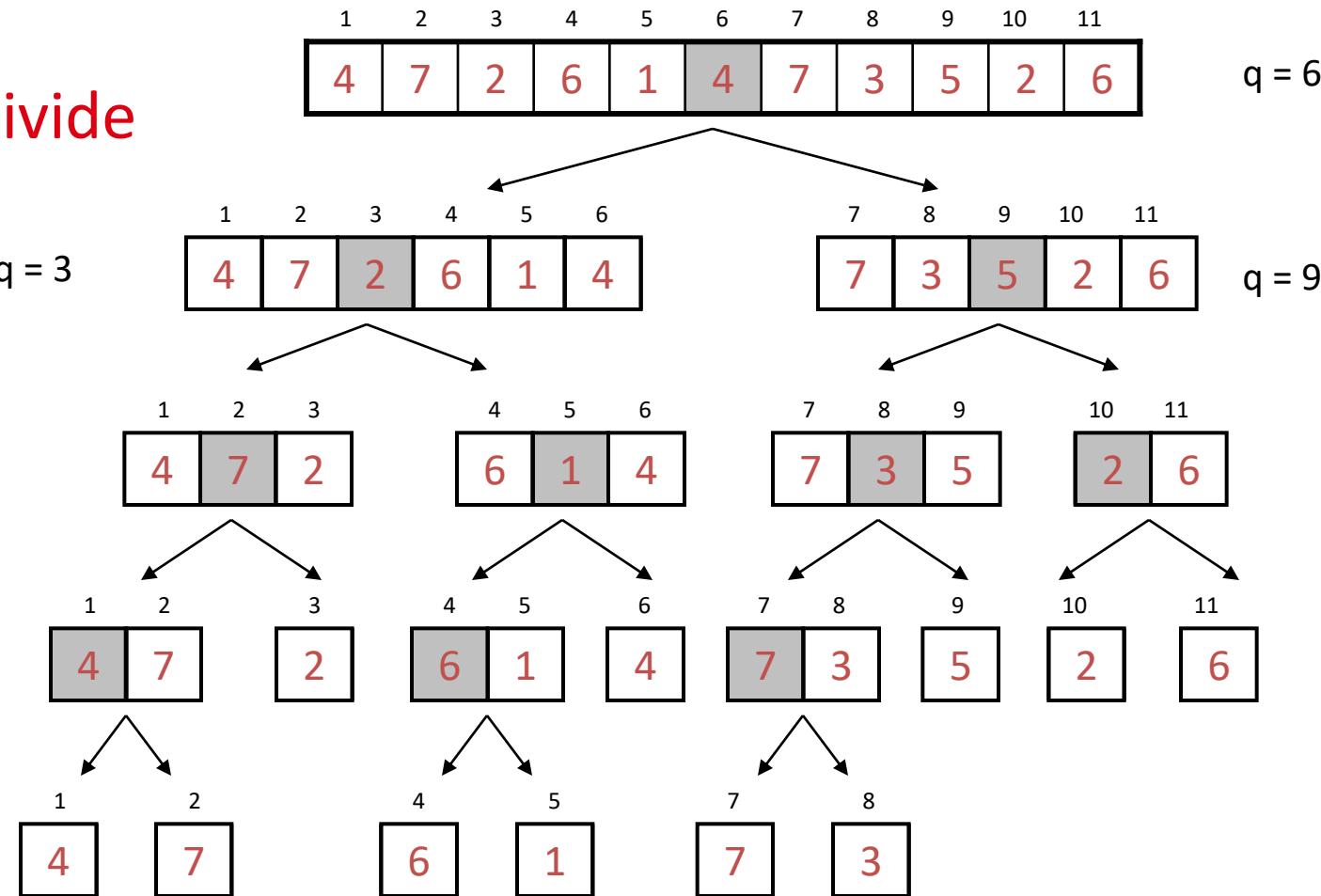
# Example – $n$ Power of 2

Conquer  
and  
Merge



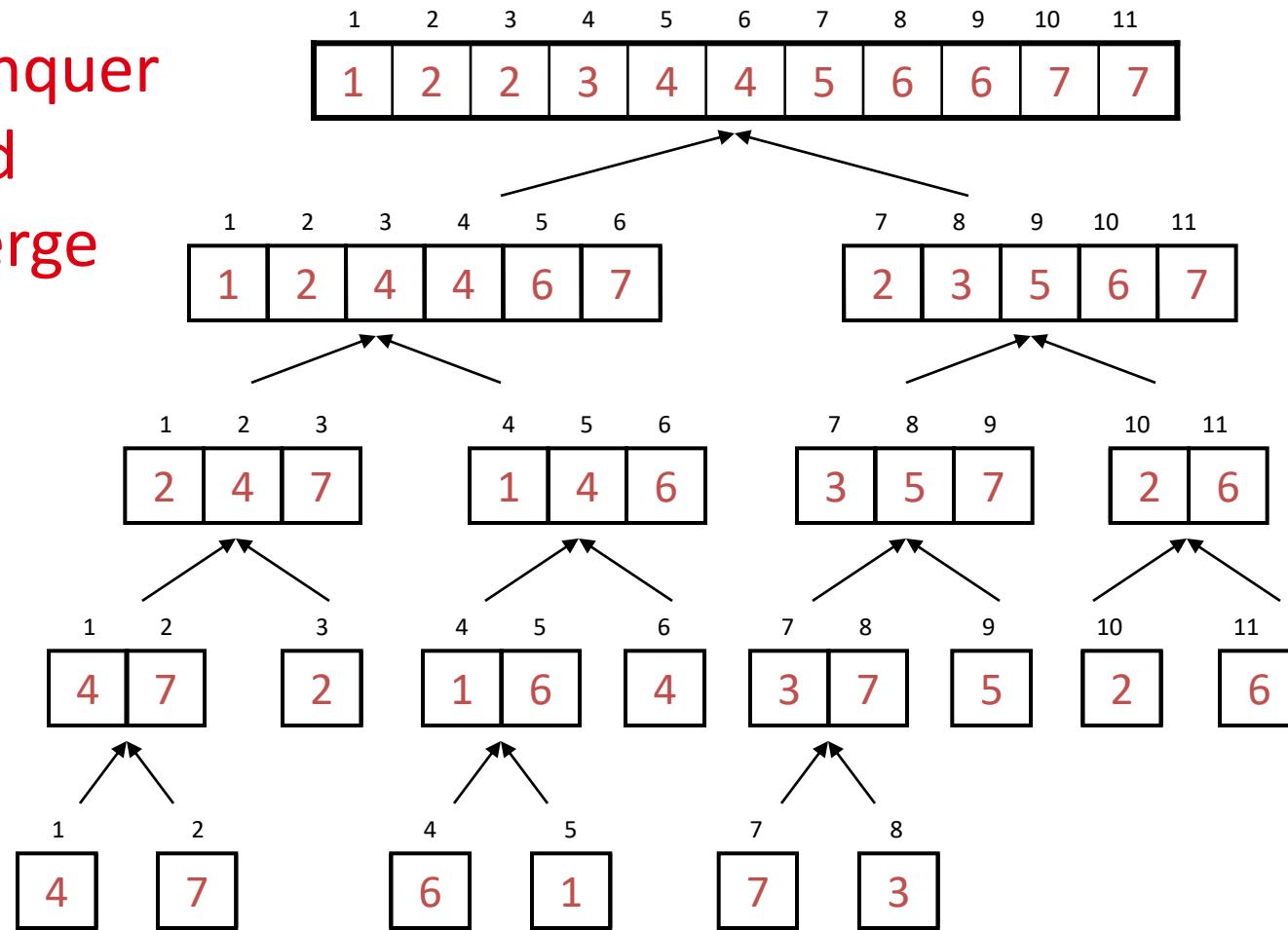
# Example – $n$ Not a Power of 2

Divide



# Example – $n$ Not a Power of 2

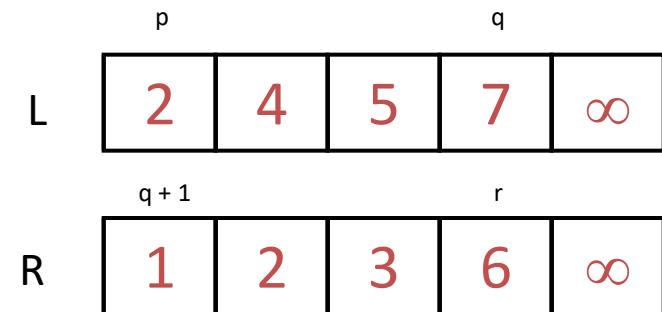
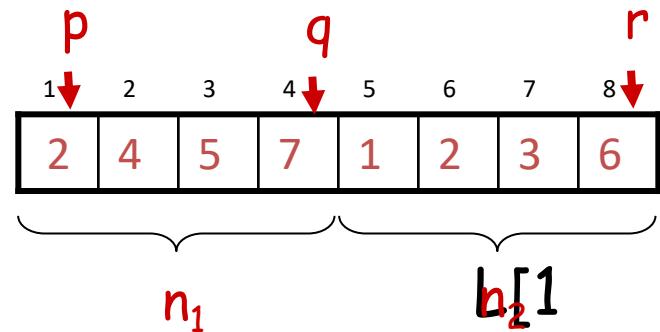
Conquer  
and  
Merge



# Merge - Pseudocode

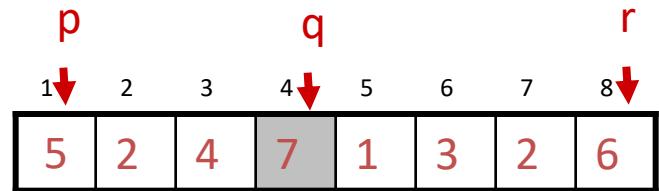
*Alg.:* MERGE( $A, p, q, r$ )

1. Compute  $n_1$  and  $n_2$
2. Copy the first  $n_1$  elements into  $L[n_1 + 1 .. n_1 + 1]$  and the next  $n_2$  elements into  $R[n_2 + 1 .. n_2 + 1]$
3.  $L[n_1 + 1] \leftarrow \infty; R[n_2 + 1] \leftarrow \infty$
4.  $i \leftarrow 1; j \leftarrow 1$
5. **for**  $k \leftarrow p$  **to**  $r$
6.   **do if**  $L[i] \leq R[j]$
7.     **then**  $A[k] \leftarrow L[i]$
8.       *i*  $\leftarrow i + 1$
9.     **else**  $A[k] \leftarrow R[j]$
10.      *j*  $\leftarrow j + 1$



# Merge Sort

*Alg.:* MERGE-SORT( $A, p, r$ )



if  $p < r$  ▷ Check for base case

then  $q \leftarrow \lfloor (p + r)/2 \rfloor$  ▷ Divide

MERGE-SORT( $A, p, q$ ) Conquer

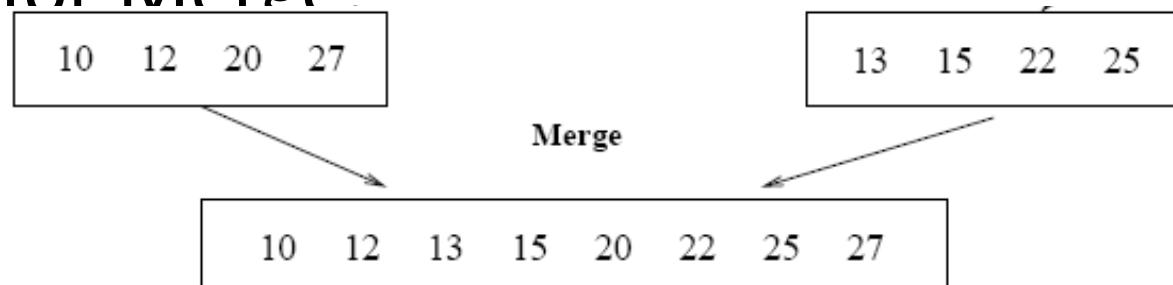
MERGE-SORT( $A, q + 1, r$ ) ▷ Conquer

MERGE( $A, p, q, r$ ) ▷ Combine

- Initial call: MERGE-SORT( $A, 1, n$ )

