SIM GLOBAL EDUCATION

UNIVERSITY OF WOLLONGONG AUSTRALIA

CSCI203 - Algorithms and Data Structures

Elementary Data Structures

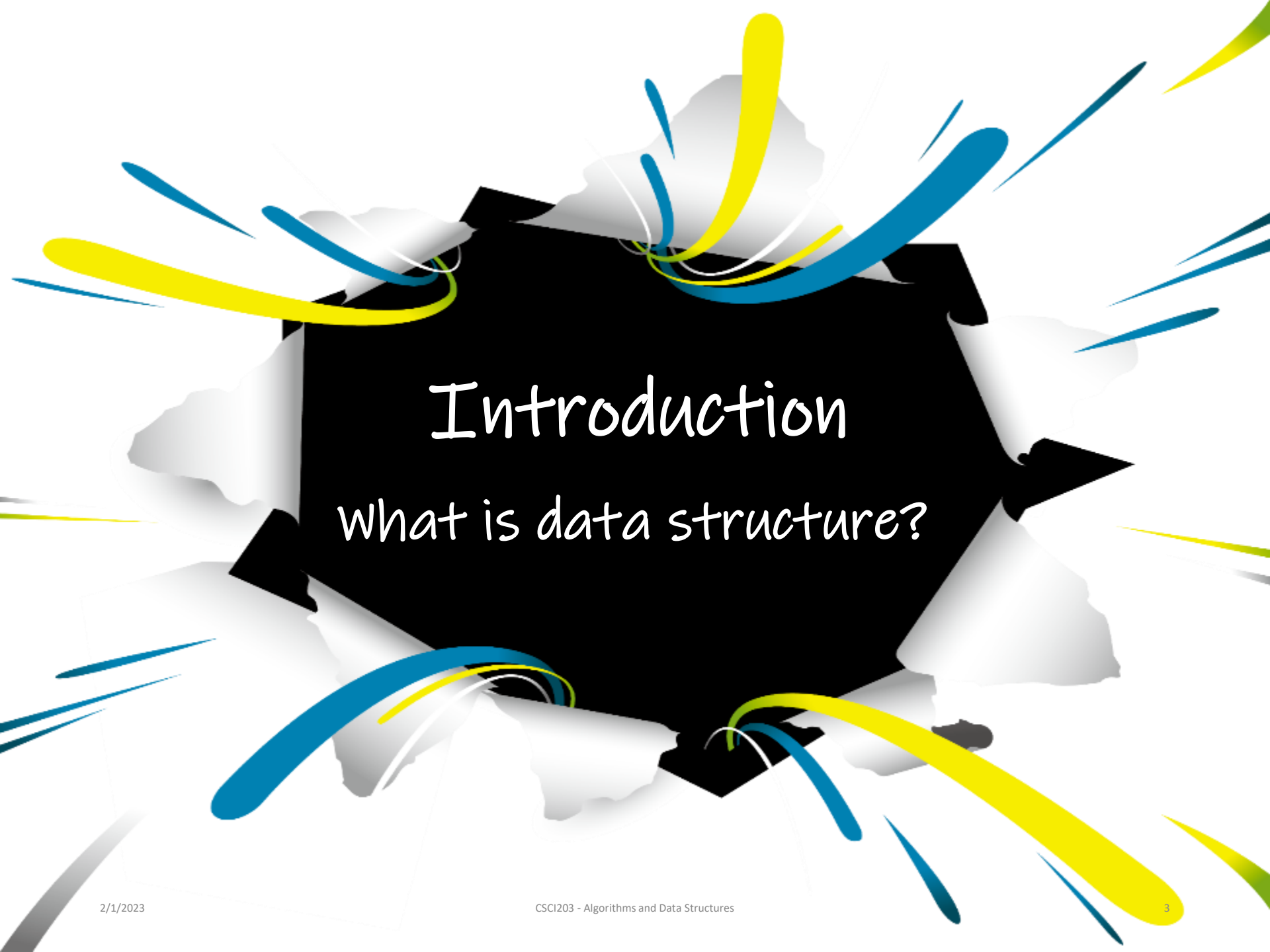
Sionggo Japit
sjapit@uow.edu.au

2 January 2023

Learning Outcome:

By the end of this lecture, you will be able to:
Explain, describe and understand the usage of the following data structures:

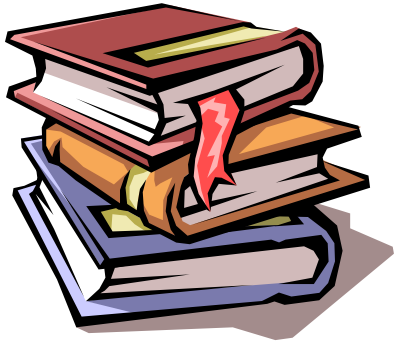
- Array
- Stack
- Queue
- Linked List
- Record (structures)
- Graph
- Tree
- Associative Tables


Introduction
What is data structure?

Elementary Data Structure

"Get your data structures correct first, and the rest of the program will write itself.."

- A **data structure** is a particular way of storing and organizing data in a computer so that the data can be used and manipulated efficiently.



Elementary Data Structure

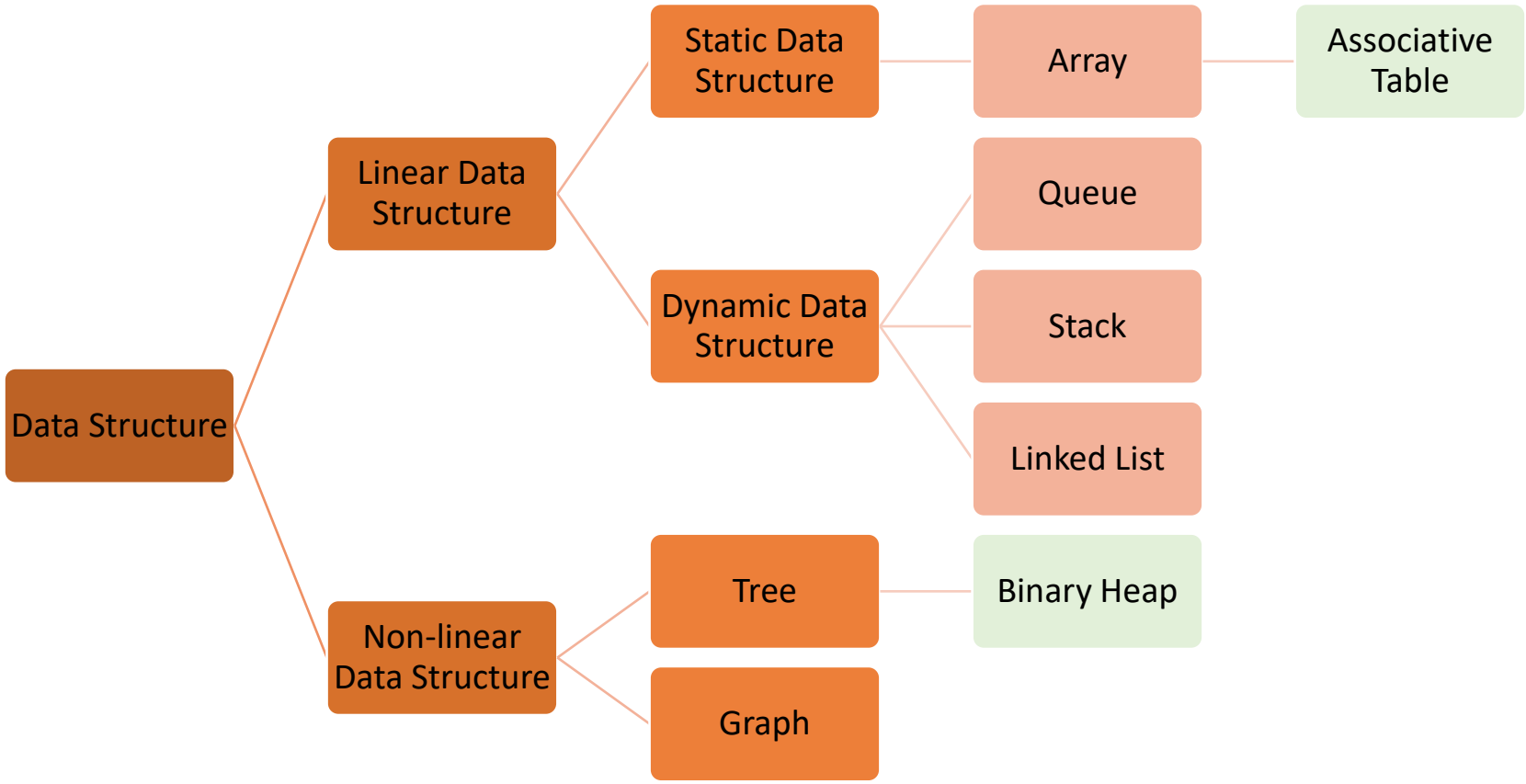
- The way in which an underlying structure is implemented can have substantial effects on program development, and on the capabilities and usefulness of the implementation.
- Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to certain tasks.

  For example, B-trees are particularly well-suited for implementation of databases, while compiler implementations usually use hash tables to look up identifiers.

Elementary Data Structure

- Data structures are used in almost every program or software system.
- Specific data structures are essential ingredients of many efficient algorithms and make possible the management of huge amounts of data, such as large databases and internet indexing services.
- Some formal design methods and programming languages emphasize data structures, rather than algorithms, as the key organizing factor in software design.

Classification of Data Structure




Array

Array

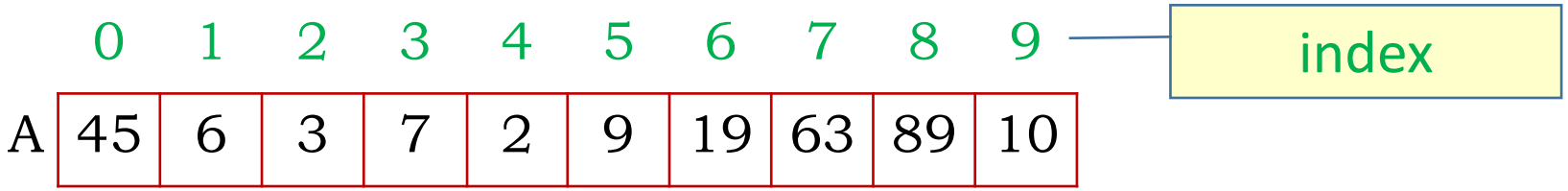
- Data: An array is a data structure consisting of a **fixed** number of **data items** of the **same type**. For example, in Java,
  - int[] digit = new int[10];

    defines a reference to an array digit consisting of 10 integer elements.
  - char[] letter = new char[26];

    defines a reference to an array letter consisting of 26 character elements.

Array

- First element of array is called lower bound, and it is always 0.
- Highest element in array is called upper bound.
- Hence, the indices of an array of 10 elements run from 0 to 9.
- An example of an array A of integer with 10 element.



Array

- Operation: Any array element is directly accessible via an index value.
  - E.g. $x = A[9]$;
    $initial = A[7]$;

- Arrays can have more than one indices; such an array is known as multidimensional arrays
  - E.g., $float\ Heights\ [20][20]$ is an array with 400 elements of floating data type.

Array

- Many computer games, be they strategy games, simulation games, or first-person conflict games use a two-dimensional “board.” A natural way to implement such games is with a two-dimensional array, where two indices, says row and column, are used to refer to the cells in the two-dimensional array.

Array

- The following diagrams are illustrations of a tic-tac-toe board and the two-dimensional integer array, board, representing them.

	X	
X	0	0

Playing board

	0	1	2
0	0	1	0
1	1	-1	-1
2	0	0	0

Board array

Array – Operation complexity

- Access an item: $O(1)$
  - Random access in constant time by specifying its index, i.e., evaluating the expression $A[i]$
- Initialize array: $O(n)$
  - We can reduce this in some instances by using virtual initialization

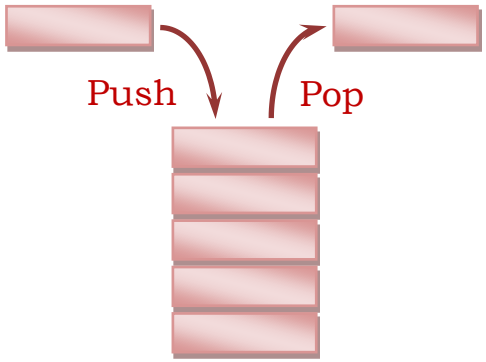
Array – Operation complexity

- Inserting at the beginning of the array: $O(n)$
  - All n items must move over one cell to make room for the added Element
- Deleting the first element from an array: $O(n)$
  - All n items must move forwards one cell to fill up the empty cell of the removed element

Stack

Stack

- A stack is a data structure which holds multiple elements of a single type
- Elements can be removed from a stack only in the reverse order to that in which they were inserted (LIFO, Last In First Out)



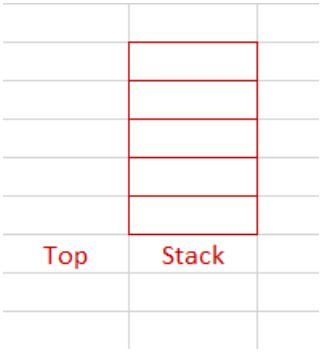
Stack

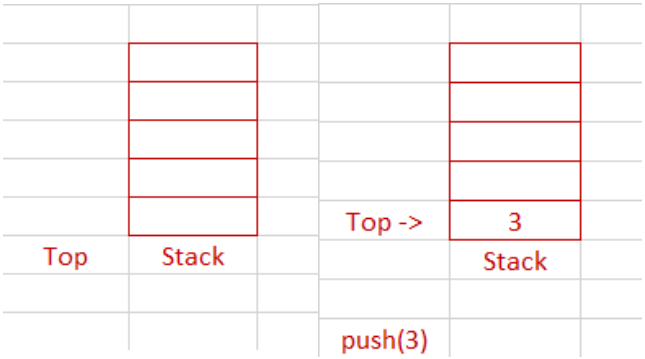
- One of the many applications of stack operations is to check whether parentheses, braces, and brackets are balanced in a computer program source file.
- Stack operations are also used in microprocessor to manage return address and arguments of methods when methods are called; the return address and arguments are pushed onto a stack, and when a method returns the return address and arguments are popped off.

Stack

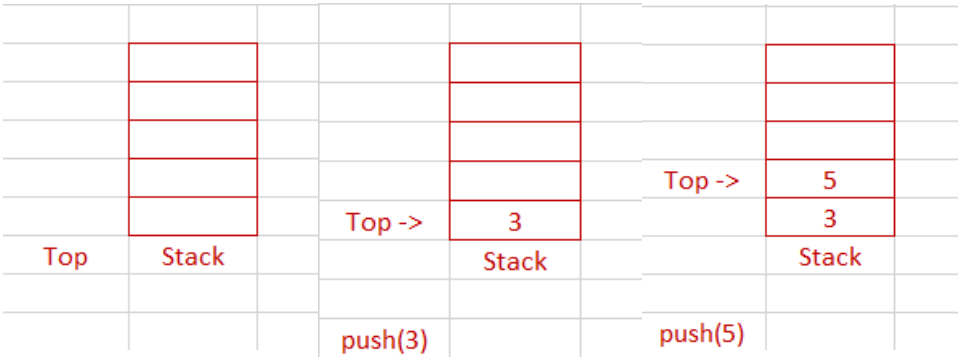
- Operations:
  - $isEempty()$: return true if the stack is empty, otherwise return false
  - $isFull()$: return true if the stack is full, otherwise return false
  - $push(val)$: Add the item val to the stack
  - $pop()$: Remove the item most recently added to the stack
  - $peek()$: Return the item most recently added to the stack without removing it

Initializing an empty stack.

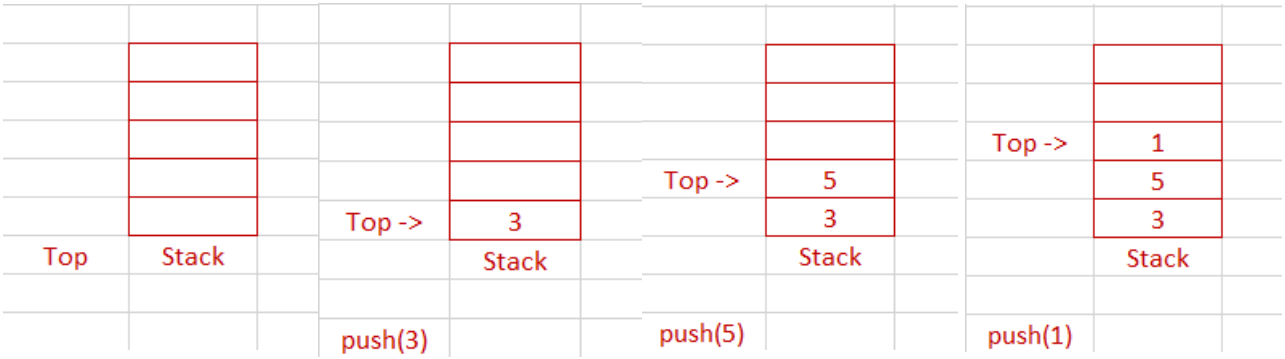




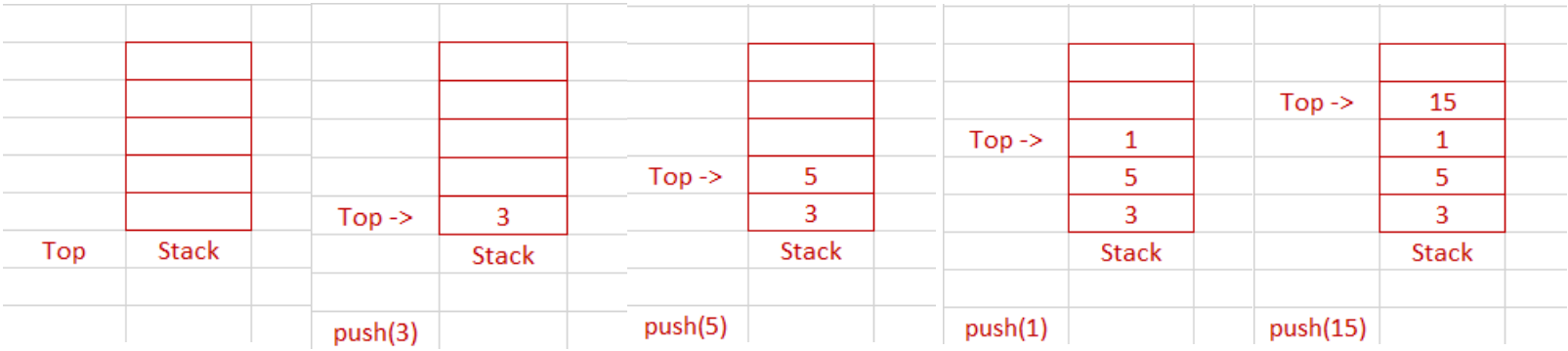
Push a value 3 into stack.



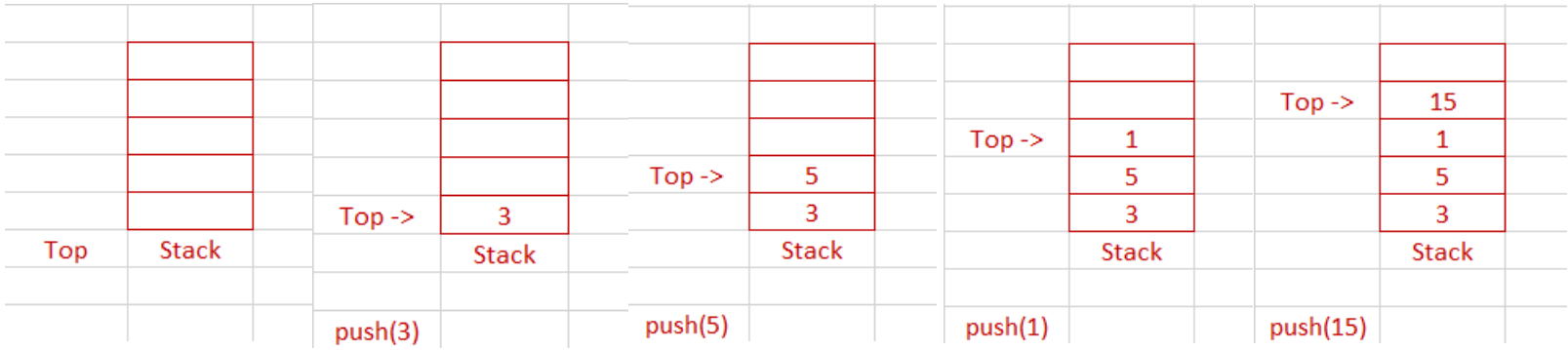
Push a value 5 into stack.

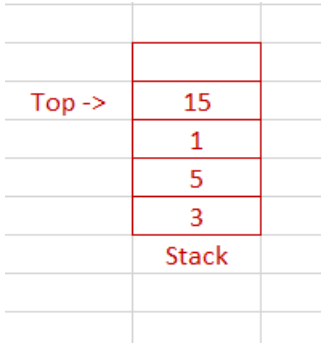


Push a value 1 into stack.

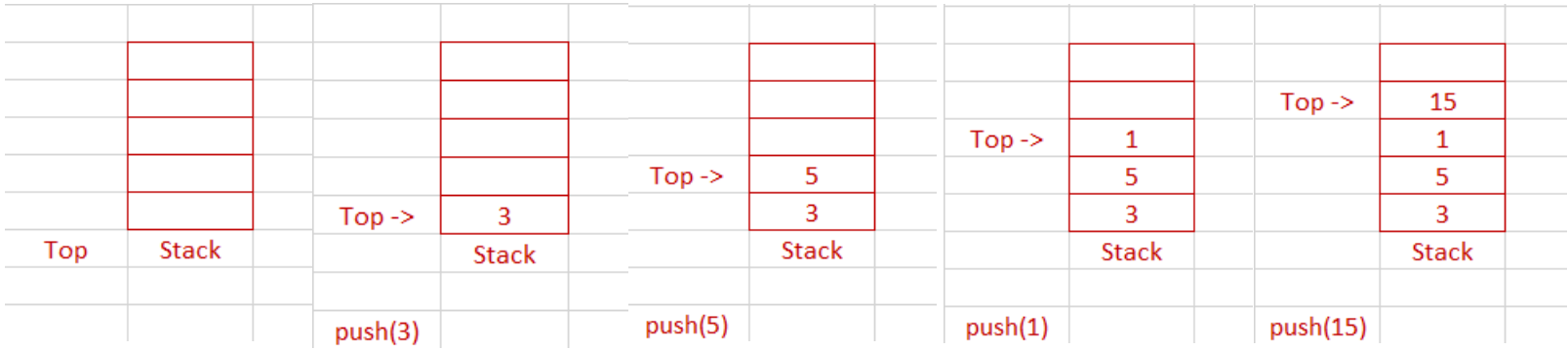


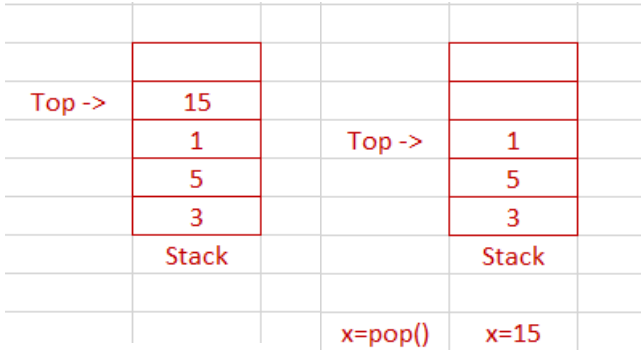
Push a value 15 into stack.



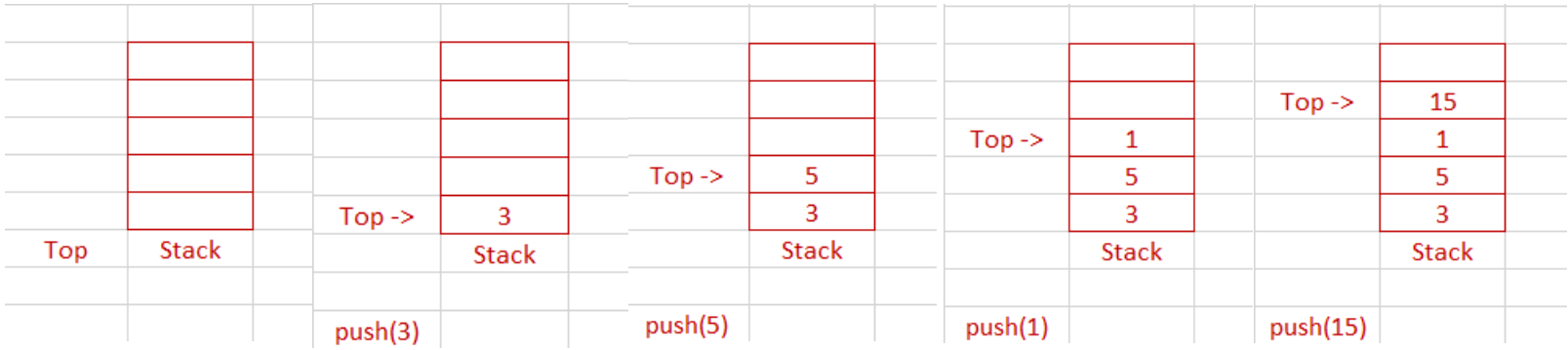


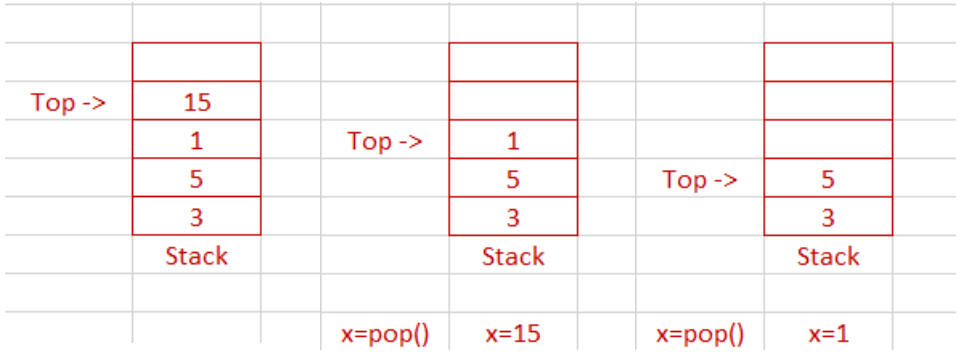
An instance of the stack.



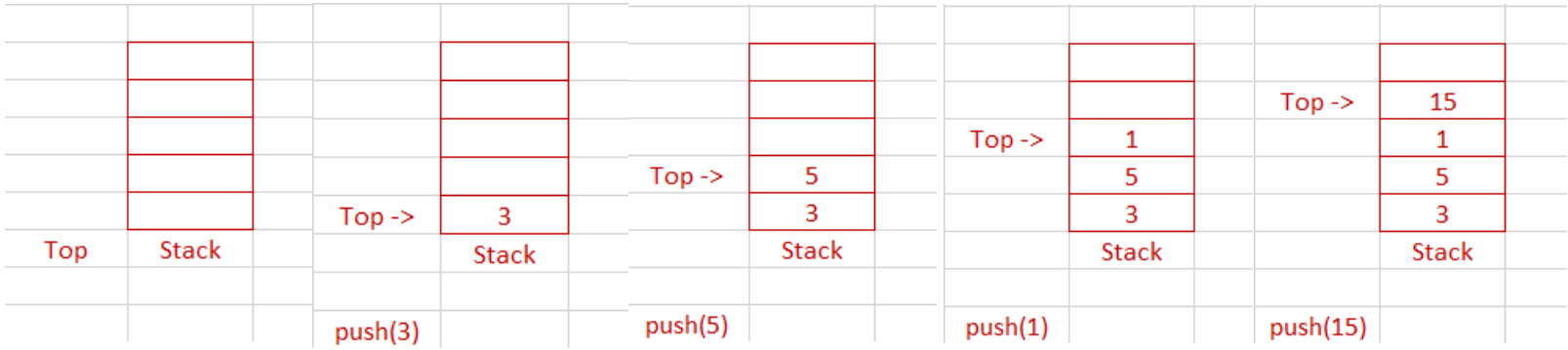


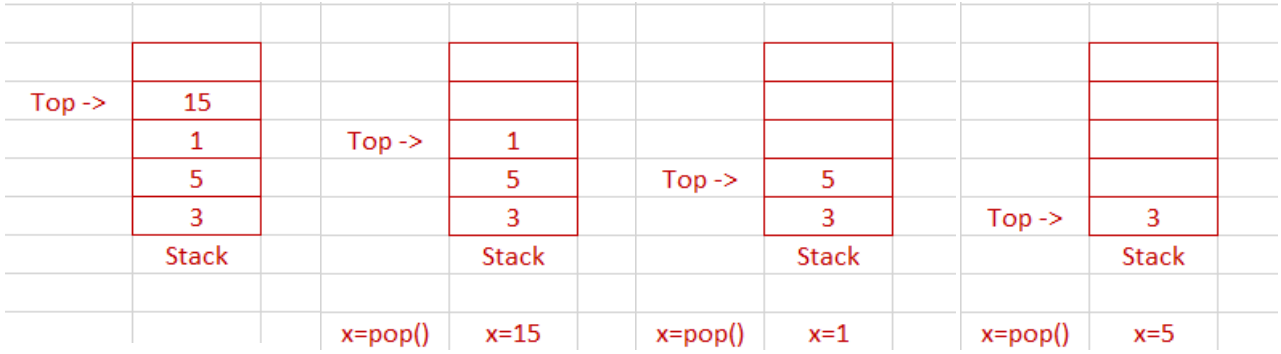
Pop from stack to x.
x receives 15



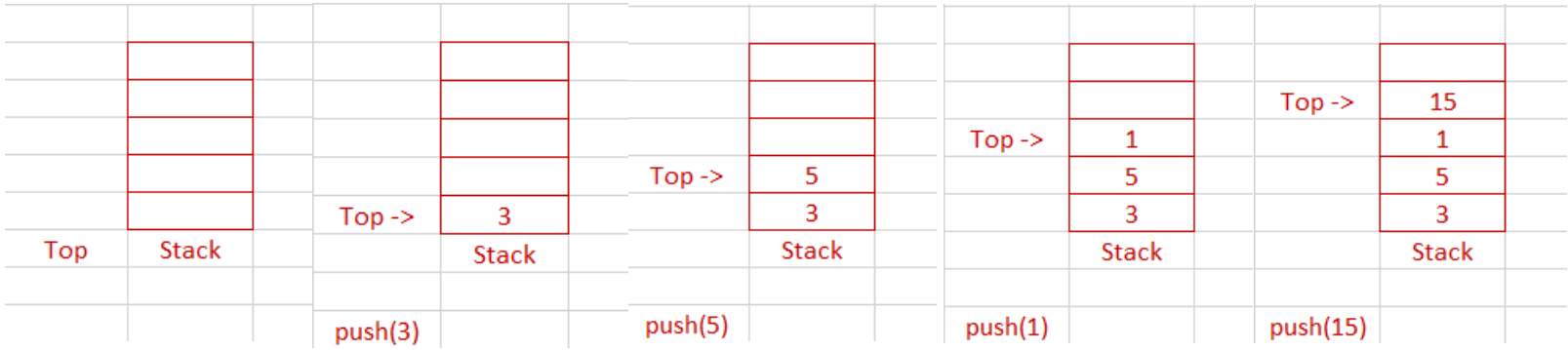


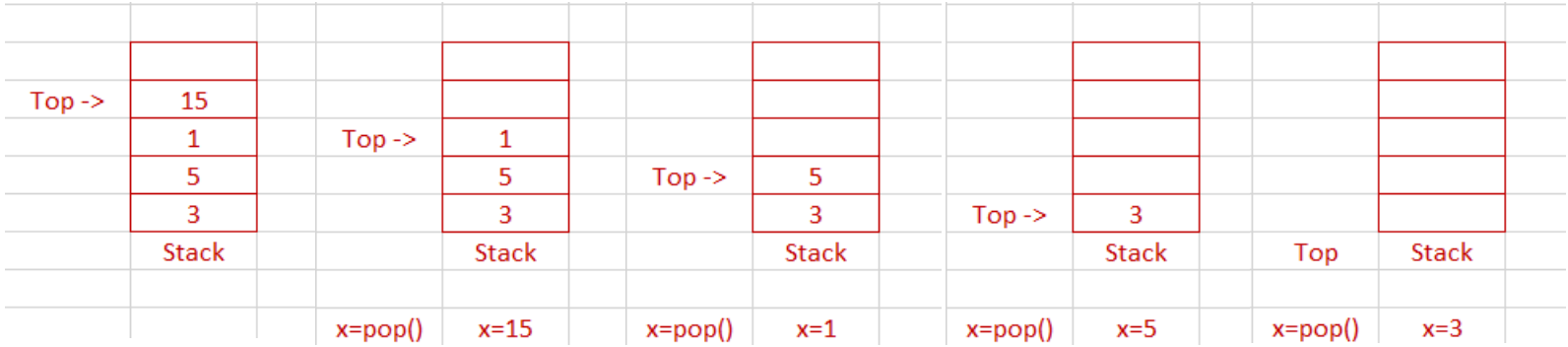
Pop from stack to x.
x receives 1



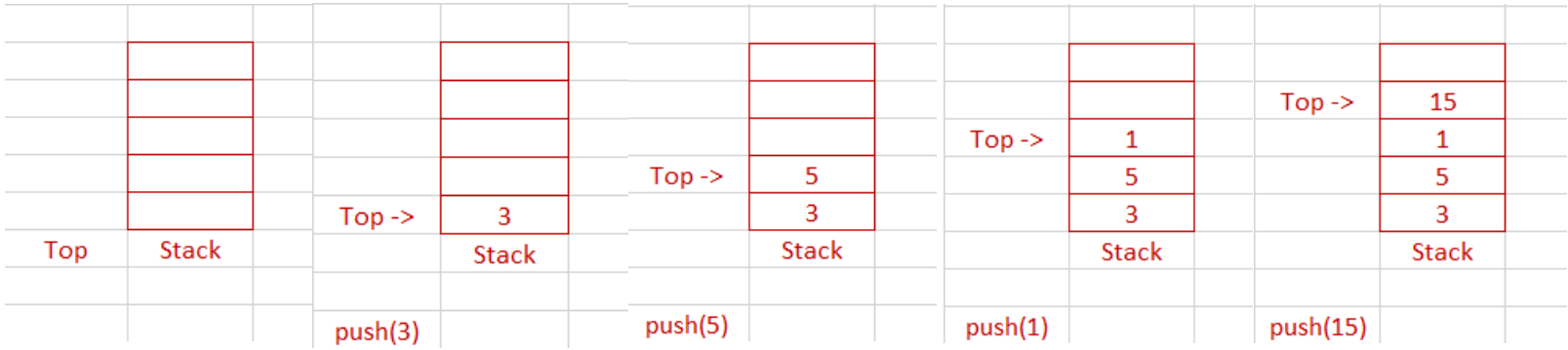


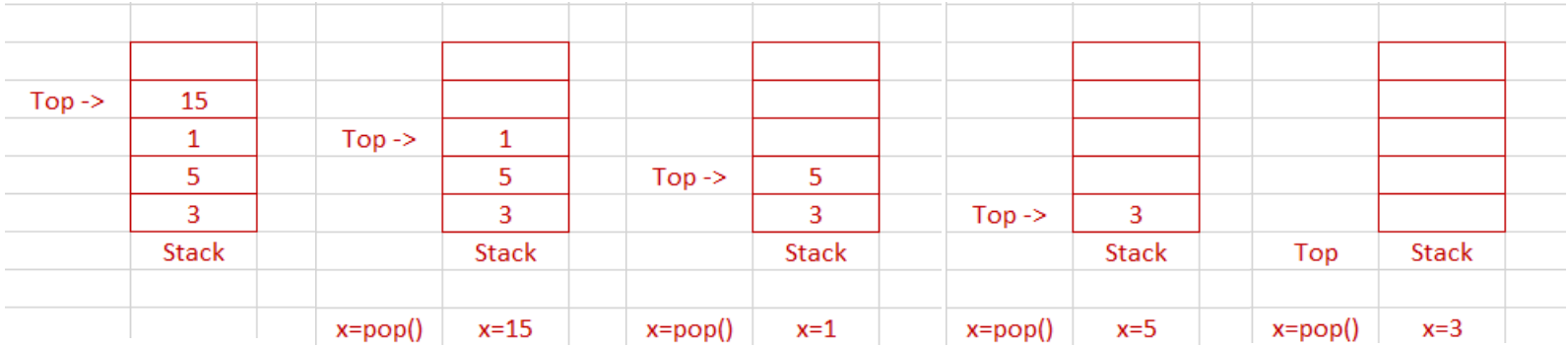
Pop from stack to x.
x receives 5





Pop from stack to x.
x receives 3


Top
Stack
Top ->
3
Stack
push(3)
Top ->
5
3
Stack
push(5)
Top ->
1
5
3
Stack
push(1)
Top ->
15
1
5
3
Stack
push(15)

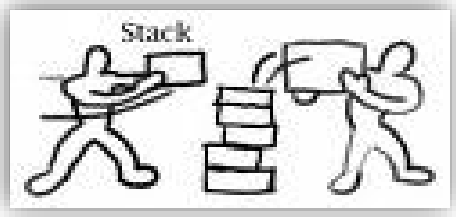

Top ->
15
1
5
3
Stack
x=pop()
x=15
Top ->
1
5
3
Stack
x=pop()
x=1
Top ->
5
3
Stack
x=pop()
x=5
Top ->
3
Stack
x=pop()
x=3
Top
Stack

Stack

- A stack can be implemented using an array "**stack**" and a variable "**top**" to index the top of the stack
  - When an item is pushed onto the stack, we increment "top" and put the item in the cell at index "top"
  - when an item is popped off the stack, we decrement.

Stack implementation

```
class Stack {
   private int maxSize;
   private double[] stackArray;
   private int top;

   // Constructor
   Stack (int s) {
            maxSize = s;
      stackArray = new double[maxSize];
             top = -1;
           } // End constructor
```

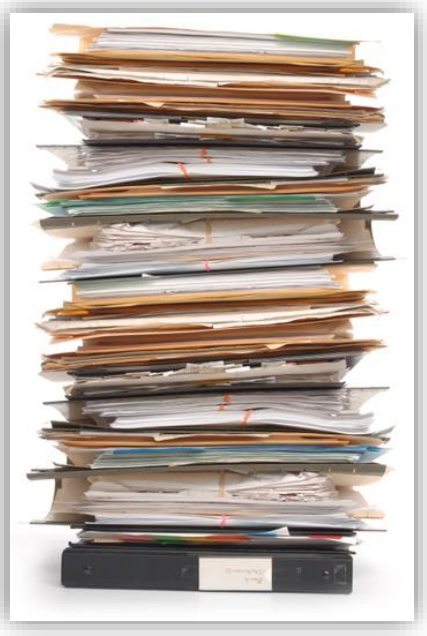


Stack implementation

```
public void push(double item) {
   stackArray[++top] =  item;
} // End push

public double pop() {
   return stackArray[top--];
} // End pop
```

Stack implementation

```
public double peek() {
   return stackArray[top];
} // End peek

public Boolean isEmpty() {
   return (top == -1);
} // End isEmpty
```



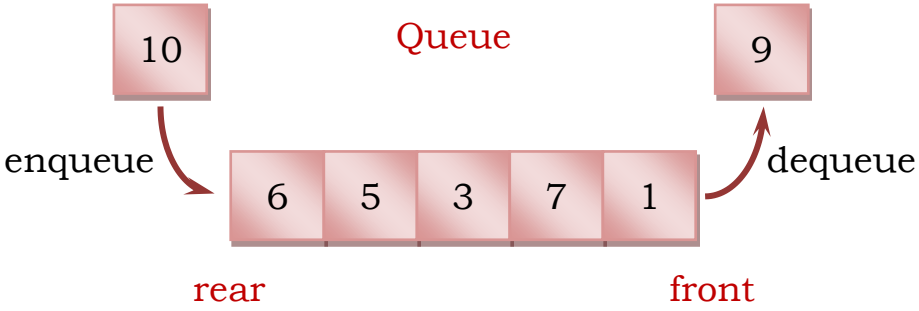
Stack implementation

```
   public Boolean isFull() {
      return (top == maxSize-1);
   } // End isFull
} // End class Stack
```


Queue

Queue

- A queue is a data structure which holds multiple elements of a single type
- Elements can be removed from a queue only in the order in which they were inserted (FIFO, First In First Out)



Queue

Operations:

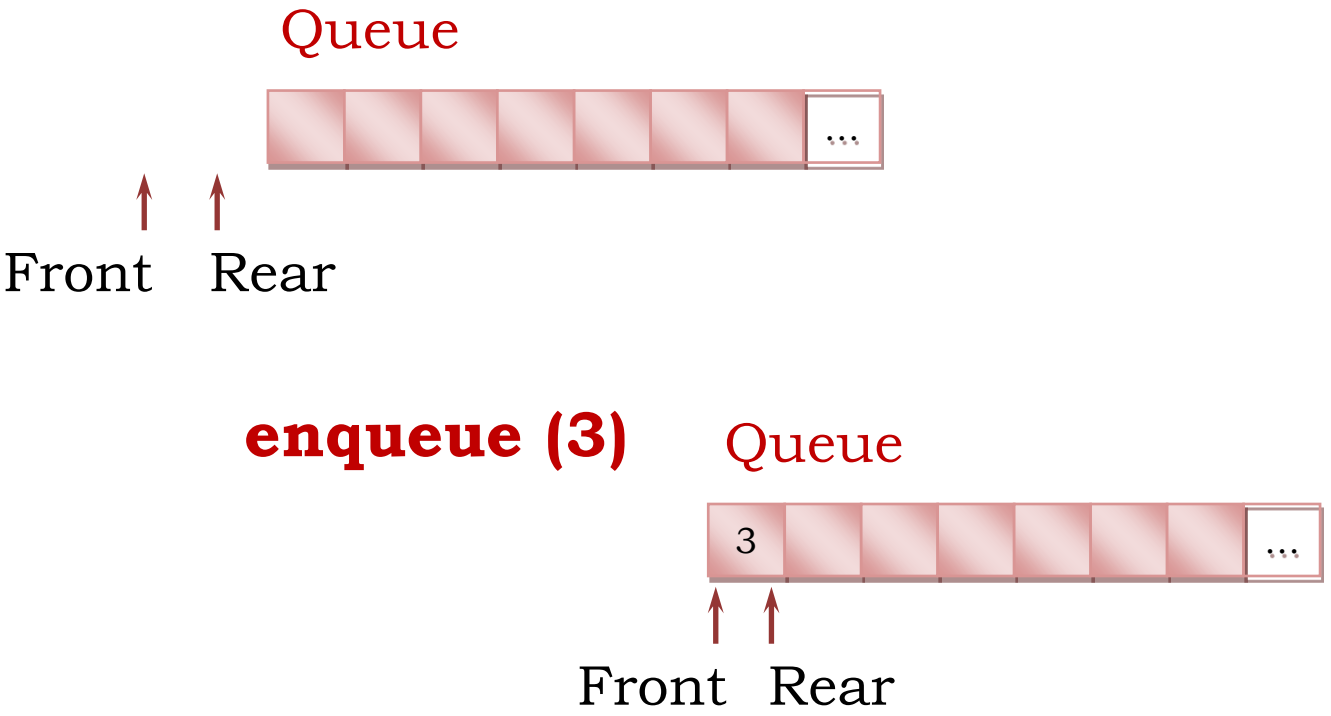
- $isEmpty()$: return true if the queue is empty otherwise, return false
- $enqueue(val)$: Add the item val to the queue. In a queue, an item is added at the rear of the queue.