# Running Time of Merge (assume last **for** loop)

- Initialization (copying into temporary arrays):
  - $\Theta(n_1 + n_2) = \Theta(n)$
- Adding the elements to the final array:
  - $n$  iterations, each taking constant time  $\Rightarrow \Theta(n)$
- Total time for Merge:
  - $\Theta(n)$



# Analyzing Divide-and Conquer Algorithms

- The recurrence is based on the three steps of the paradigm:
  - $T(n)$  – running time on a problem of size  $n$
  - **Divide** the problem into  $a$  subproblems, each of size  $n/b$ : takes  $D(n)$
  - **Conquer** (solve) the subproblems  $aT(n/b)$
  - **Combine** the solutions  $C(n)$

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

# MERGE-SORT Running Time

- **Divide:**
    - compute  $q$  as the average of  $p$  and  $r$ :  $D(n) = \Theta(1)$
  - **Conquer:**
    - recursively solve 2 subproblems, each of size  $n/2 \Rightarrow 2T(n/2)$
  - **Combine:**
    - MERGE on an  $n$ -element subarray takes  $\Theta(n)$  time  
 $\Rightarrow C(n) = \Theta(n)$
- $$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

# Solve the Recurrence

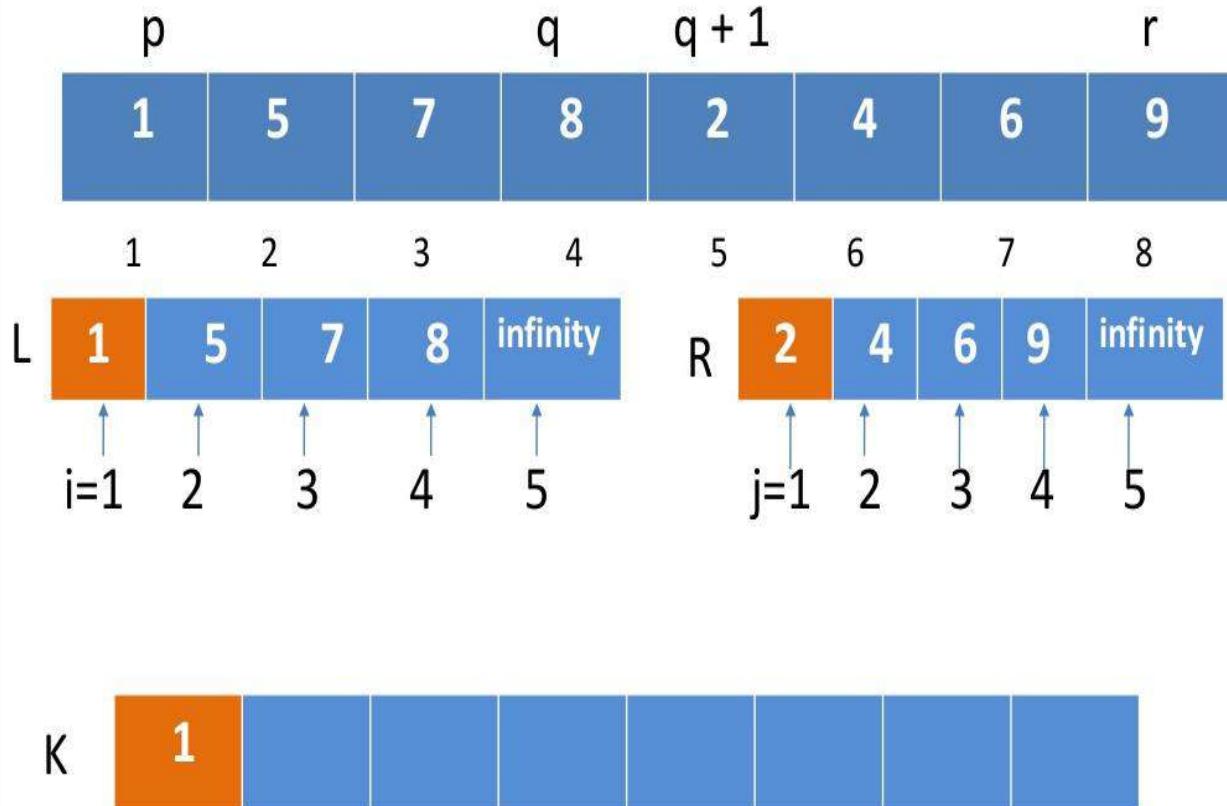
$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(n/2) + cn & \text{if } n > 1 \end{cases}$$

Use Master's Theorem:

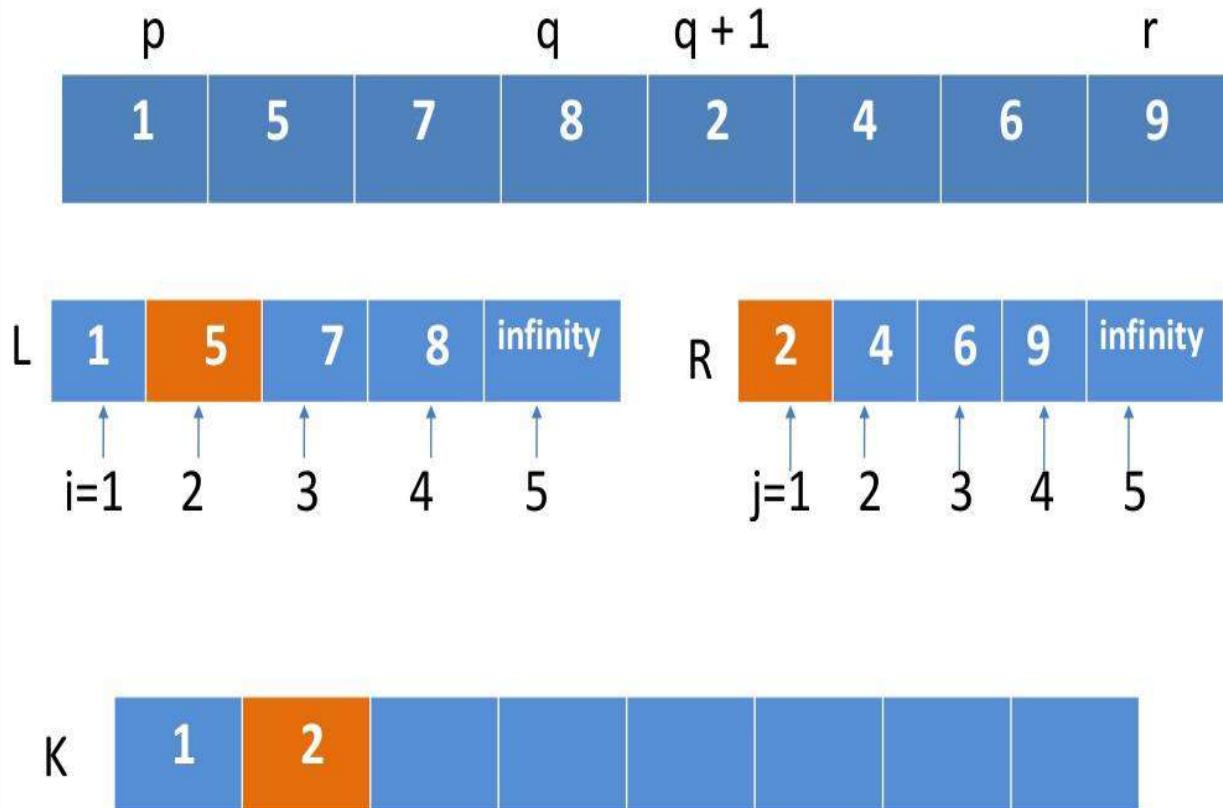
Compare  $n$  with  $f(n) = cn$

Case 2:  $T(n) = \Theta(n \lg n)$

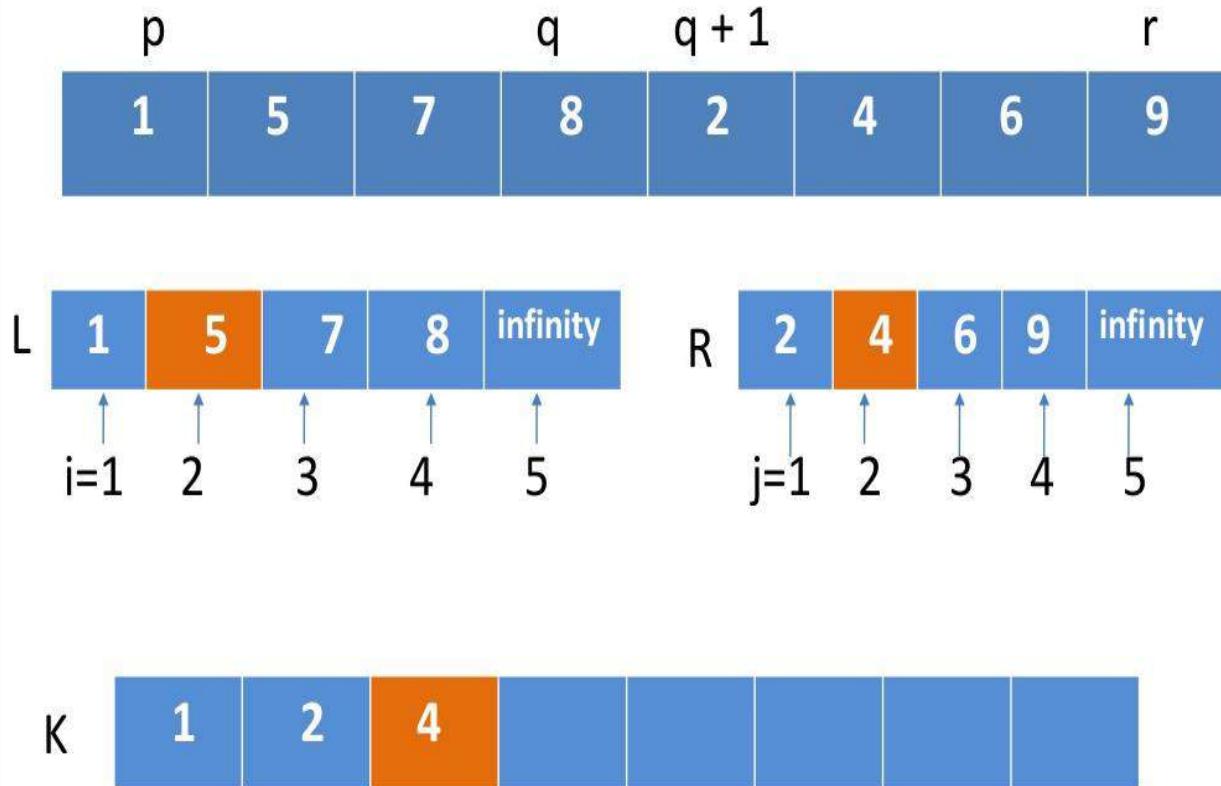
## MERGE SORT EXAMPLE :