Queue

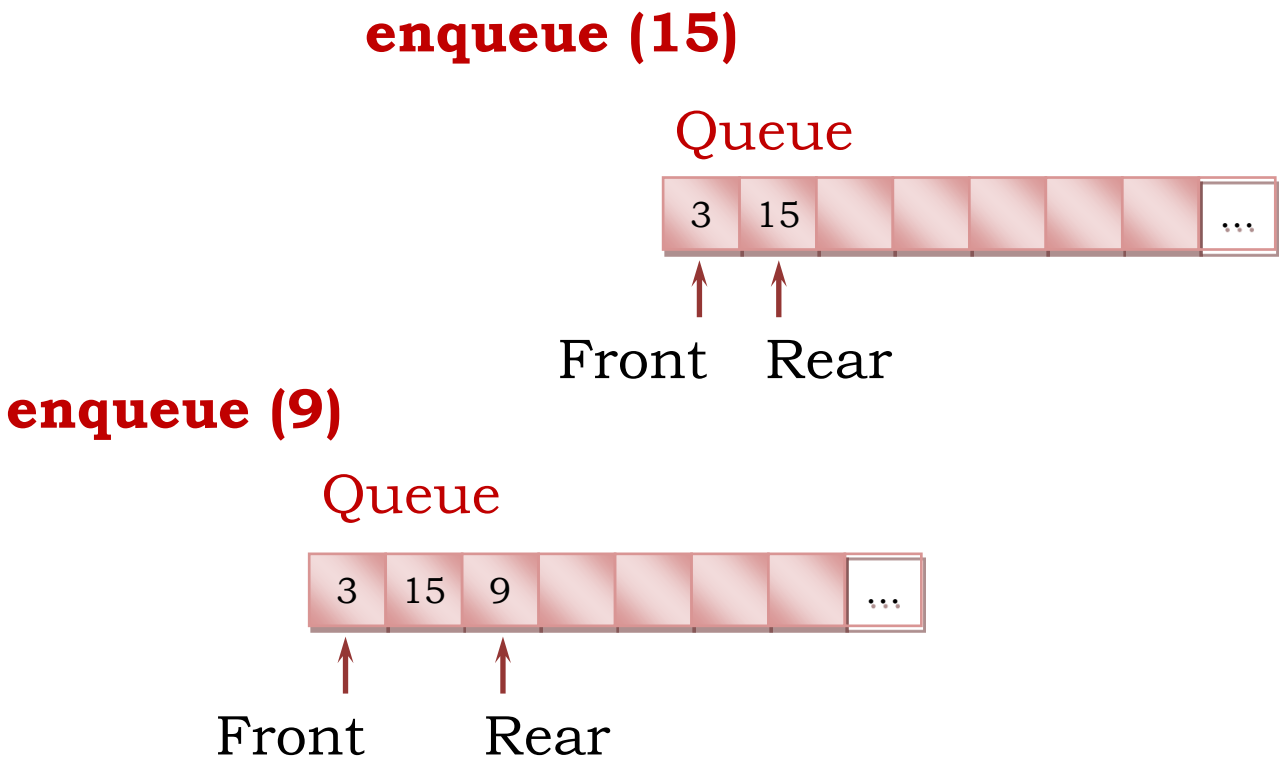
Operations: (cont…)

- $dequeue()$: remove the item least recently added to the queue. i.e., item at the front of the queue is removed.
- $front()$: return the item least recently added to the queue but do not remove it

Queue

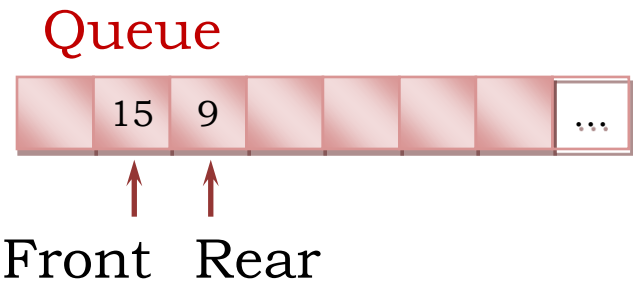


Queue

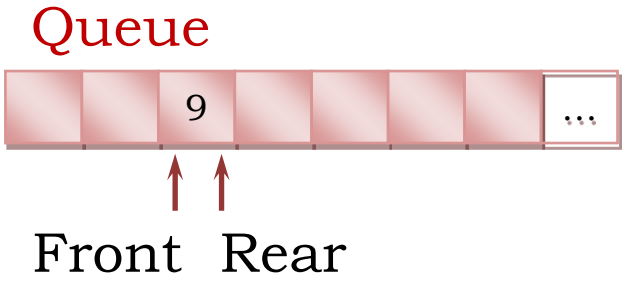

enqueue (15)
Queue
3
15
...
Front
Rear
enqueue (9)
Queue
3
15
9
...
Front
Rear

Queue

dequeue

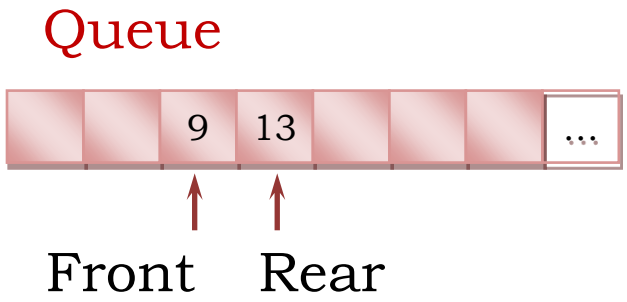


dequeue

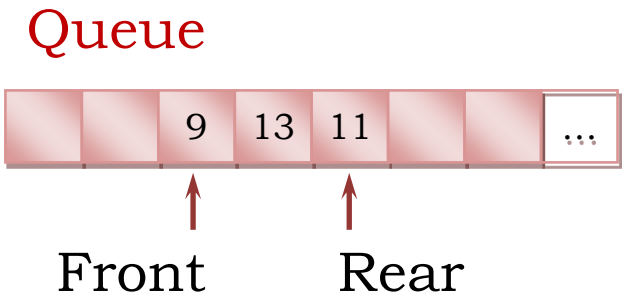


Queue

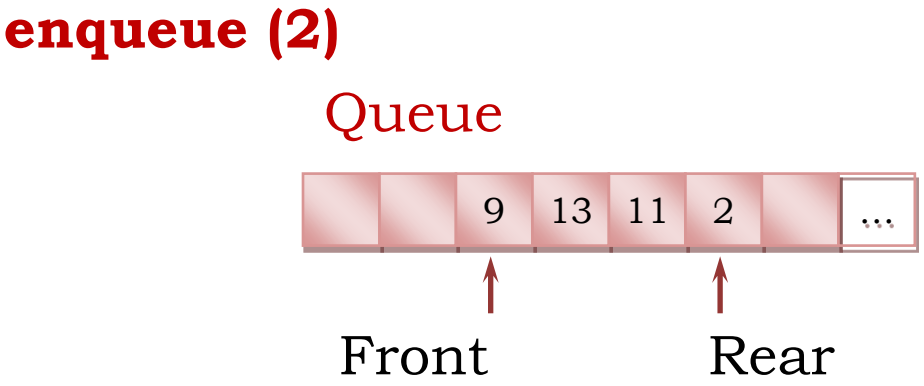
enqueue (13)



enqueue (11)



Queue



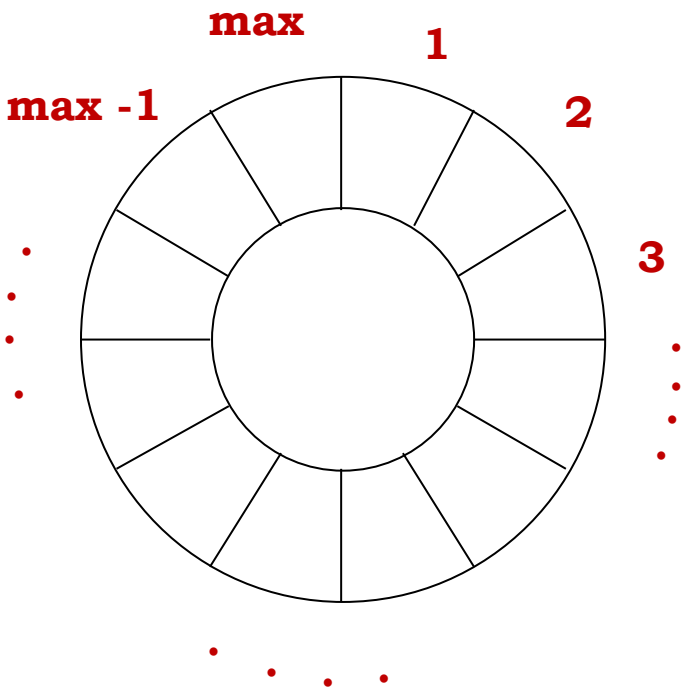
- If we continue with the enqueue and dequeue processes, both the 'Front' and 'Rear' indices will be increased and never decreased.

Queue

- This creates problem, because as the queue moves down the array, the storage space at the beginning of the array is discarded and never used again.
- To overcome this inefficient use of space, we reuse the discarded array elements at the beginning of the array.

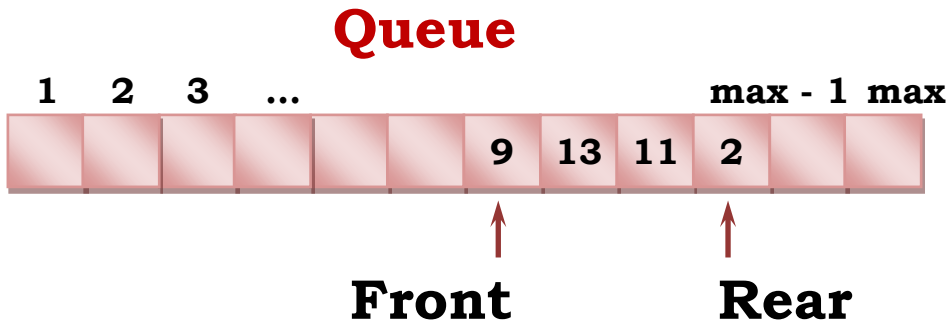
Queue

- This can be done by thinking of the array as a circle as shown below.

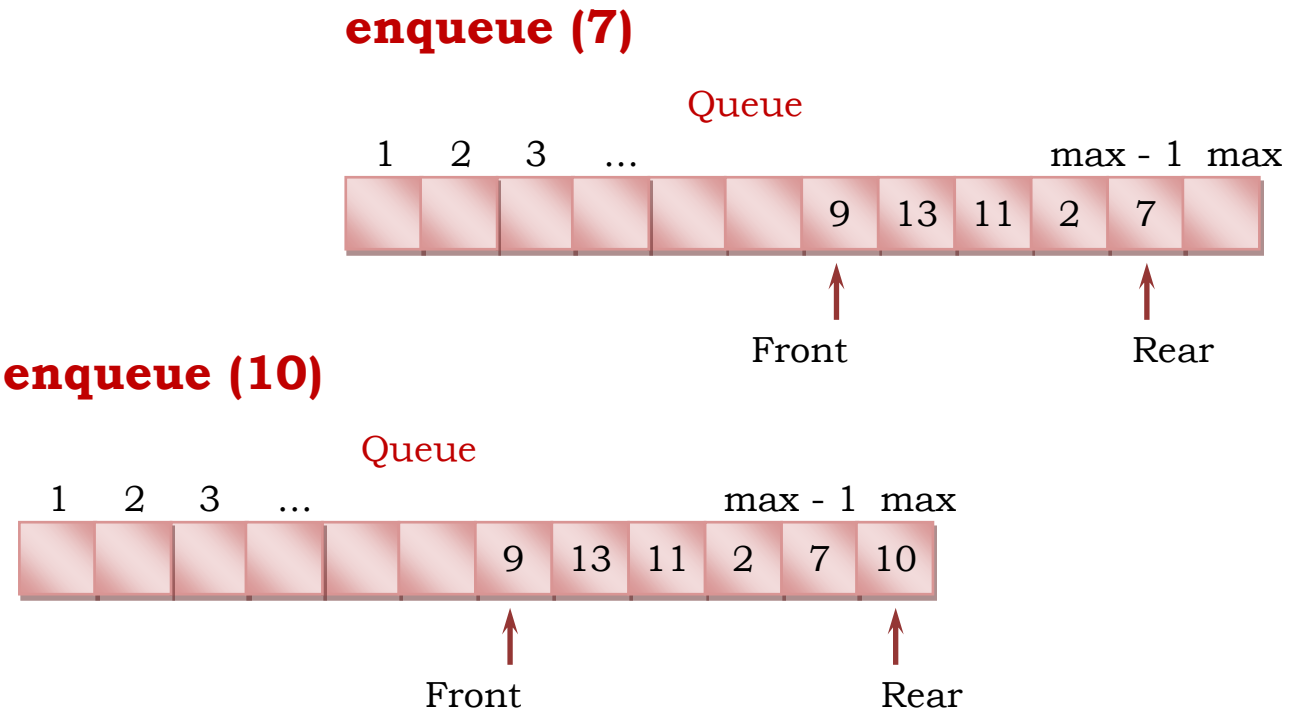


Queue

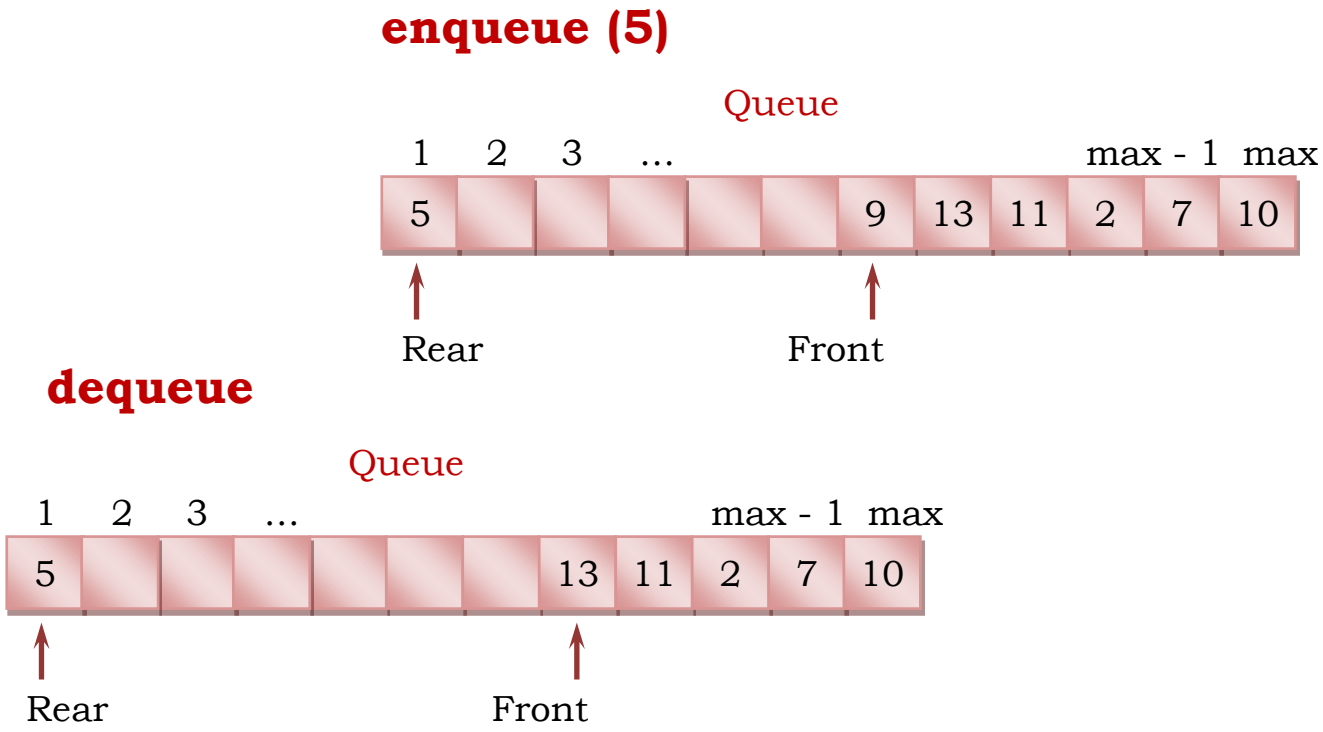
- Unwinding the circular array to analyze how to reuse the discarded array elements.



Queue


enqueue (7)
Queue
1 2 3 ... max - 1 max
9 13 11 2 7
Front
Rear
enqueue (10)
Queue
1 2 3 ... max - 1 max
9 13 11 2 7 10
Front
Rear

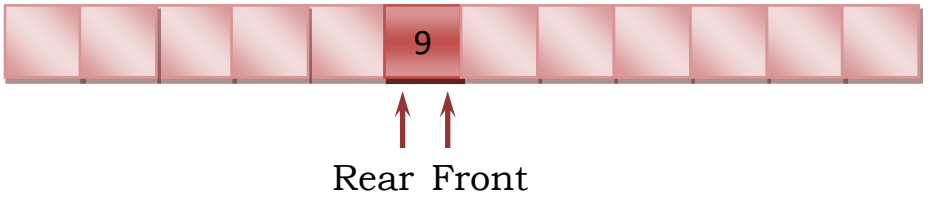
Queue


enqueue (5)
Queue
1 2 3 ... max - 1 max
5 9 13 11 2 7 10
Rear
Front
dequeue
Queue
1 2 3 ... max - 1 max
5 13 11 2 7 10
Rear
Front

Queue

Boundary conditions (scenario 1)

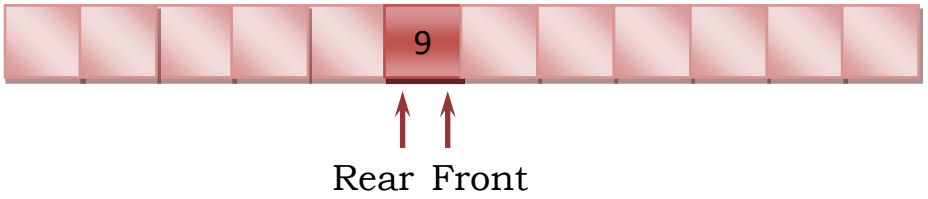
- Queue containing one item



Queue

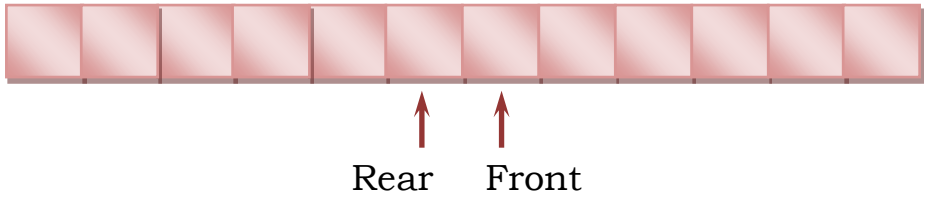
Boundary conditions (scenario 1)

- Queue containing one item



Perform a dequeue operation, and the queue becomes empty.

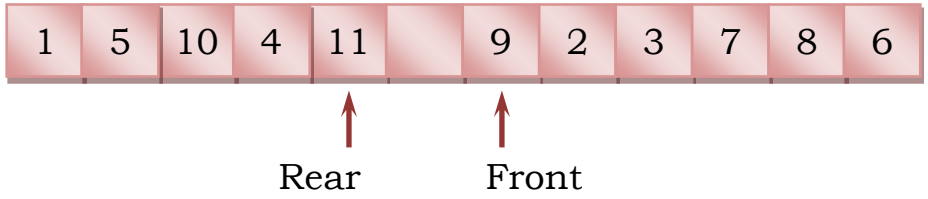
- Empty queue



Queue

Boundary conditions (scenario 2)

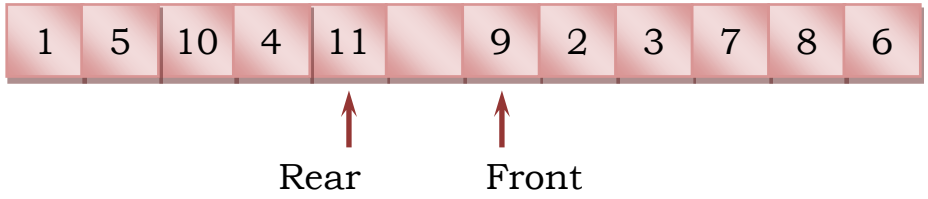
- Queue with one empty position



Queue

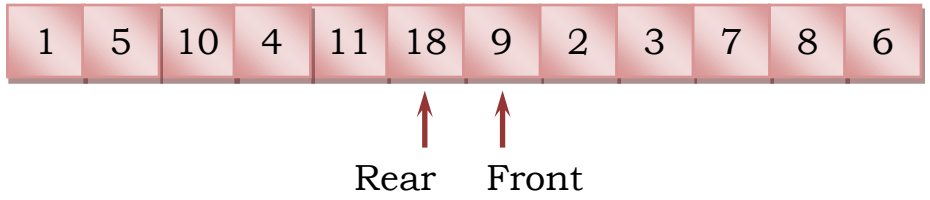
Boundary conditions (scenario 2)

- Queue with one empty position



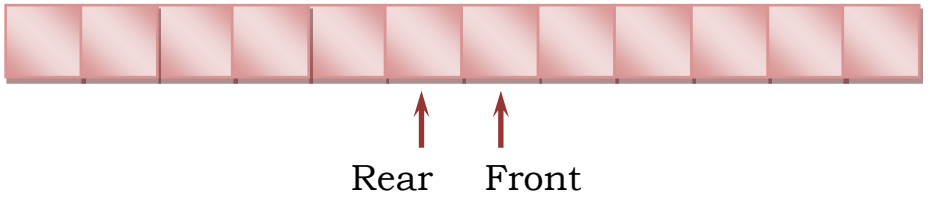
Perform an enqueue operation, and the queue becomes full.

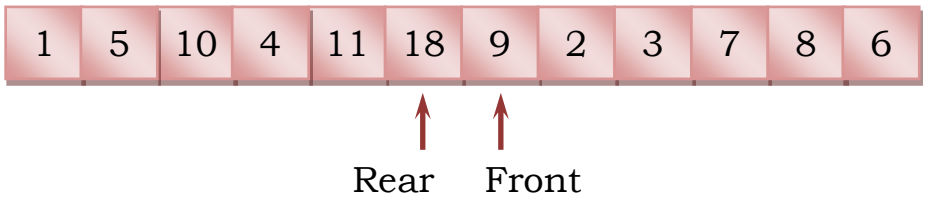
- Full queue



Queue

- Problem! How to differentiate (or know) if the queue is empty or full?





Queue

- Three possible ways to resolve the problem:
  - Empty position – Insist on leaving one empty position in the array so that the queue is considered full when the ‘Rear’ index has moved within two positions of the ‘Front’.
  - Flag – Introduce a Boolean variable that will be used when the ‘Rear’ comes just before the ‘Front’ to indicate whether the queue is full or not.

Queue

- Special value – Set one or both of the indices to some value(s), e.g., a negative value, that would otherwise never occur in order to indicate an empty or full queue.

- Are there any other possible solutions?

Queue

- A queue can be implemented using an array queue[1..N], where N is the size of the array.
- In a queue, items are added at the rear and deleted from the front.
  - Two variables are used to track the indexes of the rear and the front of the queue.
- An empty queue has both variables (Front and Rear) equal to 0.

Queue

- When an item is added to a queue
  - If the queue is empty: set both “Rear” and “Front” to 1 and put the item at index 1
  - If the queue is not empty: increment “Rear” and put the item at index “Rear”. If “Rear” is greater than the index of the last cell in the array, “wrap around” and set “Rear” to 1.

Queue

- When an item is deleted from a queue
  - Increment “Front”.
  - If “Front” is greater than the index of the last cell in the array, set “Front” to 1.
  - if the queue becomes empty, set “Rear”=“Front”=0.

- Any other more convenient way?

Queue

- When an item is deleted from a queue
  - Increment “Front”.
  - If “Front” is greater than the index of the last cell in the array, set “Front” to 1.
  - if the queue becomes empty, set “Rear”=“Front”=0.

- Any other more convenient way?
  - How about keeping a variable to count the total number of elements added to the queue? The counter is increased when an item is added to the queue and decreased when an item is removed from the queue.

Queue implementation

```
class Queue {
   private int maxSize;
   private int[] queArray;
   private int front;
   private int rear;
   private int nItems;

   // Constructor
   public Queue(int qSize) {
      maxSize = qSize;
      queArray = new int[maxSize];
      front = 0;
      rear = -1;
      nItem = 0;
   } // End constructor
```

Queue implementation

```
public void enqueue(int item) {
   if (rear == maxSize-1)
      rear = -1;
   queArray[++rear] =  item;
   nItems++;
} // End enqueue

public int dequeue() {
   int temp = queArray[front++];
   if (front == maxSize)
      front = 0;
   nItems--;
   return temp;
}
```

Queue implementation

```
public int front() {
   return queArray[front];
} // End front

public boolean isEmpty() {
   return (nItems == 0);
} // End isEmpty

public boolean isFull() {
   return (nItems == maxSize);
}
```

Queue implementation

```
  public int size() {
    return nItems;
  }
} // End class Queue
```


Linked List

List

Lists

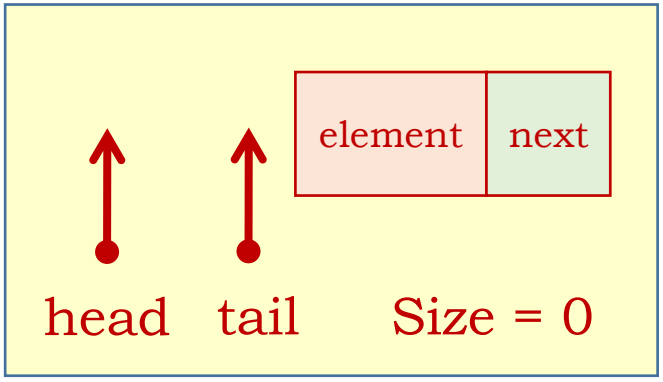
- Stack and Queue are lists data structure with certain constraints specified:
  - Stack
    - LIFO (Last-In-First-Out)
  - Queue
    - FIFO (First-In-First-Out)

List

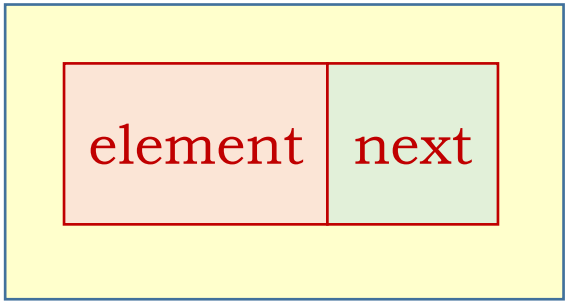
- A general list is a collection of items arranged in some order, but without the constraints of Stack or Queue; i.e., a general list allows items to be inserted into or deleted from the list at any point in the list.
- List may be implemented using:
  - Static structure, e.g., Array – We refer to it as List
  - Dynamic structure, e.g., pointers (references) – We refer to it as Linked List.
- We will discuss the concepts of List using Linked List.

Linked List

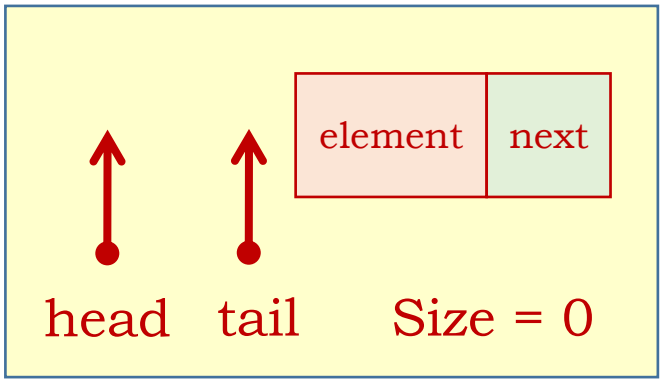
- The implementation of linked list consist of the following:
  - A ***node*** that contains two fields, a content (information) field and a next address (reference) field.
  - Two special references ***head*** and ***tail*** are maintained to reference to the first elements of the list as well as the last elements of the list.
  - In addition, often, a third variable is used to count the ***size*** (or number of elements in) the list.

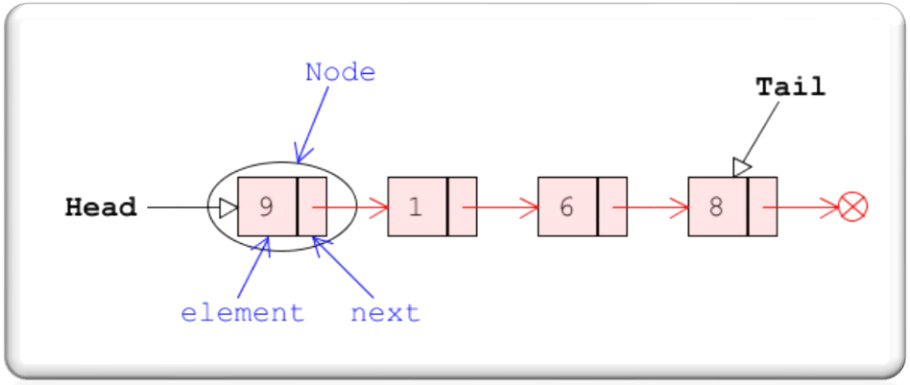


```
public class Node {
  private String element;
  private Node next;

  public Node(String s, Node n) {
    element = s;
    next = n;  }

  public String getElement() {
    return element;  }

  public Node getNext() {
    return next;  }

  public void setElement(String newElem) {
    element = newElem;  }

  public void setNext(Node newNext) {
    next = newNext;  }
}
```



```
// Singly linked list
public class SLinkedList {
  protected Node head;   // head node
  protected Node tail;   // tail node
  protected long size;   // Num of nodes

  public SLinkedList() {
    head = null;
    tail = null;
    size = 0;  }
}
```





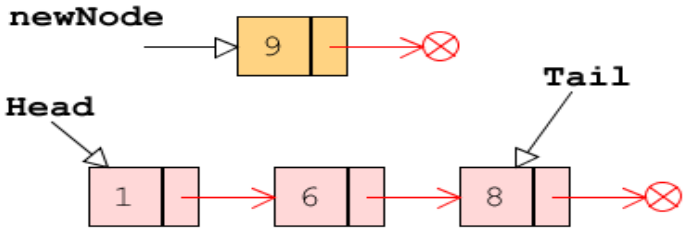
An instance of a linked list consisting of 4 elements.

Linked List: Operations

- $addFirst(newNode)$: Inserting a new node $newNode$ at the beginning of a linked list. The idea is to:
  i. create a new node $newNode$;
  ii. set the new node's $next$ link to reference to the same object as $head$ of the list, and then
  iii. set $head$ to point to the new node;
  iv. Lastly, increment the size.

Linked List: Operations

Algorithm $addFirst(newNode)$:
  newNode.$setNext(head)$
  $head \leftarrow newNode$
    $newNode \leftarrow null$
    $size \leftarrow size + 1$



Linked List: Operations

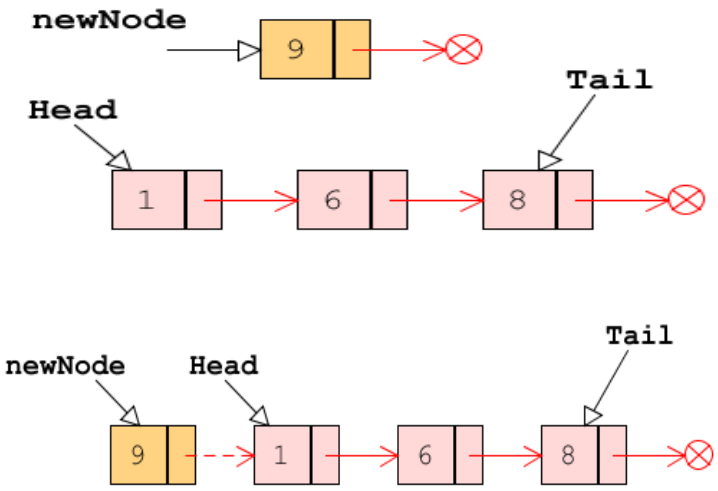
Algorithm $addFirst(newNode)$:

newNode.$setNext(head)$

$head \leftarrow newNode$

$newNode \leftarrow null$

$size \leftarrow size + 1$



Linked List: Operations

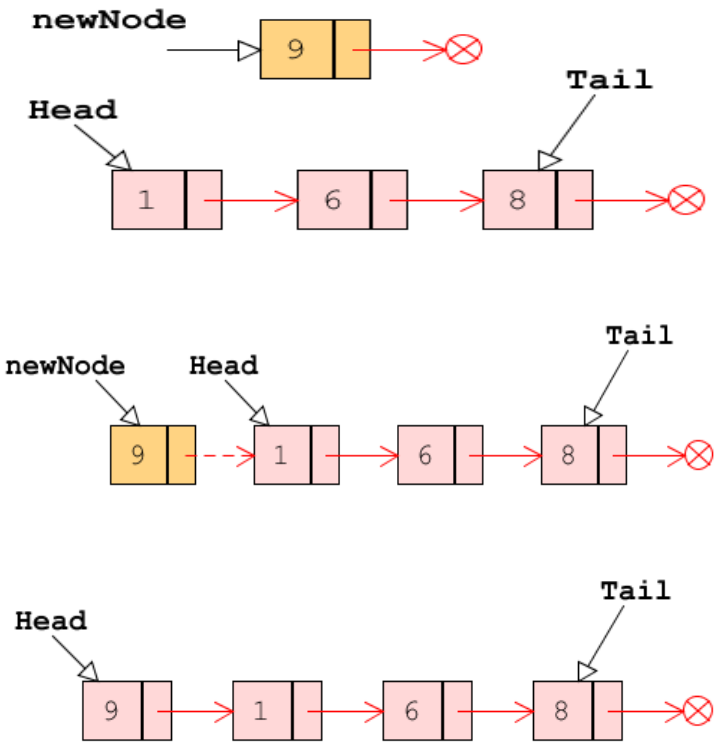
Algorithm $addFirst(newNode)$:

newNode.$setNext(head)$

$head \leftarrow newNode$

$newNode \leftarrow null$

$size \leftarrow size + 1$



Linked List: Operations

Algorithm $addFirst(newNode)$:

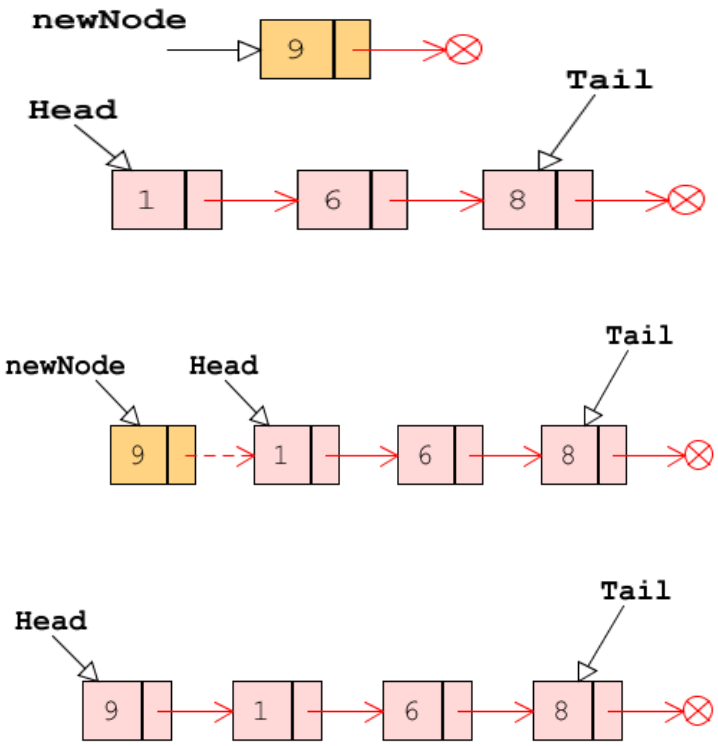
newNode.$setNext(head)$

$head \leftarrow newNode$

$newNode \leftarrow null$

$size \leftarrow size + 1$

What is the run-time complexity (Big-Θ) of the algorithm?



Linked List: Operations

Algorithm $addFirst(newNode)$:

newNode.$setNext(head)$

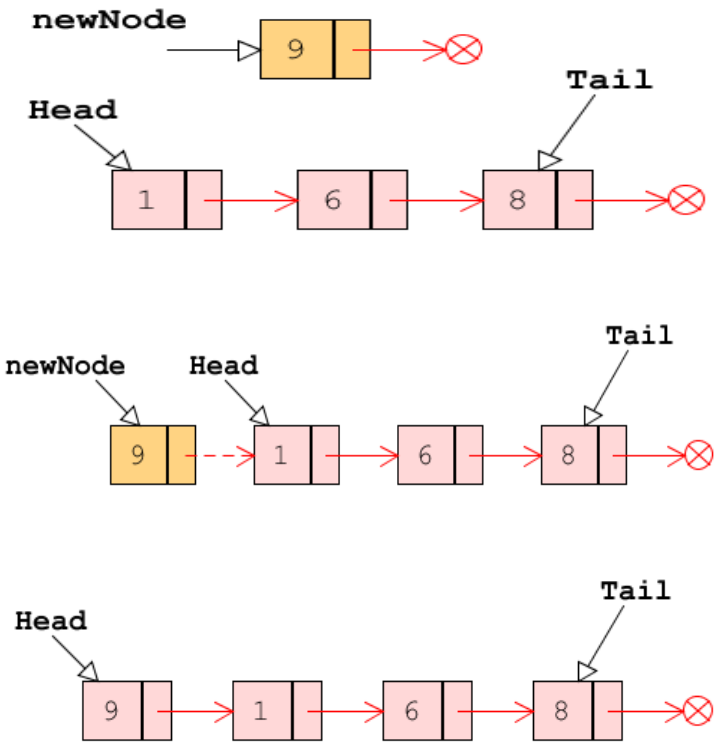
$head \leftarrow newNode$

$newNode \leftarrow null$

$size \leftarrow size + 1$

What is the run-time complexity (Big-Θ) of the algorithm? $\boldsymbol{\Theta(1)}$

Note: we use $\Theta(1)$ to denote constant steps operation.



Linked List: Operations

- $removeFirst()$:Removing a node at the beginning of a linked list. The idea is to:
  i. create a temporary node reference $tempRef$;
  ii. set $tempRef$ to reference to the same object as $head$ of the list, and then
  iii. set $head$ to reference to the node referenced to by $head.next$;
  iv. set the $tempRef$ to null; (Note, In Java we can remove the active references by assigning the value “null” to the $tempRef$. The object will be deleted by the Garbage collector.)
  v. Lastly, decrement the size of the linked list.

Algorithm $removeFirst()$:

$if\ head == null\ then$

$\quad show\ error\ message\ "List\ is\ empty."$

$tempRef \leftarrow head$

$head \leftarrow head.getNext()$

$tempRef \leftarrow null$

$size \leftarrow size - 1$

Algorithm $removeFirst()$:

$if\ head == null\ then$

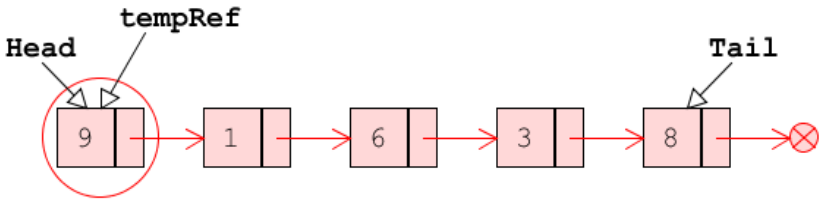
$\quad show\ error\ message\ "List\ is\ empty."$

$tempRef \leftarrow head$

$head \leftarrow head.getNext()$

$tempRef \leftarrow null$

$size \leftarrow size - 1$



Algorithm $removeFirst()$:

$if\ head == null\ then$

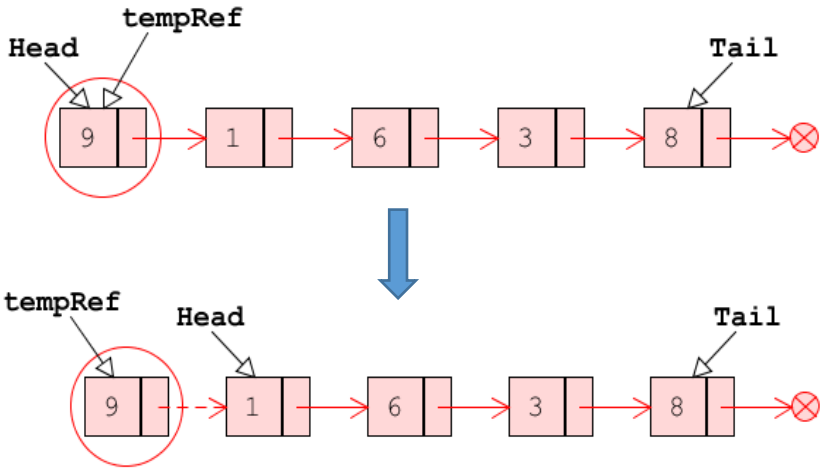
$\quad show\ error\ message\ "List\ is\ empty."$

$tempRef \leftarrow head$

$head \leftarrow head.getNext()$

$tempRef \leftarrow null$

$size \leftarrow size - 1$



Algorithm $removeFirst()$:

$if\ head == null\ then$

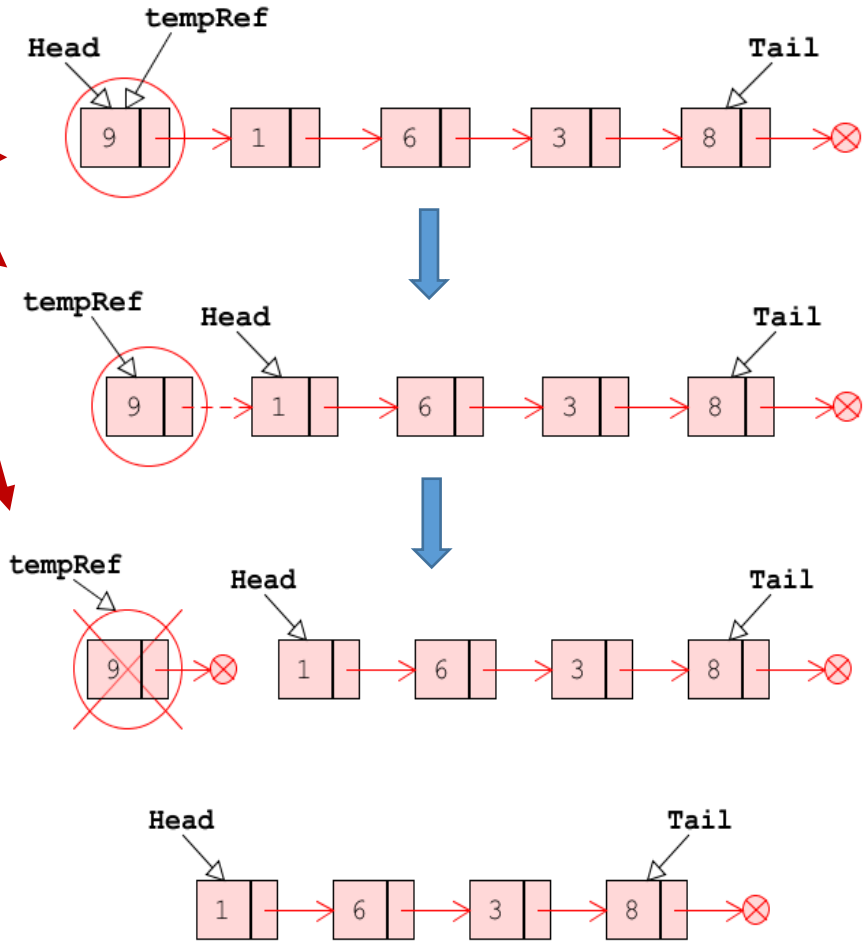
$show\ error\ message\ "List\ is\ empty."$

$tempRef \leftarrow head$

$head \leftarrow head.getNext()$

$tempRef \leftarrow null$

$size \leftarrow size - 1$



Algorithm $removeFirst()$:

$if\ head == null\ then$

$\quad show\ error\ message\ "List\ is\ empty."$

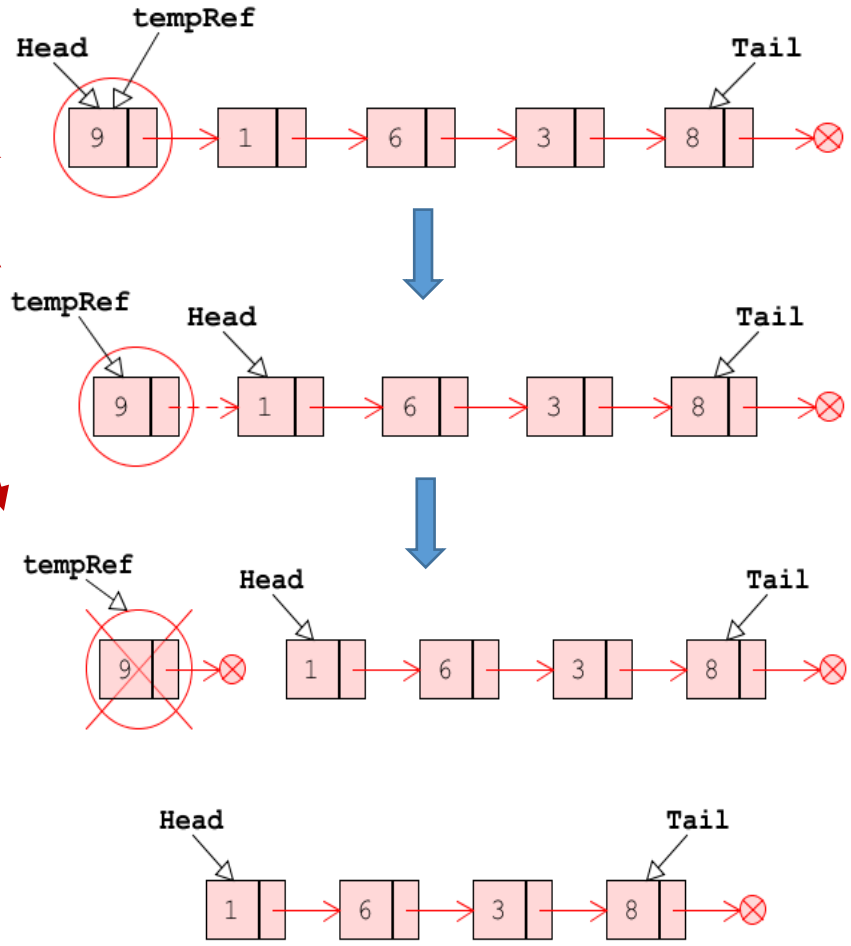
$tempRef \leftarrow head$

$head \leftarrow head.getNext()$

$tempRef \leftarrow null$

$size \leftarrow size - 1$

What is the run-time complexity (Big-Θ) of the algorithm?



Algorithm $removeFirst()$:

$if\ head == null\ then$

$\quad show\ error\ message\ "List\ is\ empty."$

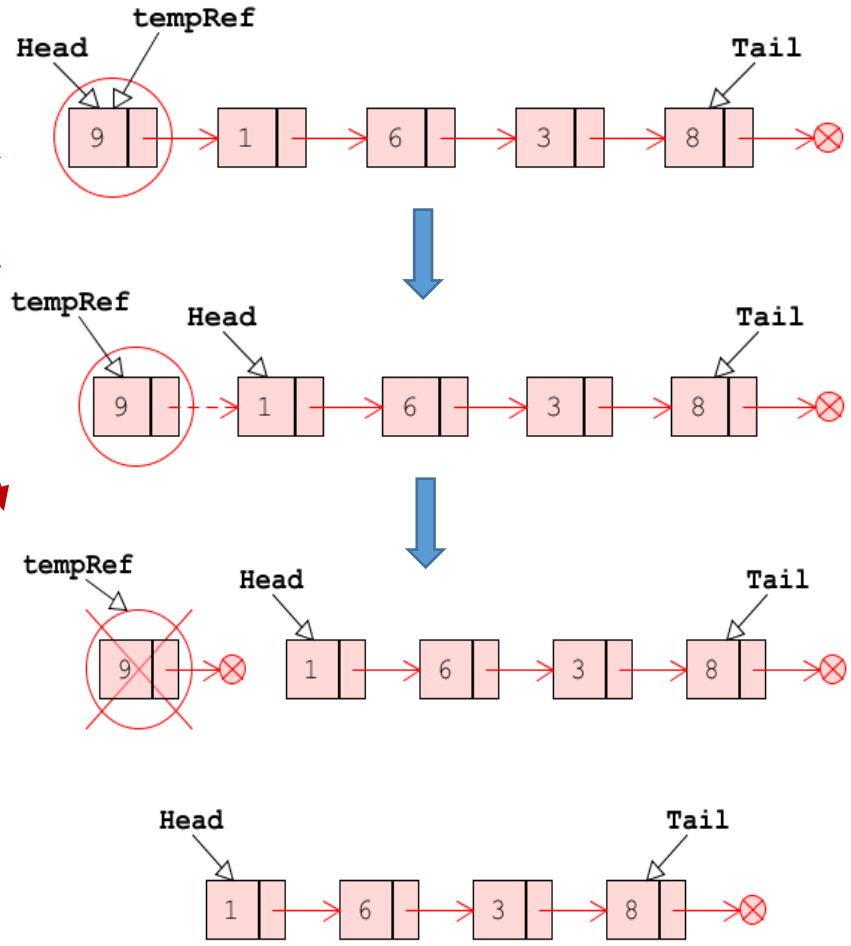
$tempRef \leftarrow head$

$head \leftarrow head.getNext()$

$tempRef \leftarrow null$

$size \leftarrow size - 1$

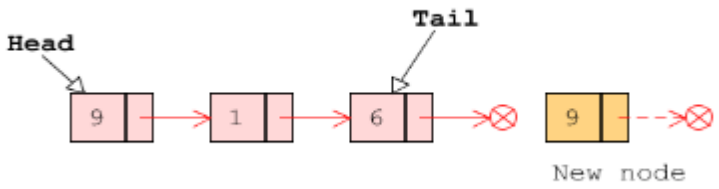
What is the run-time complexity (Big-Θ) of the algorithm? $\Theta(1)$



Linked List: Operations

- $addLast(newNode)$:Inserting a new node $newNode$ at the end of a linked list. The idea is to:
  i. Create a new node $newNode$;
  ii. Assign the new node's $next$ reference to reference to the $null$ object;
  iii. Set the $next$ reference of the $tail$ to reference to this new node $newNode$;
  iv. Assign the $tail$ reference itself to this new node;
  v. And finally, increment the $size$ of the list.

Linked List: Operations



Algorithm $addLast(newNode)$:

$newNode.setNext(null)$

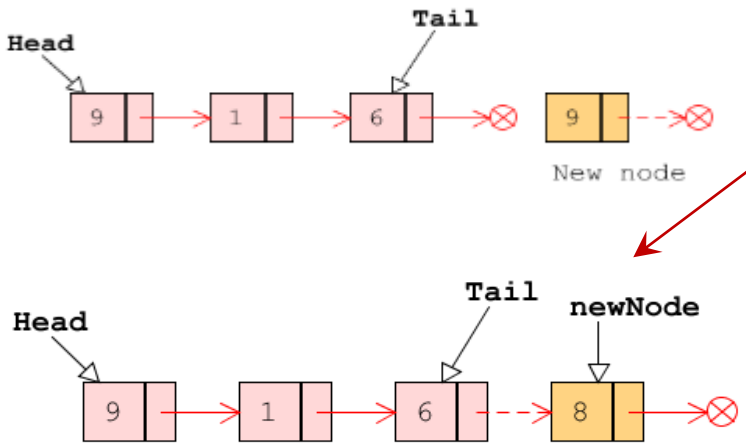
$tail.setNext(newNode)$

$tail \leftarrow newNode$

$newNode \leftarrow null$

$size \leftarrow size + 1$

Linked List: Operations



Algorithm $addLast(newNode)$:

$newNode.setNext(null)$

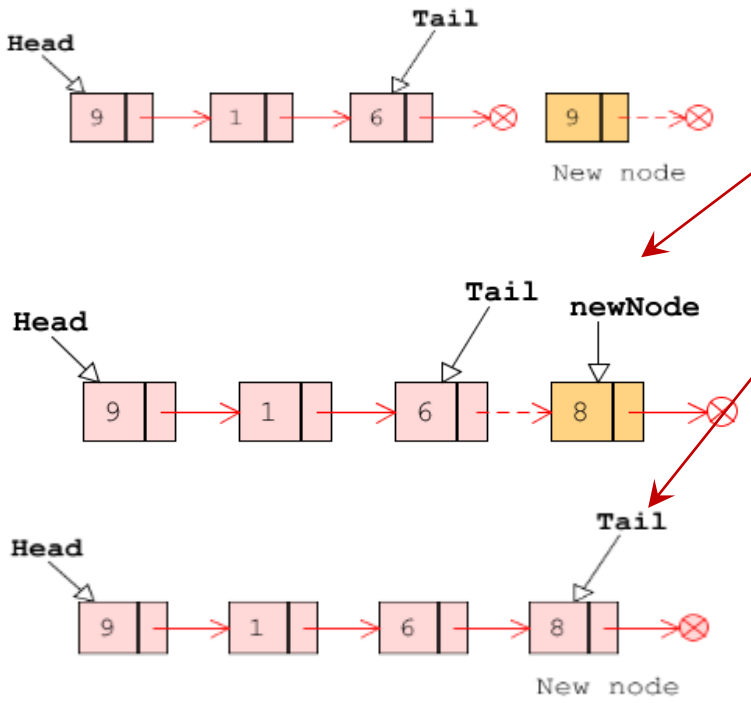
$tail.setNext(newNode)$

$tail \leftarrow newNode$

$newNode \leftarrow null$

$size \leftarrow size + 1$

Linked List: Operations



Algorithm $addLast(newNode)$:

$newNode.setNext(null)$

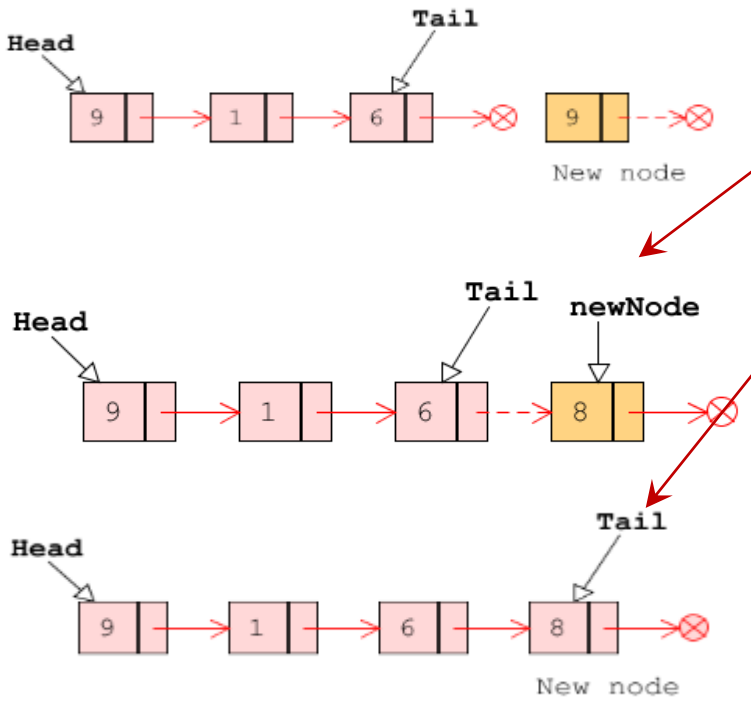
$tail.setNext(newNode)$

$tail \leftarrow newNode$

$newNode \leftarrow null$

$size \leftarrow size + 1$

Linked List: Operations



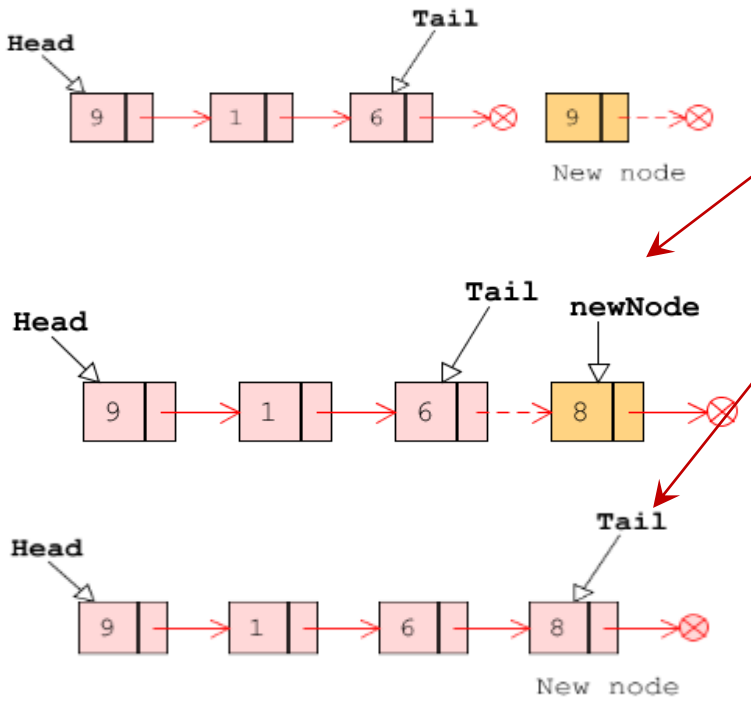
Algorithm $addLast(newNode)$:

$newNode.setNext(null)$
$tail.setNext(newNode)$

$tail \leftarrow newNode$
$newNode \leftarrow null$
$size \leftarrow size + 1$

What is the run-time complexity (Big-Θ) of the algorithm?

Linked List: Operations



Algorithm $addLast(newNode)$:

$newNode.setNext(null)$
$tail.setNext(newNode)$

$tail \leftarrow newNode$
$newNode \leftarrow null$
$size \leftarrow size + 1$

What is the run-time complexity (Big-Θ) of the algorithm? $\boldsymbol{\Theta(1)}$

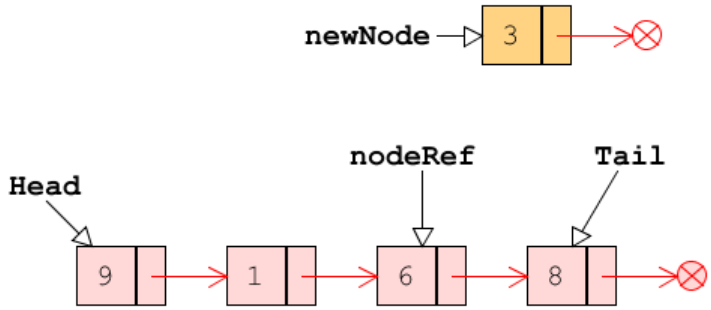
Linked List: Operations

- $removeLast()$:
- Unfortunately, remove last operation is not as easy as remove first operation. We will come back to discuss this operation in a while.

Linked List: Operations

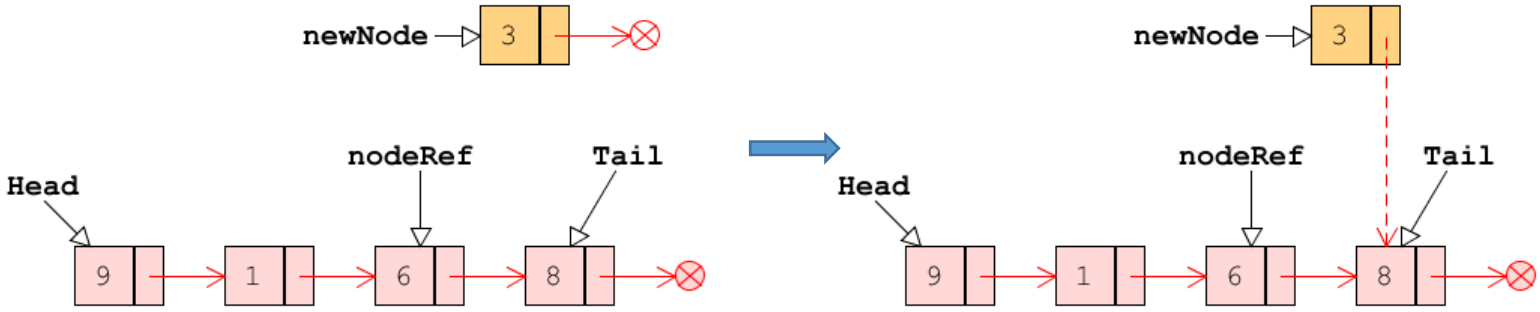
- $addAfter(nodeRef, newNode)$:Inserting a new node $newNode$ at the position after the $nodeRef$. The idea is to:
  i. Locate a position in the linked list where the new node is to be inserted. Use a node reference $nodeRef$ to reference the position.
  ii. Create a new node $newNode$;
  iii. Assign the new node's $next$ reference to reference to the object the $nodeRef.next$ is referencing to;
  iv. Set the $next$ reference of the $nodeRef$ to reference to this new node $newNode$;
  v. And finally, increment the $size$ of the list.

For example, add a newNode after nodeRef (second node).



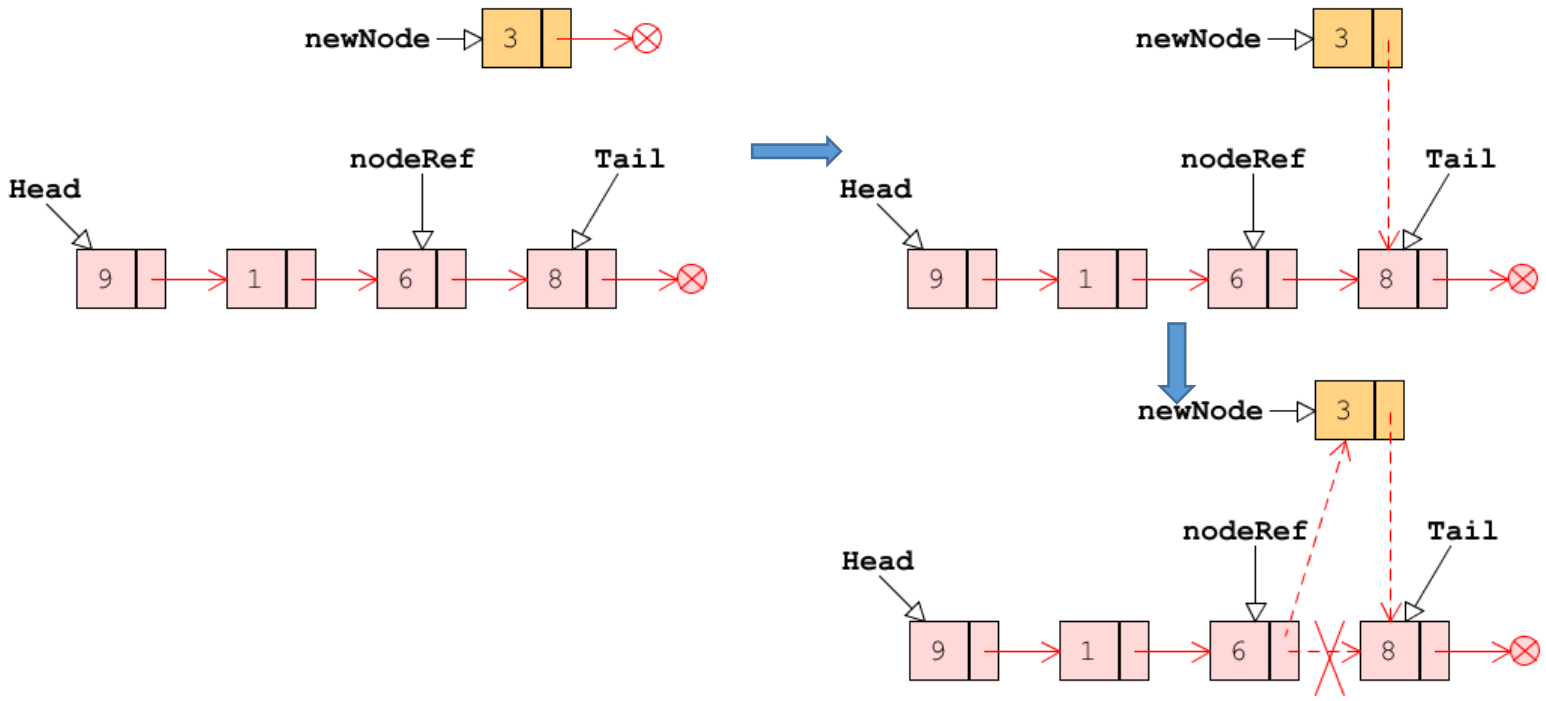
Algorithm $addAfter(nodeRef, newNode)$:
$newNode.setNext(nodeRef.next)$
$nodeRef.setNext(newNode)$
$size \leftarrow size + 1$

For example, add a newNode after nodeRef (second node).



Algorithm $addAfter(nodeRef, newNode)$:
$newNode.setNext(nodeRef.next)$
$nodeRef.setNext(newNode)$
$size \leftarrow size + 1$

For example, add a newNode after nodeRef (second node).



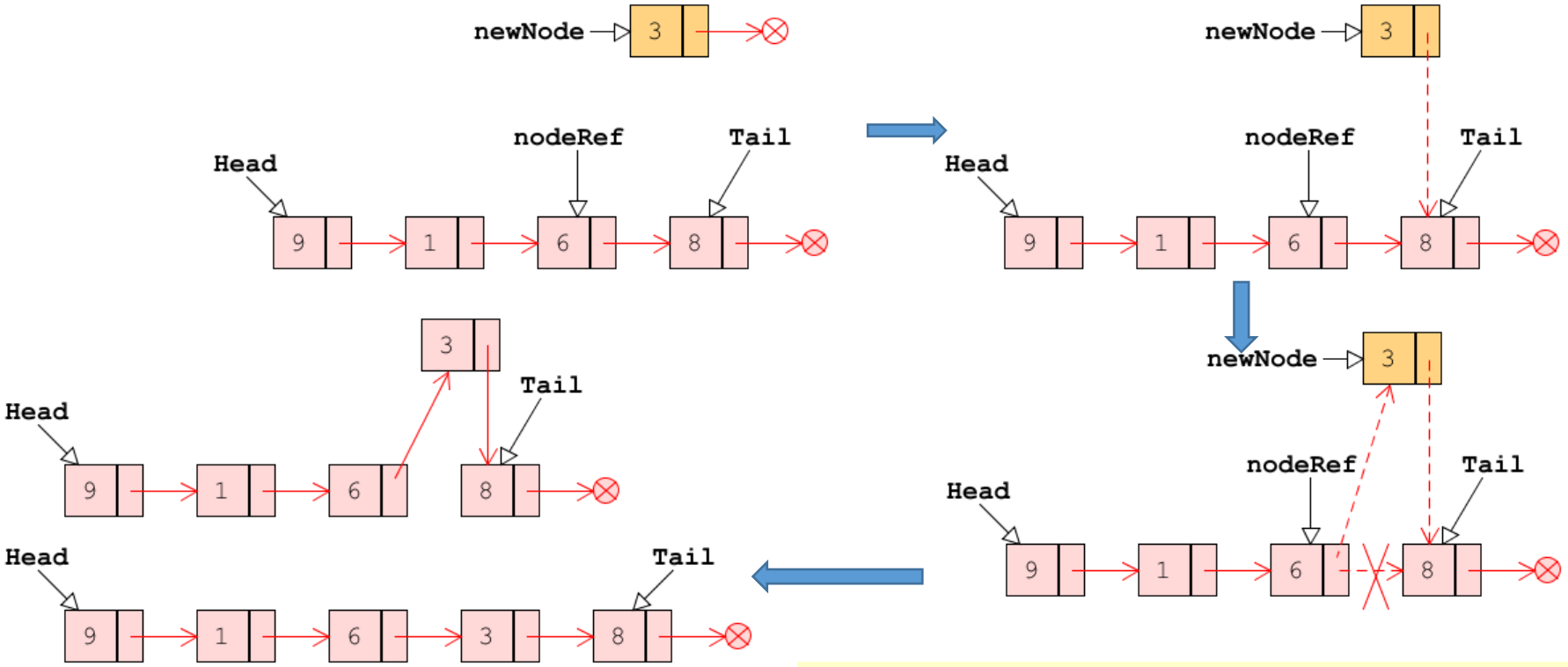
Algorithm $addAfter(nodeRef, newNode)$:

$newNode.setNext(nodeRef.next)$

$nodeRef.setNext(newNode)$

$size \leftarrow size + 1$

For example, add a newNode after nodeRef (second node).



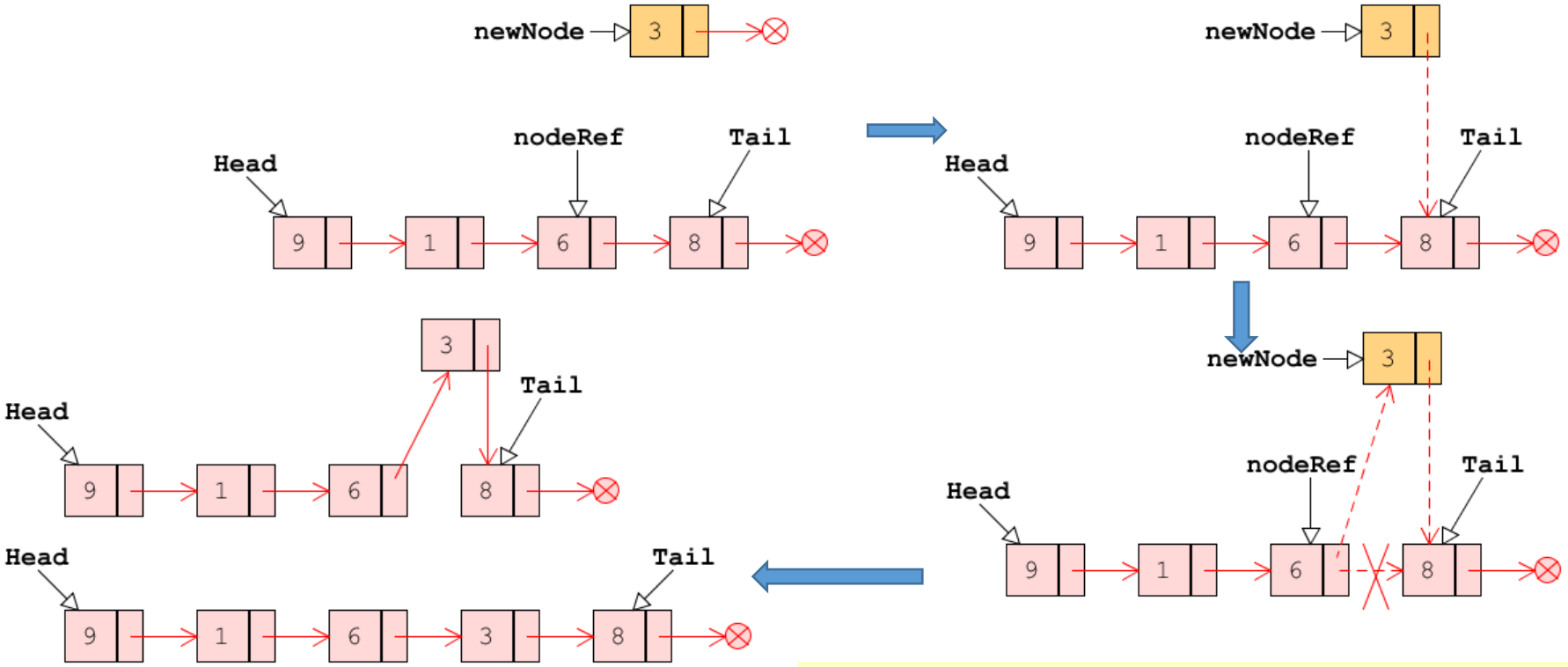
Algorithm $addAfter(nodeRef, newNode)$:

$newNode.setNext(nodeRef.next)$

$nodeRef.setNext(newNode)$

$size \leftarrow size + 1$

For example, add a newNode after nodeRef (second node).



What is the run-time complexity (Big-Θ) of the algorithm?

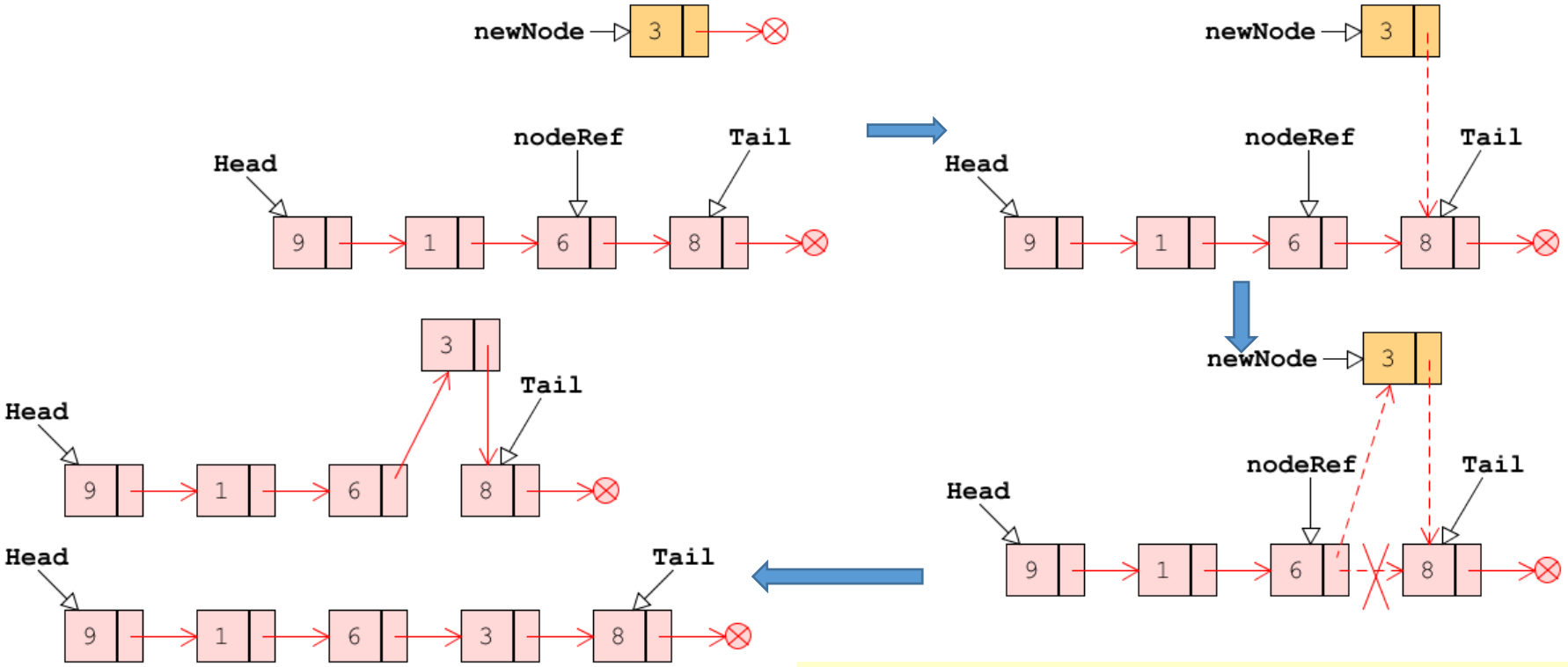
Algorithm $addAfter(nodeRef, newNode)$:

$newNode.setNext(nodeRef.next)$

$nodeRef.setNext(newNode)$

$size \leftarrow size + 1$

For example, add a newNode after nodeRef (second node).



What is the run-time complexity (Big-Θ) of the algorithm? $\boldsymbol{\Theta(1)\ or\ \Theta(n)}$**???**

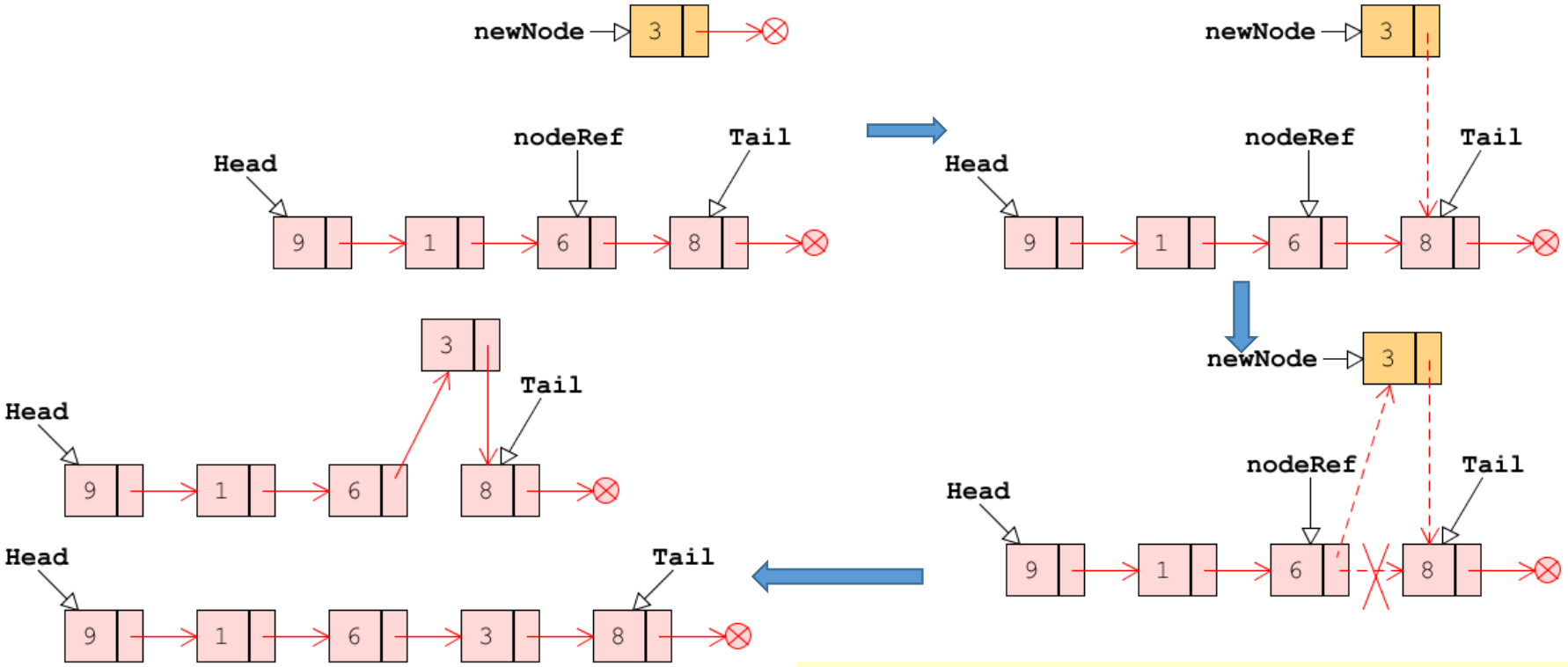
Algorithm $addAfter(nodeRef, newNode)$:

$newNode.setNext(nodeRef.next)$

$nodeRef.setNext(newNode)$

$size \leftarrow size + 1$

For example, add a newNode after nodeRef (second node).



What is the run-time complexity (Big-Θ) of the algorithm? $\boldsymbol{\Theta(1)\ or\ \Theta(n)}$**???**

Algorithm $addAfter(nodeRef, newNode)$:

$newNode.setNext(nodeRef.next)$

$nodeRef.setNext(newNode)$

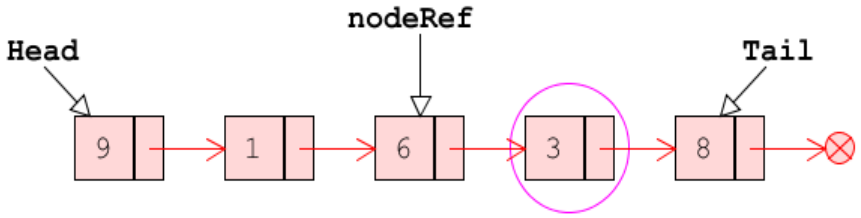
$size \leftarrow size + 1$

Will come back to this in a while.

Linked List: Operations

- $removeAfter(nodeRef, newNode)$:Removing a node at a position after a node reference $nodeRef$. The idea is to:
  i. Locate a position in the linked list where the new node is to be removed. Use a node reference $nodeRef$ to reference the position;
  ii. Set the reference $next$ of $nodeRef$ to reference to the object the $nodeRef.next.next$ is referencing to;
  iii. And finally, increment the $size$ of the list.

For example, delete a node after nodeRef (second node).



Algorithm $removeAfter(nodeRef)$:

$tempRef \leftarrow nodeRef.next$

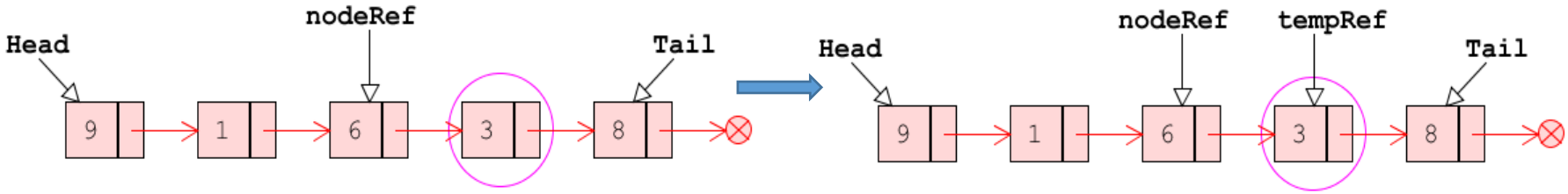
$nodeRef.setNext(tempRef.next)$

$tempRef.setNext(null)$

$nodeRef \leftarrow null; tempRef \leftarrow null$

$size \leftarrow size - 1$

For example, delete a node after nodeRef (second node).



Algorithm $removeAfter(nodeRef)$:

$tempRef \leftarrow nodeRef.next$

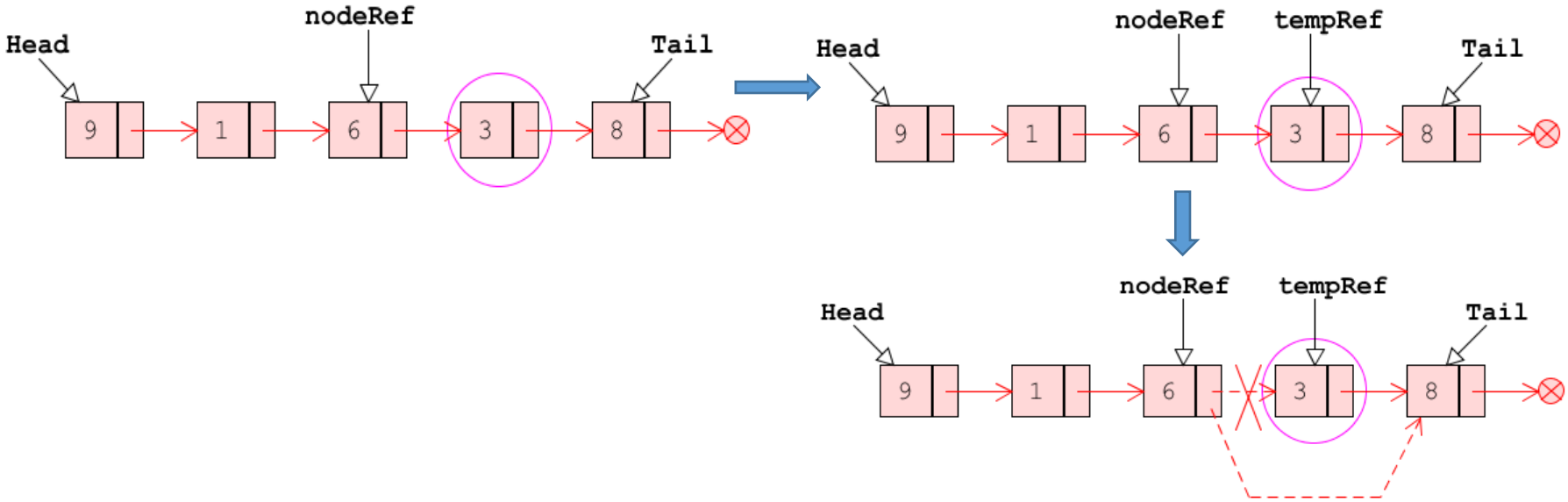
$nodeRef.setNext(tempRef.next)$

$tempRef.setNext(null)$

$nodeRef \leftarrow null; tempRef \leftarrow null$

$size \leftarrow size - 1$

For example, delete a node after nodeRef (second node).



Algorithm $removeAfter(nodeRef)$:

$tempRef \leftarrow nodeRef.next$

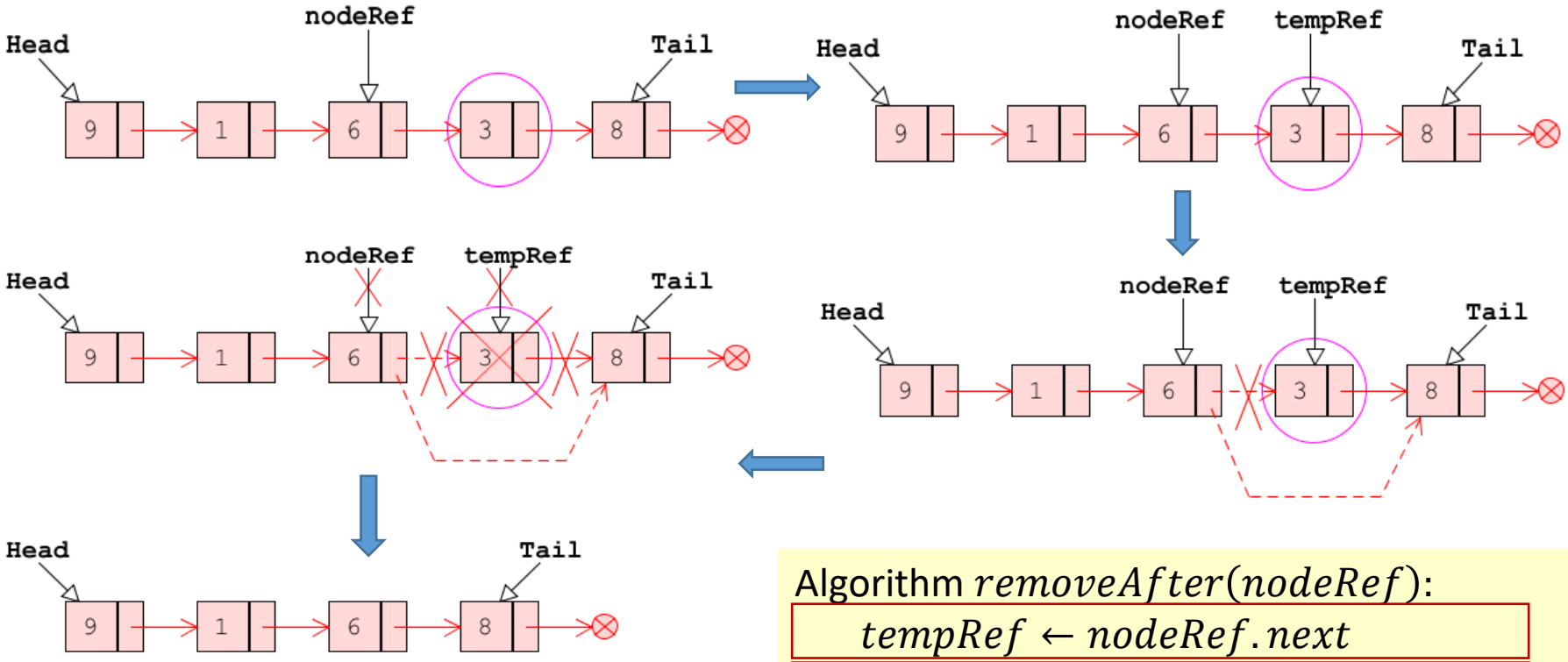
$nodeRef.setNext(tempRef.next)$

$tempRef.setNext(null)$

$nodeRef \leftarrow null; tempRef \leftarrow null$

$size \leftarrow size - 1$

For example, delete a node after nodeRef (second node).