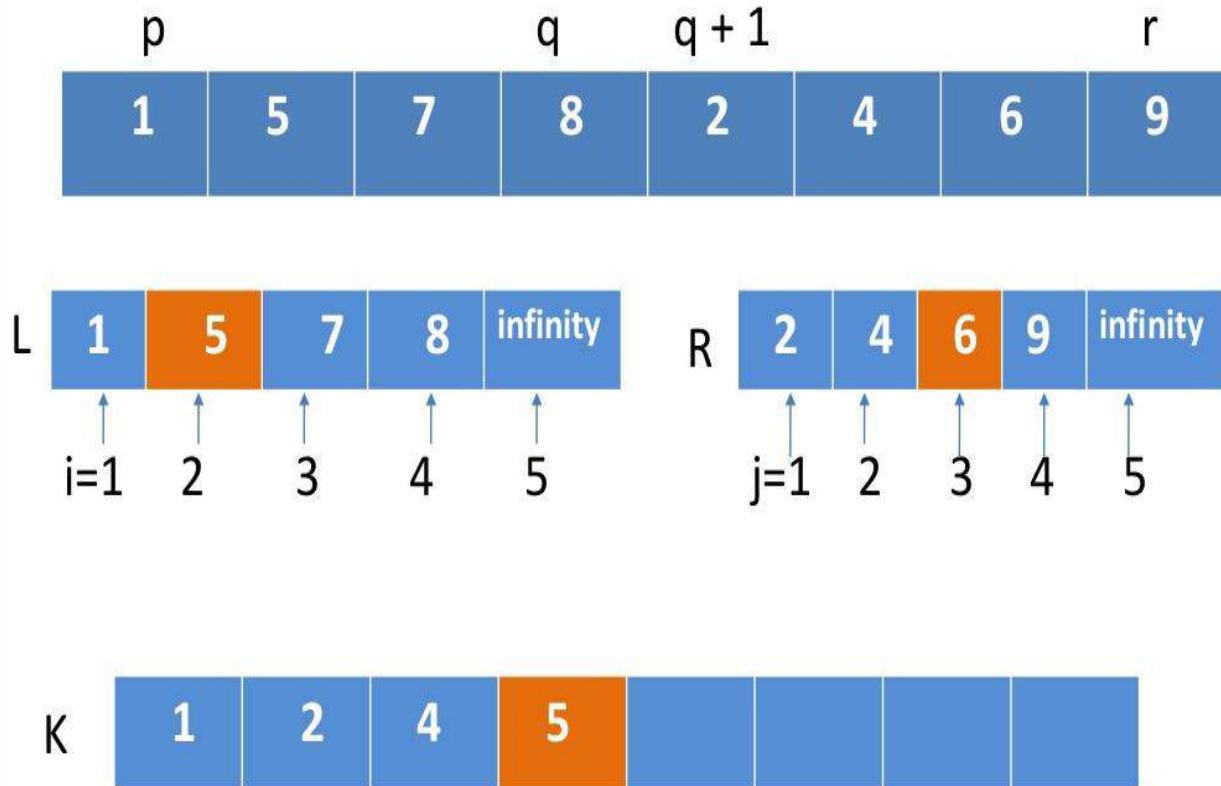
## MERGE SORT EXAMPLE :



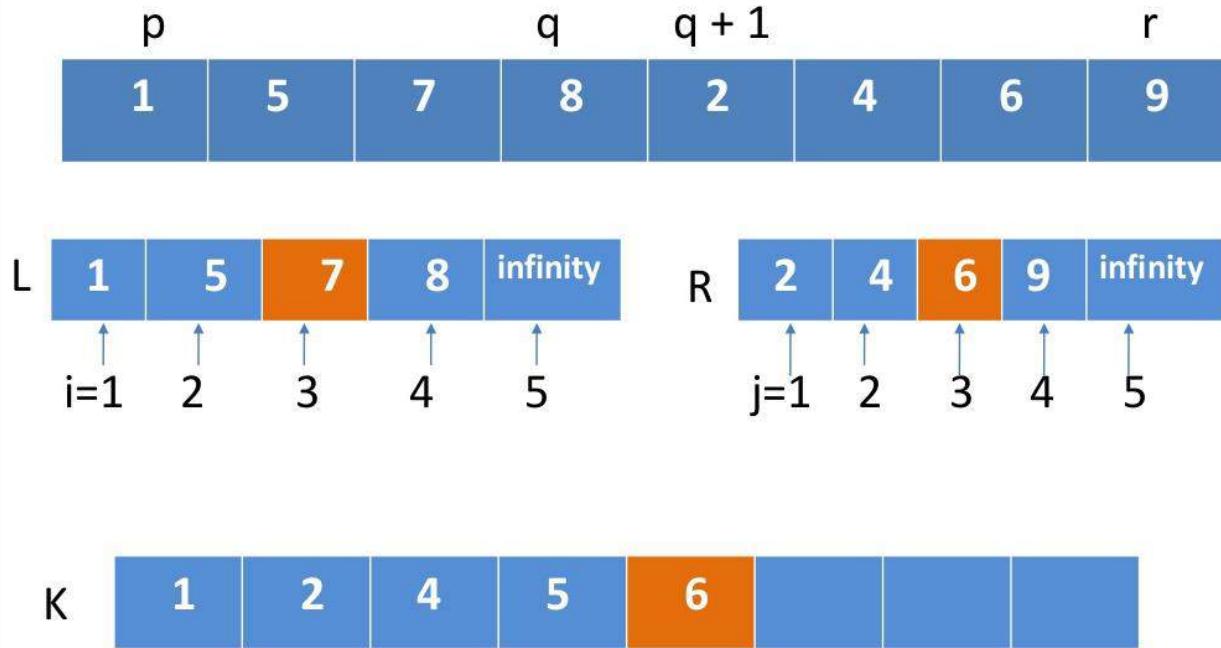
## MERGE SORT EXAMPLE :



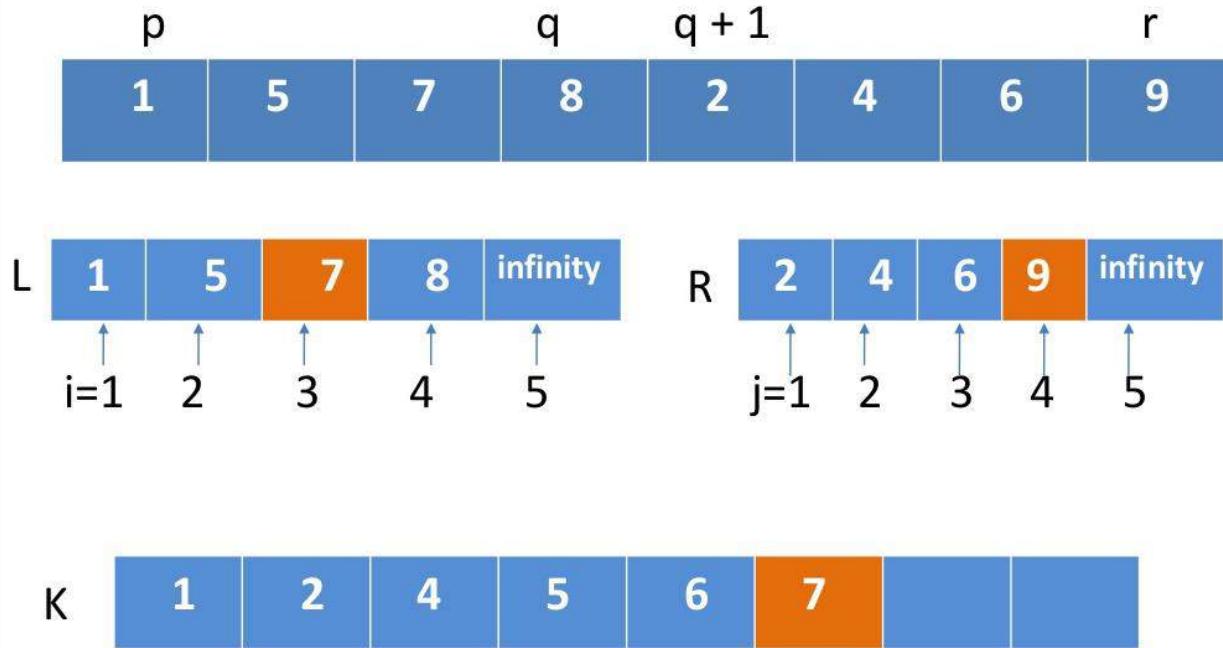
## MERGE SORT EXAMPLE :



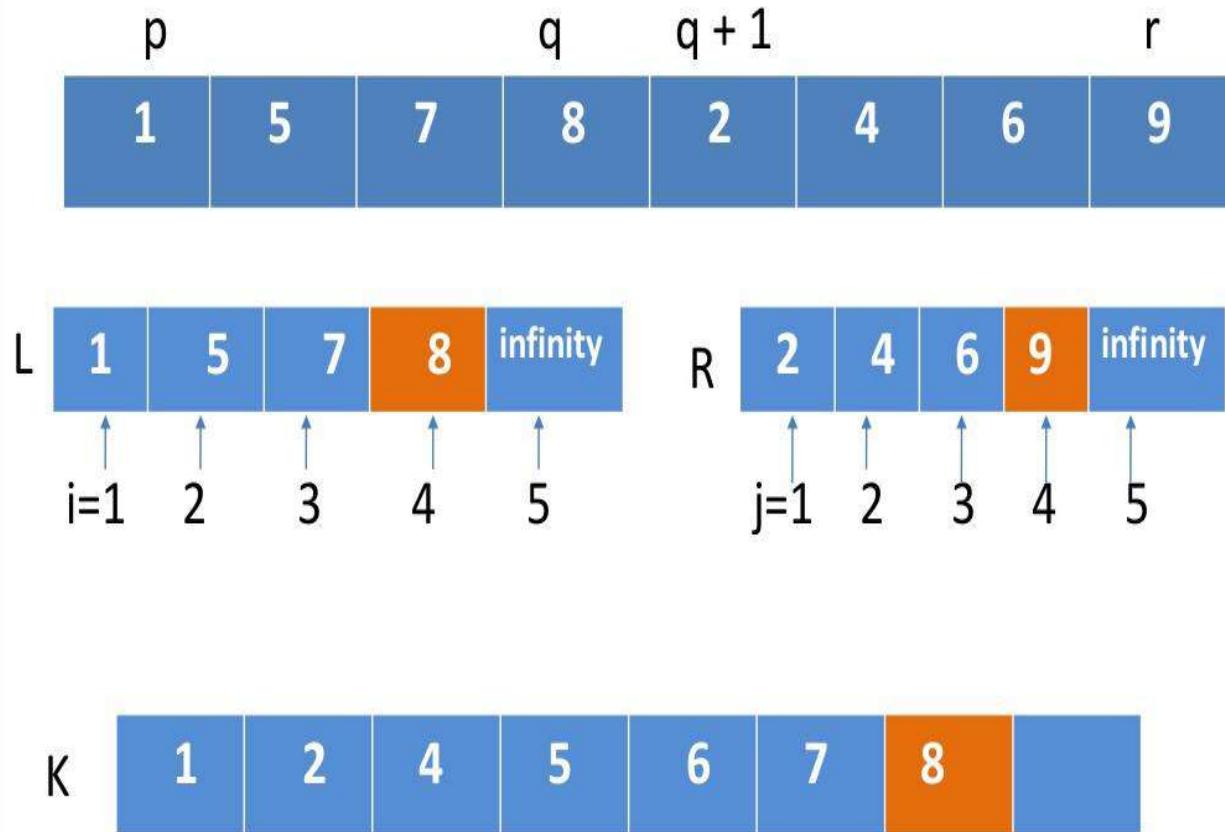
## MERGE SORT EXAMPLE :