Algorithm $removeAfter(nodeRef)$:

$tempRef \leftarrow nodeRef.next$

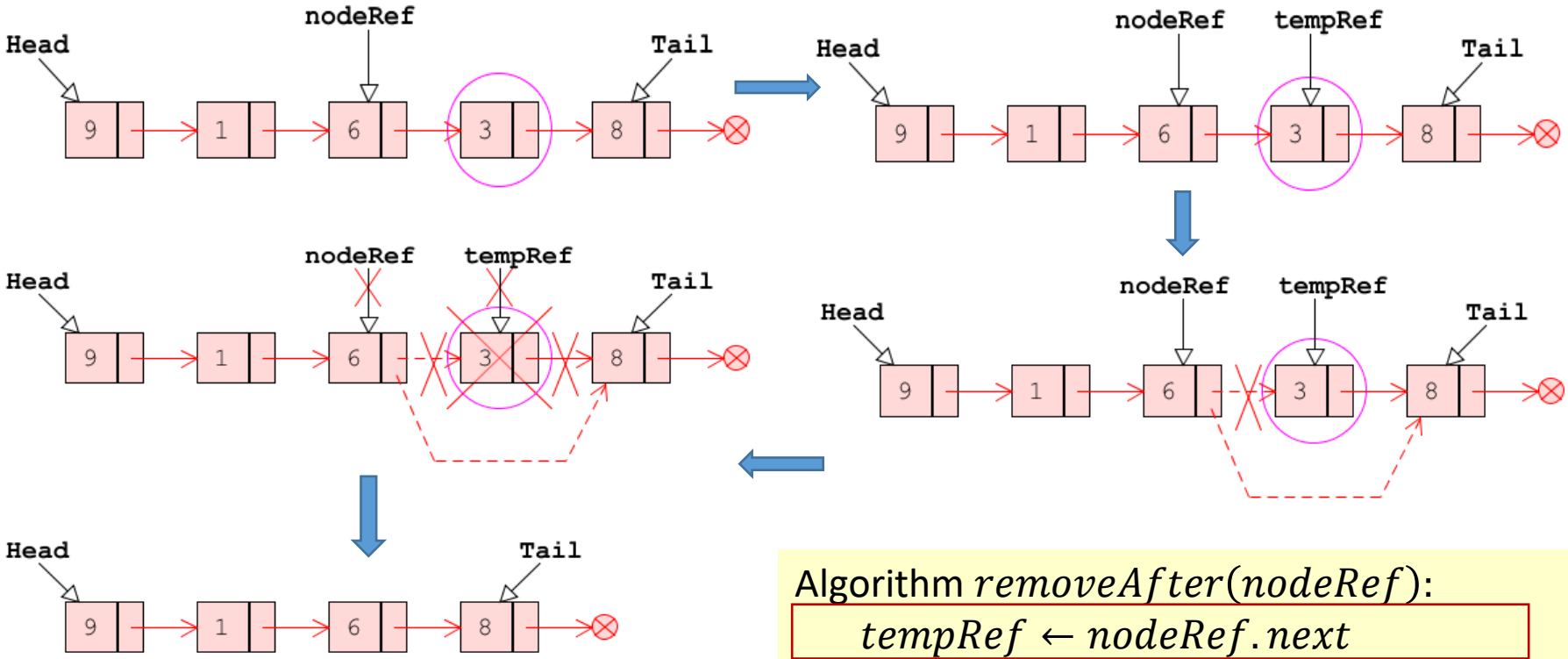
$nodeRef.setNext(tempRef.next)$

$tempRef.setNext(null)$

$nodeRef \leftarrow null; tempRef \leftarrow null$

$size \leftarrow size - 1$

For example, delete a node after nodeRef (second node).



What is the run-time complexity (Big-Θ) of the algorithm?

Algorithm $removeAfter(nodeRef)$:

$tempRef \leftarrow nodeRef.next$

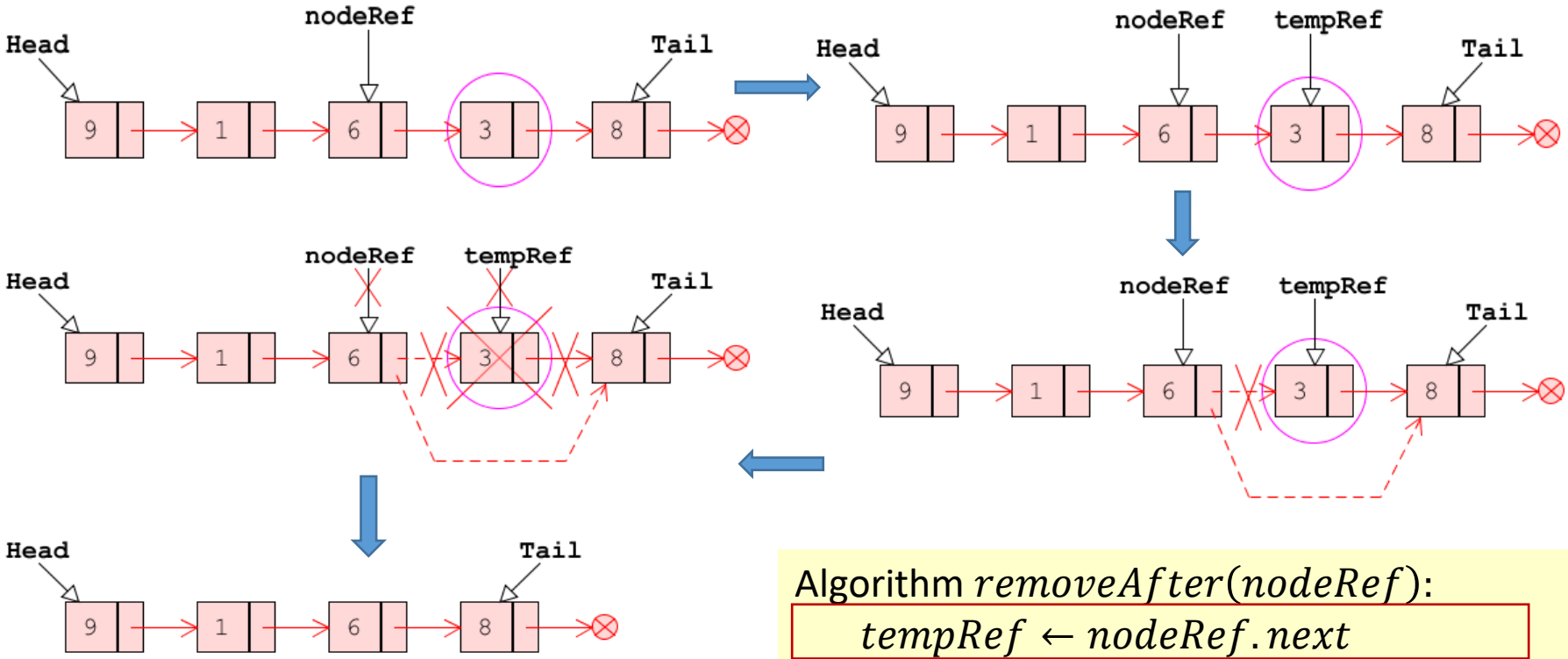
$nodeRef.setNext(tempRef.next)$

$tempRef.setNext(null)$

$nodeRef \leftarrow null; tempRef \leftarrow null$

$size \leftarrow size - 1$

For example, delete a node after nodeRef (second node).



What is the run-time complexity (Big-Θ) of the algorithm? **$\Theta(1)$ *or* $\Theta(n)$???**

Algorithm $removeAfter(nodeRef)$:

$tempRef \leftarrow nodeRef.next$

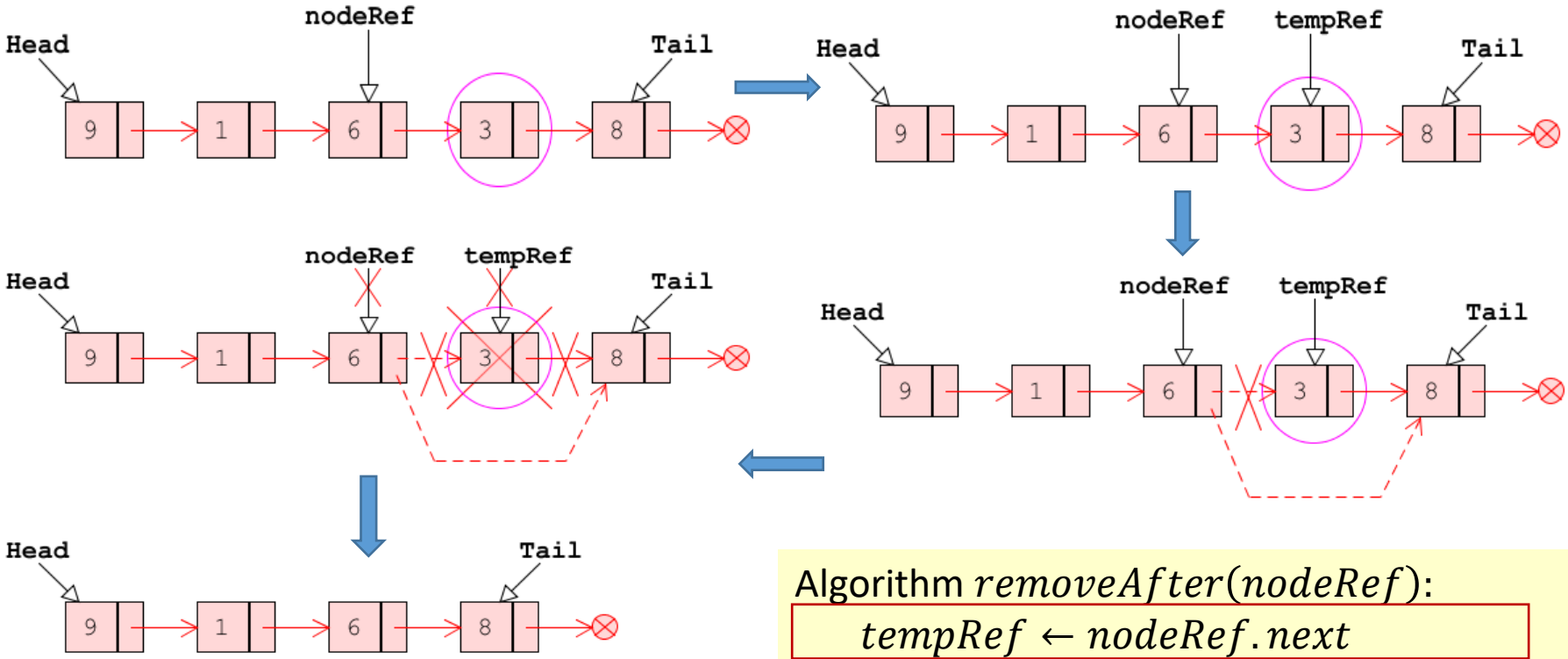
$nodeRef.setNext(tempRef.next)$

$tempRef.setNext(null)$

$nodeRef \leftarrow null; tempRef \leftarrow null$

$size \leftarrow size - 1$

For example, delete a node after nodeRef (second node).



What is the run-time complexity (Big-Θ) of the algorithm? **$\Theta(1)$ *or* $\Theta(n)$???**

Algorithm $removeAfter(nodeRef)$:

$tempRef \leftarrow nodeRef.next$

$nodeRef.setNext(tempRef.next)$

$tempRef.setNext(null)$

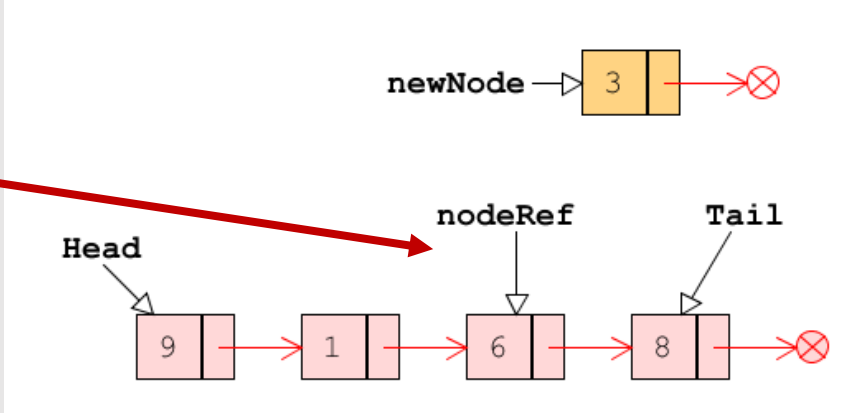
$nodeRef \leftarrow null; tempRef \leftarrow null$

$size \leftarrow size - 1$

Will come back to this in a while.

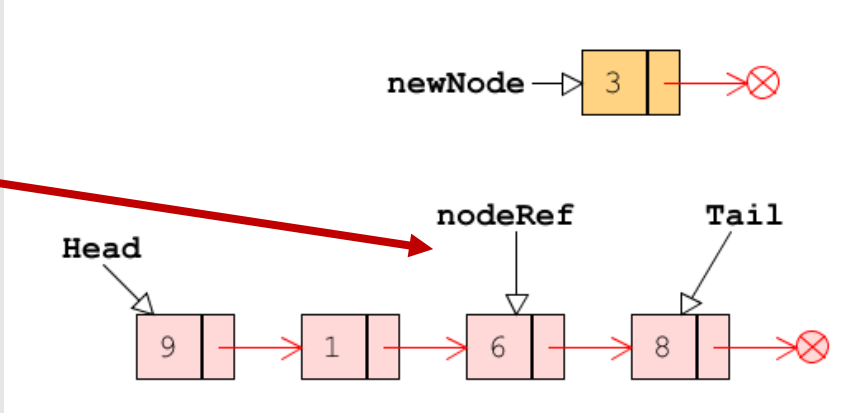
Linked List: Operations

Question: How to get $nodeRef$ to reference to the correct node?



Linked List: Operations

Question: How to get $nodeRef$ to reference to the correct node?



List traversal – An operation that visits each node in a list starting from the beginning (head) of the list to the end (tail) of the list.

Linked List: Operations

Algorithm $listTraversal()$:

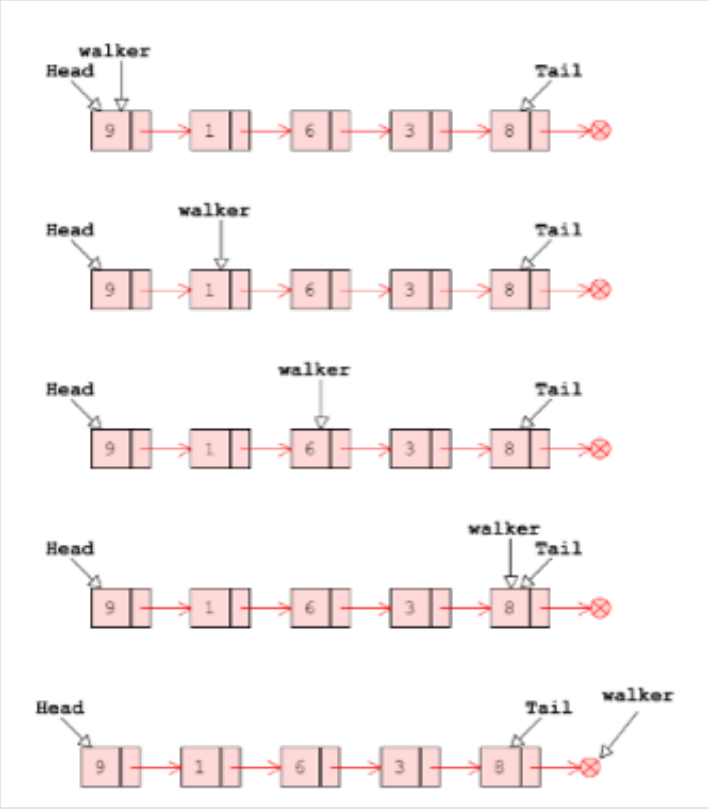
$walker \leftarrow head$

$while\ walker\ is\ not\ null\ \{$

$visit\ node$

$walker \leftarrow walker.next$

$\}$



Linked List: Operations

Algorithm $listTraversal()$:

$walker \leftarrow head$

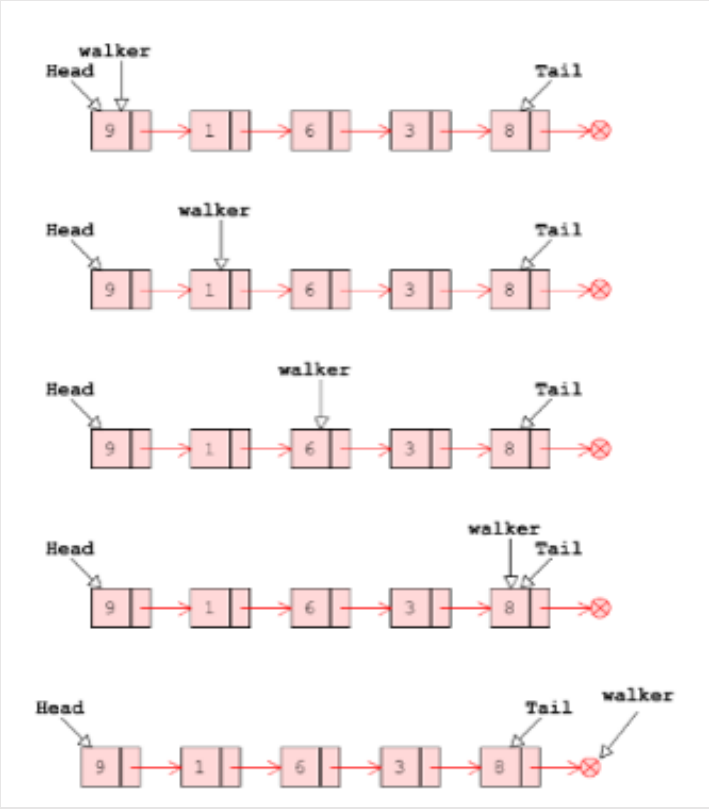
$while\ walker\ is\ not\ null\ \{$

$visit\ node$

$walker \leftarrow walker.next$

$\}$

What is the run-time complexity (Big-Θ) of the algorithm?



Linked List: Operations

Algorithm $listTraversal()$:

$walker \leftarrow head$

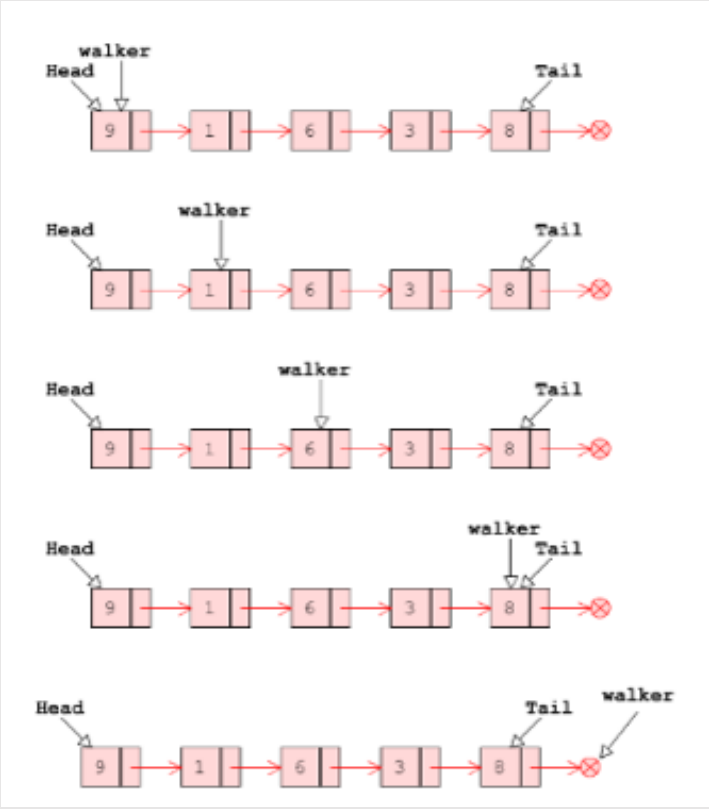
$while\ walker\ is\ not\ null\ \{$

$visit\ node$

$walker \leftarrow walker.next$

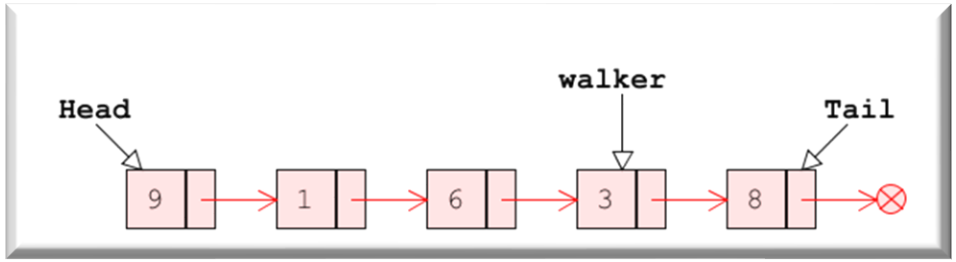
$\}$

What is the run-time complexity (Big-Θ) of the algorithm? $\boldsymbol{\Theta(n)}$



Linked List: Operations

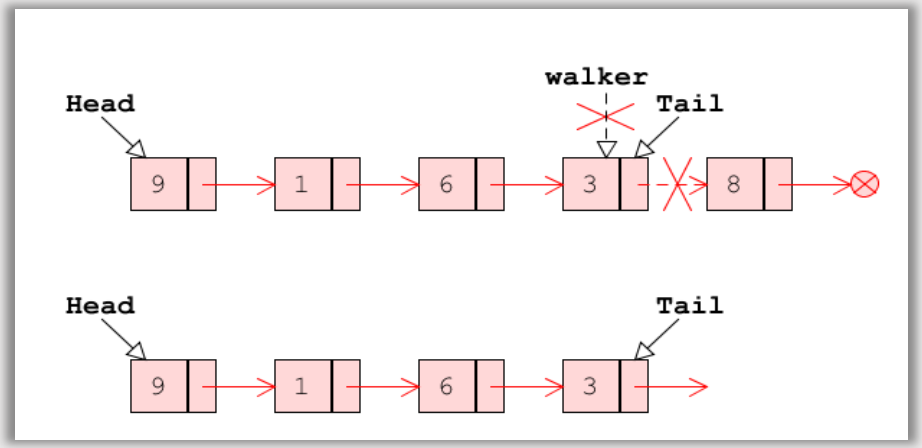
- Back to $revomeLast()$ operation.



- To remove the last node in a linked list, we first need to traverse the list until the node one just before the last node ($tail$).

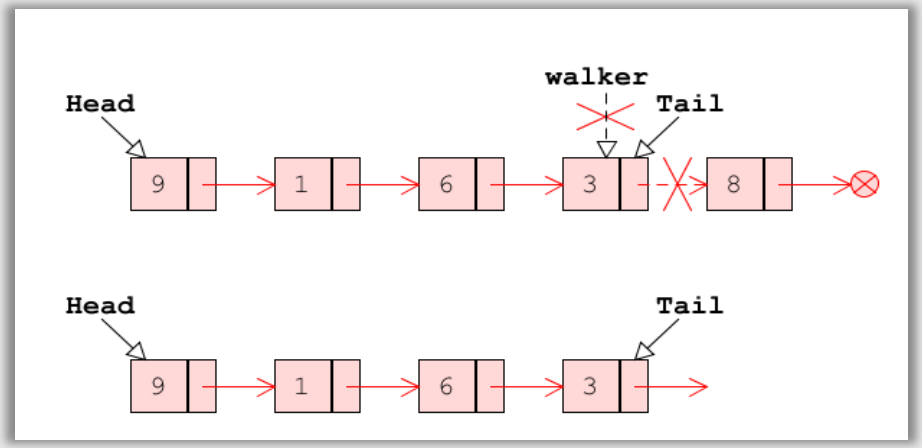
Linked List: Operations

- After we have the $walker$ reference to the position of the node, we can then use the $removeAfter(walker)$ to remove the last node.



Linked List: Operations

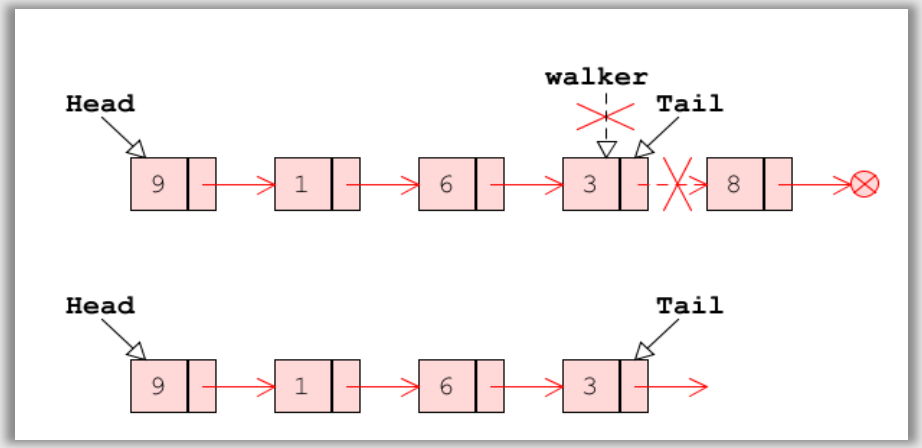
- After we have the $walker$ reference to the position of the node, we can then use the $removeAfter(walker)$ to remove the last node.



What is the run-time complexity (Big-Θ) of the algorithm?

Linked List: Operations

- After we have the $walker$ reference to the position of the node, we can then use the $removeAfter(walker)$ to remove the last node.

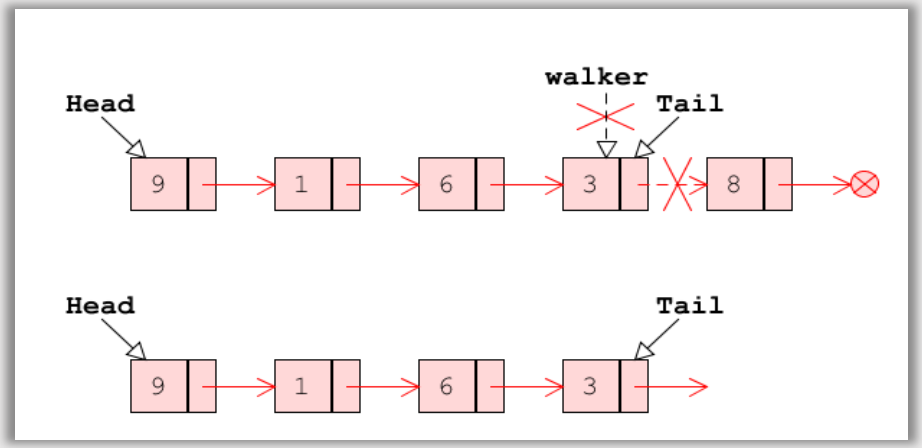


What is the run-time complexity (Big-Θ) of the algorithm?

$\boldsymbol{\Theta(n) or\ \Theta(1)?}$

Linked List: Operations

- After we have the $walker$ reference to the position of the node, we can then use the $removeAfter(walker)$ to remove the last node.



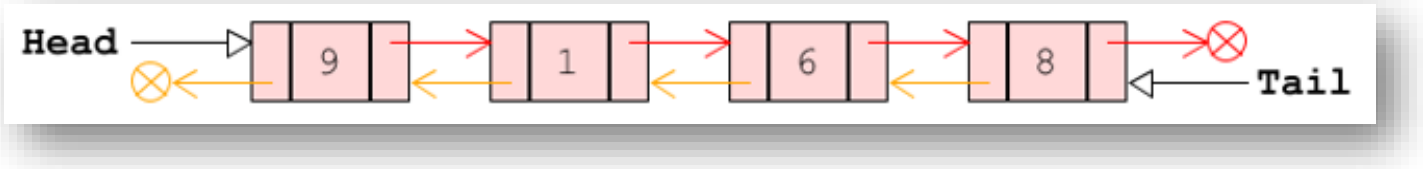
What is the run-time complexity (Big-Θ) of the algorithm?

$$\boldsymbol{\Theta(n) or\ \Theta(1)?}$$

The run-time complexity of the algorithm removeAfter(walker, nodePtr) or removeLast(walker, nodePtr) is constant; that is $\Theta(1)$. However, the overall process of removing a node from a linked list is $\Theta(n)$.

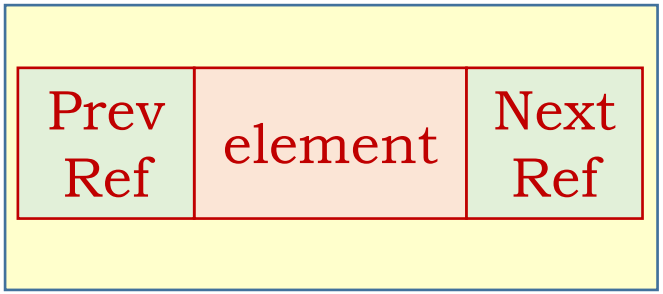
Linked List: Doubly linked list

- A list may also have a pointer back to the previous node in the list. This list is known as Doubly Linked List.



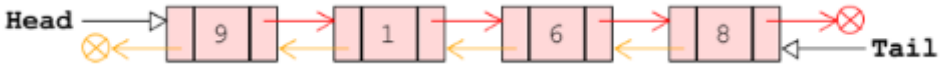
- The head and tail pointers are maintained to point to the first and last elements of the list.

```
public class DNode {
  private String element;
  private DNode prev, next;

  public DNode(String s, DNode p, DNode n) {
    element = s;
    prev = p;
    next = n;  }

  public String getElement() {
    return element;  }
  public DNode getPrev() { return prev; }
  public DNode getNext() { return next; }
  public void setElement(String newElem) {
    element = newElem;  }
  public void setPrev(DNode newPrev) {
    prev = newPrev;  }
  public void setNext(DNode newNext) {
    next = newNext;  }
}
```



Doubly Linked List - Operations

- Inserting or removing elements at either end of a doubly linked list is straight-forward to do.
- With the previous link (reference), the need to traverse the list from the beginning to the node just before the tail become unnecessary. This simplifies the operation to remove node at the end of doubly linked list.



Algorithm $removeLast()$:

$if\ size == 0\ then$

$show\ error\ message$ "The list is empty."

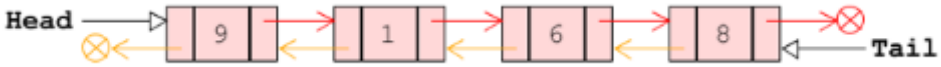
$tempRef \leftarrow tail.getPrev()$

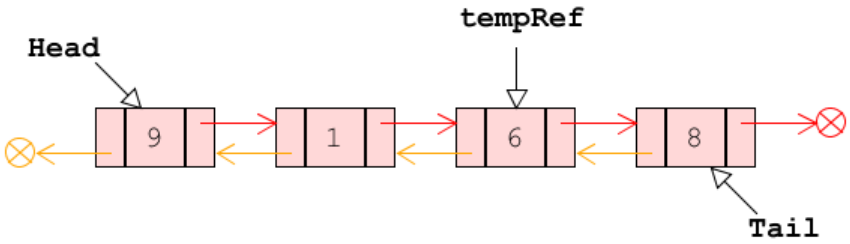
$tail.prev \leftarrow null$

$tail \leftarrow tempRef$

$tail.setNext(null)$

$size \leftarrow size - 1$





Algorithm $removeLast()$:

$if\ size == 0\ then$

$show\ error\ message$ "The list is empty."

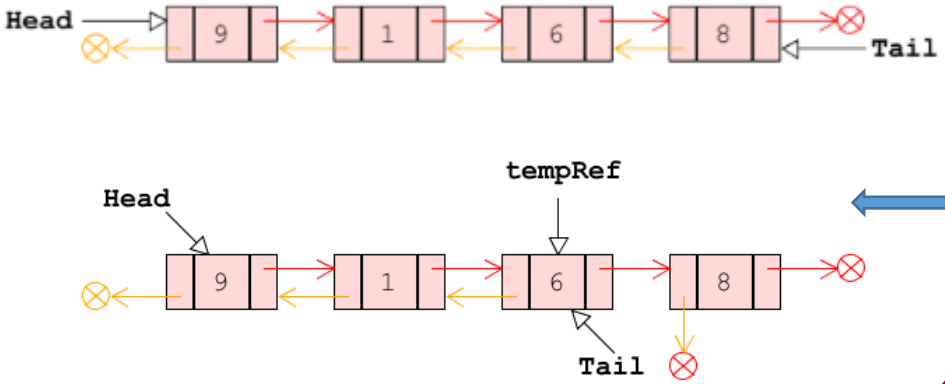
$tempRef \leftarrow tail.getPrev()$

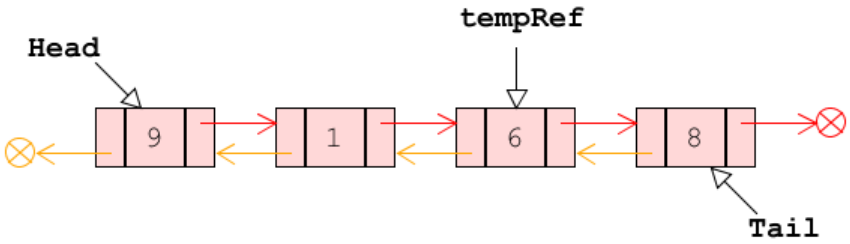
$tail.prev \leftarrow null$

$tail \leftarrow tempRef$

$tail.setNext(null)$

$size \leftarrow size - 1$





Algorithm $removeLast()$:

if $size == 0$ *then*

show error message "The list is empty."

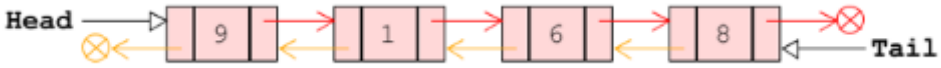
$tempRef \leftarrow tail.getPrev()$

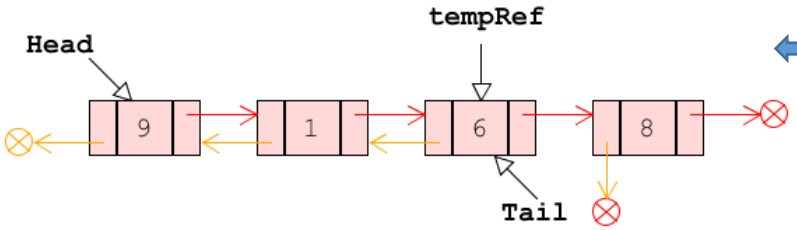
$tail.prev \leftarrow null$

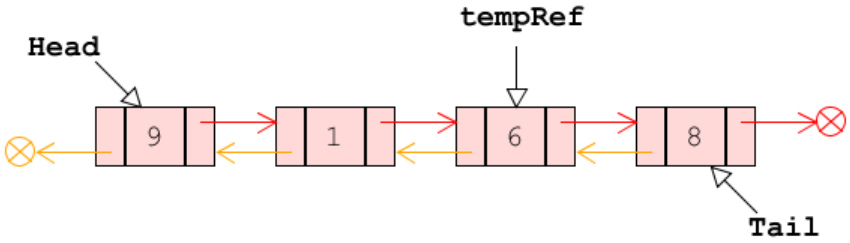
$tail \leftarrow tempRef$

$tail.setNext(null)$

$size \leftarrow size - 1$







Algorithm *removeLast*():

if size == 0 *then*

show error message "The list is empty."

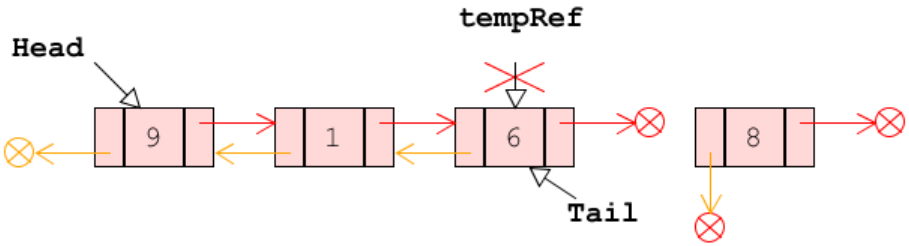
tempRef ← *tail.getPrev*()

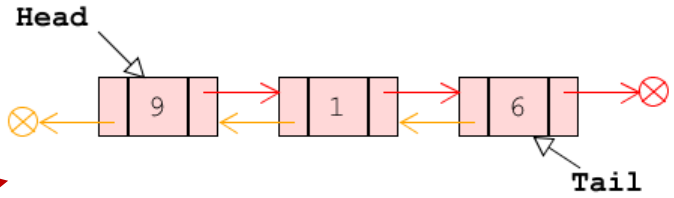
tail.prev ← *null*

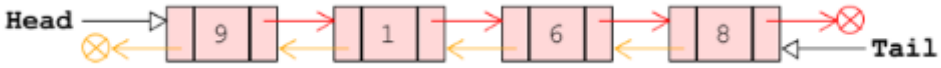
tail ← *tempRef*

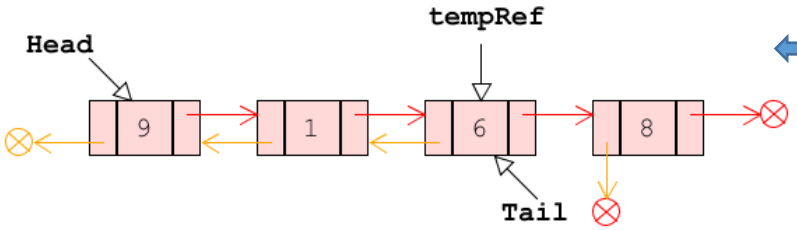
tail.setNext(*null*)

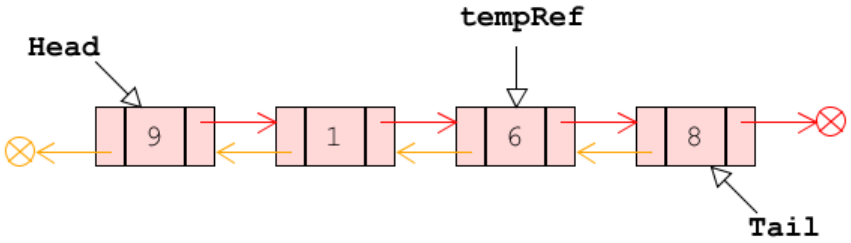
size ← *size* − 1











Algorithm *removeLast*():

if size == 0 *then*

show error message "The list is empty."

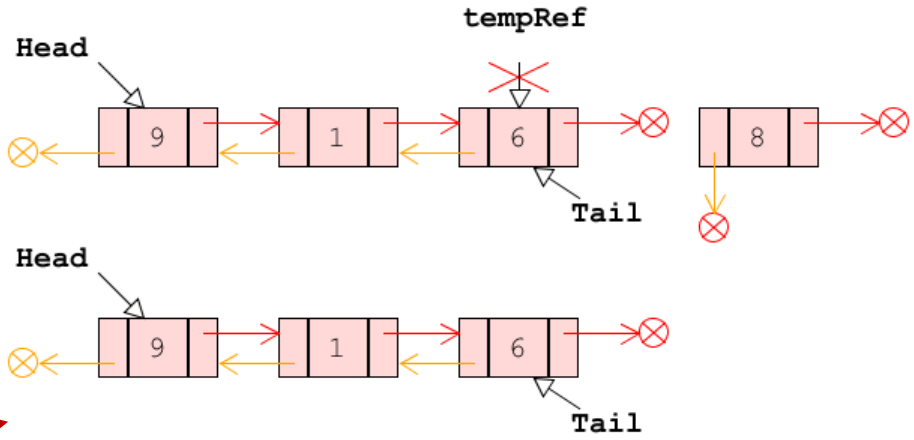
tempRef ← *tail.getPrev*()

tail.prev ← *null*

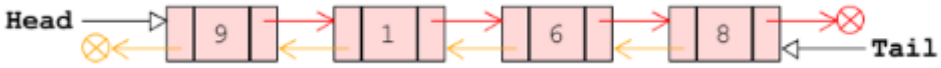
tail ← *tempRef*

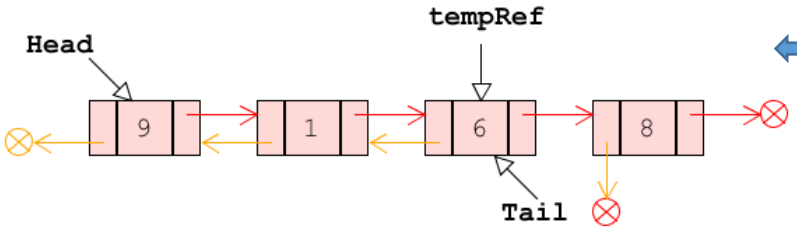
tail.setNext(*null*)

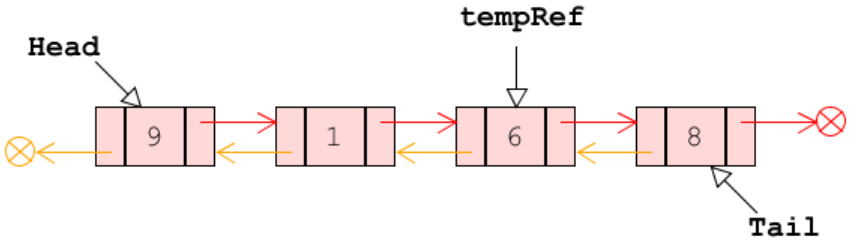
size ← *size* − 1



What is the run-time complexity of the removeLast() method?







Algorithm *removeLast*():

if size == 0 *then*

show error message "The list is empty."