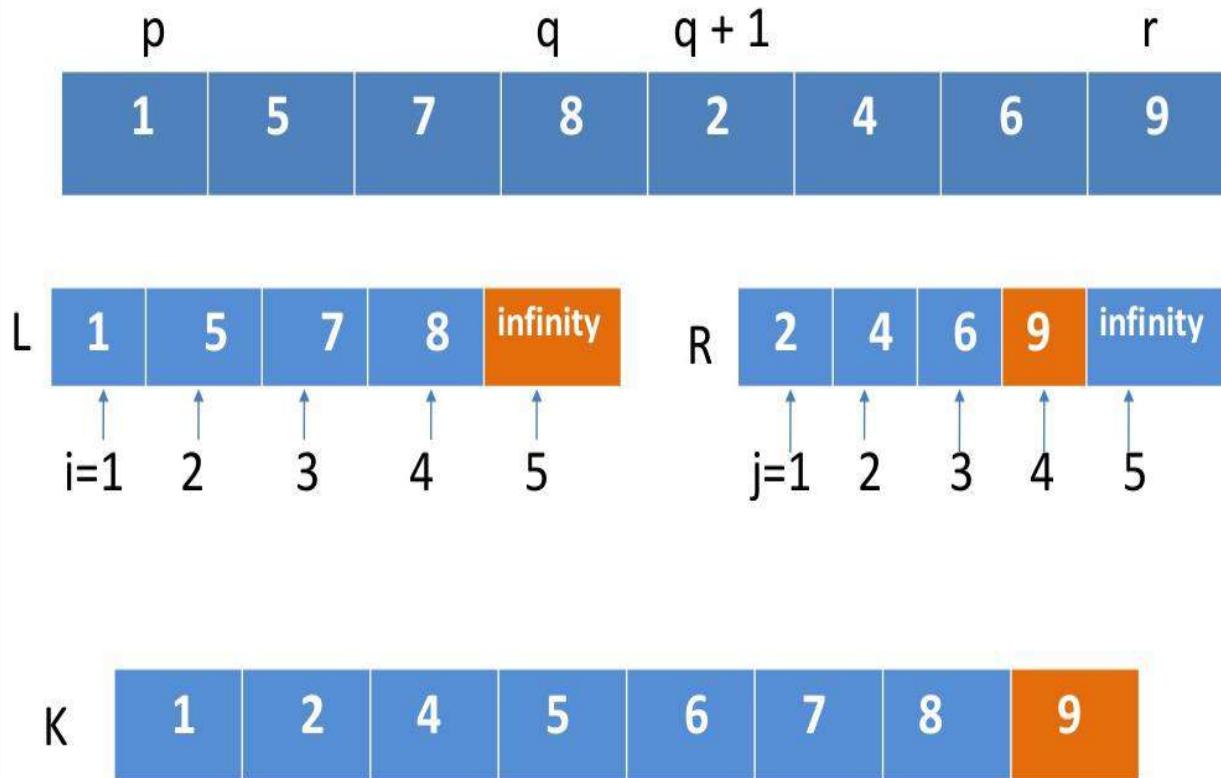
## MERGE SORT EXAMPLE :



## MERGE SORT EXAMPLE :



## MERGE SORT EXAMPLE :



4	3	1	9	8	2	4	7
↑ 1	↑ 2	3	4	5	6	7	8 ↑ high

$$A[low] < A[pivot] \Rightarrow low = low + 1 = 2 + 1 = 3$$

4	3	1	9	8	2	4	7
↑ 1	2	3 ↑ low	4	5	6	7	8 ↑ high

$$A[low] < A[pivot] \Rightarrow low = low + 1 = 3 + 1 = 4$$

4	3	1	9	8	2	4	7
↑ 1	2	3	4 ↑ low	5	6	7	8 ↑ high

$A[low] > A[pivot]$ , so fix low pointer, and move high pointer

$$A[high] > A[pivot] \Rightarrow high = high - 1 = 8 - 1 = 7$$

4	3	1	9	8	2	4	7
1	2	3	4	5	6	7	8
↑ pivot			↑ low			↑ high	

$A[high] \leq A[pivot]$ , so fix high pointer.

$low < high$ , so exchange  $A[low]$  and  $A[high]$ , and move low on right,  
 $low = low + 1 = 4 + 1 = 5$

4	3	1	4	8	2	9	7
1	2	3	4	5	6	7	8
↑ pivot			↑ low			↑ high	

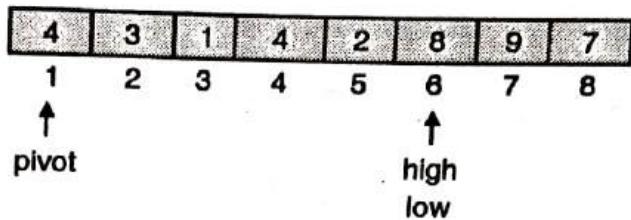
$A[low] > A[pivot]$ , so fix low pointer and move high pointer.

$$high = high - 1 = 7 - 1 = 6$$

4	3	1	4	8	2	9	7
1	2	3	4	5	6	7	8
↑ pivot				↑ low	↑ high		

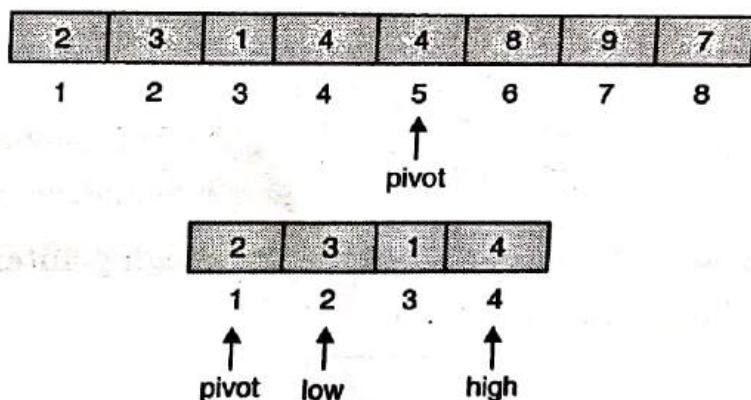
$A[high] \leq A[pivot]$ , so fix high pointer.

Here,  $\text{low} < \text{high}$ , so exchange  $A[\text{low}]$  and  $A[\text{high}]$ , forward  $\text{low}$  pointer, so,  $\text{low} = \text{low} + 1 = 5 + 1 = 6$ .



$A[low] > A[pivot]$ , so fix low pointer move high pointer,  
 $high = high - 1 = 6 - 1 = 5$

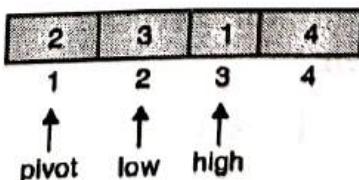
We have two sub-problems,  $p_1 = \{2, 3, 1, 4\}$ ,  $p_2 = \{8, 9, 7\}$ , both list are sorted recursively. Lets first sort sub list  $p_1$ .



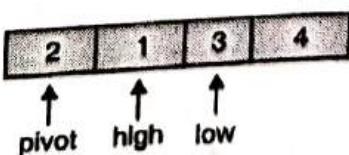
$A[low] > A[pivot]$ , so fix low pointer.

**A[high] > A [pivot], move high pointer.**

$$\text{high} = \text{high} - 1 = 4 - 1 = 3$$



$A[high] \leq A[pivot]$ , and  $low < high$ , so swap  $A[low]$  and  $A[high]$ , and forward low pointer.

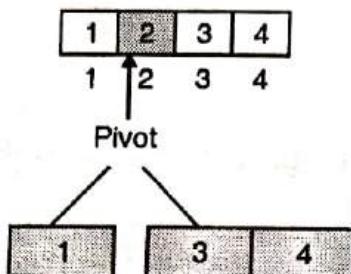


$A[\text{low}] > A[\text{pivot}]$ , fix low pointer move high pointer

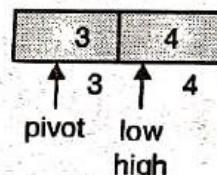
$$\text{high} = \text{high} - 1 = 3 - 1 = 2$$

$A[\text{high}] \leq A[\text{pivot}]$ , so fix high pointer.

$\text{low} > \text{high}$ , so exchange  $A[\text{pivot}]$  and  $A[\text{high}]$ . This will divide the list.

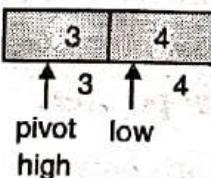


Left sub-list has only one element, so it is sorted. Right sub-list is sorted in similar way.



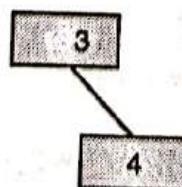
$A[\text{low}] > A[\text{pivot}]$ , so fix low pointer and move high pointer.

$$\text{high} = \text{high} - 1 = 4 - 1 = 3$$

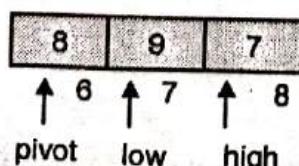


$A[\text{high}] \leq A[\text{pivot}]$ , so fix high pointer.

$\text{low} > \text{high}$ , so swap  $A[\text{pivot}]$  and  $A[\text{high}]$ , which are same in this case. This action will split the list as shown



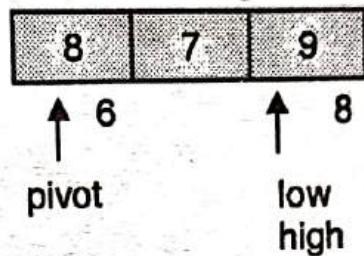
There exists only right sub-list and it has only one element. So it is sorted.  
Let us process  $p_2$ .



$A[\text{low}] > A[\text{pivot}]$ , so fix low pointer.

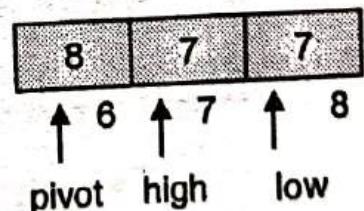
$A[\text{high}] \leq A[\text{pivot}]$ , so fix high pointer.

$\text{Low} < \text{high}$ , so swap  $A[\text{low}]$  and  $A[\text{high}]$ . Forward low pointer.



$A[\text{low}] > A[\text{pivot}]$ , so fix low pointer and move high pointer.

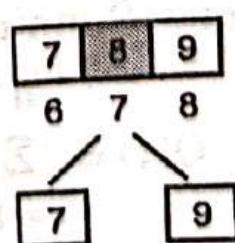
$$\text{high} = \text{high} - 1, 8 - 1 = 7$$



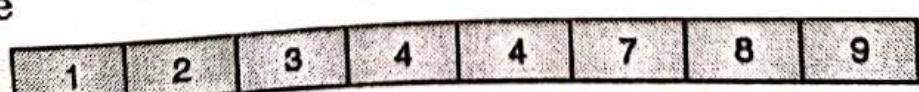
$A[\text{high}] \leq A[\text{pivot}]$ , so fix high pointer.

$\text{low} > \text{high}$ , so exchange  $A[\text{pivot}]$  and  $A[\text{high}]$ .

This will split the list.