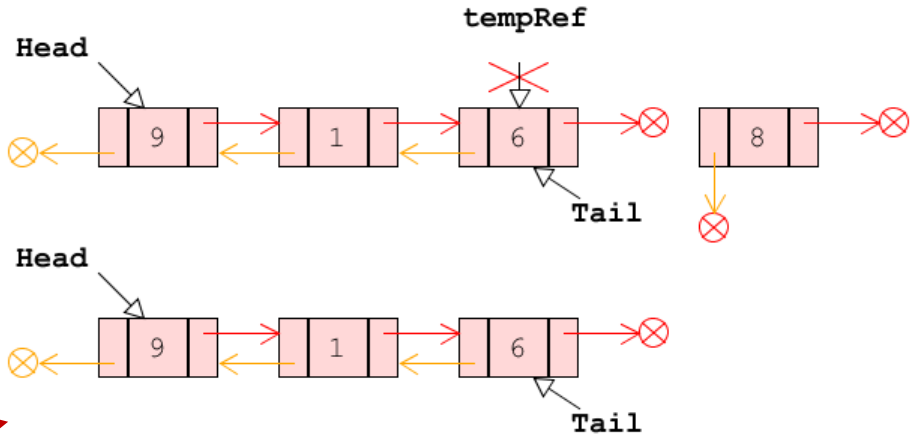
tempRef ← *tail.getPrev*()

tail.prev ← *null*

tail ← *tempRef*

tail.setNext(*null*)

size ← *size* − 1



What is the run-time complexity of the removeLast() method? $\boldsymbol{\Theta(1)}$

Doubly Linked List - Operations

- Do the following as exercises:
  - addFirst(newNode)
  - addLast(newNode)
  - addAfter(tempRef, newNode)
  - addBefore(tempRef, newNode)
  - Remove(tempRef) // Note: this is not the
    // same as removeAfter()
    // or removeBefore()

Field names

	PatientID	Surname	Firstname	Sex	DOB	Age	Town	Test	Practice	GP
records	1	Poole	Janet	F	01/05/1967	43	Newcastle	✓	West Road	Barnie
	2	Robinson	Christopher	M	23/11/1985	24	Durham	✓	East Gate	Priestley

Records

Records (Structures)

Records (Structures)

- A record is a data structure consisting of a fixed number of items
- Unlike an array, the elements in a record may be of differing types and are named.
- E.g., type person = record

  name: string
  age: integer
  height: real
  female: Boolean
  children: array[1:10] of string

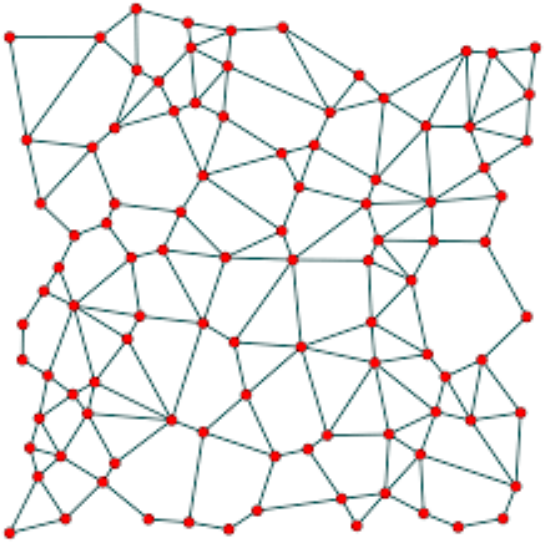
name	age	height	female	Children

Records (Structures)

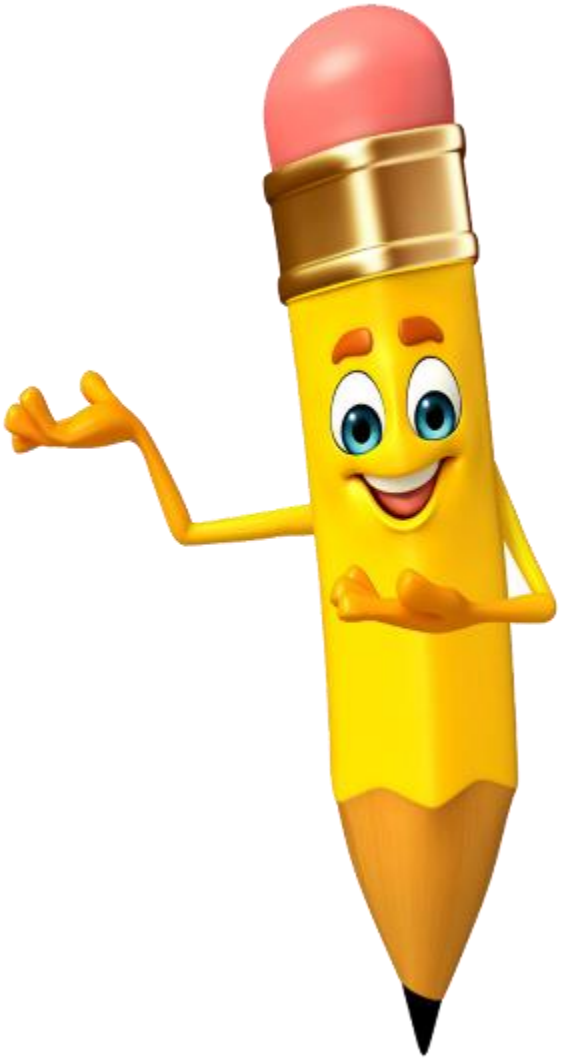
- An array may appear as a field in a record
- Records may appear as elements of an array
  - E.g., staff: array[1..50] of person
- Records may be implemented using static structure like array or dynamic structure like pointers (references).
- A dynamic-structure implemented records are addressed by a pointer (reference)
  - E.g., type boss = ^person declares boss as a type of object person.

Records (Structures)

- Fields of a record are accessible via the field name
  - E.g., staff[5].age, boss^.name



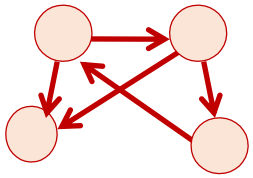
Graphs



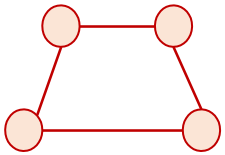
Graphs

Graphs

- A graph is a set of nodes joined by edges
- A graph may be directed or undirected

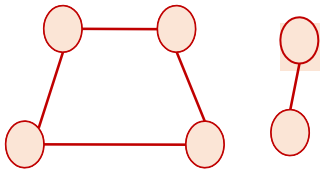


directed



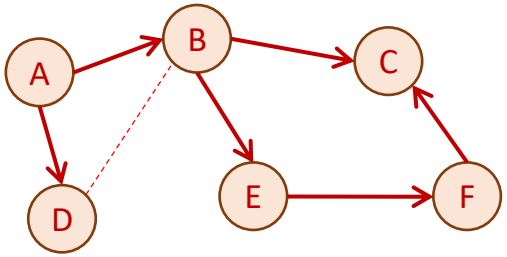
undirected

- A graph is connected if you can get from any node to any other node by following edges



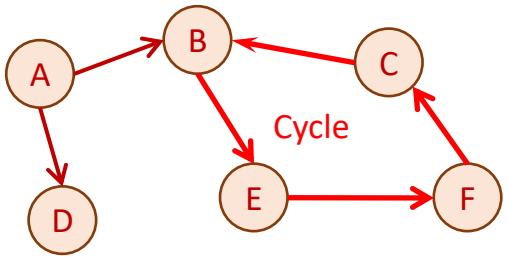
Graphs

- A directed graph is strongly connected if you can get from any node to any other node by following directed edges

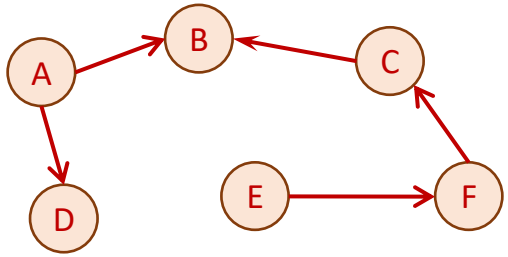


Graphs

- An acyclic graph is one in which there is no path from any node back to the same node that does not involve retracing edges



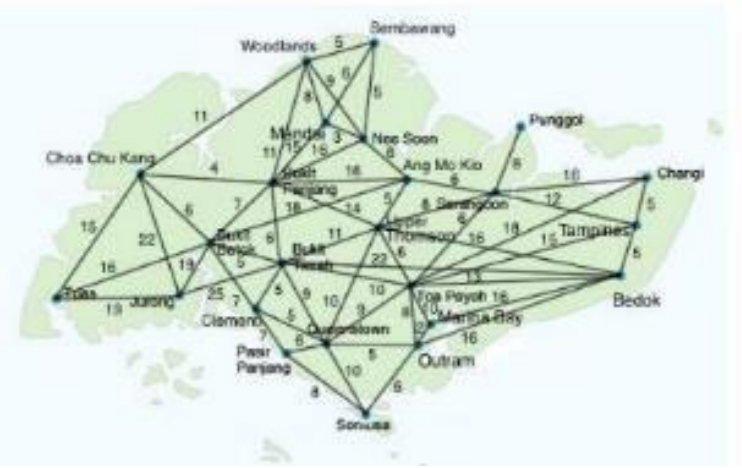
Cyclic

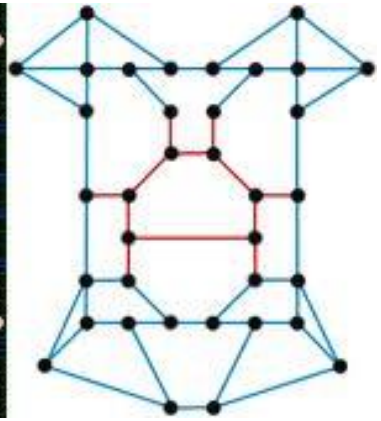


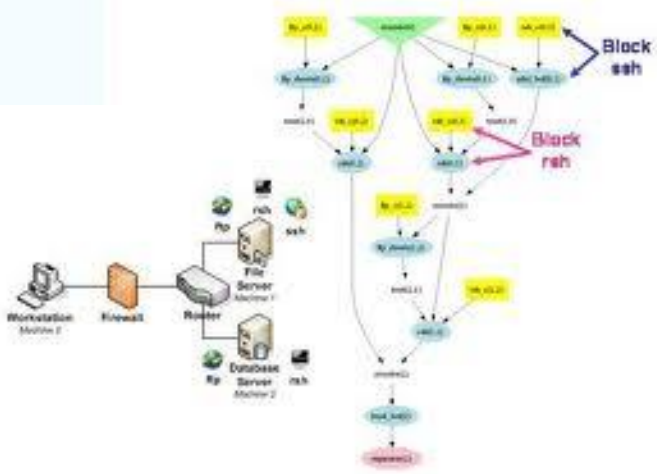
Acyclic

Graph

Possible applications:





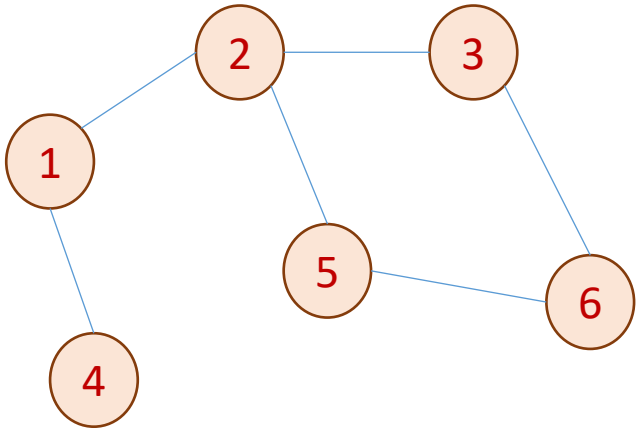


Graphs

- Adjacency matrix
  - Represent a graph G by showing the relationship between each pair of vertices of the graph.
  - '1' or 'T' is used to indicate if the two vertices are connected by an edge (that is, adjacent to each other), otherwise, '0' or 'F'.
- In undirected weighted graph: if $Adj[i][j] = 'T'$ then $Adj[j][i] = 'T'$

Graphs

$O(n^2)$

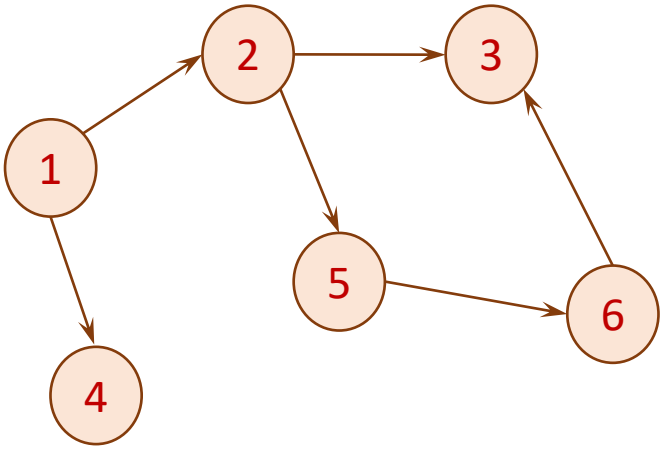


A	1	2	3	4	5	6
1	F	T	F	T	F	F
2	T	F	T	F	T	F
3	F	T	F	F	F	T
4	T	F	F	F	F	F
5	F	T	F	F	F	T
6	F	F	T	F	T	F

$$A = \begin{pmatrix} F & T & F & T & F & F \\ T & F & T & F & T & F \\ F & T & F & F & F & T \\ T & F & F & F & F & F \\ F & T & F & F & F & T \\ F & F & T & F & T & F \end{pmatrix}$$

Graphs

$O(n^2)$



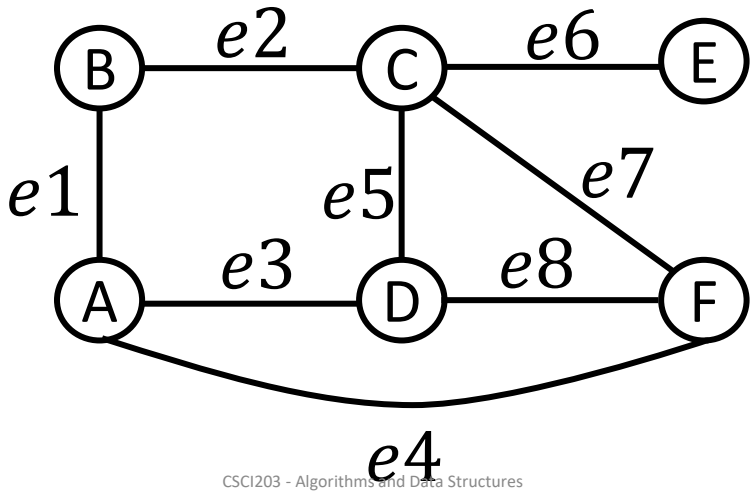
Directed graphs

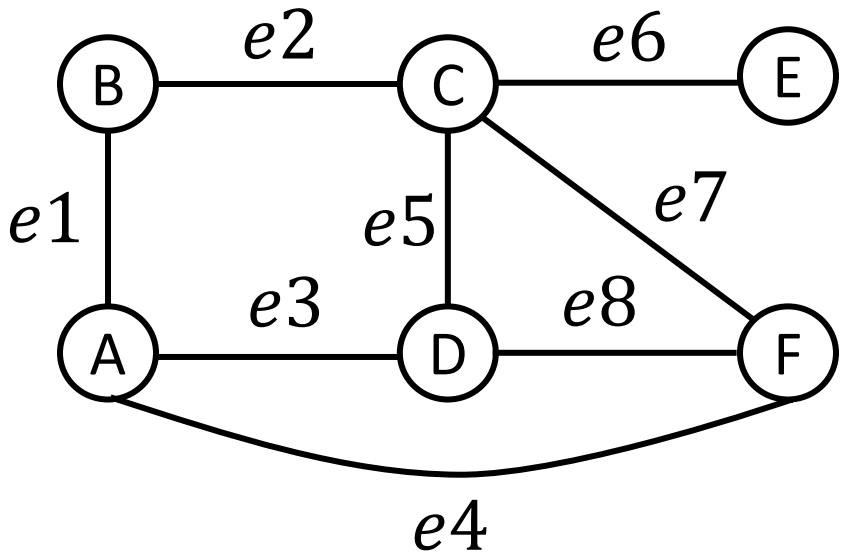
A	1	2	3	4	5	6
1	F	T	F	T	F	F
2	F	F	T	F	T	F
3	F	F	F	F	F	F
4	F	F	F	F	F	F
5	F	F	F	F	F	T
6	F	F	T	F	F	F

$$A = \begin{pmatrix} F & T & F & T & F & F \\ F & F & T & F & T & F \\ F & F & F & F & F & F \\ F & F & F & F & F & F \\ F & F & F & F & F & T \\ F & F & T & F & F & F \end{pmatrix}$$

Graphs

- Incidence matrix
  - Represent a graph G by showing the relationship between every vertex and every edge.
  - '1' or 'T' is used to indicate a vertex is incident to the edge, '0' or 'F' otherwise.
  - Not suitable for a graph with self-loop.





M:	e1	e2	e3	e4	e5	e6	e7	e8
A	1	0	1	1	0	0	0	0
B	1	1	0	0	0	0	0	0
C	0	1	0	0	1	1	1	0
D	0	0	1	0	1	0	0	1
E	0	0	0	1	0	0	1	1
F	0	0	0	0	0	1	0	0

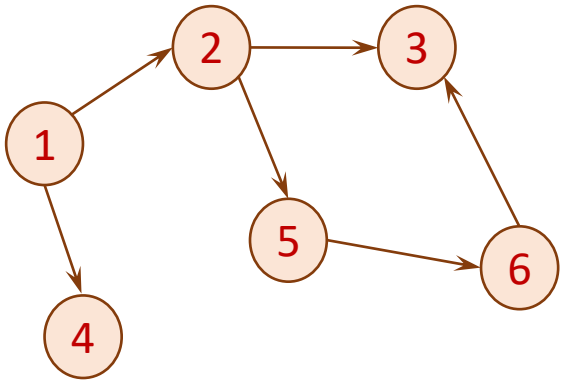
Preferred

M:	e1	e2	e3	e4	e5	e6	e7	e8
A	1	0	1	1	0	0	0	0
B	1	1	0	0	0	0	0	0
C	0	1	0	0	1	1	1	0
D	0	0	1	0	1	0	0	1
E	0	0	0	1	0	0	1	1
F	0	0	0	0	0	1	0	0

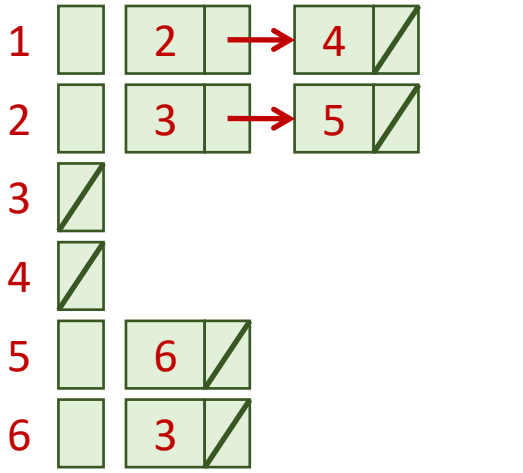
$$A = \begin{pmatrix} 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Graphs

- Edge list
  - type listgraph: array[1..nodes] of record
    - value: stuff
    - neighbours: list

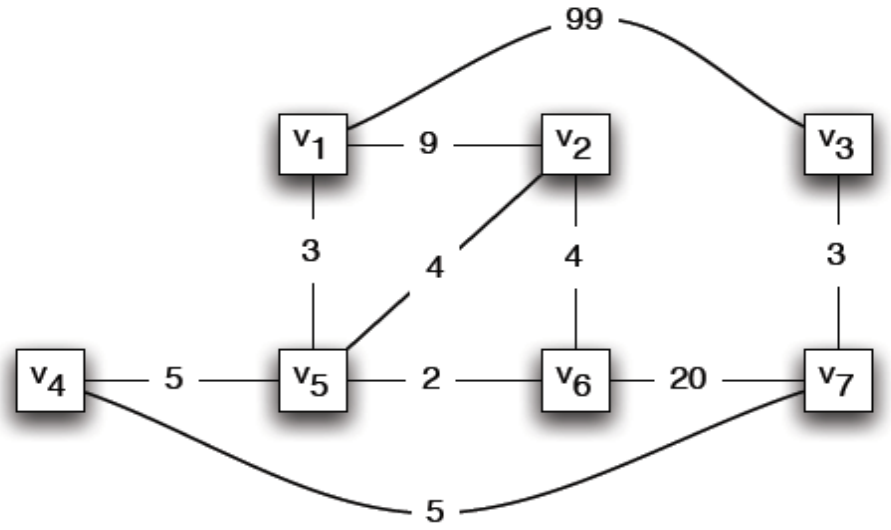


O(n+m)
m:number of edges

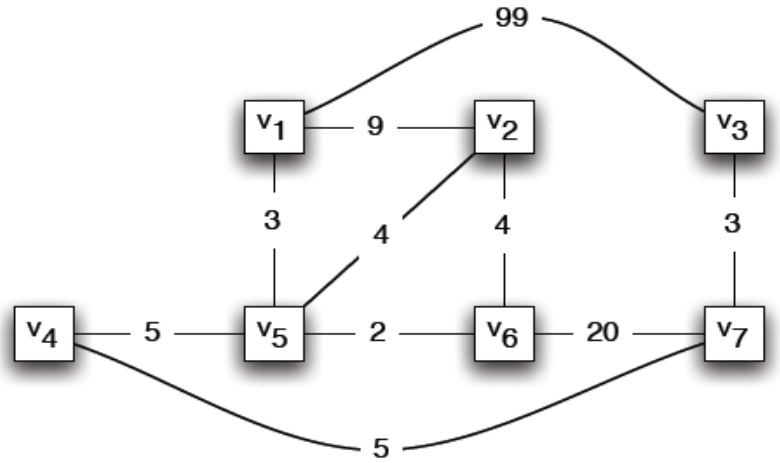


Graphs

- How to represent weighted graphs (shown below) using adjacency matrix, incidence matrix as well as adjacency list?

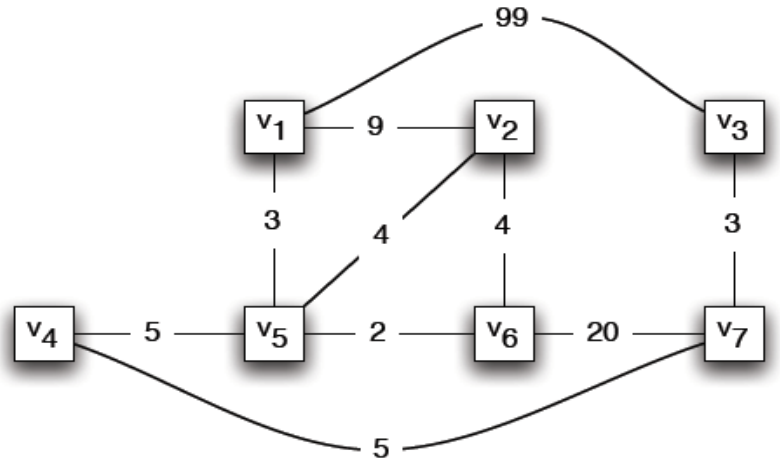


Adjacency matrix



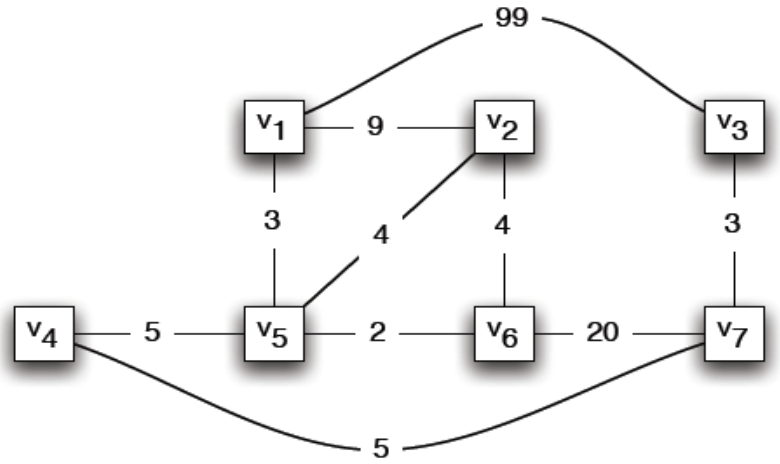
A	v1	v2	v3	v4	v5	v6	v7
V1	∞	9	99	∞	3	∞	∞
V2	9	∞	∞	∞	4	4	∞
V3	99	∞	∞	∞	∞	∞	3
V4	∞	∞	∞	∞	5	∞	5
V5	3	4	∞	5	∞	2	∞
V6	∞	4	∞	∞	2	∞	20
v7	∞	∞	3	5	∞	20	∞

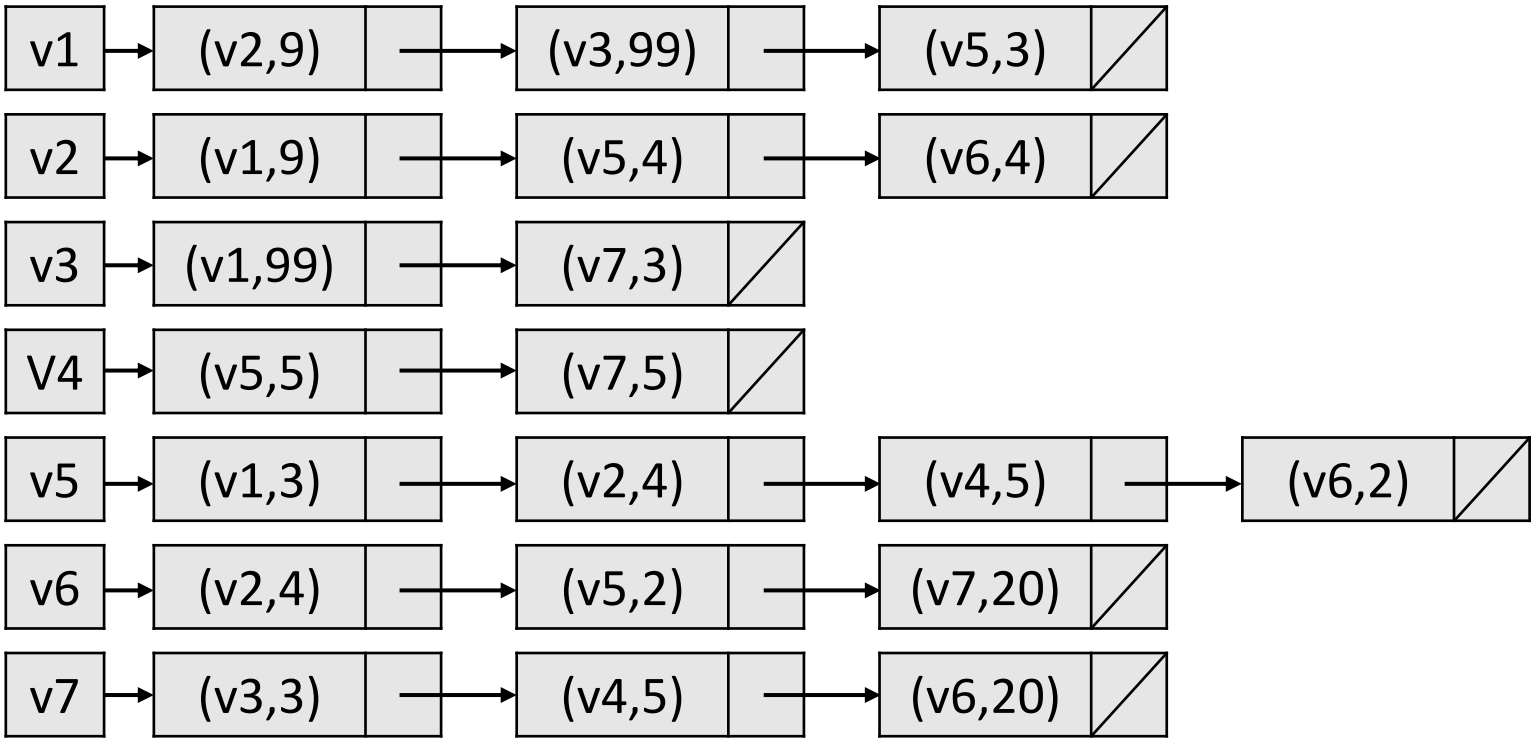
Incidence matrix



A	e12	e13	e15	e25	e26	e37	e45	e47	e56	e67
V1	9	99	3	∞	∞	∞	∞	∞	∞	∞
V2	9	∞	∞	4	4	∞	∞	∞	∞	∞
V3	∞	99	∞	∞	∞	3	∞	∞	∞	∞
V4	∞	∞	∞	∞	∞	∞	5	5	∞	∞
V5	∞	∞	3	4	∞	∞	5	∞	2	∞
V6	∞	∞	∞	∞	4	∞	∞	∞	2	20
V7	∞	∞	∞	∞	∞	3	∞	5	∞	20

Adjacency list






Trees

Tree

- In computer science, a tree is an abstract model of a hierarchical structure.
  - A tree is an acyclic connected undirected graph
  - There is only one path between any pair of nodes
  - Rooted trees have a special node, the root from which the tree “grows”

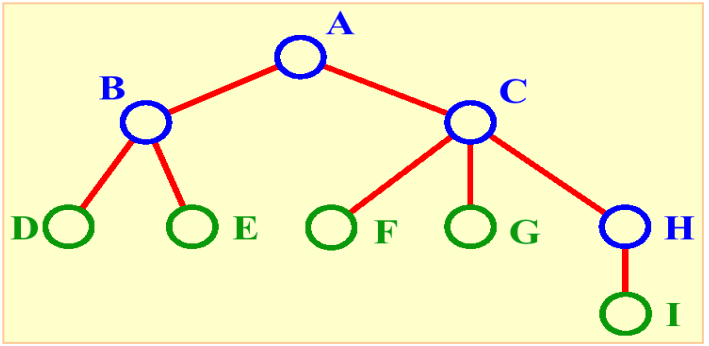
Tree

- Applications:
  - Organization charts
  - File systems
  - Programming environments

Tree

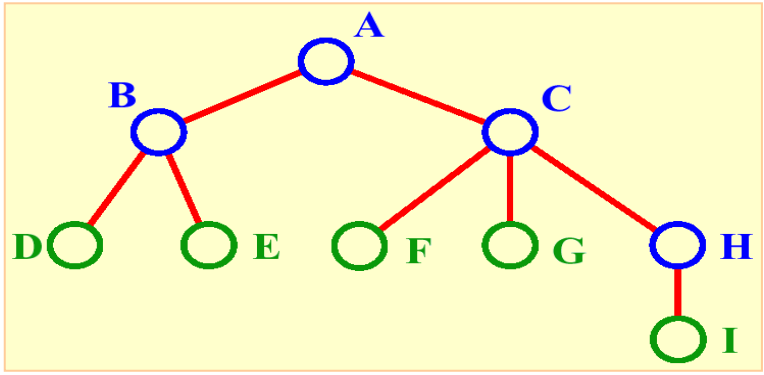
Naming nodes

- A node with no children is called a **leaf (external node)**
  - E.g., D, E, F, G and I



- A node with at least one children is called an **internal node**
  - E.g., A, B, C and H

Tree



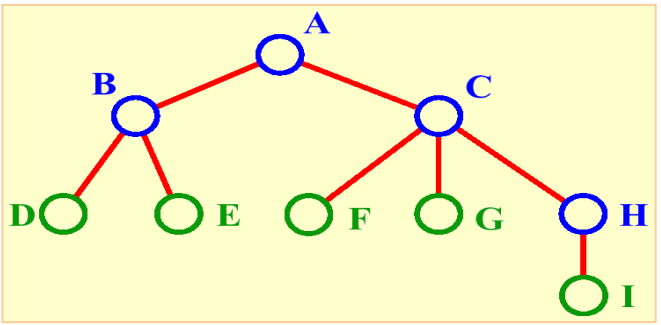
Naming nodes (con..)

- *B* is the ***parent*** of D and E.
- *A* is ***ancestor*** of D and E.
- D and E are ***descendants*** of *A*.
- *C* is the ***sibling*** of B
- *D* and *E* are the ***children*** of B.

Tree

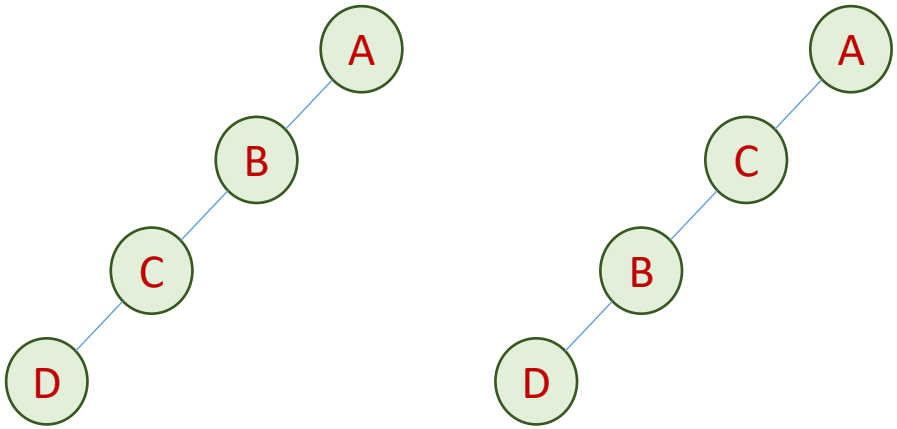
Naming nodes (con..)

- *D* and *F* are ***cousin***
- *A* is the ***root*** node.



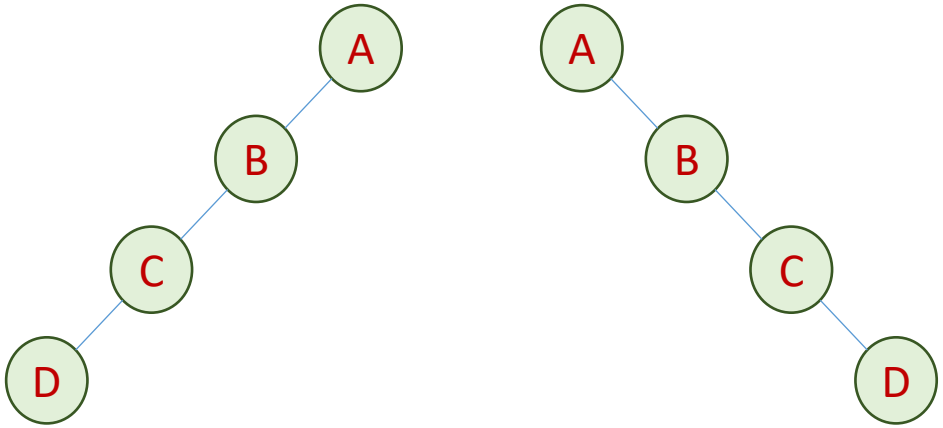
Tree

- The order of nodes in a tree is significant
  - E.g., these are different trees



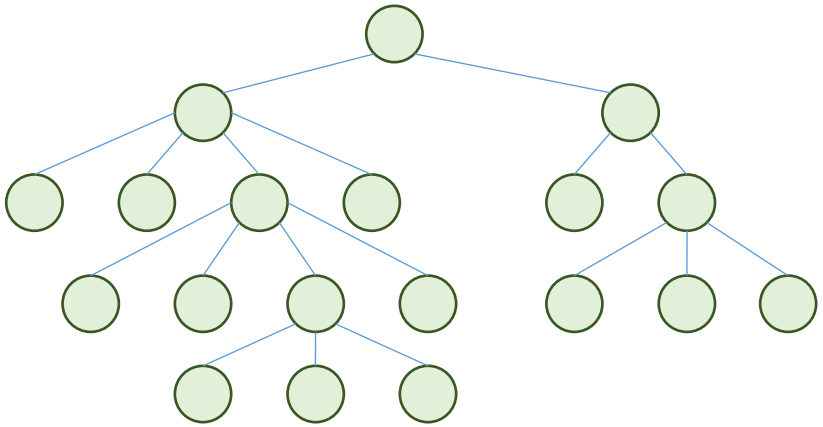
Tree

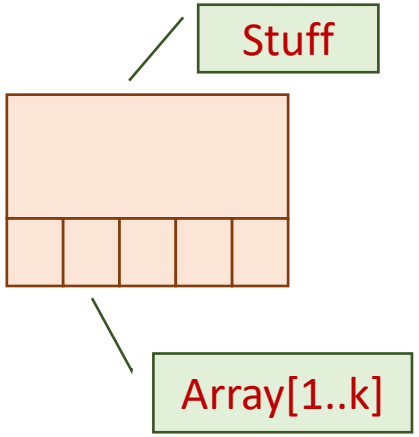
- The placement of nodes in a tree is significant
  - E.g., these are different trees



Tree

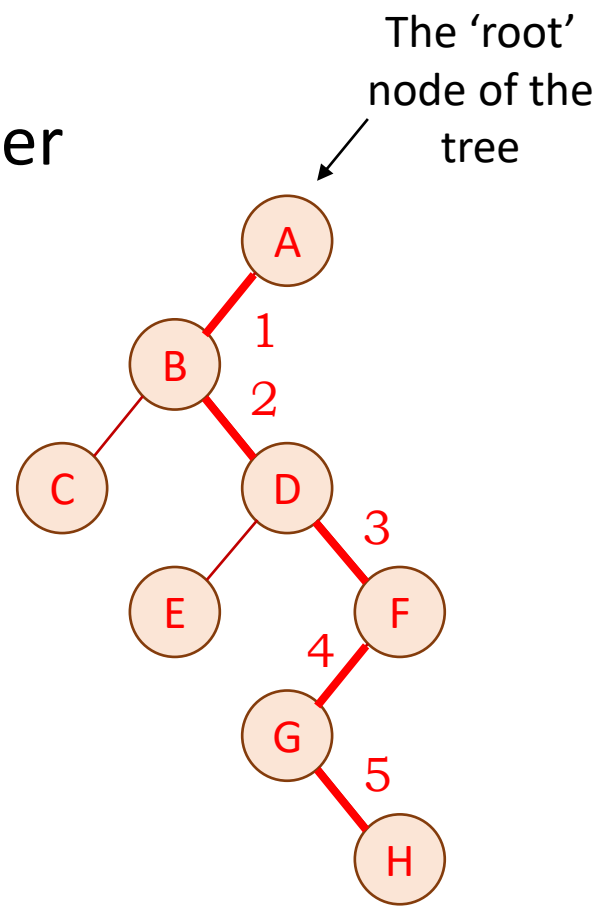
- If each node of a tree can have no more than k children, we say it is a ***k-ary* tree**
  - type k-ary_node: record
    - value: stuff
    - child: array[1..k] of ^k-ary_node





Tree

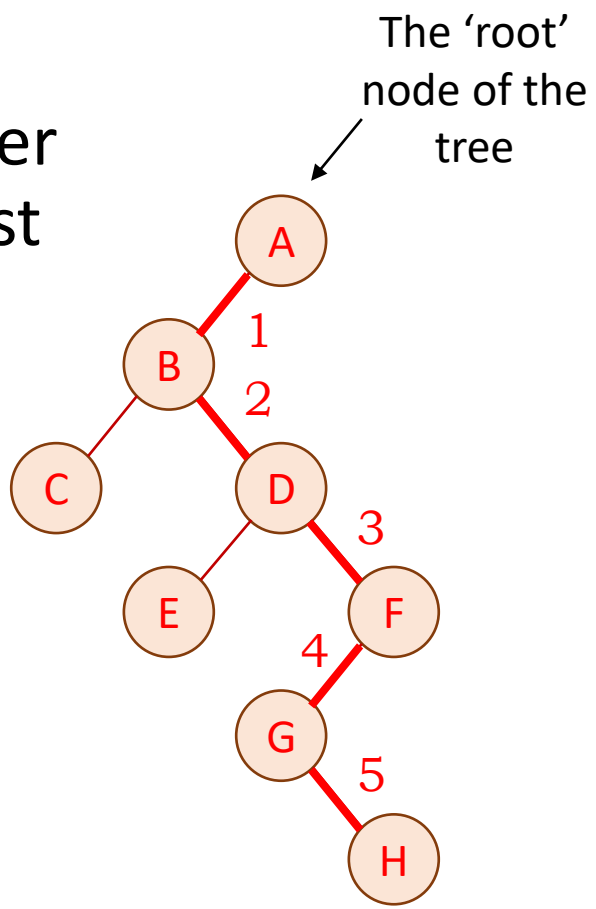
- The height of a node is the number of edges to its most distance leaf node.



Tree

- The height of a node is the number of edges from the node to its most distance leaf node.

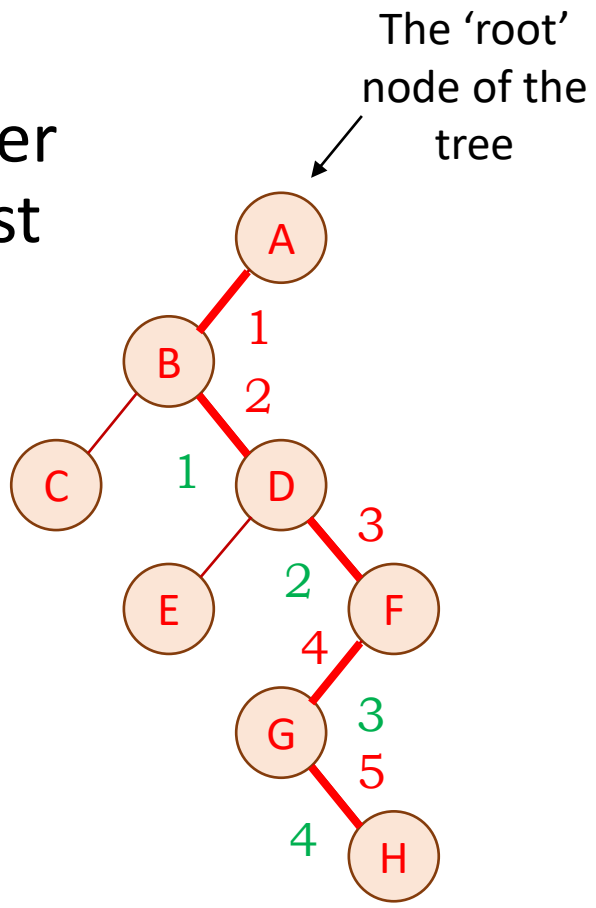
The height of the node A is 5.



Tree

- The height of a node is the number of edges from the node to its most distance leaf node.

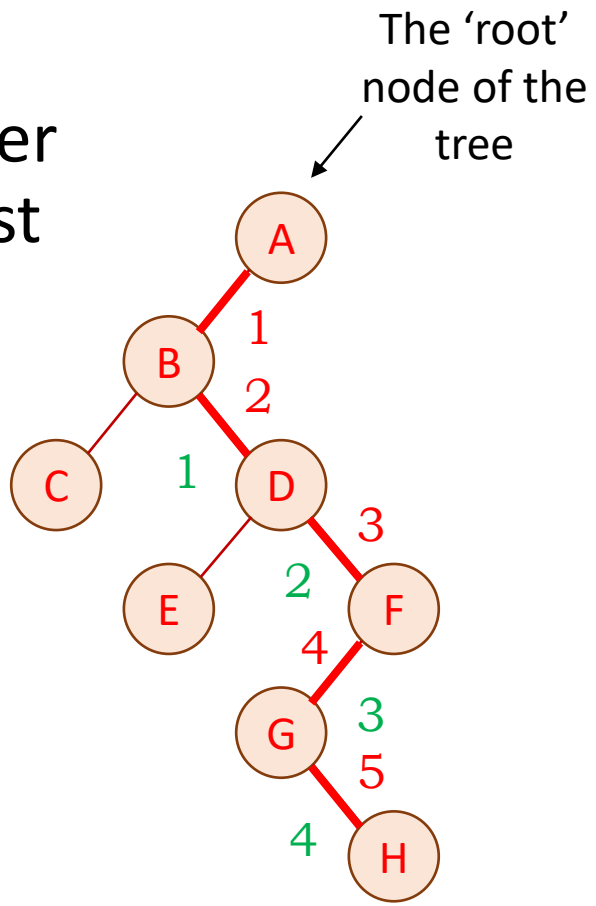
The height of the node A is 5.
The height of the node B is 4.



Tree

- The height of a node is the number of edges from the node to its most distance leaf node.

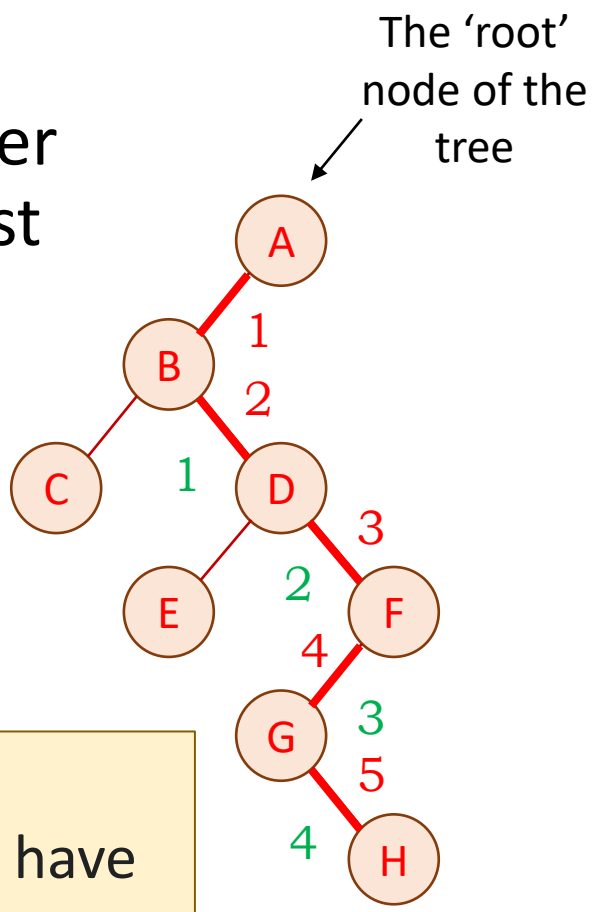
The height of the node A is 5.
The height of the node B is 4.
The height of the node C is 0.



Tree

- The height of a node is the number of edges from the node to its most distance leaf node.

The height of the node A is 5.
The height of the node B is 4.
The height of the node C is 0.
The height of the node G is 1.

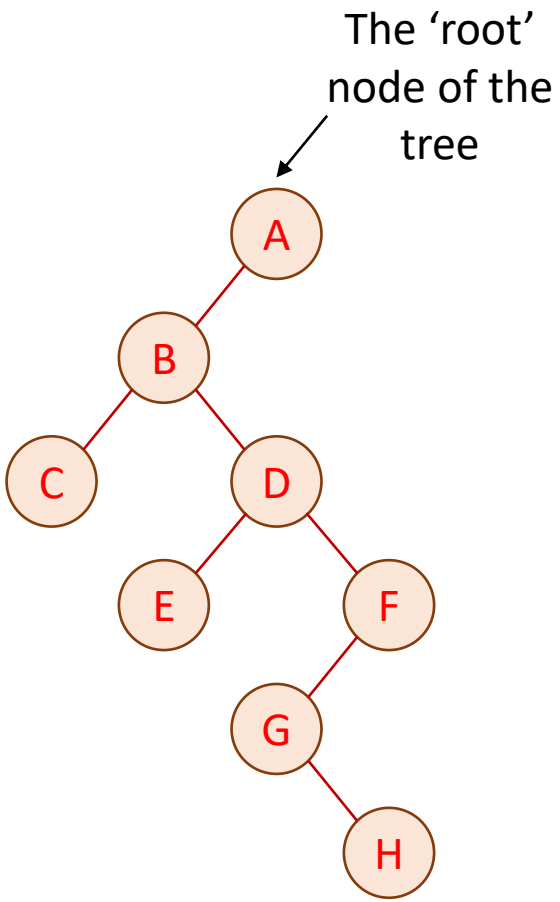


Note:

- The height of the leaf node of a path will have a height of 0.
- The height of the root of a tree is the height of the tree.

Tree

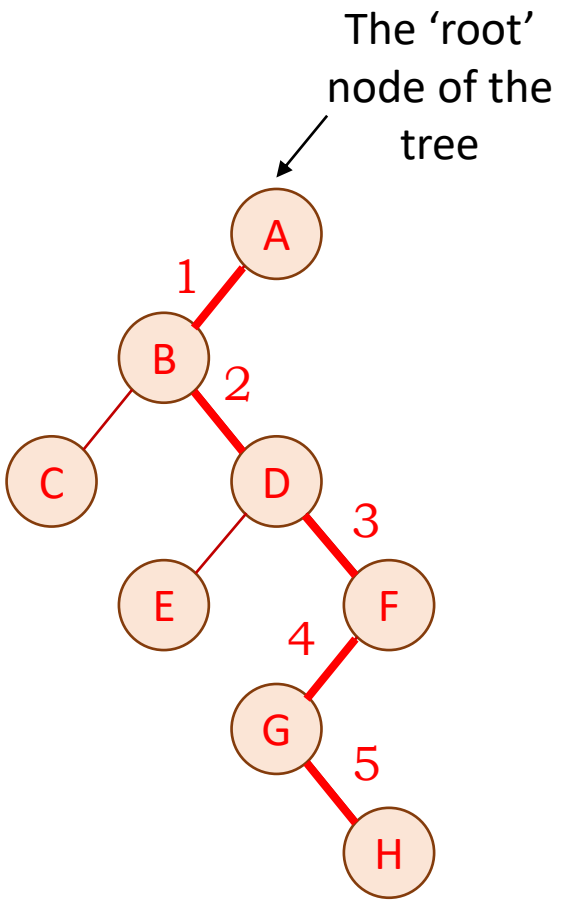
- The depth of a node is the number of edges in the path from the root node to that node.



Tree

- The depth of a node is the number of edges in the path from the root node to that node.

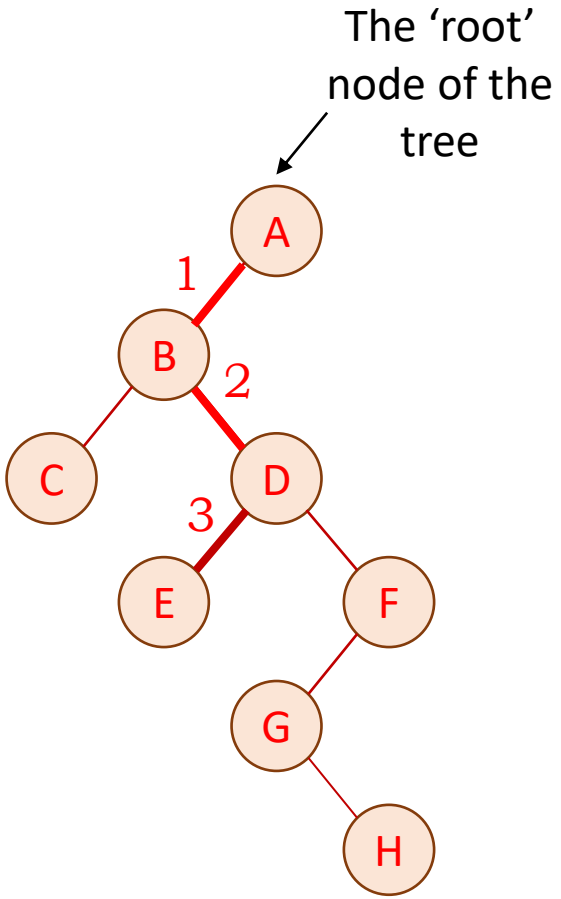
  The depth of the node H is 5.



Tree

- The depth of a node is the number of edges in the path from the root node to that node.

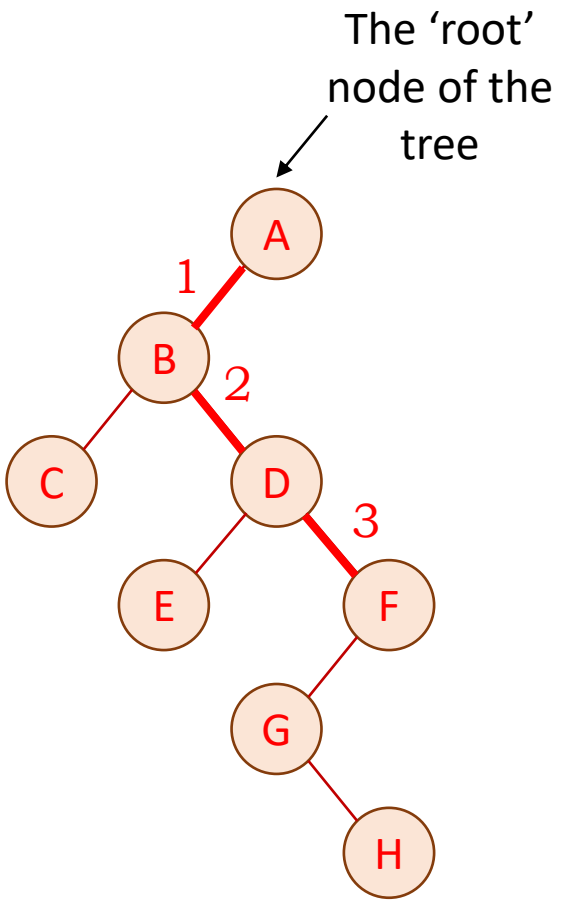
The depth of the node H is 5.
The depth of the node E is 3.



Tree

- The depth of a node is the number of edges in the path from the root node to that node.

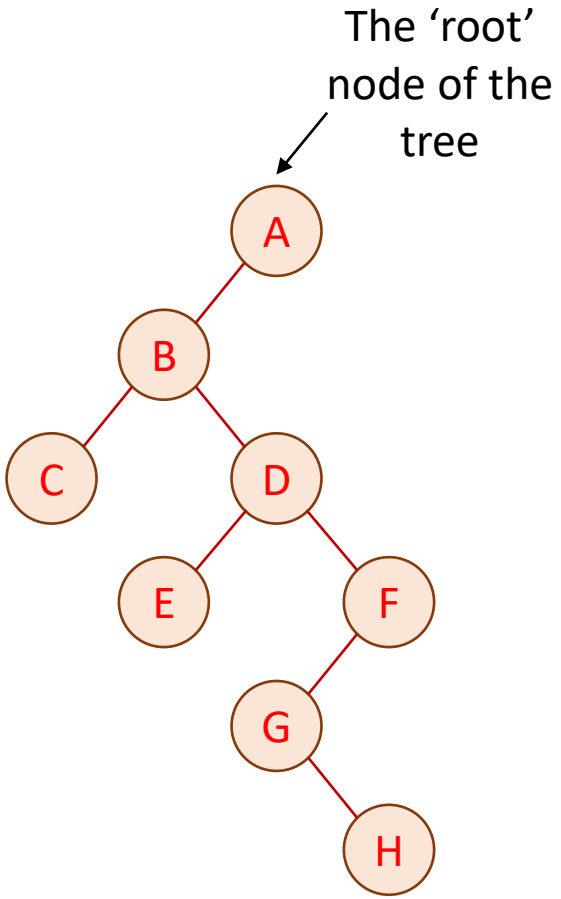
The depth of the node H is 5.
The depth of the node E is 3.
The depth of the node F is 3.



Tree

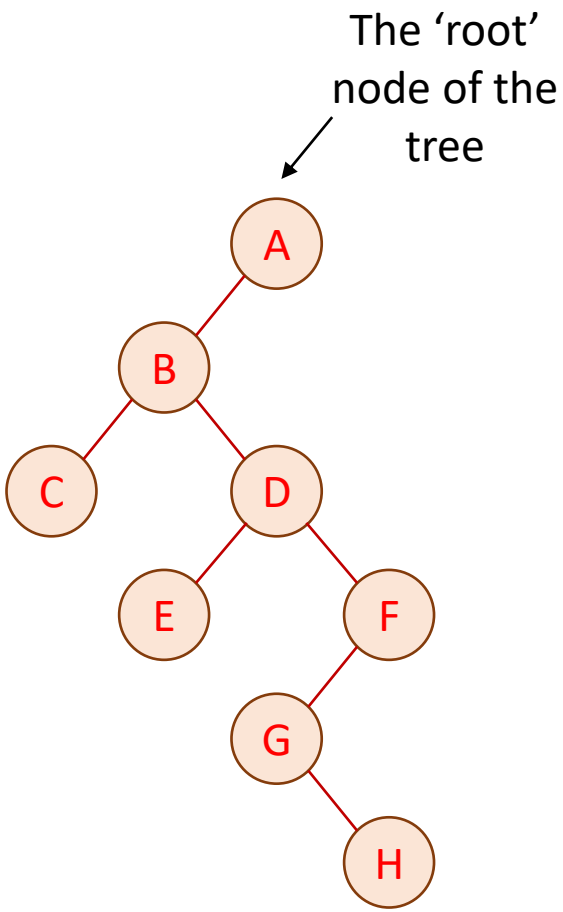
- The depth of a node is the number of edges in the path from the root node to that node.

The depth of the node H is 5.
The depth of the node E is 3.
The depth of the node F is 3.
The depth of the node A is 0.



Tree

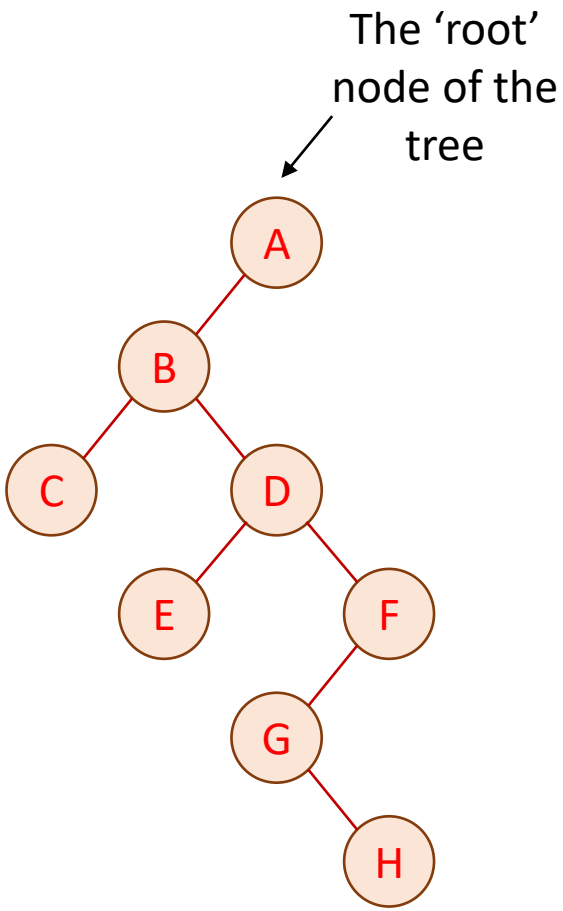
- The level of a node is equal to the number of edges along the unique path between the node and the root node of the tree.



Tree

- The level of a node is equal to the number of edges along the unique path between the node and the root node of the tree.

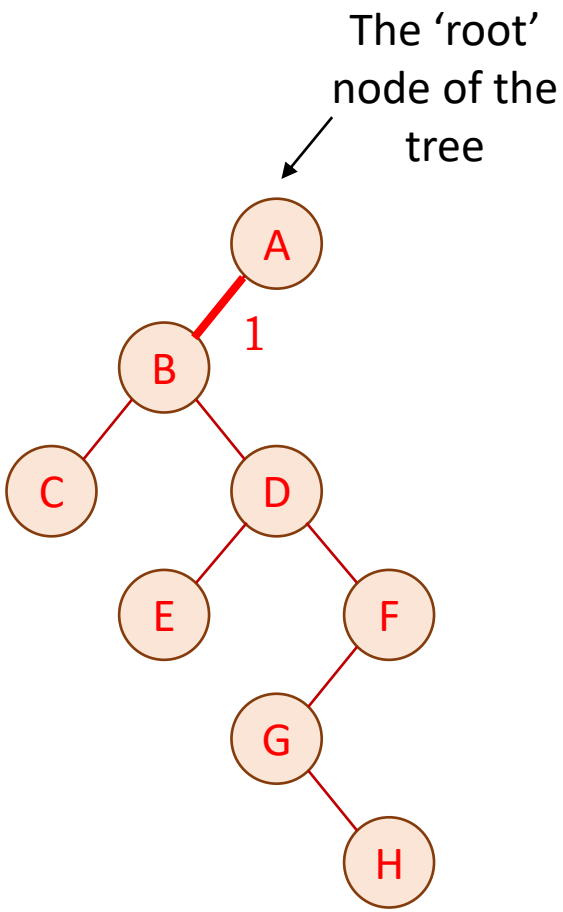
The level of the node A = 0.



Tree

- The level of a node is equal to the number of edges along the unique path between the node and the root node of the tree.

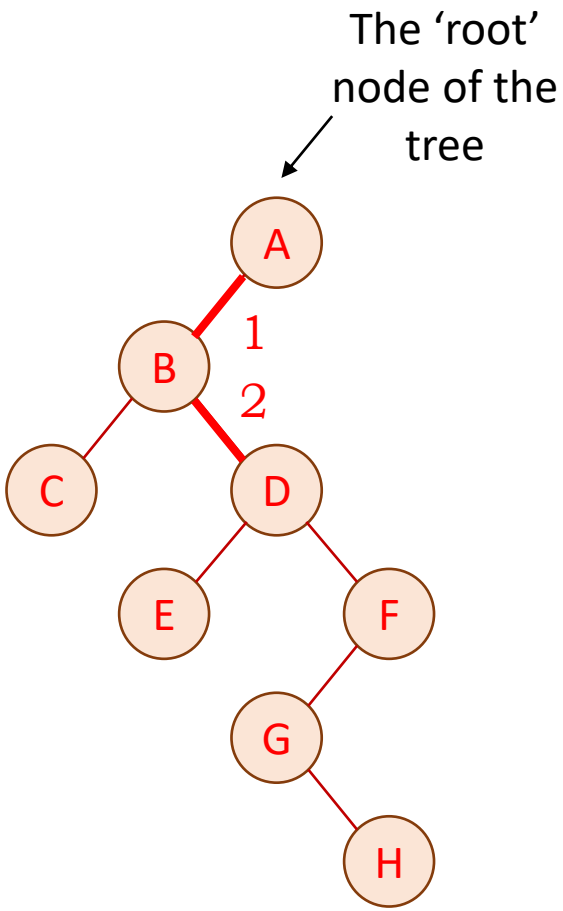
The level of the node A = 0.
The level of the node B = 1.



Tree

- The level of a node is equal to the number of edges along the unique path between the node and the root node of the tree.

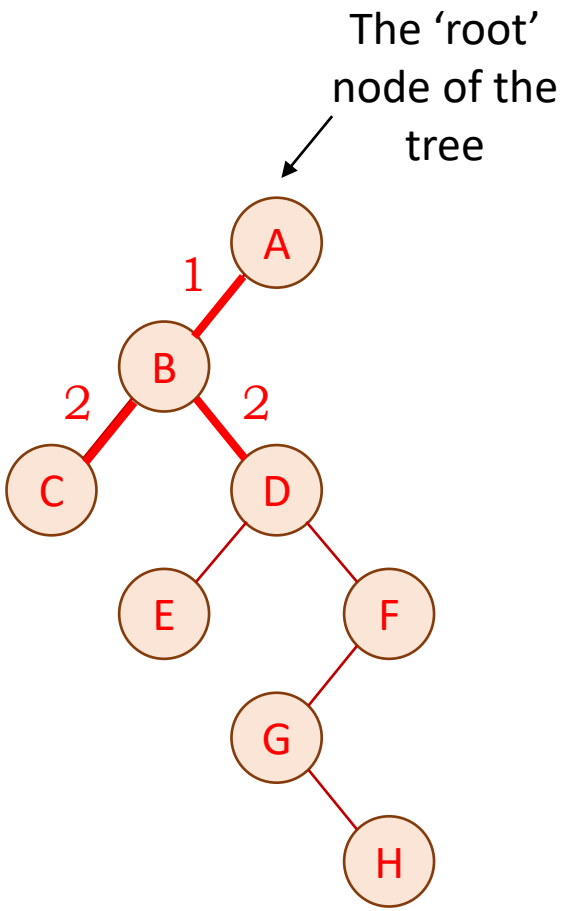
The level of the node A = 0.
The level of the node B = 1.
The level of the node D = 2.



Tree

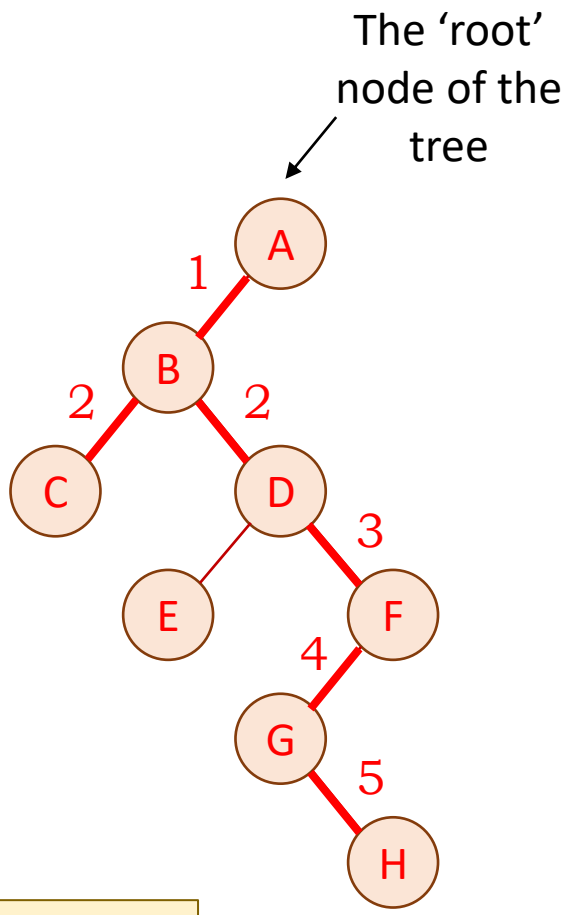
- The level of a node is equal to the number of edges along the unique path between the node and the root node of the tree.

The level of the node A = 0.
The level of the node B = 1.
The level of the node D = 2.
The level of the node C = 2.



Tree

- The level of a node is equal to the number of edges along the unique path between the node and the root node of the tree.

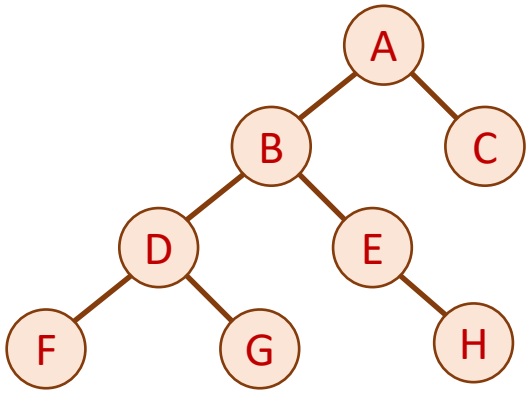


The level of the node A = 0.
The level of the node B = 1.
The level of the node D = 2.
The level of the node C = 2.
The level of the node H = 5.

Note:

- The depth of a node will be the same as the level of a node.

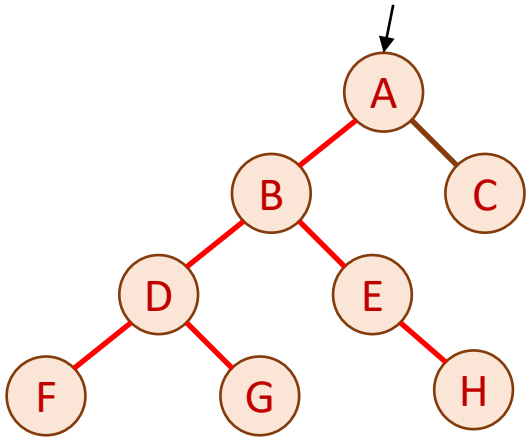
Tree



Node	Height	Depth	Level
A			
B			
C			
D			
E			
F			
G			
H			

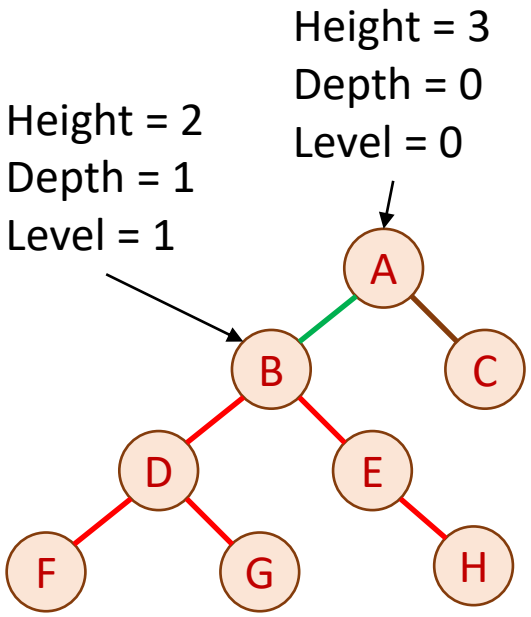
Tree

Height = 3
Depth = 0
Level = 0



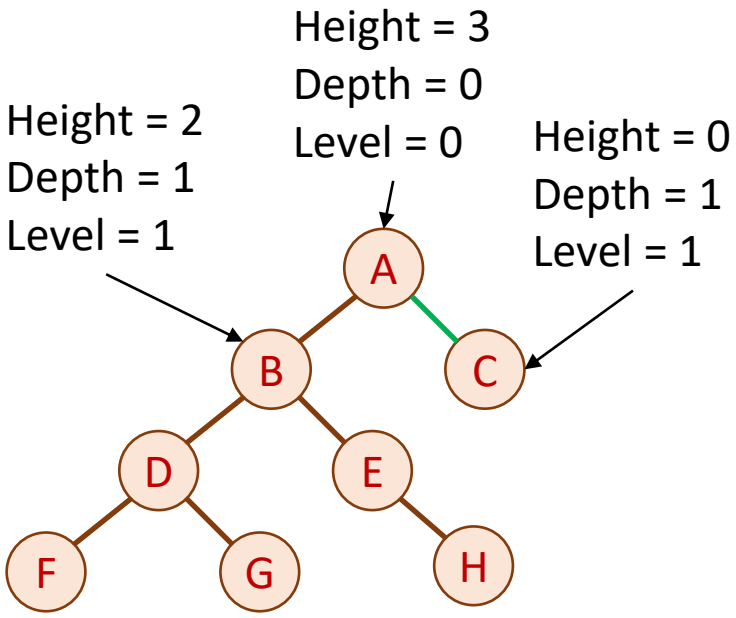
Node	Height	Depth	Level
A	3	0	0
B			
C			
D			
E			
F			
G			
H			

Tree



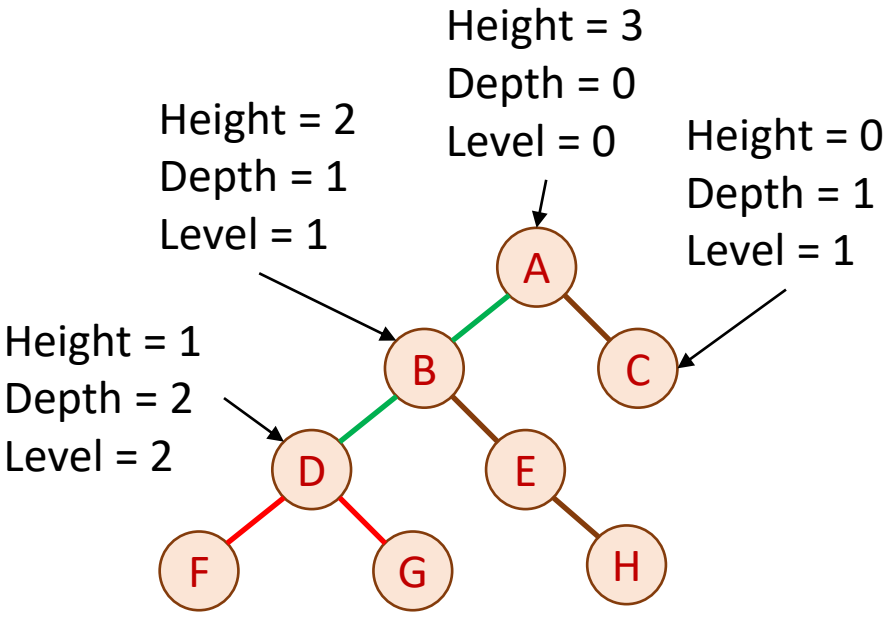
Node	Height	Depth	Level
A	3	0	0
B	2	1	1
C			
D			
E			
F			
G			
H			

Tree



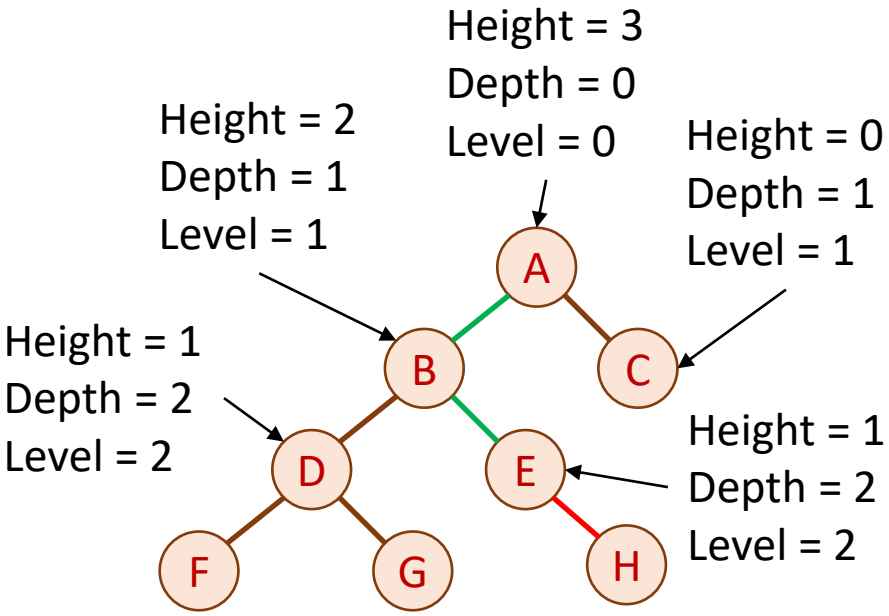
Node	Height	Depth	Level
A	3	0	0
B	2	1	1
C	0	1	1
D			
E			
F			
G			
H			

Tree



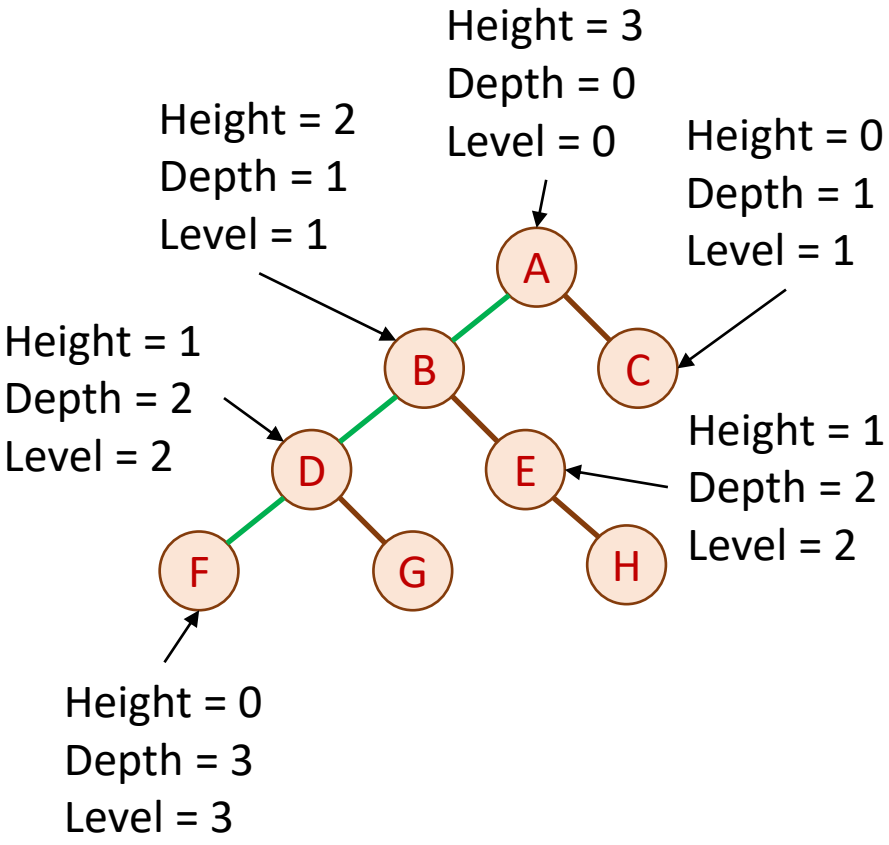
Node	Height	Depth	Level
A	3	0	0
B	2	1	1
C	0	1	1
D	1	2	2
E			
F			
G			
H			

Tree



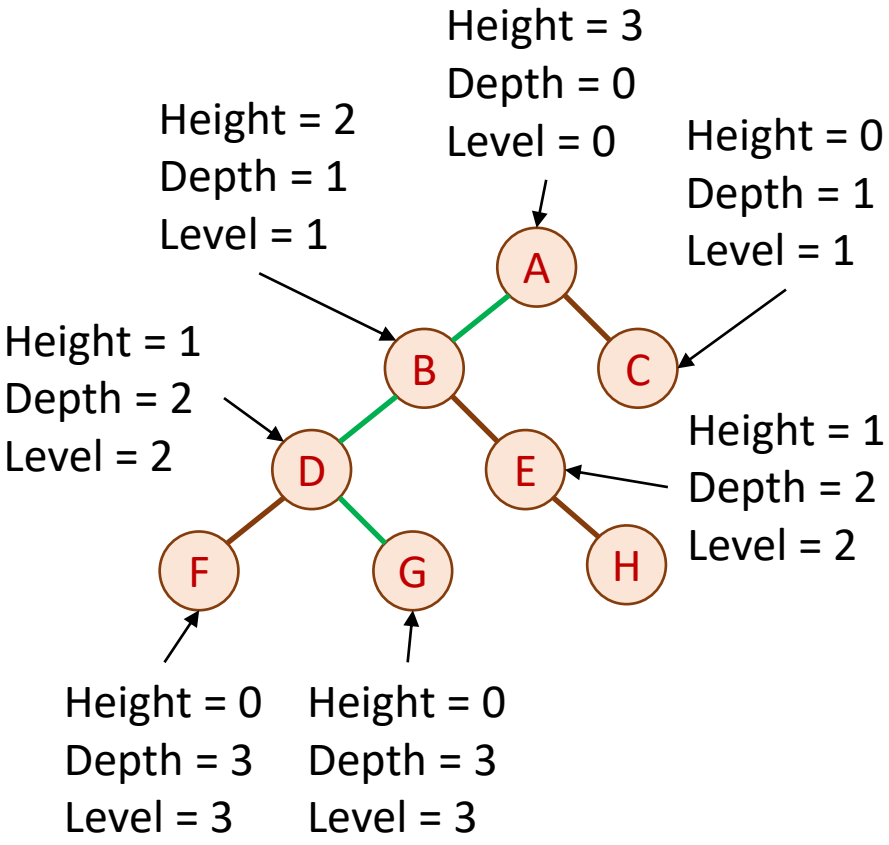
Node	Height	Depth	Level
A	3	0	0
B	2	1	1
C	0	1	1
D	1	2	2
E	1	2	2
F			
G			
H			

Tree



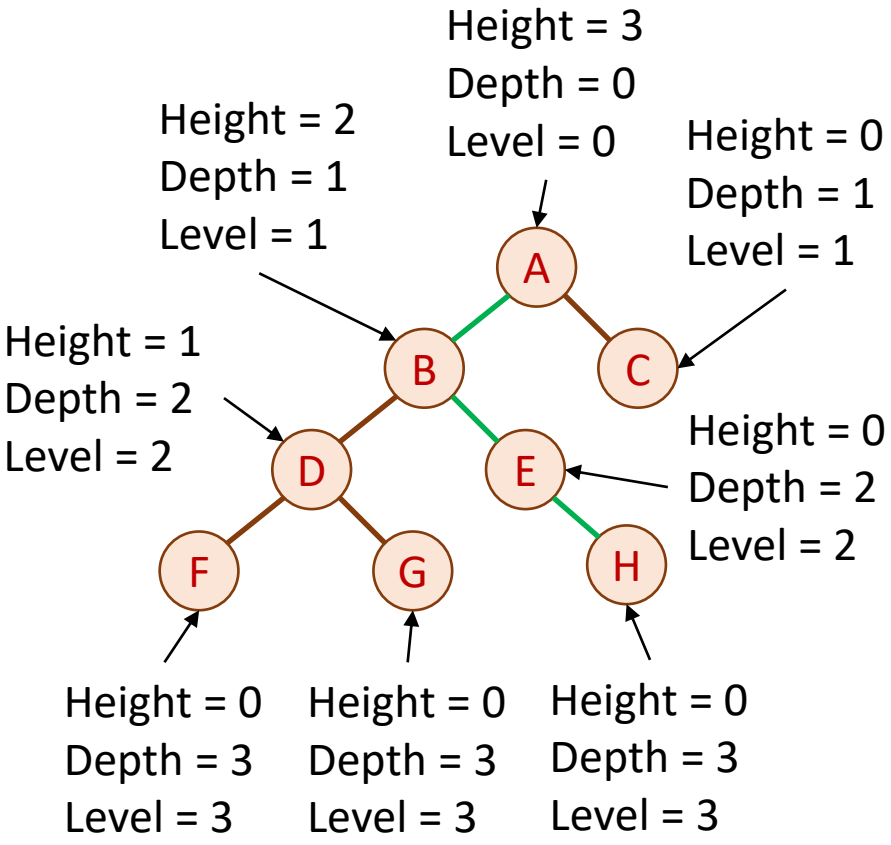
Node	Height	Depth	Level
A	3	0	0
B	2	1	1
C	0	1	1
D	1	2	2
E	1	2	2
F	0	3	3
G			
H			

Tree



Node	Height	Depth	Level
A	3	0	0
B	2	1	1
C	0	1	1
D	1	2	2
E	1	2	2
F	0	3	3
G	0	3	3
H			

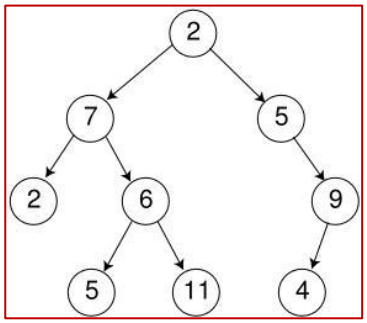
Tree



Node	Height	Depth	Level
A	3	0	0
B	2	1	1
C	0	1	1
D	1	2	2
E	0	2	2
F	0	3	3
G	0	3	3
H	0	3	3

Binary Tree

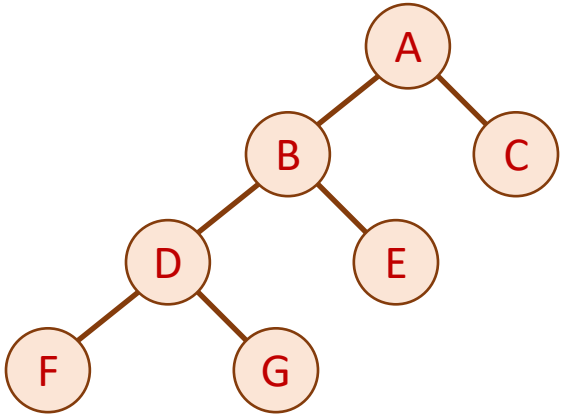
- If $k = 2$ we have a **binary tree**
  - type binary_node: record
    - value: stuff
    - left: ^binary_node
    - right: ^binary_node



- If a node has one child, that child is designated as either a *left child* or a *right child* (but not both).
- If a node has two children, one child is designated a left child and the other a *right child*.