Both, sub-list have size one, so they are already sorted. So finally array would be



# QuickSort

# Partition

```
private static int partition(A, int l, int h) {  
  
    pivot = l;    // remember pivot  
  
    int low = l + 1; // index of first unknown value  
    int high = h;   // index of last unknown value  
    while (low <= high) { // while some values still unknown  
        if (A[low] <= A[pivot])  
            low++;  
        else if (A[high] > A[pivot])  
            high--;  
        else {  
            swap(A[low], A[high]);  
            low++;  
            high--;  
        }  
    }  
    swap(A[pivot],A[high]); // put the pivot value between the two  
                           // sublists and return its index  
    return high;  
}
```

# Quicksort

```
public static void sort(A, int l, int h) {  
    if (l < h) {  
        int pivotIndex = partition(A, l, h);  
        sort(A, l, pivotIndex - 1);  
        sort(A, pivotIndex + 1, h);  
    }  
}
```

- D:\CamScanner 07-14-2020 10.54.07.pdf

## \* Time complexity

① For Best case

$$\begin{aligned}T(n) &= \frac{1}{2}T(n/2) + O(1) \\&= \Theta(n \log n)\end{aligned}$$

②

For Worst case

$$\begin{aligned}T(n) &= T(n-1) + O(1) \\&= T(n-1) + C \\&= T(n-2) + C(n-1) + C \\&= T(n-3) + C(n-2) + C(n-1) + C \\&\vdots \\&= T(1) \\&= C + C2 + C3 + \dots + Cn \\&= \Theta(n^2) \\&= \frac{n(n+1)}{2} \\&= \Theta(n^2).\end{aligned}$$

# Optimization Problems

- A problem that may have many feasible solutions.
- Each solution has a value
- In maximization problem, we wish to find a solution to maximize the value
- In the minimization problem, we wish to find a solution to minimize the value

# **Technique to Solve a Problem**

---

- Greedy Method
- Dynamic Programming
- Branch and Bound

# Greedy Algorithms

---

- Many optimization problems can be solved using a greedy approach
  - The basic principle is that local optimal decisions may be used to build an optimal solution
  - But the greedy approach may not always lead to an optimal solution overall for all problems
  - The key is knowing which problems will work with this approach and which will not
- We will study
  - **The Knapsack Problem**

# Greedy Technique

- Greedy algorithms are simple and straightforward.
- They are shortsighted in their approach in the sense that they take decisions on the basis of information at hand without worrying about the effect these decisions may have in the future.
- They are easy to invent, easy to implement and most of the time quite efficient.
- Many problems cannot be solved correctly by greedy approach.  
Greedy algorithms are used to solve optimization problems

# DP

- Set Of Coins,  $D = \{1,5,6,9\}$
- Make Change Of Taka,  $n = 11$

- Result :  
Minimum Number of coin : 2.  
The coins : 5, 6.

- Set Of Coins,  $D = \{1,5,8,10\}$
- Make Change Of Taka,  $n = 15$

- Result :  
Minimum Number of coin : 2.  
The coins : 10, 5.

# GREEDY

- Set Of Coins,  $D = \{9,6,5,1\}$
- Make Change Of Taka,  $n = 11$

- Result :  
Minimum Number of coin : 3.  
The coins : 9(1), 1(2).

- Set Of Coins,  $D = \{10,8,5,1\}$
- Make Change Of Taka,  $n = 15$

- Result :  
Minimum Number of coin : 2.  
The coins : 10(1), 5(1).

## Definition:



**Knapsack problem states that:** Given a set of items, each with a weight and a profit, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total profit is as large as possible.

## **Version of knapsack:**

□ **There are two versions of knapsack problem:**

1. **0/1 Knapsack Problem:** Items are indivisible; you either take them or not. And it is solved using Dynamic Programming(DP).

2. **Fractional Knapsack Problem:** Items are divisible; you can take any fraction of an item. And it is solved using Greedy Algorithm.



# Explanation:



**Camera**  
Weight: 2kg  
Value: 1000tk



**Laptop**  
Weight: 3kg  
Value: 20000tk



**Necklace**  
Weight: 1kg  
Value: 3000tk



**Knapsack**  
Weight: 7kg  
????



**Vase**  
Weight: 4kg  
Value: 4000tk

## \* Greedy technique

① Knapsack ; capacity( $M$ ) = 20

	$J_1$	$J_2$	$J_3$
Profit	25	24	15
Weight	18	15	10

\* algorithm

- for  $i = 1$  to  $n$  calculate  $P/W$

- Sort item in decreasing order of  $P/W$  ratio.

- for  $i = 1$  to  $n$

if  $M > 0$  &  $w_i \leq M$

$$M = M - w_i$$

$$P = P + p_i ;$$

else

break ;

if  $M > 0$

$$P = P + p_i \left( \frac{M}{w_i} \right) ;$$

break ;

$$\Theta(n) + \Theta(\log n) + \Theta(n)$$

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

① money about profit

$$= 25 + \frac{2}{15} \times 24 = 28.2$$

$\frac{2}{15}$	$y_{18}$
----------------	----------

② money about weight

$$15 + \frac{10}{15} \times 24$$

$$= 31$$

$\frac{10}{15}$	$y_{10}$
obj3	$y_{10}$

③ obj 1 =  $\frac{25}{18} = 1.3$   
 obj 2 =  $\frac{24}{15} = 1.6$

$$\text{obj3} = \frac{15}{10} = 1.5$$

[money about both]

decrease added

$$\text{total profit} - 24 + 7.5$$

$$- 31.5$$

$\frac{5}{10}$	$y_{15}$
obj2	$y_{15}$

## Example of fractional knapsack:

Example: Number of items n=3, Capacity m=45

&

the weights ~~fits are given below:~~

Weight ( $w_i$ )	40	60	50
Profit ( $p_i$ )	40	60	50

Now find out the fraction of chosen items with maximum p

Greedy approach to solve  
Fractional knapsack problem :

- Find the unit  $u_i$  using the formula  $u_i = p_i/w_i$ .
- Find the fraction of the items  $x_i$  that will be taken in order to get maximum profit.

## SOLUTION:

**Step 1:** Find the unit  $u_i$ .

**Step 2:** Sort the item in descending order of  $u_i$ .

**Step 3:** Find the maximum profit & the fraction  $x_i$  of the items.

$w_i$	20	20	20
$p_i$	60	50	40
$u_i = p_i/w_i$	3	2.5	2
$x_i$	1	1	1/4

Therefore, maximum profit =  $(60 \times 1) + (50 \times 1) + 40 \times 1/4 = 120$ .

Ans: Maximum Profit = 120.

Taken fraction of the item:

Weight	20	20	20
Profit	60	50	40
Fraction	1	1	1/4

## *Application of knapsack:*

**Knapsack problems appear in real world decision making processes in a wide variety of fields, such as finding the least wasteful way to cut raw materials , selection of investment and selection of assets.**

## Huffman Coding-

- Huffman Coding is a famous Greedy Algorithm.
- It is used for the lossless compression of data.
- It uses variable length encoding.
- It assigns variable length code to all the characters.
- The code length of a character depends on how frequently it occurs in the given text.
- The character which occurs most frequently gets the smallest code.
- The character which occurs least frequently gets the largest code.
- It is also known as **Huffman Encoding**.

- **Major Steps in Huffman Coding-**
- 
- There are two major steps in Huffman Coding-
- Building a Huffman Tree from the input characters.
- Assigning code to the characters by traversing the Huffman Tree.

- **Huffman Tree-**
- 
- The steps involved in the construction of Huffman Tree are as follows-
- 
- **Step-01:**
- 
- Create a leaf node for each character of the text.
- Leaf node of a character contains the occurring frequency of that character.
- 
- **Step-02:**
- 
- Arrange all the nodes in increasing order of their frequency value.

- **Step-03:**
  - Considering the first two nodes having minimum frequency,
  - Create a new internal node.
  - The frequency of this new node is the sum of frequency of those two nodes.
  - Make the first node as a left child and the other node as a right child of the newly created node.
  -
- **Step-04:**
  - Keep repeating Step-02 and Step-03 until all the nodes form a single tree.
  - The tree finally obtained is the desired Huffman Tree.

- **Time Complexity-**
- 
- The time complexity analysis of Huffman Coding is as follows-
- `extractMin( )` is called  $2 \times (n-1)$  times if there are  $n$  nodes.
- As `extractMin( )` calls `minHeapify( )`, it takes  $O(n \log n)$  time.
- 
- Thus, Overall time complexity of Huffman Coding becomes  **$O(n \log n)$** .
- Here,  $n$  is the number of unique characters in the given text.

# Formula1

---

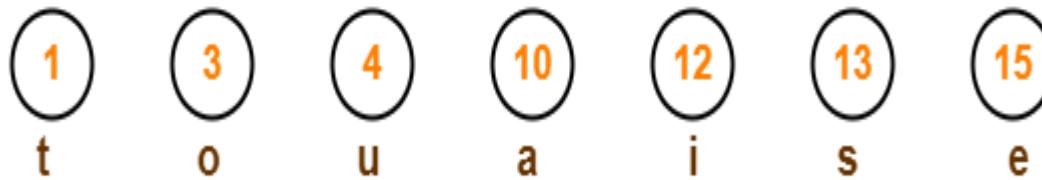
$$\text{Average code length per character} = \frac{\sum (\text{frequency}_i \times \text{code length}_i)}{\sum \text{frequency}_i}$$
$$= \sum (\text{probability}_i \times \text{code length}_i)$$

- **Formula-02:**
- 
- Total number of bits in Huffman encoded message
- = Total number of characters in the message x Average code length per character
- =  $\sum ( \text{frequency}_i \times \text{Code length}_i )$

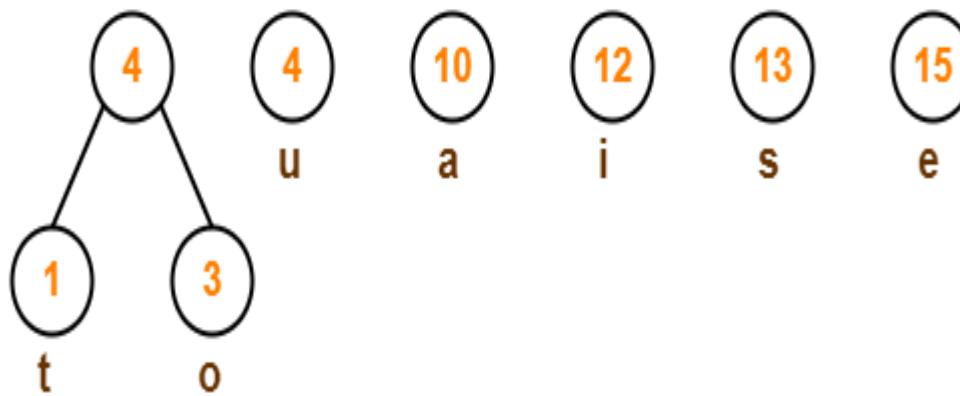
- **PRACTICE PROBLEM BASED ON HUFFMAN CODING-**
- 
- **Problem-**
- 
- A file contains the following characters with the frequencies as shown. If Huffman Coding is used for data compression, determine-
  - Huffman Code for each character
  - Average code length
  - Length of Huffman encoded message (in bits)

Characters	Frequencies
a	10
e	15
i	12
o	3
u	4
s	13
t	1

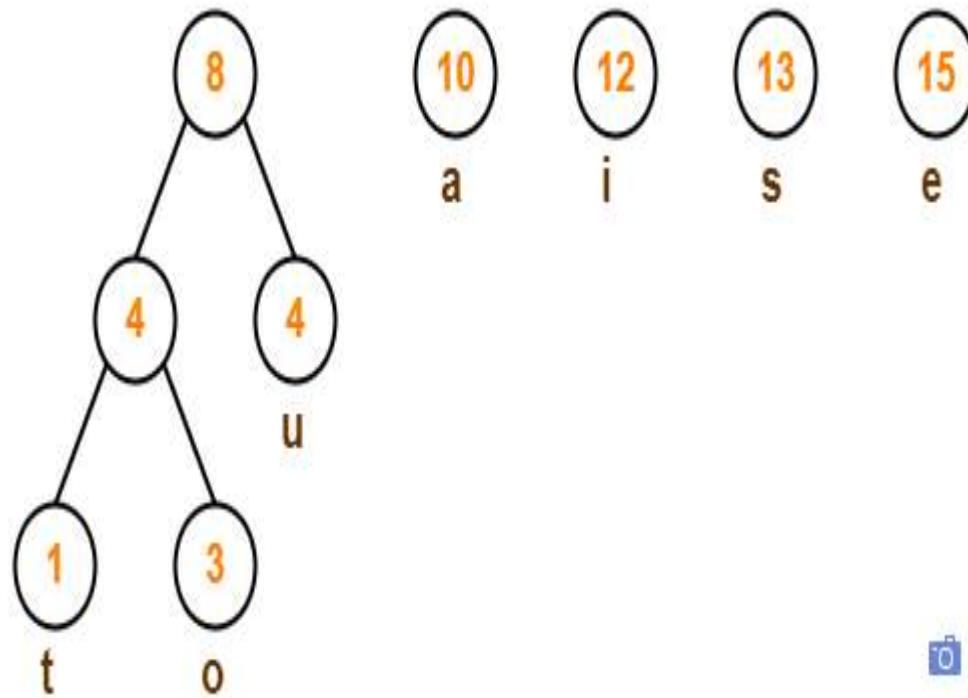
Step-01:



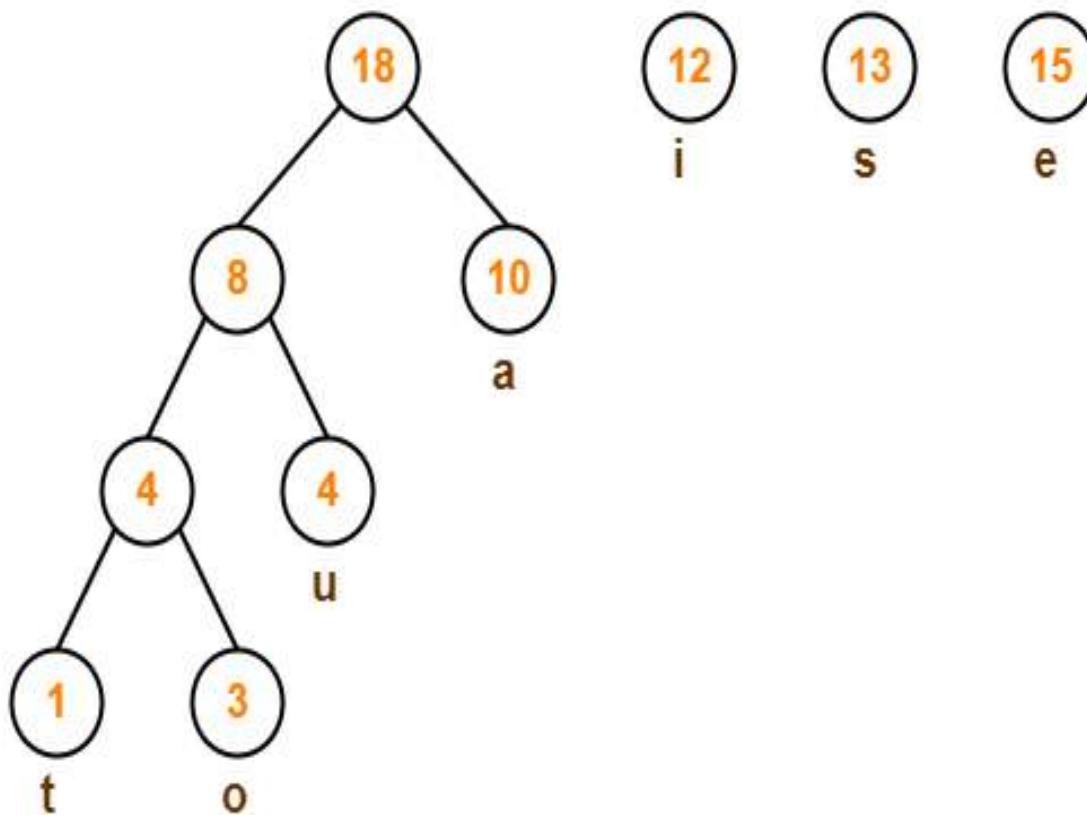
Step-02:



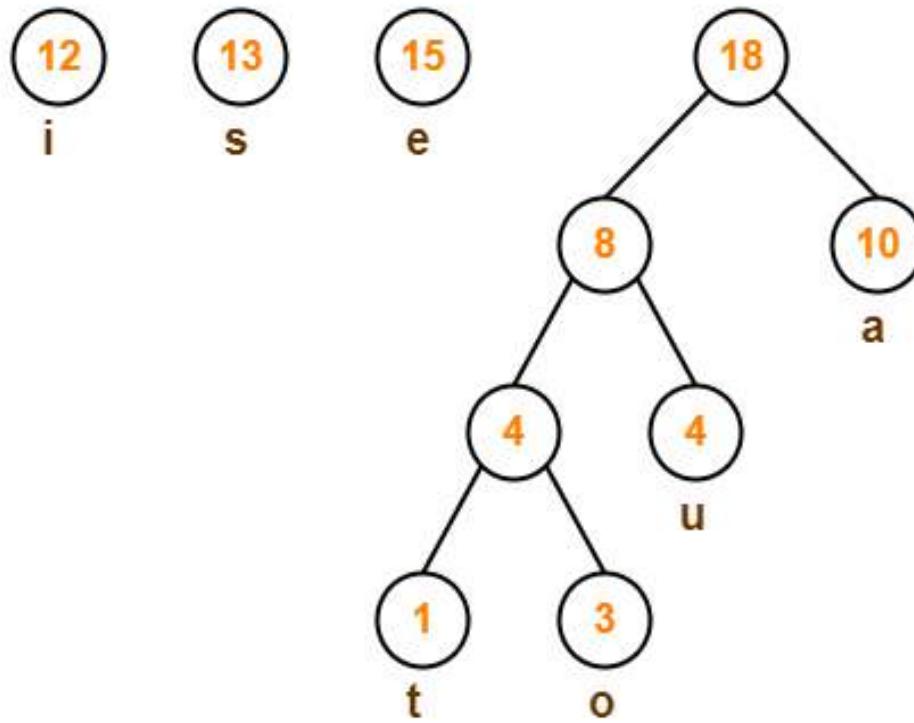
### Step-03:



Step-04:

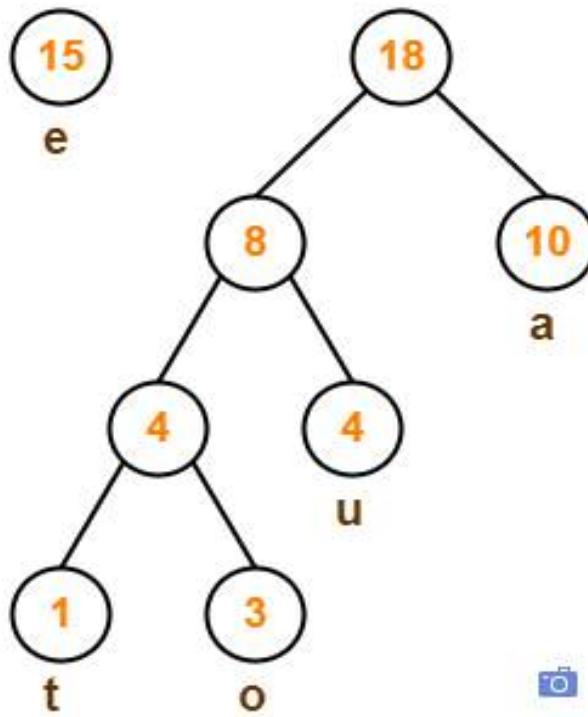
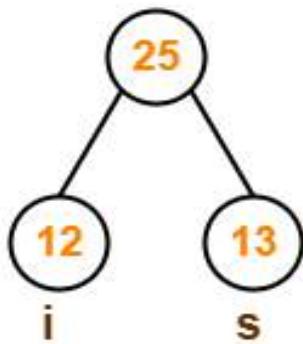


Step-05:

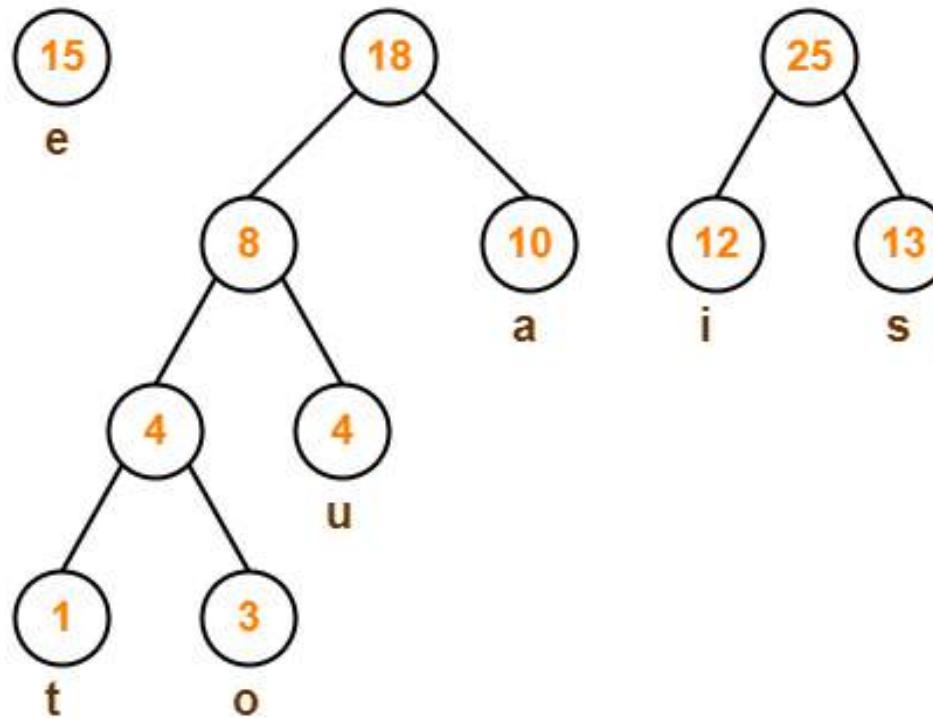


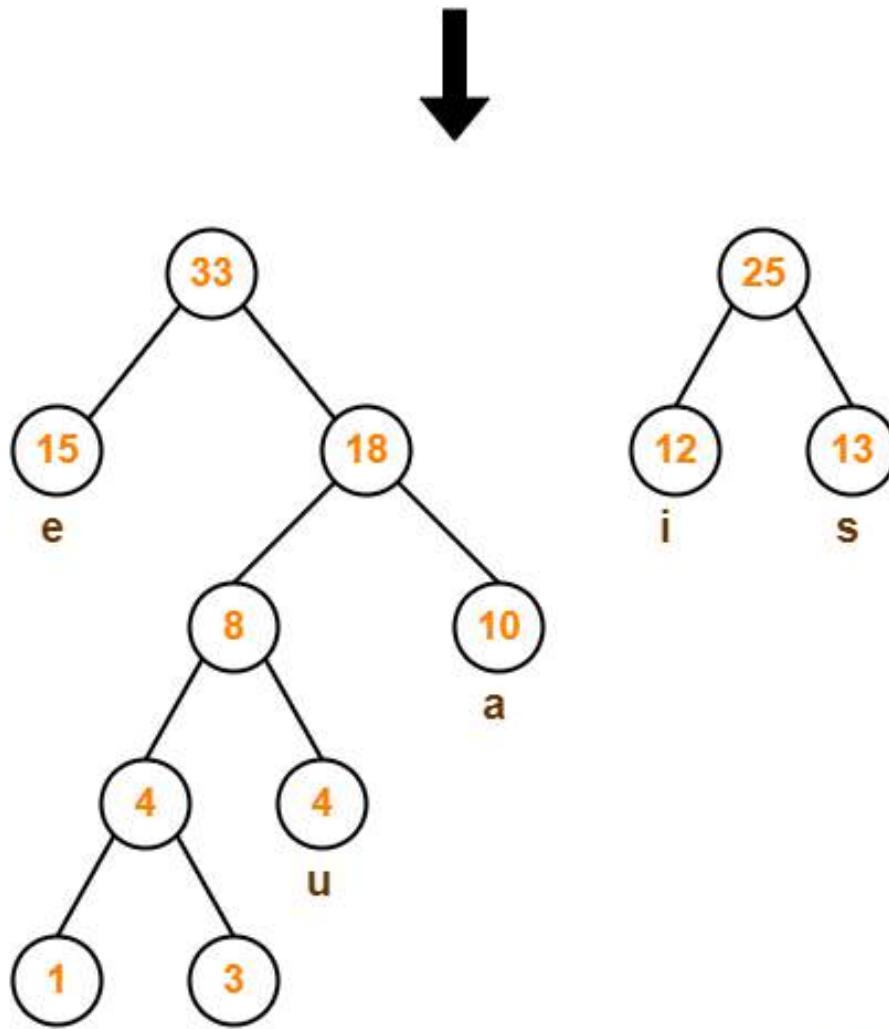
t

o

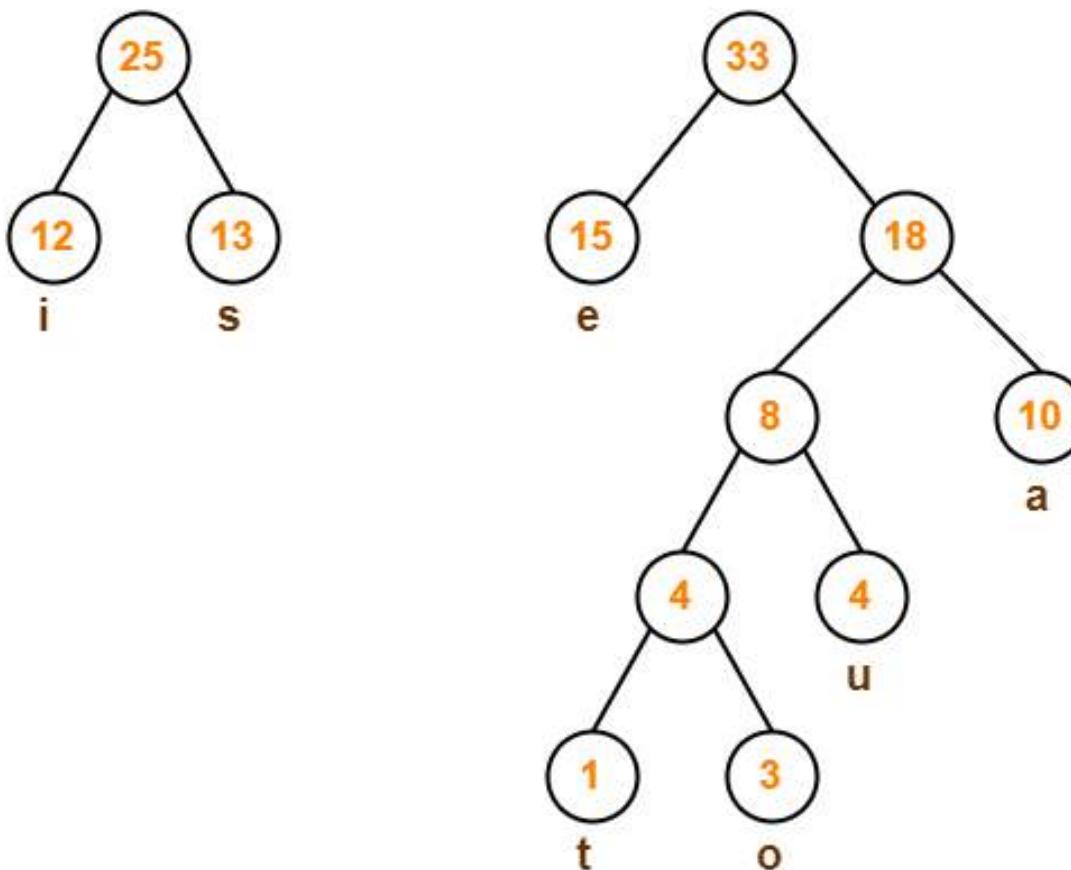


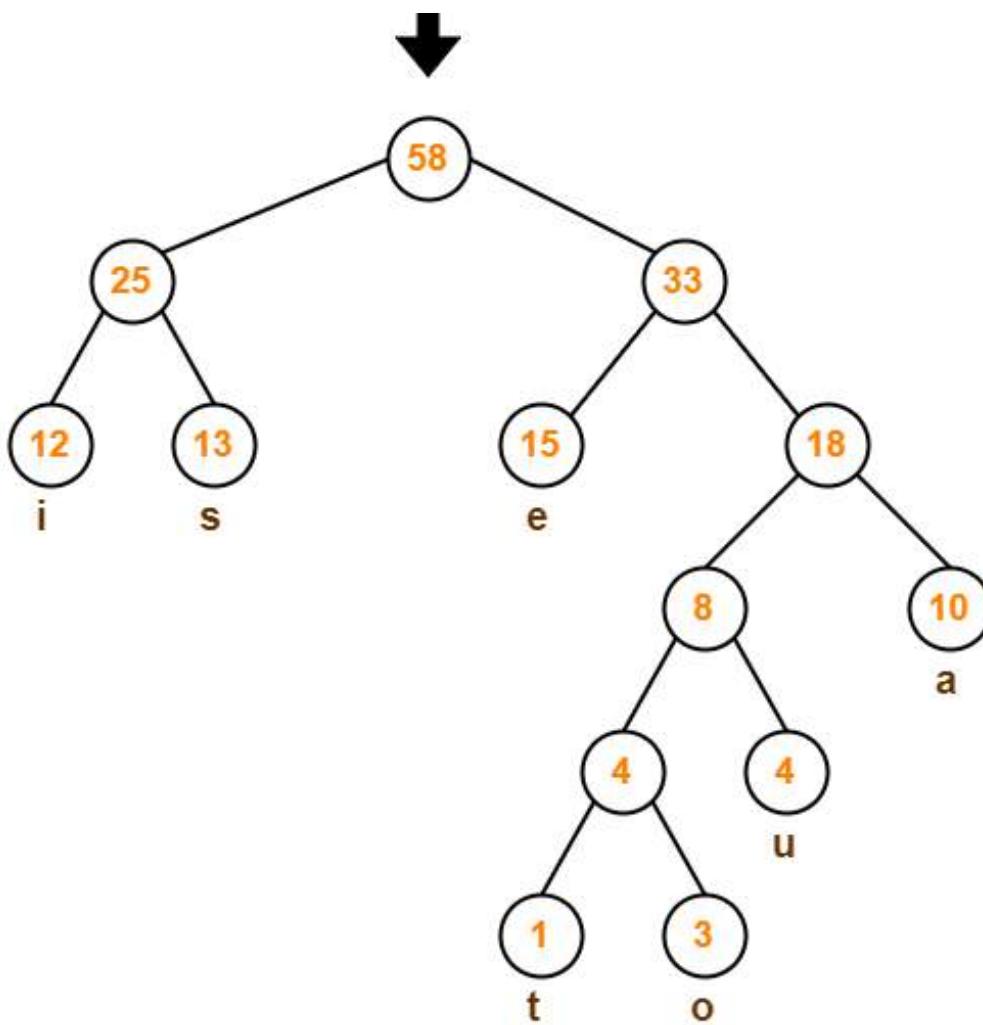
Step-06:



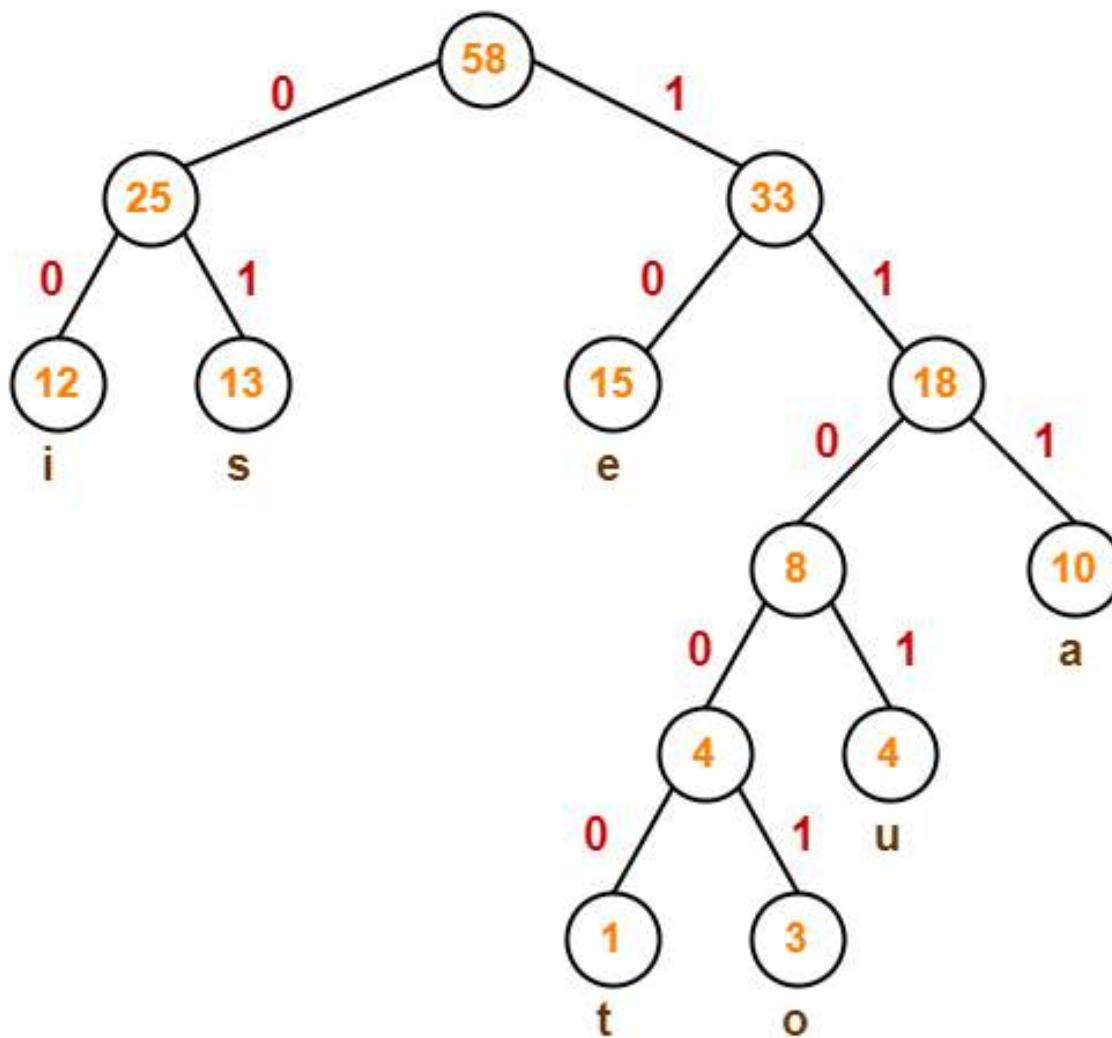


Step-07:





After assigning weight to all the edges, the modified Huffman Tree is-



## 1. Huffman Code For Characters-

To write Huffman Code for any character, traverse the Huffman Tree from root node to the leaf node of that character.

Following this rule, the Huffman Code for each character is-

- a = 111
- e = 10
- i = 00
- o = 11001
- u = 1101
- s = 01
- t = 11000

From here, we can observe-

- Characters occurring less frequently in the text are assigned the larger code.
- Characters occurring more frequently in the text are assigned the smaller code.

## 2. Average Code Length-

Using formula-01, we have-

Average code length

$$= \sum (\text{frequency}_i \times \text{code length}_i) / \sum (\text{frequency}_i)$$

$$= \{ (10 \times 3) + (15 \times 2) + (12 \times 2) + (3 \times 5) + (4 \times 4) + (13 \times 2) + (1 \times 5) \} / (10 + 15 + 12 + 3 + 4 + 13 + 1)$$

$$= 2.52$$

### 3. Length of Huffman Encoded Message-

Using formula-02, we have-

Total number of bits in Huffman encoded message

= Total number of characters in the message x Average code length per character

=  $58 \times 2.52$

= 146.16

$\approx 147$  bits

# Job Sequencing With Deadlines-

Here-

You are given a set of jobs.