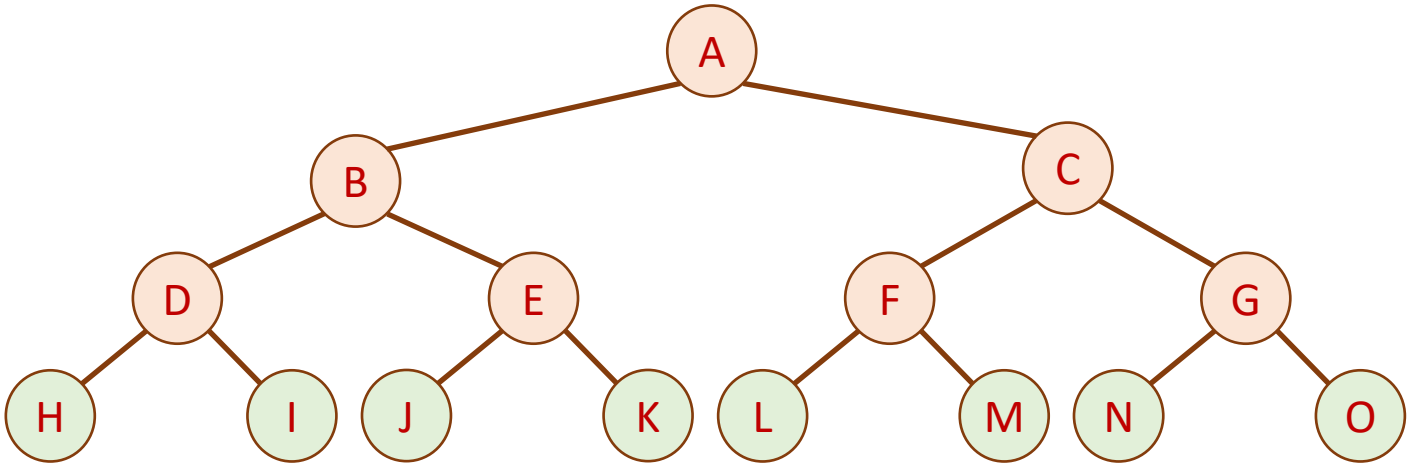
Types of binary tree

- **Full** binary tree
  - All internal nodes have exactly two children



Types of binary tree

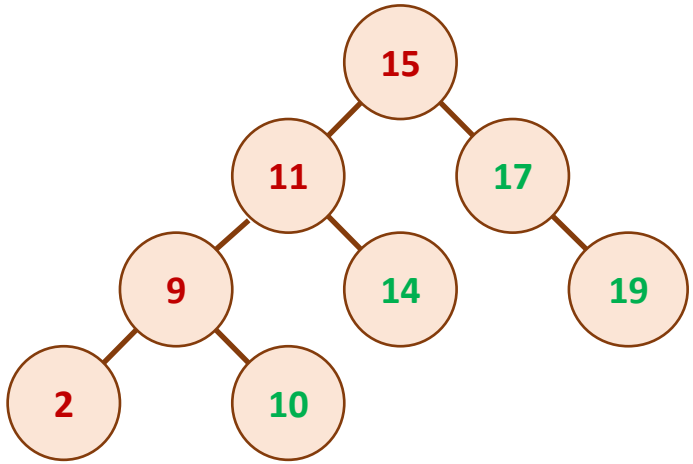
- **Complete** binary tree
  - **Full** binary tree
  - **All leaves** have exactly the **same depth**



Binary search tree

A **Search** tree is:

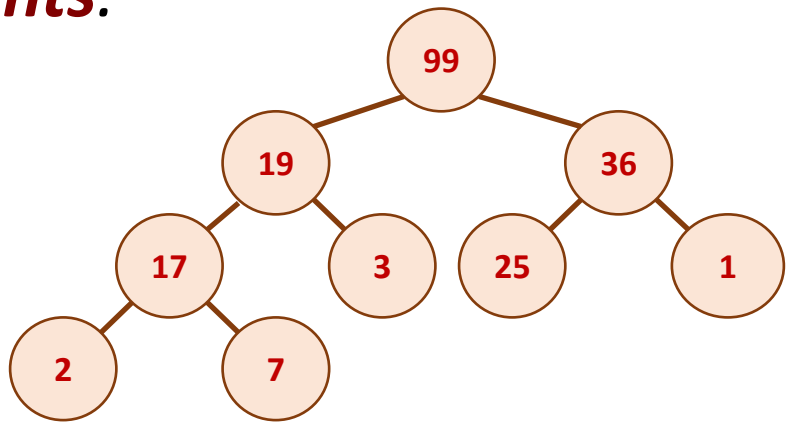
- A binary tree
- The value in each node is greater than or equal to all the values in its left child or any of that child's descendants
- The value in each node is less than or equal to all the values in its right child or any of that child's descendants



Heap

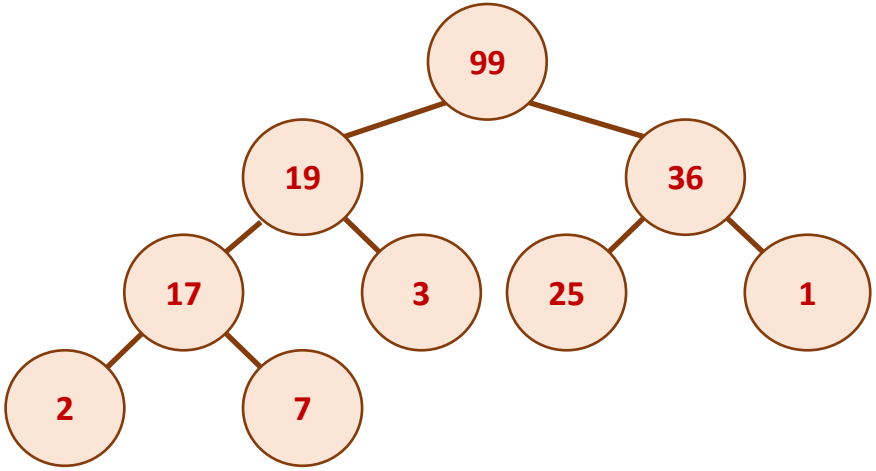
*A **heap** structure:*

- *a **binary** tree with the following properties:*
  - *The tree is **complete** or **nearly complete***
  - *The key value of each node is **greater than** or **equal to** the key value in **each of its descendents**.*



Max-Heap / Min-Heap

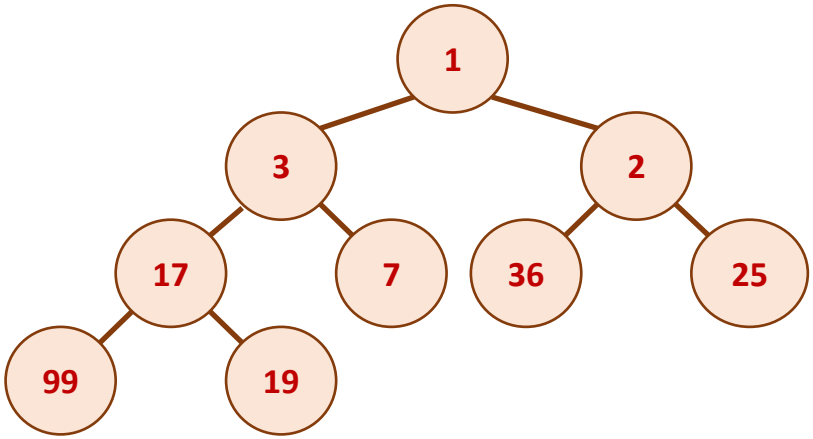
- If the key value is ***greater than*** or ***equal to*** the key values of the subtrees, the heap is called ***Max-heap***.



Max-heap

Max-Heap / Min-Heap

- If the key value is reversed; i.e., the key value is ***smaller than*** or ***equal to*** the key values of the subtrees, the heap is called ***Min-Heap***.



Min-heap

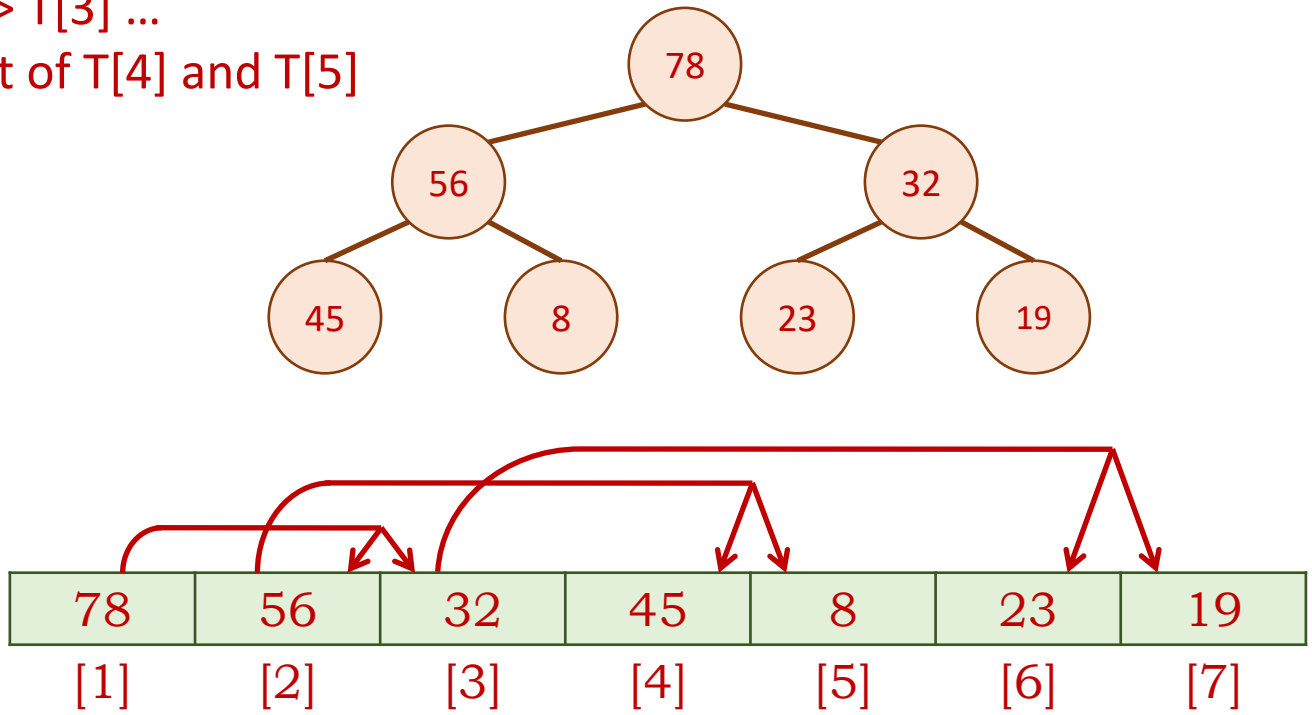
Max-Heap Implementation

- A **max heap** can be implemented using an array where
  - $T[1]$ is the root of the tree
  - $T[i]$ is the parent of $T[2i]$ and $T[2i+1]$
  - $T[i] \geq T[2i]$ and $T[i] \geq T[2i+1]$

Max-Heap Implementation

- T[1] is the parent of T[2] and T[3]
- T[1] > T[2], T[1] > T[3] …
- T[2] is the parent of T[4] and T[5]

…

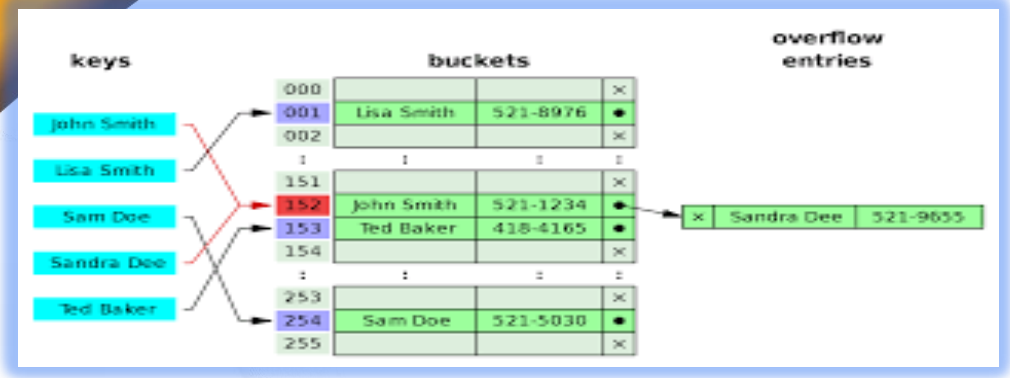


How to add a node to a heap?

- We will look at how to add a node to a heap during tutorial session.

How to delete a node from a heap?

- We will look at how to delete a node from a heap during tutorial session.



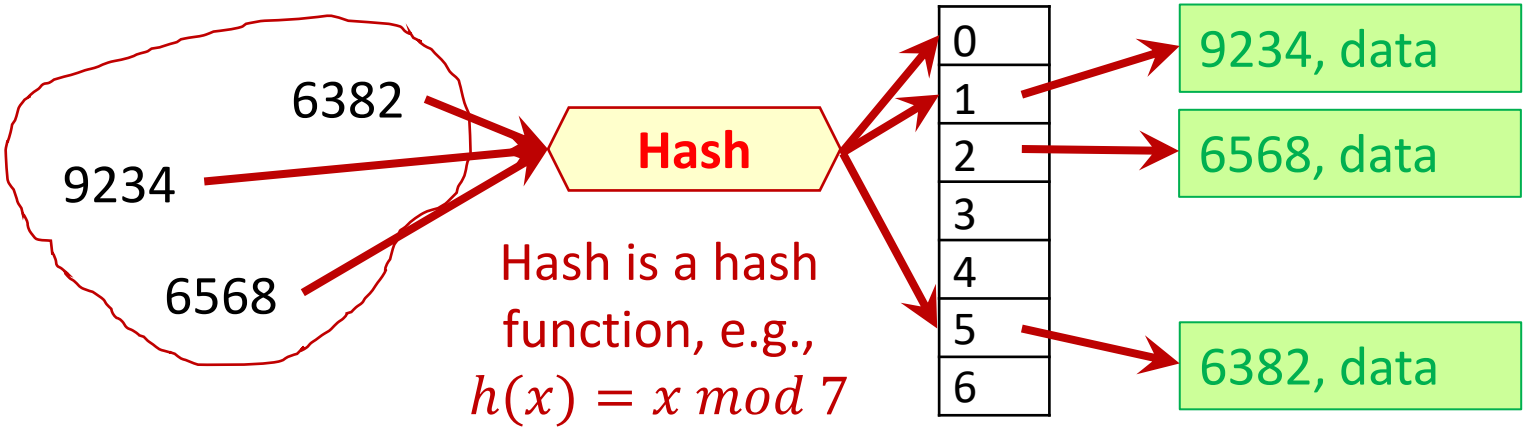
Associative Tables

Associative Tables (Hash table)

- An **associative table** (also known as **hash table**) behaves like an array with no restriction on index value
- Conceptually, an hash table is a collection of (key, value) pairs.
- Unlike an array, there is no guarantee that item access is $\Theta(1)$
- Storage is not necessarily required for unreferenced items

Associative Tables (Hash table)

- Operations on hash tables involve hash function, an algorithm that associates large data sets of variable length, commonly known as keys, to smaller data sets of a fixed length.



Associative Tables (Hash table)

- When a key is given a fixed address in the hash table, it is referred to as close addressing hash system. For example, separate chaining (further explanation and examples will be given later.)
- When a key is not given to a fixed address in the hash table, it is referred to as open addressing hash system. For example, linear probing, quadratic probing, and double hashing are open addressing systems (further explanation and examples will be given later.)

Associative Tables (Hash table)

- Unfortunately, it is very hard to device a one-to-one mapping hash function. Most hash functions have many-to-one mapping.
- For example, using the same hash function h(x)=x mod 11, the following two keys are hashed to the same slot:

$$H(6382) = 6382\ mod\ 7 = 5$$
$$H(7822) = 7824\ mod\ 11 = 5$$

- This is known as "collision"; that is, when two keys are associated to the same hash value.

Associative Tables (Hash table)

- The occurrence of collision become higher as the load factor of the hash table increases.
- Load factor is defined as ratio of number of keys in the hash table over the size of the hash tables (number of slots).

$$\alpha = \frac{n}{m}$$

where:

$\alpha - load\ factor$

$n - number\ of\ keys\ in\ the\ hash\ table$

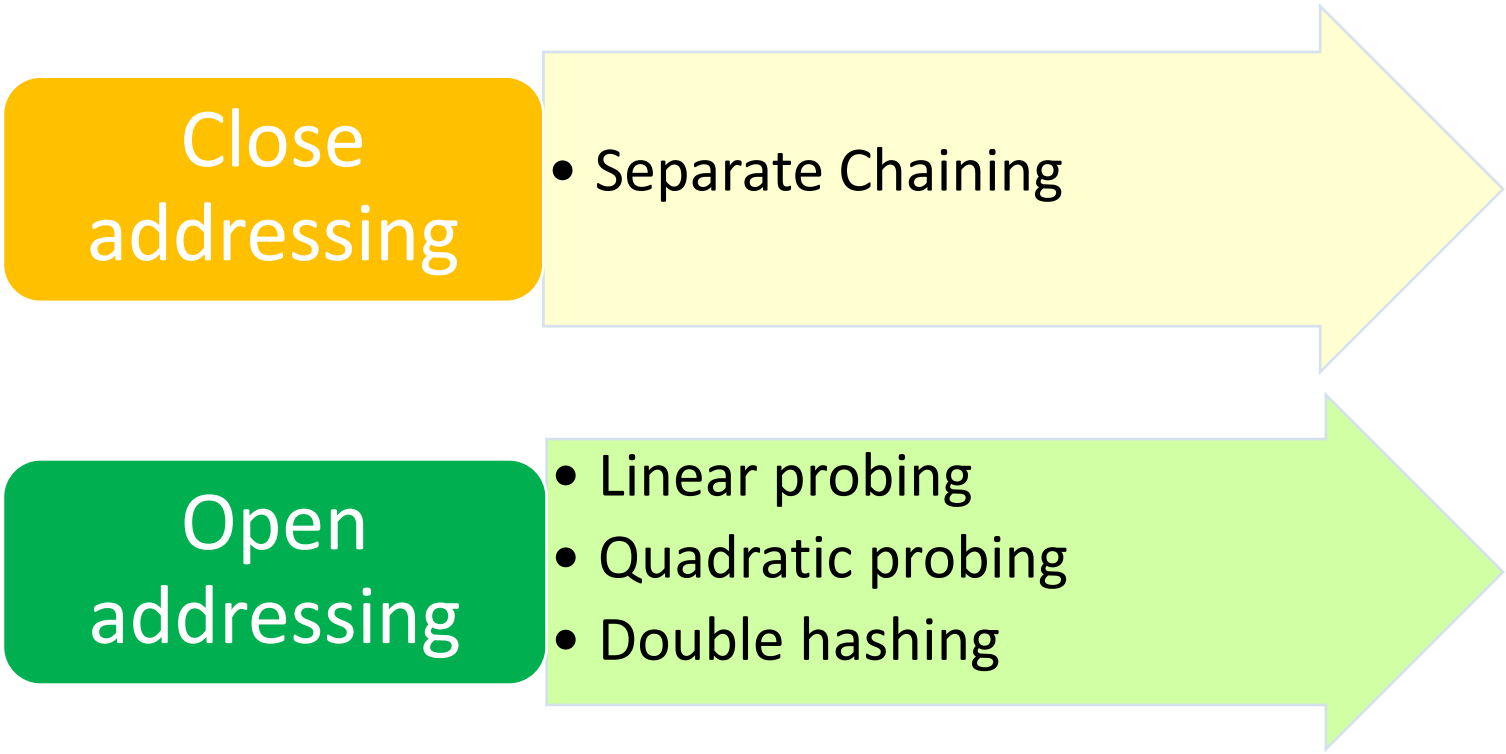
$m - size\ of\ the\ hash\ tables$

Associative Tables (Hash table)

- Load factor is a measure of the fullness of a hash table.
- A load factor of 0 indicates the hash table is empty.
- A load factor of 1 indicates the hash table is full.
- To search, insert or delete keys from or to a hash table, we need to first examine the number of keys in the list $T[h(k)]$ to check if their keys are equal to k.
- The costs of these operations is related to the load factor of the table, that is, the fullness of the table.

Handling collision

Collision is inevitable. Commonly used techniques to handle collision:



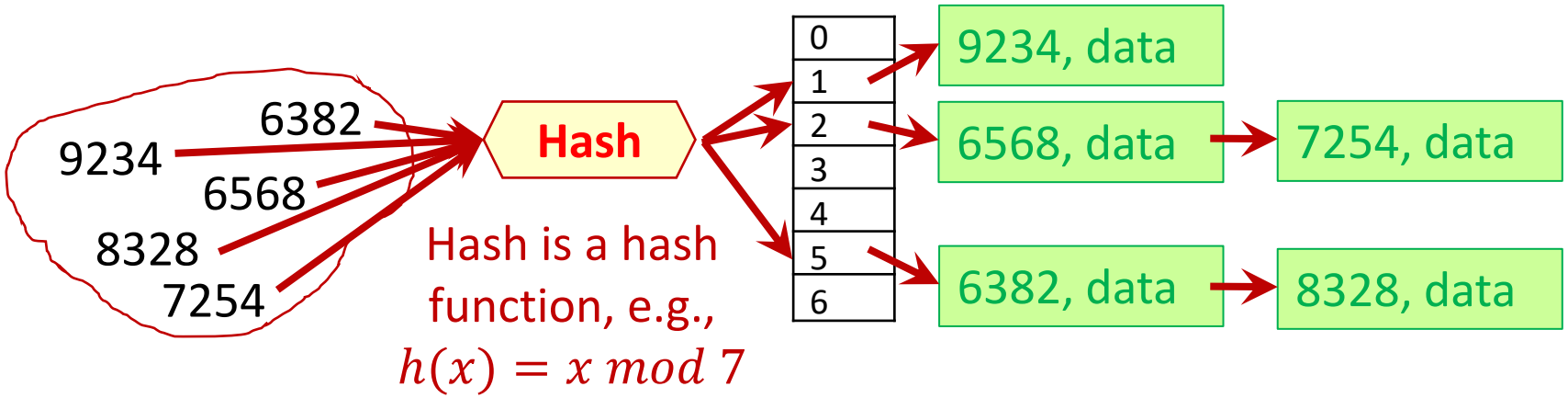
Close Addressing

Separate Chaining

Collision Resolution Using Chaining

Idea:

- Put all objects whose keys are hashed to the same slot into a linked list.
- Slot k contains a reference to the head of the linked list of all objects that are hashed to slot k.



Collision Resolution Using Chaining

- From this point onwards, in our discussion, I will assume the implementation of hash table involves only the key of the object.
- When inserting an key k, the algorithm inserts the key at the beginning of the linked list at location $h(k)$.
- To find a key k, the algorithm simply searches for the key in the linked list at location $h(k)$.
- To delete a key, the algorithm removes the key from the linked list at location $h(k)$.

Collision Resolution Using Chaining

Run-time complexity analysis:

- Worst case: Worst case scenario happen when all n keys are hashed to the same slot. Then search/delete operations may take $\Theta(n)$.
- Average case: depending on the design of the hash function and hence the distribution of n keys among the m slots, that is, the load factor of the hash table.
- Note: In chaining, size of the hash table equals to the number of linked list, hence, α, load factor, is the average length of the linked lists.

Collision Resolution Using Chaining

- To calculate the expected (average) number of probes, we need to assume the hash function is a simple uniform hashing.
- Simple uniform hashing refers to hashing implementation in which any n elements is equally likely to hash into any of the m slots, independently of where any other element has hashed to.

Separate Chaining

- Detail analysis of the number of probe is not within the scope of this module. For separate chaining, the following are the expected successful and unsuccessful probe:

Successful probe	Unsuccessful probe
$1 + \frac{\alpha}{2}$	α

$where\ \alpha\ is\ the\ load\ factor\ of\ the\ hash\ table.$

Example 1

- Given a hash table using separate chaining collision resolution with load factor $\frac{7}{11}$, what is the expected number of probes in an unsuccessful search and the expected number of probes in a successful search?
- Unsuccessful search:

$$\alpha = \frac{7}{11} = 0.64 \text{ probe.}$$

Example 1

- Successful search:

$$1 + \frac{\alpha}{2} = 1 + \frac{\frac{7}{11}}{2} = 1 + 0.3182 = 1.32 \text{ probes.}$$

We will look at the computation of average number of probes with empirical case during tutorial sessions.

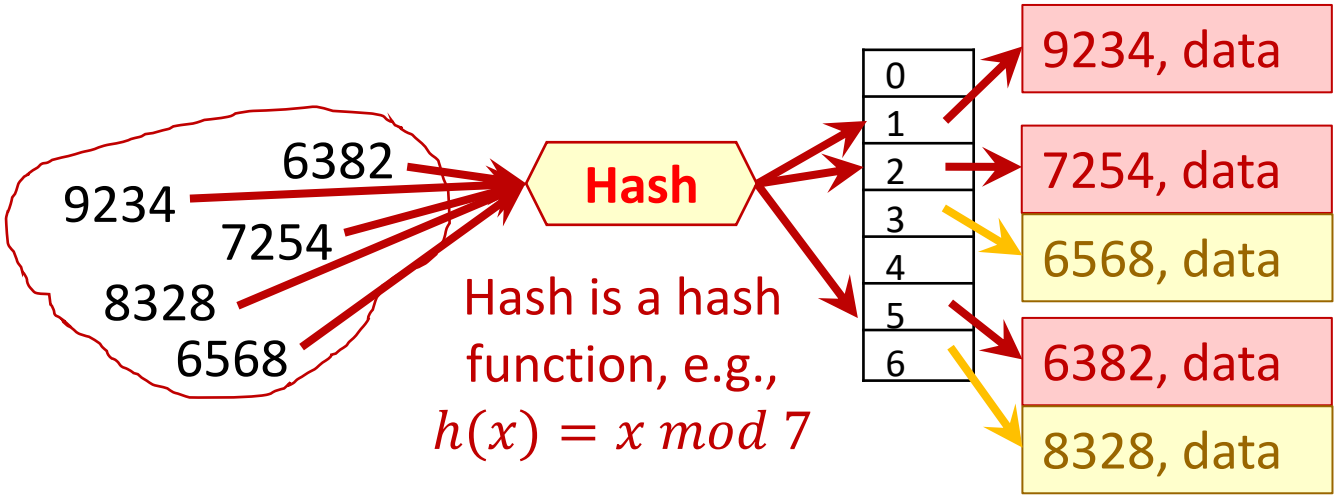
Open Addressing

Linear Probing

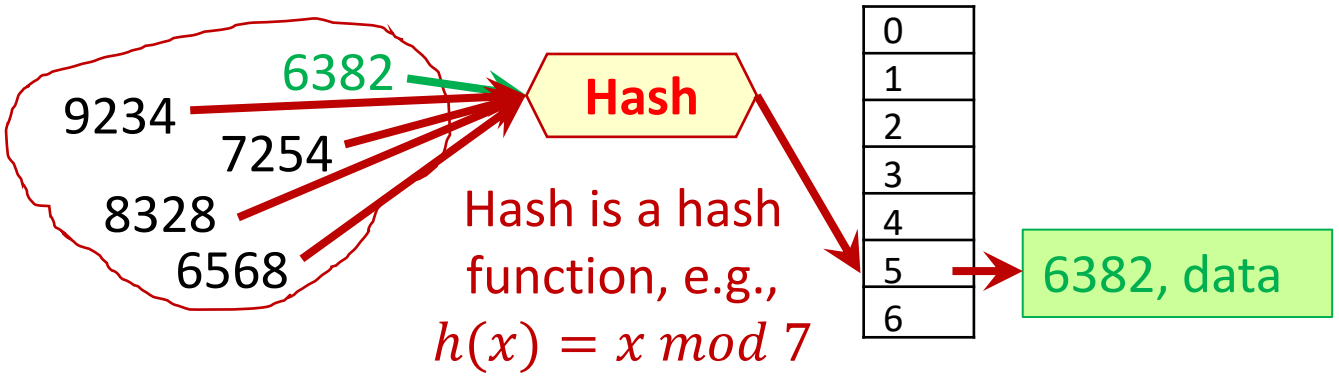
Linear probing

Idea:

- When there is a collision, we scan forwards for the next open bucket (slot), wrap around when we reach the last slot, to place the value.

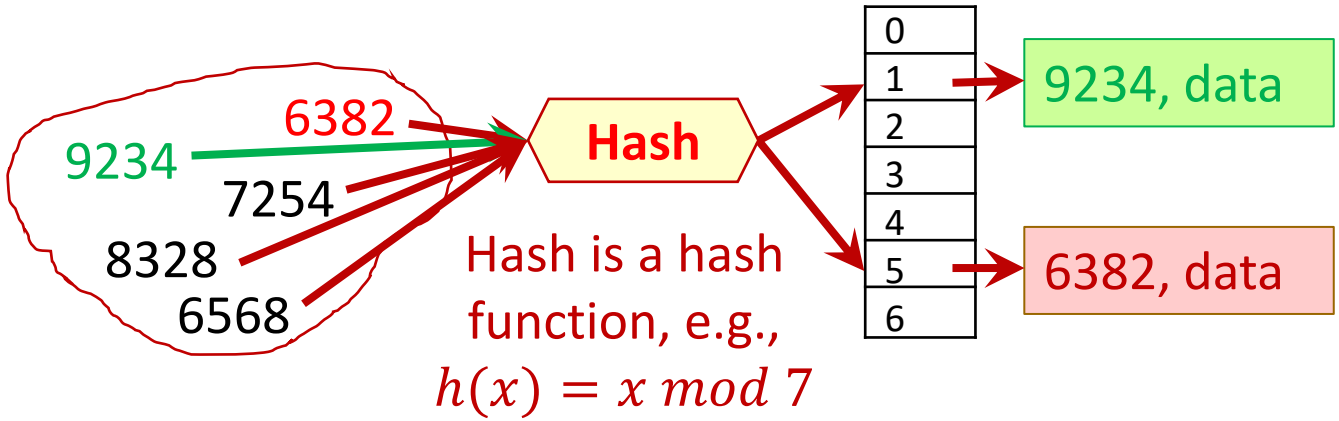


Step-by-step example: insert 6382



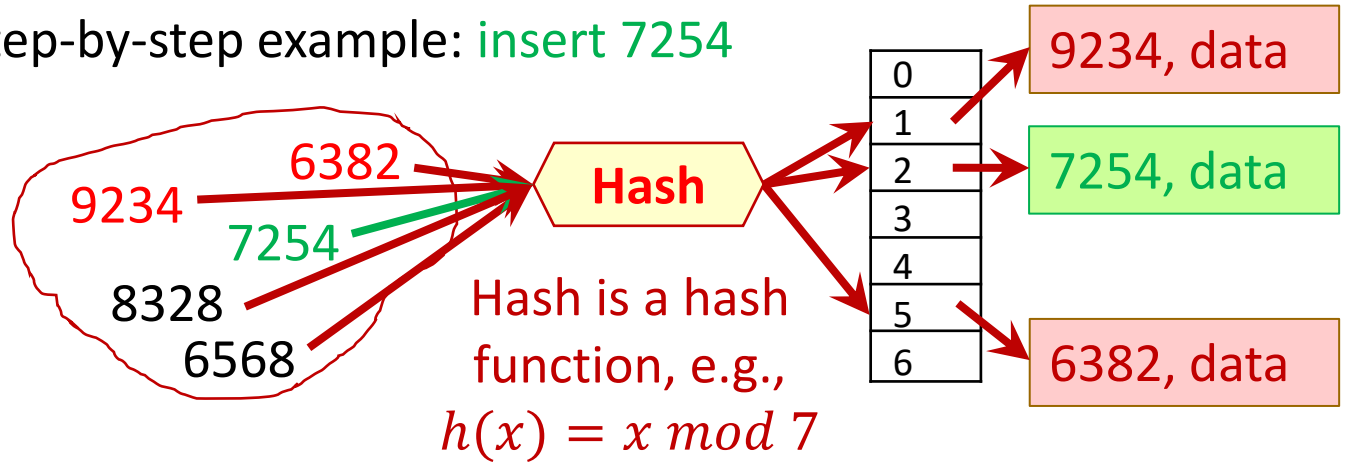
Insert object with key 6382.

Step-by-step example: insert 9234



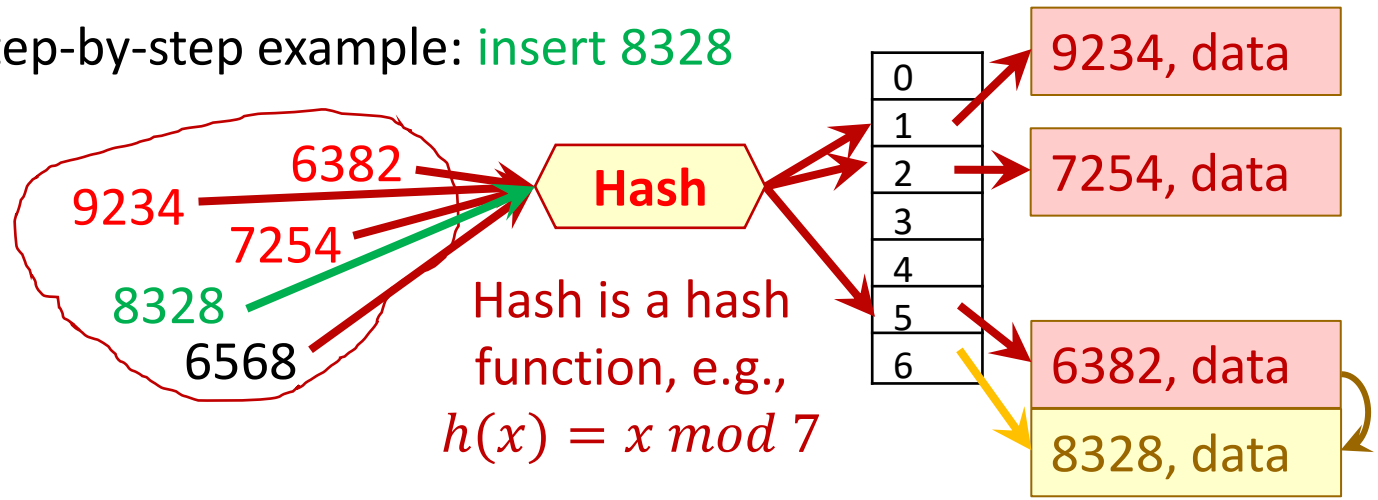
Insert object with key 9234.

Step-by-step example: insert 7254



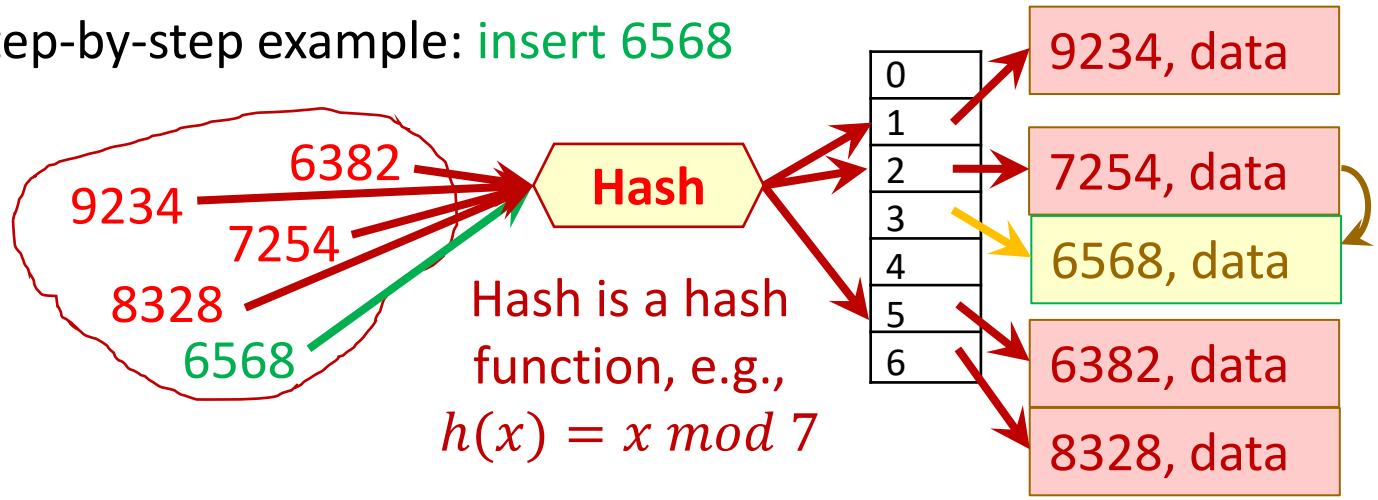
Insert object with key 7254.

Step-by-step example: insert 8328



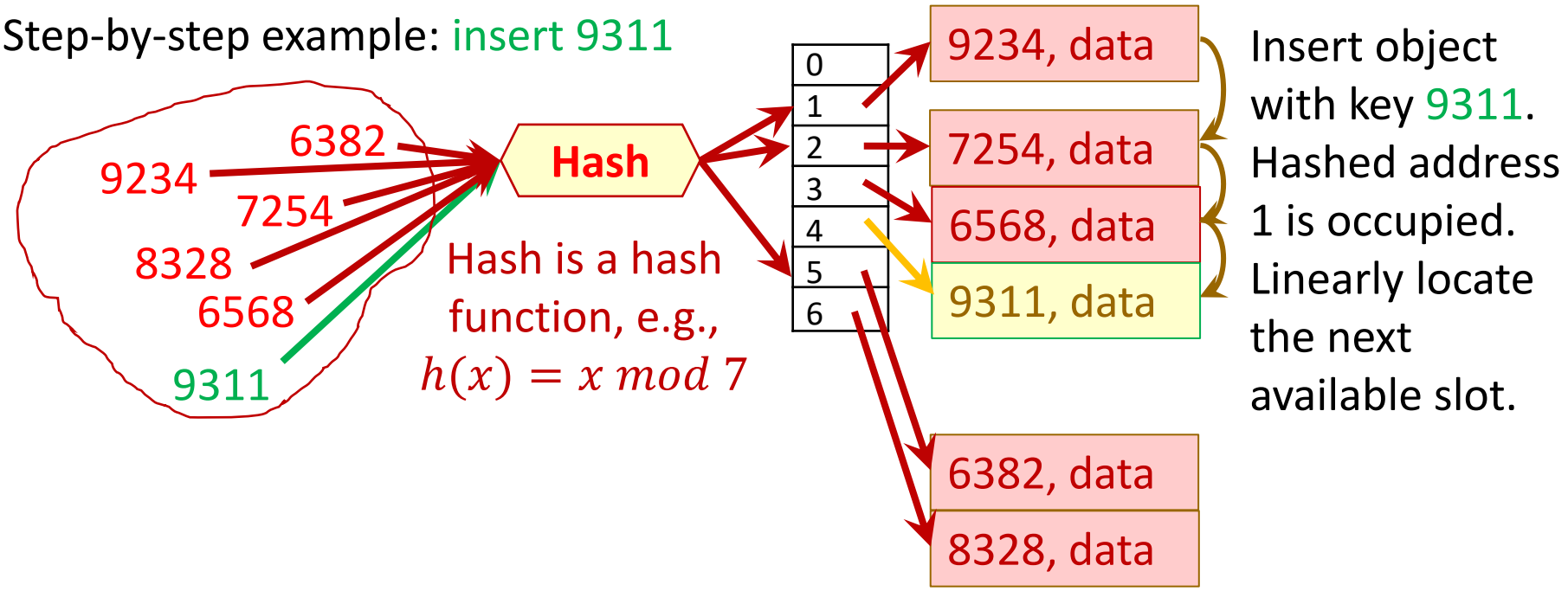
Insert object with key 8328. Hashed address 5 is occupied. Linearly locate the next available slot.

Step-by-step example: insert 6568

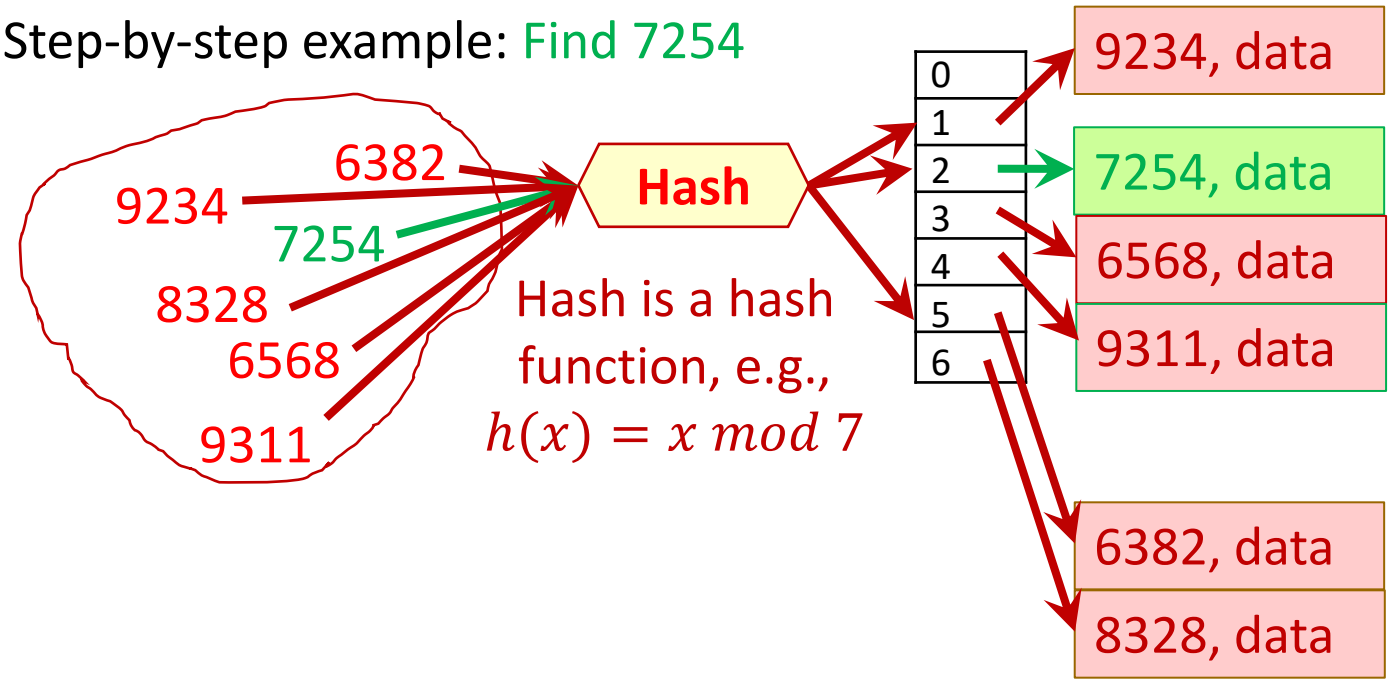


Insert object with key 6568. Hashed address 2 is occupied. Linearly locate the next available slot.

Step-by-step example: insert 9311

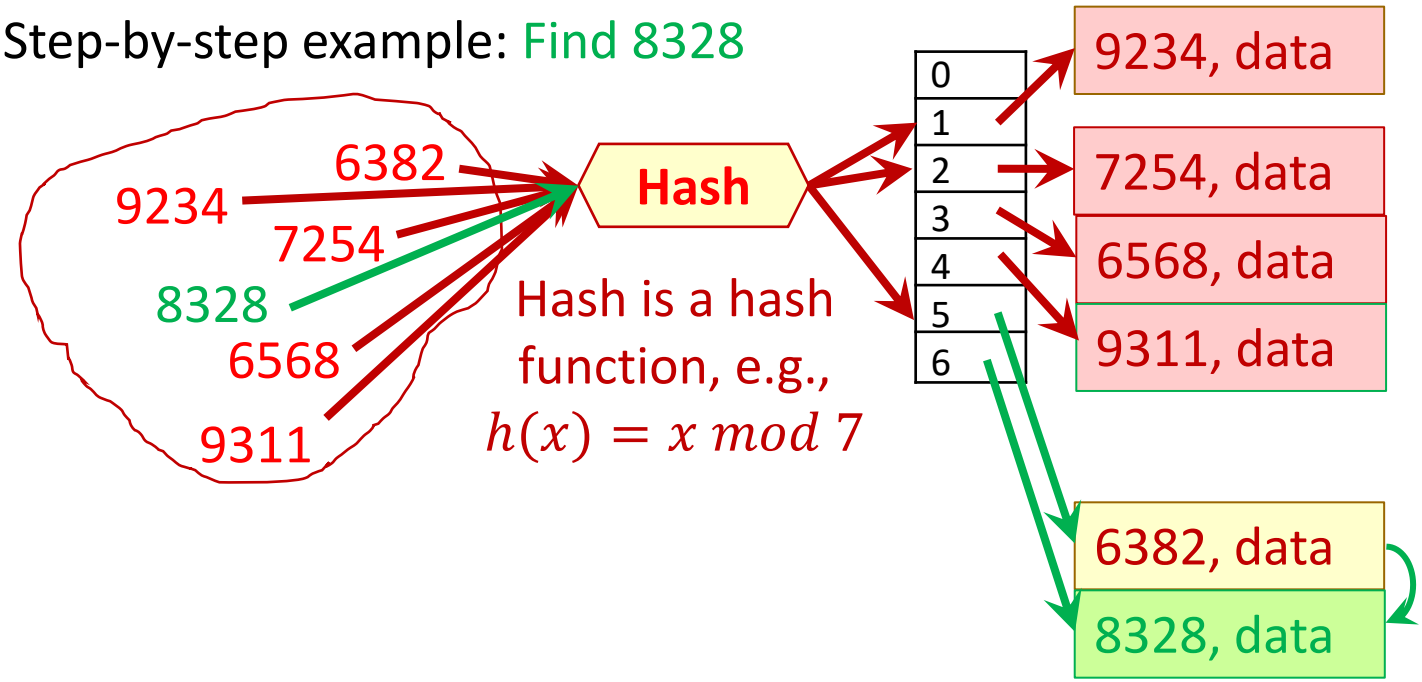


Step-by-step example: Find 7254



Find object with key 7254. Object is found in slot 2 $(h(7254))$.