Each job has a defined deadline and some profit associated with it.

The profit of a job is given only when that job is completed within its deadline.

Only one processor is available for processing all the jobs.

Processor takes one unit of time to complete a job.

## Greedy Algorithm-

Greedy Algorithm is adopted to determine how the next job is selected for an optimal solution.

The greedy algorithm described below always gives an optimal solution to the job sequencing problem-

### **Step-01:**

Sort all the given jobs in decreasing order of their profit.

### **Step-02:**

Check the value of maximum deadline.

Draw a Gantt chart where maximum time on Gantt chart is the value of maximum deadline.

### **Step-03:**

Pick up the jobs one by one.

Put the job on Gantt chart as far as possible from 0 ensuring that the job gets completed before its deadline.

# problem

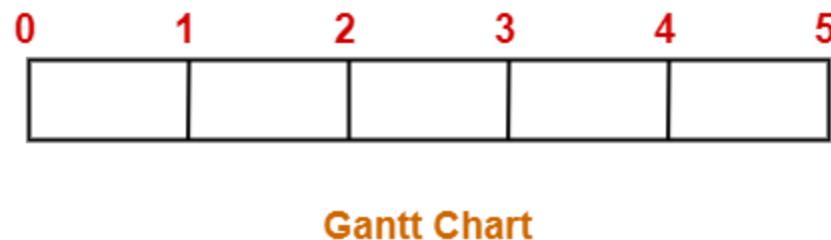
Jobs	J1	J2	J3	J4	J5	J6
Deadlines	5	3	3	2	4	2
Profits	200	180	190	300	120	100

Sort all the given jobs in decreasing order of their profit-

Jobs	J4	J1	J3	J2	J5	J6
Deadlines	2	5	3	3	4	2
Profits	300	200	190	180	120	100

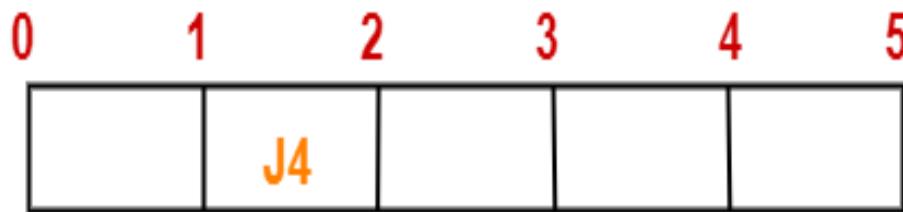
Value of maximum deadline = 5.

So, draw a Gantt chart with maximum time on Gantt chart = 5 units as shown-



We take job J4.

Since its deadline is 2, so we place it in the first empty cell before deadline 2 as-



We take job J1.

Since its deadline is 5, so we place it in the first empty cell before deadline 5 as-



We take job J3.

Since its deadline is 3, so we place it in the first empty cell before deadline 3 as-



We take job J2.

Since its deadline is 3, so we place it in the first empty cell before deadline 3.

Since the second and third cells are already filled, so we place job J2 in the first cell as-



Now, we take job J5.

Since its deadline is 4, so we place it in the first empty cell before deadline 4 as-



- Now,
- The only job left is job J6 whose deadline is 2.
- All the slots before deadline 2 are already occupied.
- Thus, job J6 can not be completed.

- The optimal schedule is-
- **J2 , J4 , J3 , J5 , J1**
- This is the required order in which the jobs must be completed in order to obtain the maximum profit.

- Maximum earned profit
- = Sum of profit of all the jobs in optimal schedule
- = Profit of job J2 + Profit of job J4 + Profit of job J3 + Profit of job J5 + Profit of job J1
- =  $180 + 300 + 190 + 120 + 200$
- = 990 units

```
for i = 1 to n do
    Set k = min(dmax, DEADLINE(i)) //where DEADLINE(i) denotes deadline of

    while k >= 1 do
        if timeslot[k] is EMPTY then
            timeslot[k] = job(i)
            break
        endif

        Set k = k - 1
    endwhile

endfor
```

- On Average , $N$  jobs search  $N/2$  slots.
- This would take  $O(N^2)$  time.

# Optimal merge pattern

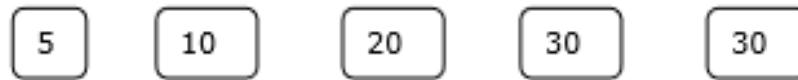
# Introduction:

- Merge two files each has n & m elements, respectively:
- takes  $O(n+m)$ .
- Given n files What's the minimum time needed to merge all n files? .
- Example:  $(F_1, F_2, F_3, F_4, F_5) = (20, 30, 10, 5, 30)$ .
- $M_1 = F_1 \& F_2 = 20+30 = 50$
- $M_2 = M_1 \& F_3 = 50+10 = 60$
- $M_3 = M_2 \& F_4 = 60+5 = 65$
- $M_4 = M_3 \& F_5 = 65+30 = 95$

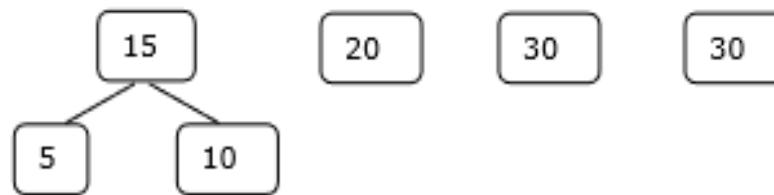
- Optimal merge pattern: Greedy method.
- Sort the list of files: (5,10, 20, 30, 30)= (F4, F3, F1, F2, F5)

- Merge the first two files:
- $(5, 10, 20, 30, 30) = (15, 20, 30, 30)$
- Merge the next two files:
- $(15, 20, 30, 30) = (30, 30, 35)$
- Merge the next two files:
- $(30, 30, 35) = (35, 60)$
- Merge the last two files:
- $(35, 60) = (95)$
- Total time:  $15 + 35 + 60 + 95 = 205$
- This is called a 2-way merge pattern

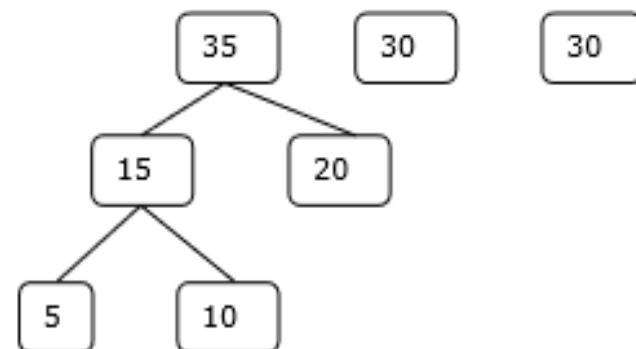
Initial Set



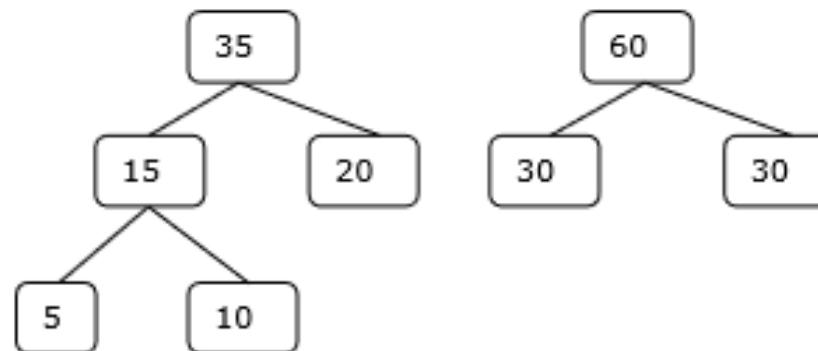
Step-1



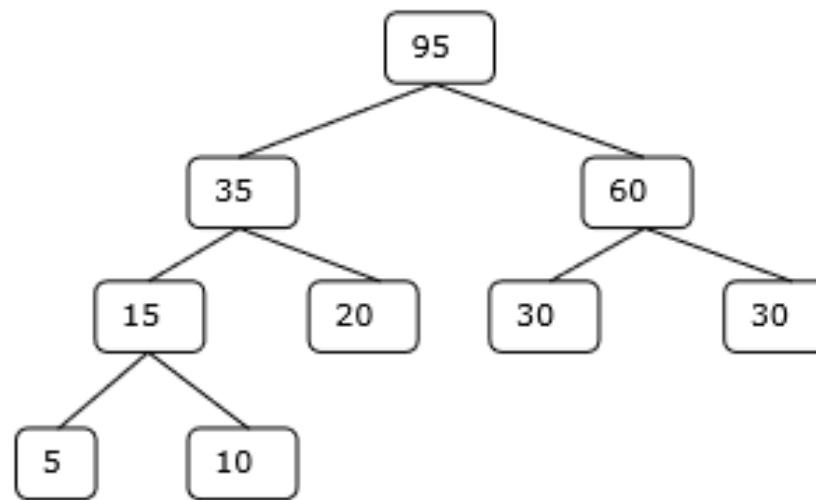
Step-2



Step-3



Step-4



Hence, the solution takes  $15 + 35 + 60 + 95 = 205$  number of comparisons.

- **Algorithm:**

- Least (L): find a tree in L whose root has the smallest weight.
- Function : Tree (L,n).  
    Integer i;  
    Begin  
        For i=1 to n -1 do  
            Get node (T)     /\* create a node pointed by T \*/  
            Left child (T)= Least (L)     /\* first smallest \*/  
            Right child (T)= Least (L)   /\* second smallest \*/  
            Weight (T) = weight (left child (T))  
                     + weight (right child (T))  
            Insert (L,T);   /\* insert new tree with root T in L \*/  
        End for  
        Return (Least (L)) /\* tree left in L \*/  
    End.

# Time Complexity

L is represented as a min-heap. Value in the root is  $\leq$  the values of its children.

$$O(\text{Least}) = O(1)$$

$$O(\text{Insert}) = O(\log n)$$

$$\Rightarrow T = O(n \log n).$$

# Optimal Tap Storage

Given programs stored on a computer tape and length of each program is where , find the order in which the programs should be stored in the tape for which the Mean Retrieval Time (MRT given as ) is minimized.

Example:

Input : n = 3

L[] = { 5, 3, 10 }

Output : Order should be { 3, 5, 10 } with MRT = 29/3

- Now suppose there are 4 songs in a tape of audio lengths 5, 7, 3 and 2 mins respectively.
- In order to play the fourth song, we need to traverse an audio length of  $5 + 7 + 3 = 15$  mins and then position the tape head.
- Retrieval time of the data is the time taken to retrieve/access that data in its entirety.
- Hence retrieval time of the fourth song is  $15 + 2 = 17$  mins.

- The sequential access of data in a tape has some limitations.
- Order must be defined in which the data/programs in a tape are stored so that least MRT can be obtained.
- Hence the order of storing becomes very important to reduce the data retrieval/access time.

ORDER	TOTAL RETRIEVAL TIME	MEAN RETRIEVAL TIME
1 1 2 3	$2 + (2 + 5) + (2 + 5 + 4) = 20$	20/3
2 1 3 2	$2 + (2 + 4) + (2 + 4 + 5) = 19$	19/3
3 2 1 3	$5 + (5 + 2) + (5 + 2 + 4) = 23$	23/3
4 2 3 1	$5 + (5 + 4) + (5 + 4 + 2) = 25$	25/3
5 3 1 2	$4 + (4 + 2) + (4 + 2 + 5) = 21$	21/3
6 3 2 1	$4 + (4 + 5) + (4 + 5 + 2) = 24$	24/3

- Time complexity of the above program is the time complexity for sorting, that is  $O(n \log n)$ .
- If you use bubble sort, it will take  $O(n^2)$ .