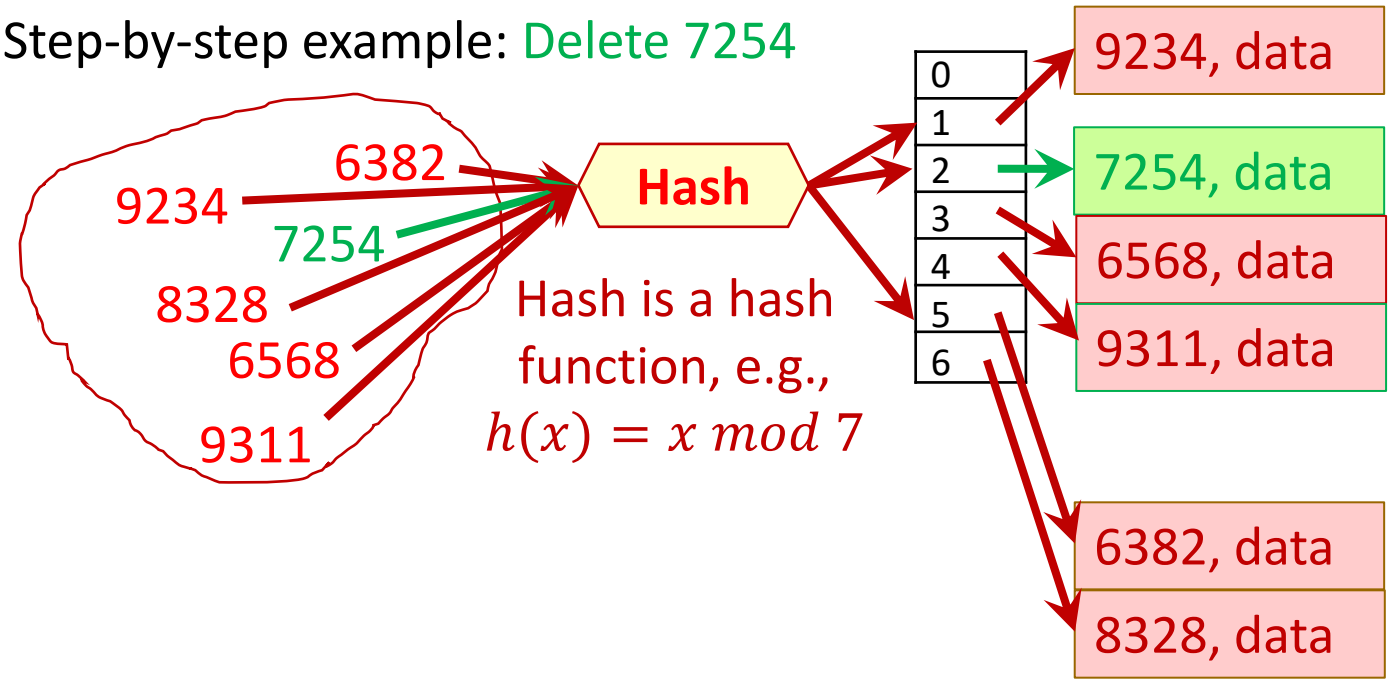
Step-by-step example: Find 8328



Find object with key 8328. Hash value $(h(8328))$ directs the search to slot 5. The key found in slot 5 does not match. Sequentially probe the next slot, and the object with key 8328 is found in slot 6.

Delete object from hash table

Step-by-step example: Delete 7254

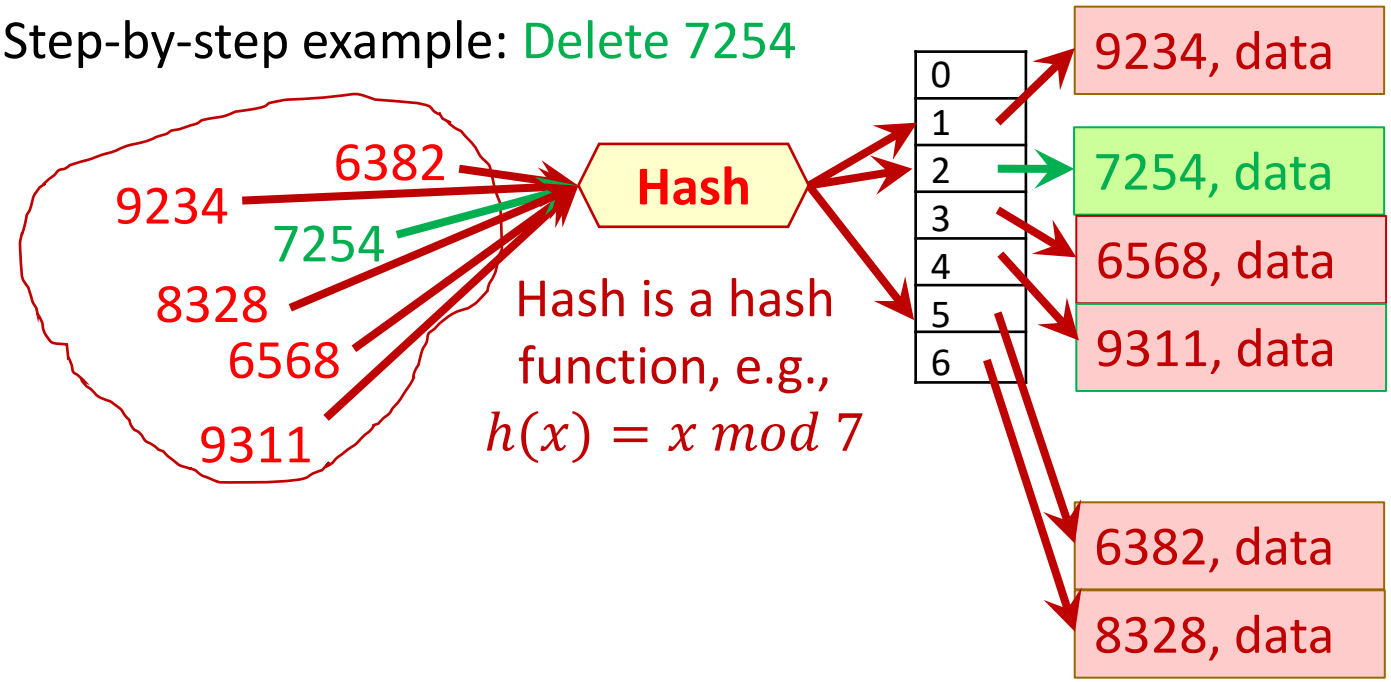


Delete object with key 7254.

Hash the key and the object is found in slot 2 $(h(7254))$.

What then?

Step-by-step example: Delete 7254



Delete object with key 7254.

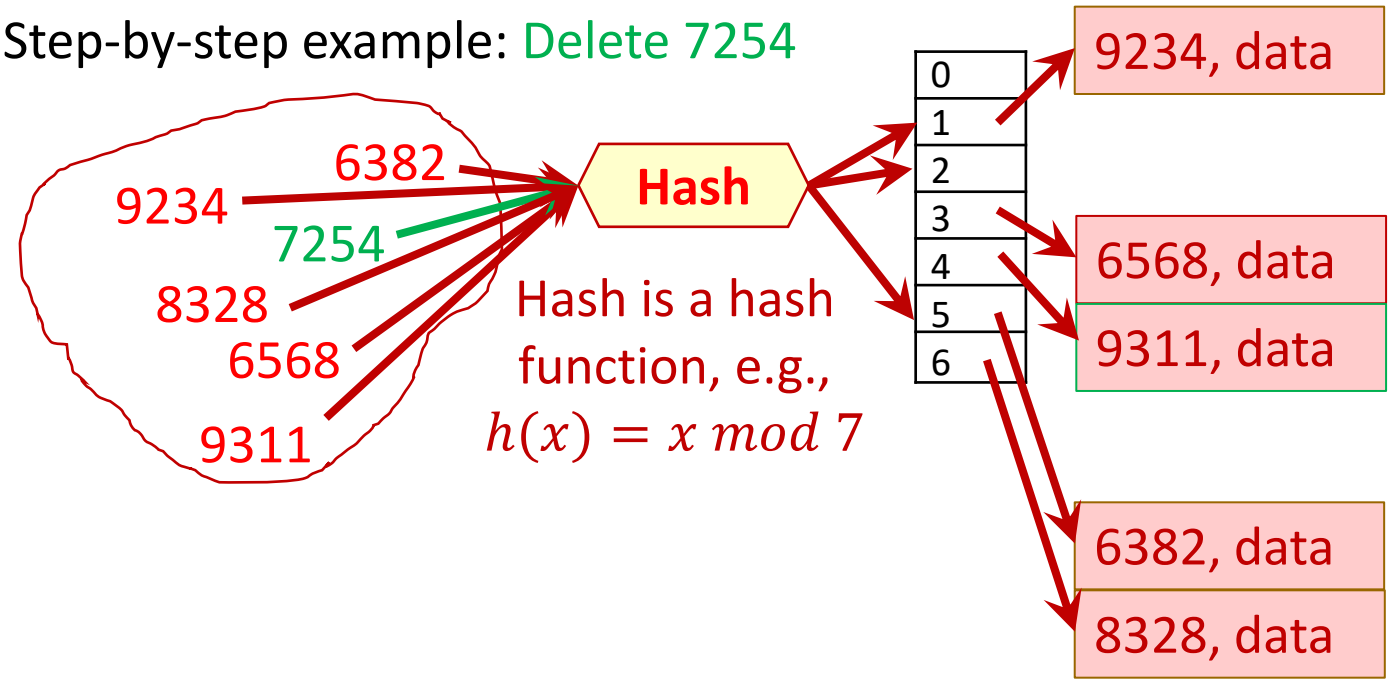
Hash the key and the object is found in slot 2 $(h(7254))$.

What then?

Remove the object from the hash table.

☹☹☹

Step-by-step example: Delete 7254



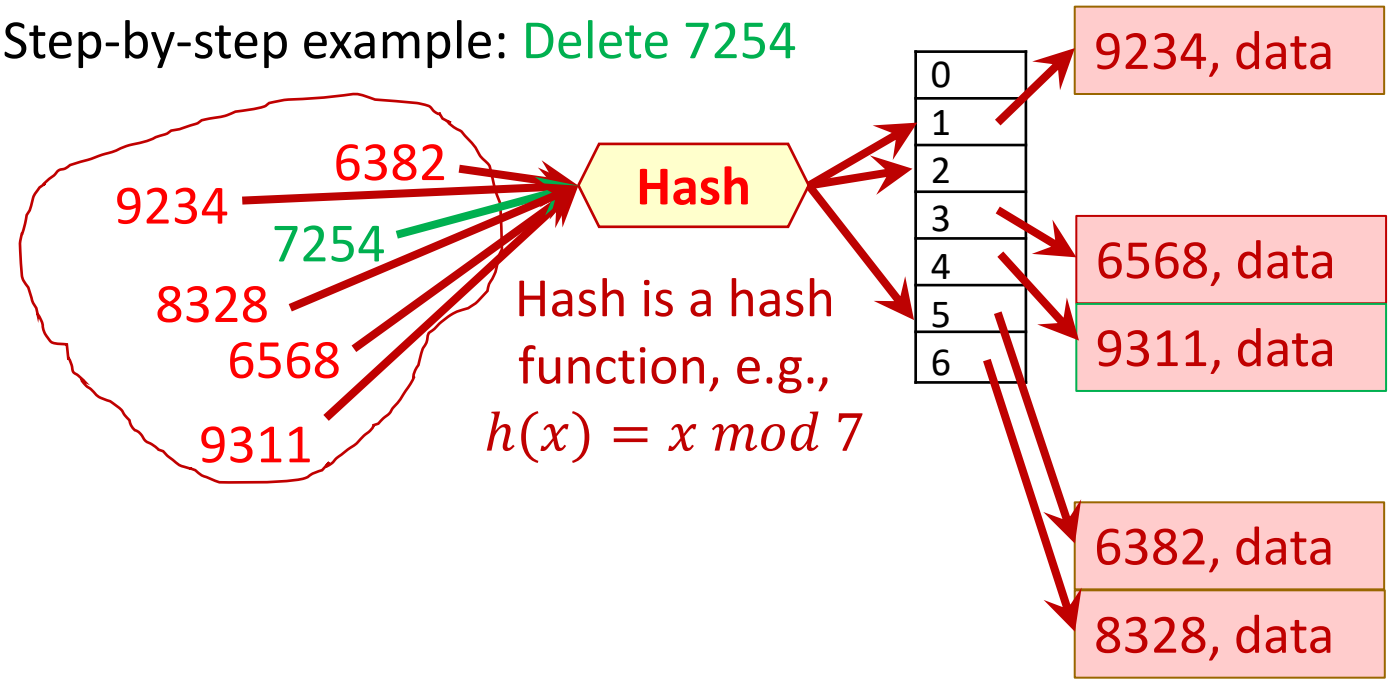
Delete object with key 7254.

Hash the key and the object is found in slot 2 $(h(7254))$.

What then?

Warning!!! To delete an object, we cannot remove the object from the hash table. Removing object from hash table will cause problem in subsequent searching process. We need to employ a 'Lazy Deletion' approach.

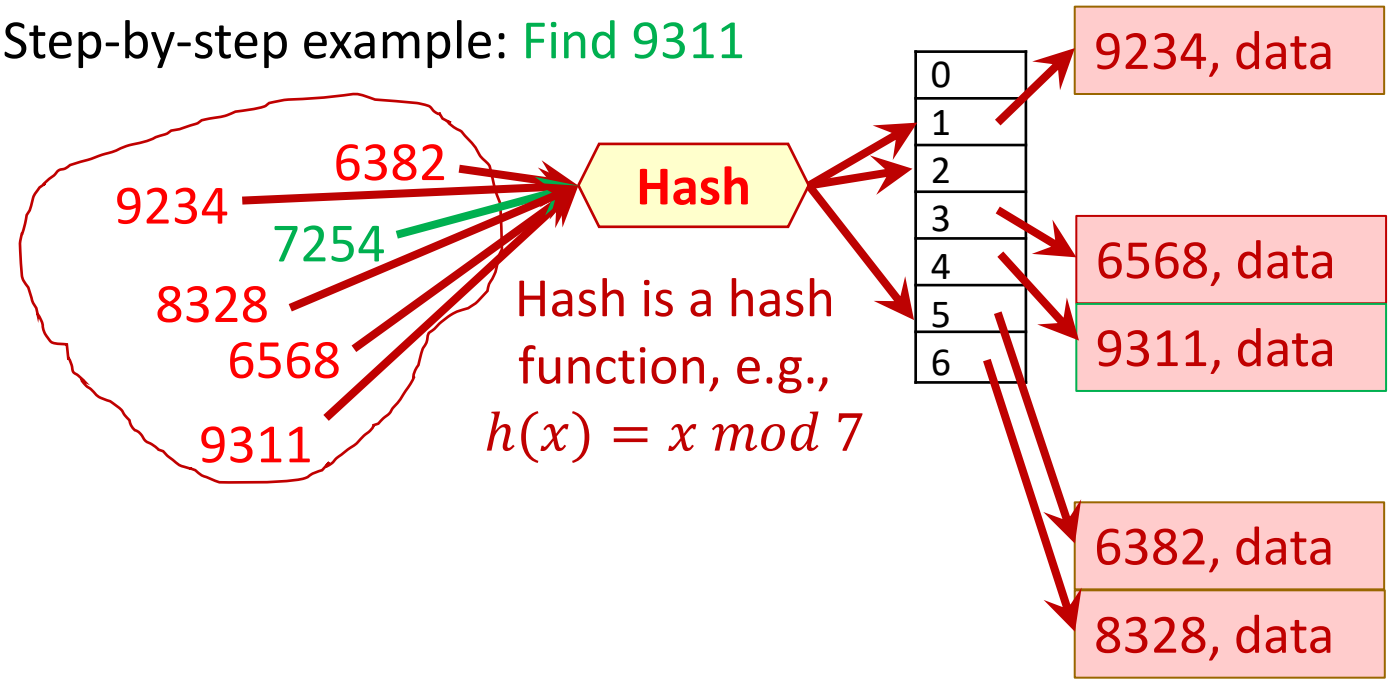
Step-by-step example: Delete 7254



Delete object with key 7254. Object is found in slot 2 $(h(7254))$. What then? Remove the object from the hash table. ☹☹☹

What is the problem with removing the object from the hash table?

Step-by-step example: Find 9311



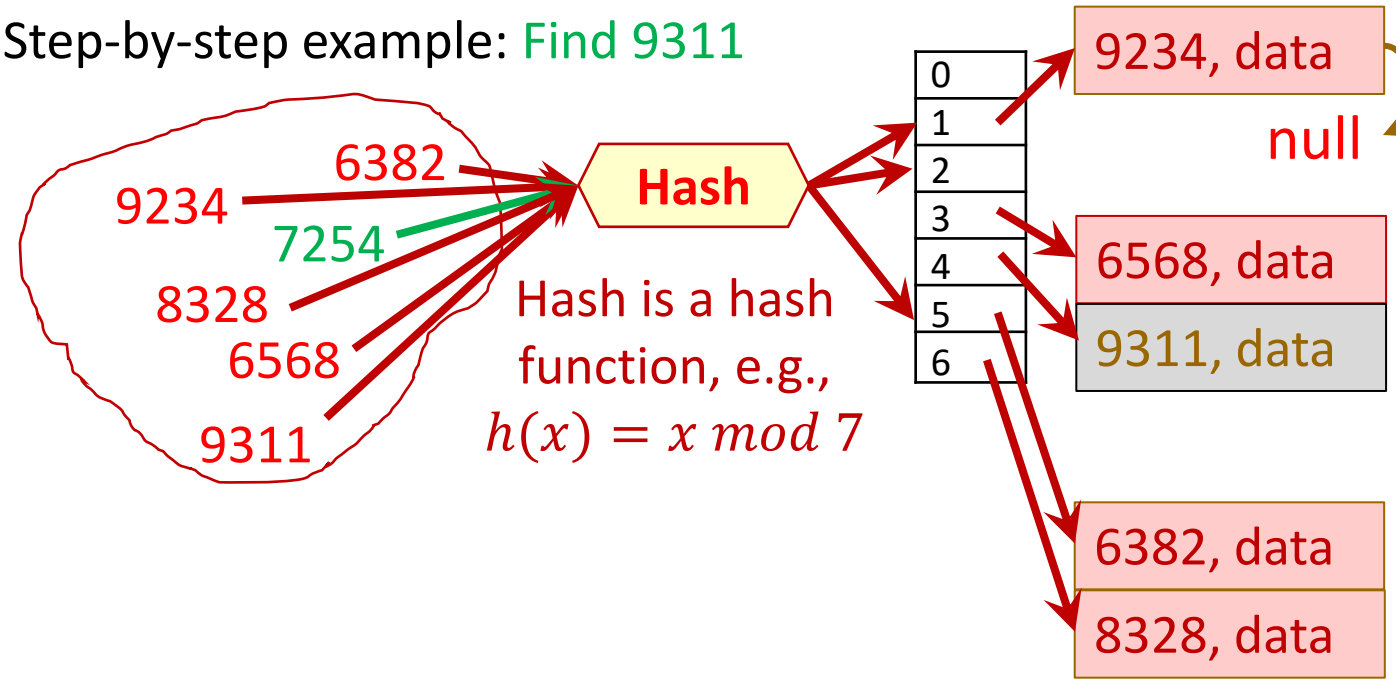
Find object with key 9311.

Hash the key and the object is found in slot 1 $(h(9311))$.

What then?

Assuming that the object with key 7254 is removed from the hash table as shown above. Next, we do a search for object with key 9311. Can we succeed?

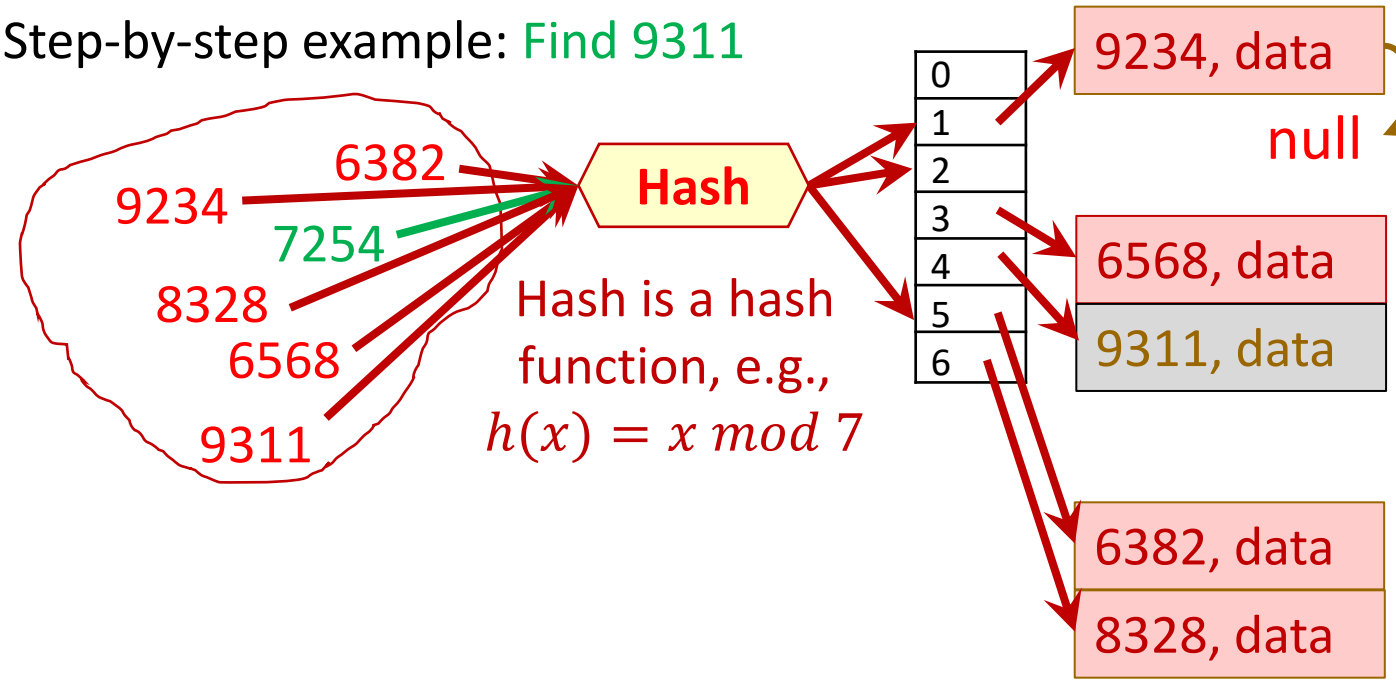
Step-by-step example: Find 9311



Find object with key 9311.

Hash the key and the search goes to slot 1. The key 9234 is not matching with the key 9311, the algorithm will probe the next slot, and a null is encountered.

Step-by-step example: Find 9311



Find object with key 9311.

Hash the key and the search goes to slot 1. The key 9234 is not matching with the key 9311, the algorithm will probe the next slot, and a null is encountered.

This will lead to unsuccessful search!!!

What is a lazy deletion?

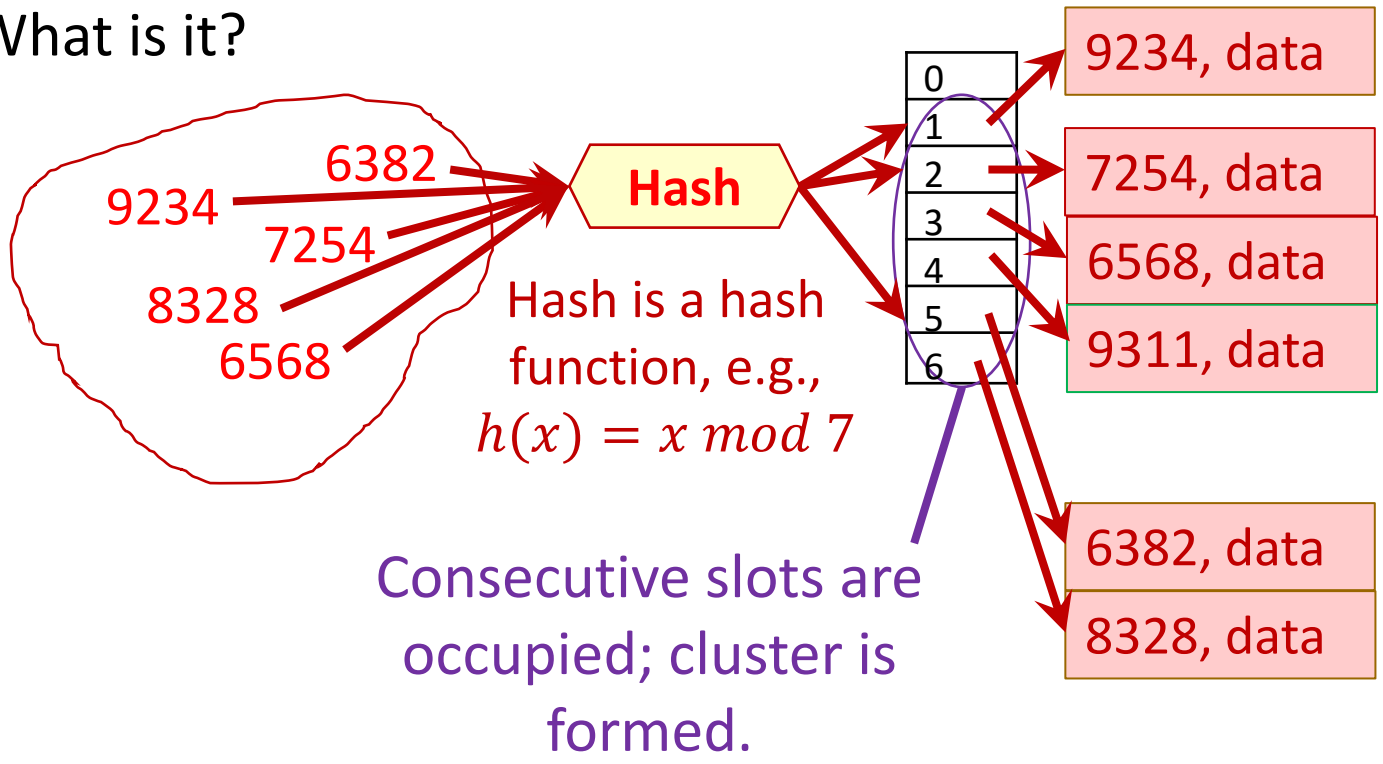
- Lazy deletion – instead of physically remove the object from the hash table, each slot of the hash table is given three different states (status):
  - Occupied
  - Deleted
  - Empty

What is a lazy deletion?

- When a value is removed from linear probed hash table, we change the status of the slot to “deleted”, instead of physically remove the object from the slot.
- This implementation will change the logic to insert. During insertion, if collision takes place, a probe sequence that detect “deleted” slot is considered “empty” and the new object can be inserted in the slot.

Primary Clustering

What is it?



Primary Clustering

- A cluster is a collection of consecutive occupied slots.
- Linear probing has a tendency to create primary clusters, because when collision takes place, the next available slots will be used, hence, cluster is formed.
- Cluster will increase the run-time complexity to find/insert/delete objects.
- To avoid forming of primary cluster, we modify the probe sequence to spread the distribution of keys such that slot that is d distance away is used, instead of next available slot.

Expected Number of Probes (Linear Probing)

- For linear probing, the following are the expected (average) successful and unsuccessful probe:

Successful probe	Unsuccessful probe
$\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$	$\frac{1}{2}\left(1+\left(\frac{1}{1-\alpha}\right)^2\right)$

$where\ \alpha\ is\ the\ load\ factor\ of\ the\ hash\ table.$

Example

- Given a hash table using linear probing collision resolution with load factor $\frac{7}{11}$, what is the expected number of probes in an unsuccessful search and the expected number of probes in a successful search?
- Unsuccessful search:

$$\frac{1}{2}\left(1+\left(\frac{1}{1-\alpha}\right)^2\right)=\frac{1}{2}\left(1+\left(\frac{1}{1-\frac{7}{11}}\right)^2\right)=\frac{1}{2}(1+2.75^2)$$

$$=4.28 \text{ probes.}$$

Example

- Successful search:

$$\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)=\frac{1}{2}\left(1+\frac{1}{1-\frac{7}{11}}\right)=\frac{1}{2}\left(1+\frac{1}{0.3636}\right)$$
$$=\frac{1}{2}(1+2.75)=1.875\ probes.$$

We will look at the computation of average number of probes with empirical case during tutorial sessions.

Open Addressing

Quadratic Probing

Quadratic Probing

Idea:

- When collision takes place, quadratic probing uses a formula that produces more "scattering" spread of keys to avoid primary clustering.

The probe sequence of quadratic probing is:

$hash(key)\ mod\ m$

$hash(key + 1)\ mod\ m$

$hash(key + 4)\ mod\ m$

$hash(key + 9)\ mod\ m$

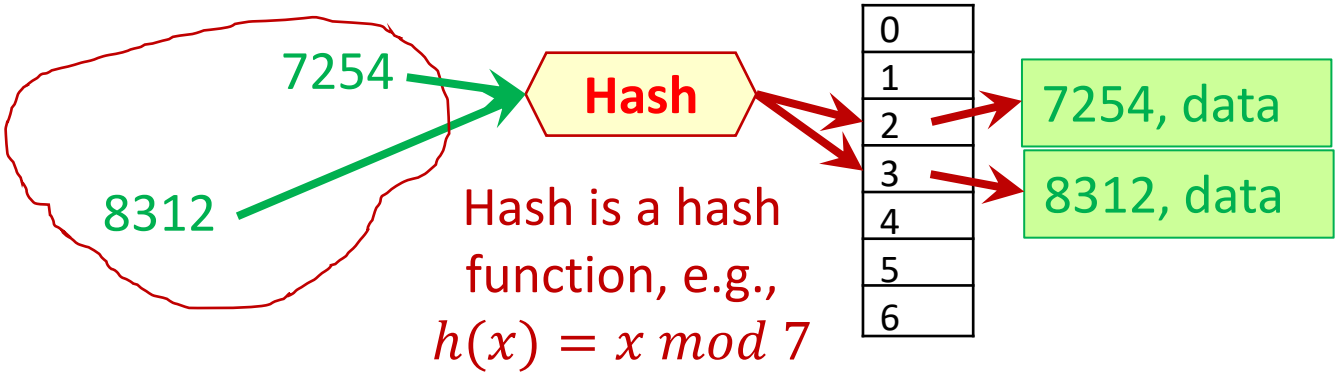
$hash(key + 16)\ mod\ m$

$\dots$

$\boldsymbol{hash(key + i^2)\ mod\ m}$

Quadratic Probing

Step-by-step example: insert 7254 and followed by 8312

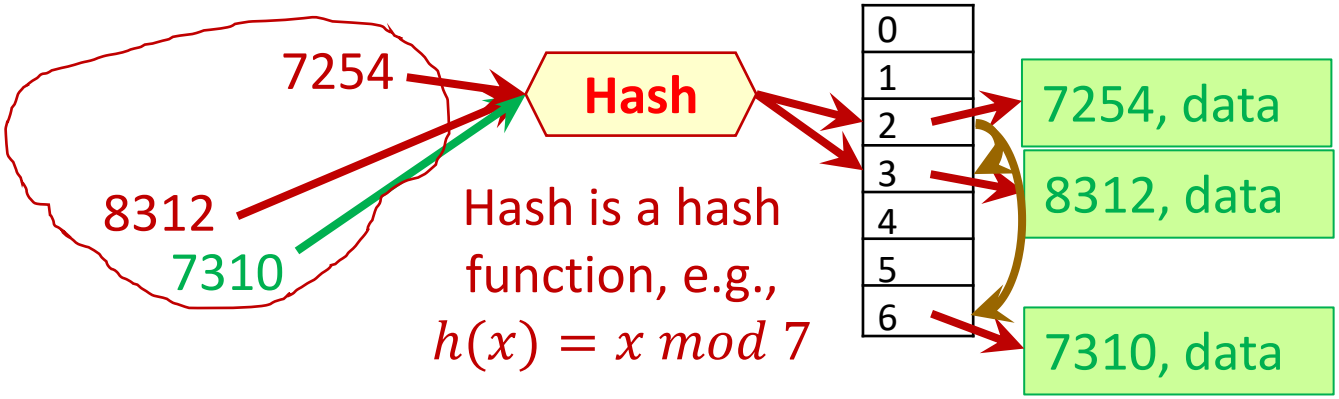


Hash is a hash function, e.g., $h(x) = x \ mod \ 7$

Insert object with key 7254, 8312.

Quadratic Probing

Step-by-step example: insert 7310

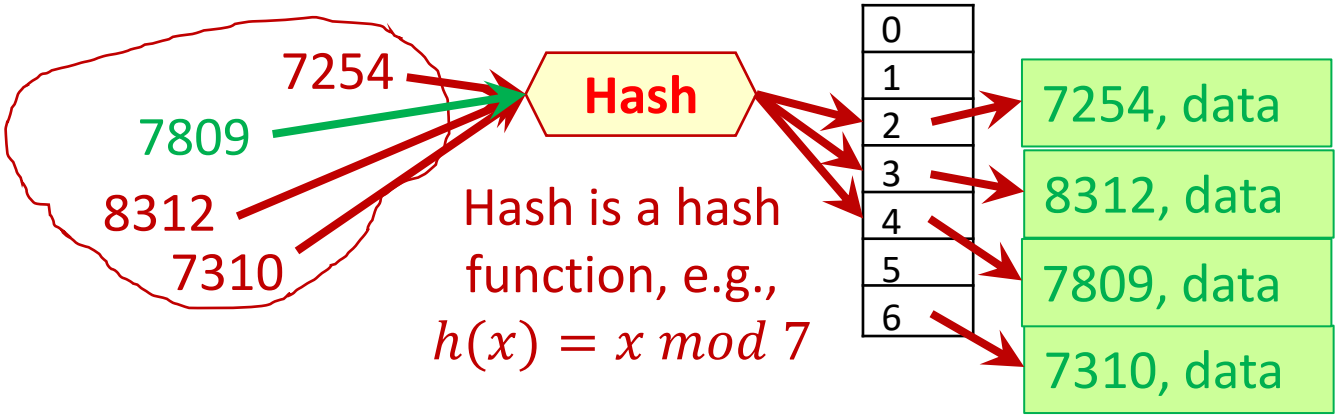


Insert object with key 7310. $h(7310) -> 2$, which is occupied. First probe leads to slot 3, which is occupied. Second probe leads to slot 6.

Slots 4 and 5 are skipped, hence avoid forming primary cluster.

Quadratic Probing

Step-by-step example: insert 7809



Insert object with key 7809. $h(7809) \rightarrow 4$, Object is inserted to slot 4.

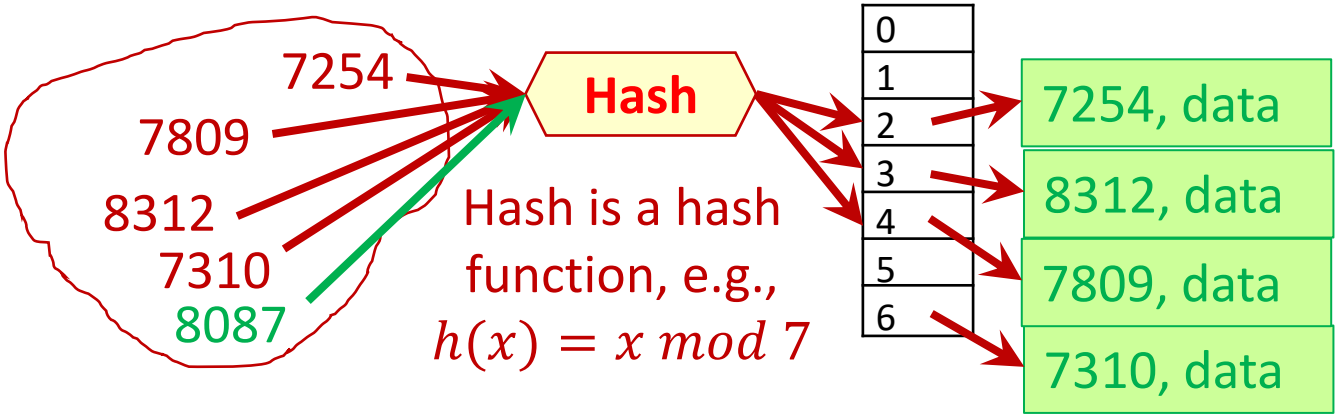
Problem With Quadratic Probing

- In quadratic probing, instead of forming primary clusters, quadratic probing forms a secondary clusters.
- Secondary clusters are formed along the path of probing.
- Secondary clusters are formed because of the same pattern in probing of keys due to the mathematical formula created (quadratic equation).
  - For example, if two keys have the same hashed value (home location), their probe sequences are going to be the same.

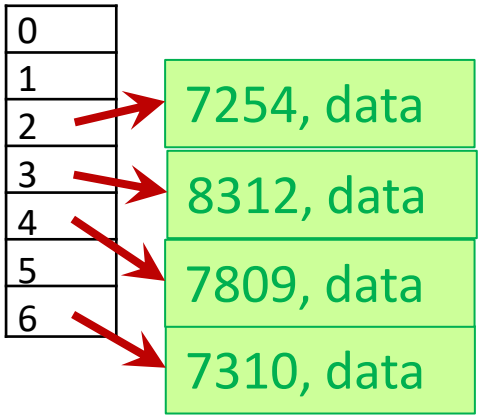
Problem With Quadratic Probing

- Quadratic probing can fail to find an empty slot even if are empty slots exist.
- Try insert 8087 into the hash table.

Step-by-step example: insert 8087



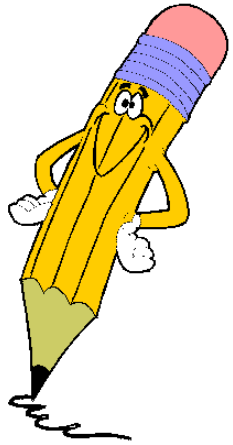
Insert object with key 8087.
$h(8087) \rightarrow 2$.



Noted that the hash table is not full, there are empty slots, but the probe sequence is unable to locate the empty slots.

The following are the quadratic probe sequences:

$h(8087) mod\ 7 \rightarrow 2$
$h(8087+1) mod\ 7 \rightarrow 3$
$h(8087+4) mod\ 7 \rightarrow 6$
$h(8087+9) mod\ 7 \rightarrow 4$
$h(8087+16) mod\ 7 \rightarrow 4$
$h(8087+25) mod\ 7 \rightarrow 6$
$h(8087+36) mod\ 7 \rightarrow 3$
$h(8087+49) mod\ 7 \rightarrow 2$
$h(8087+64) mod\ 7 \rightarrow 3$
$h(8087+81) mod\ 7 \rightarrow 6$
$\ldots$



Quadratic Probing

- Because of the spread, however, secondary clusters are not as serious compared to primary clusters.
- How to avoid secondary clusters then?

Quadratic Probing

- Because of the spread, however, secondary clusters are not as serious compared to primary clusters.
- How to avoid secondary clusters then?

Double hashing

Open Addressing

Double Hashing

Double Hashing

Idea:

- A sequence of possible slots to insert an element are produced using two hash functions.
- The first hash function is to determine the first slot;
- if the slot is occupied (collision),
  - The second hash function is used to determine the increment for the probe sequence.

Double Hashing

- The probe sequence of double hashing function has the following form:

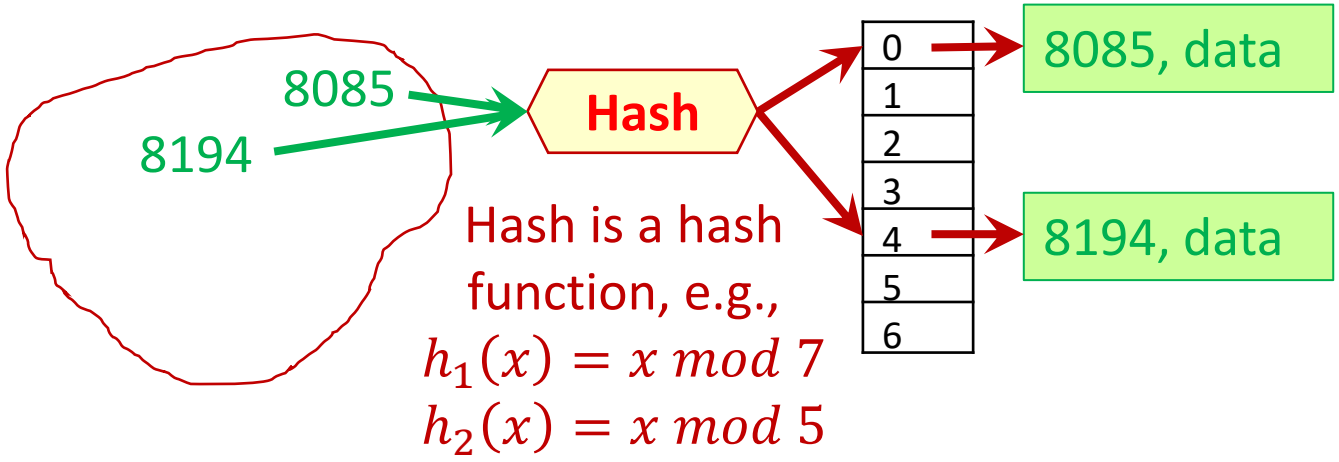
$$hash_1(key)$$
$$hash_1(key) + \left(1 \times hash_2(key)\right) mod\ m$$
$$hash_1(key) + \left(2 \times hash_2(key)\right) mod\ m$$
$$hash_1(key) + \left(3 \times hash_2(key)\right) mod\ m$$
$$\dots$$

$hash_2$ is called the secondary hash function.

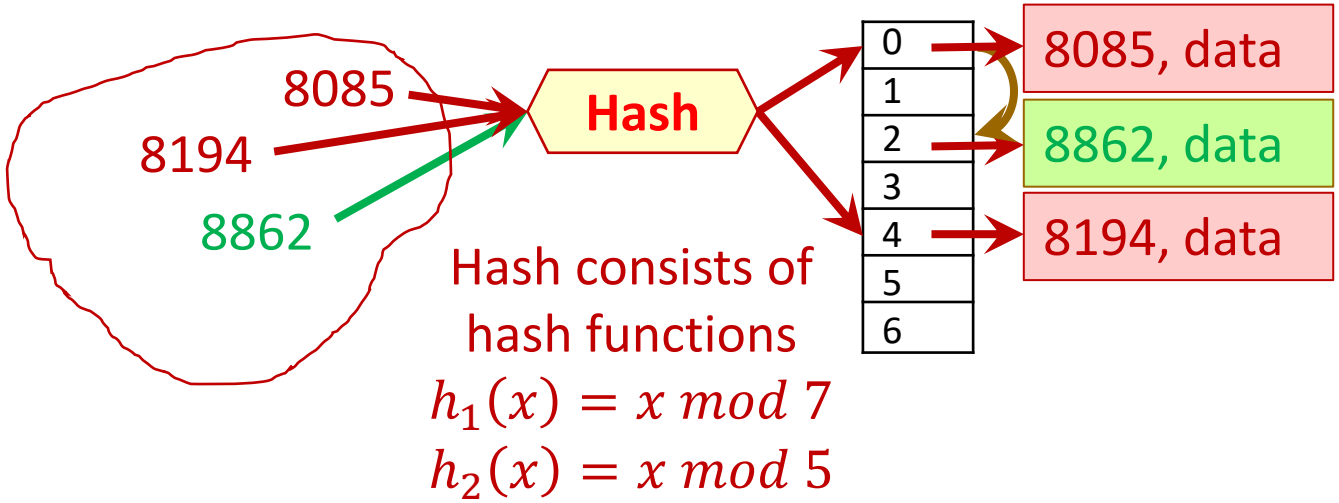
Double Hashing

- Using the following two hash functions as example:
  - $hash_1(k) = k \ mod \ 7$
  - $hash_2(k) = k \ mod \ 5$
- We first hash key 8085 followed by key 8194.

Step-by-step example: insert 8085 followed by 8194



Step-by-step example: insert 8862

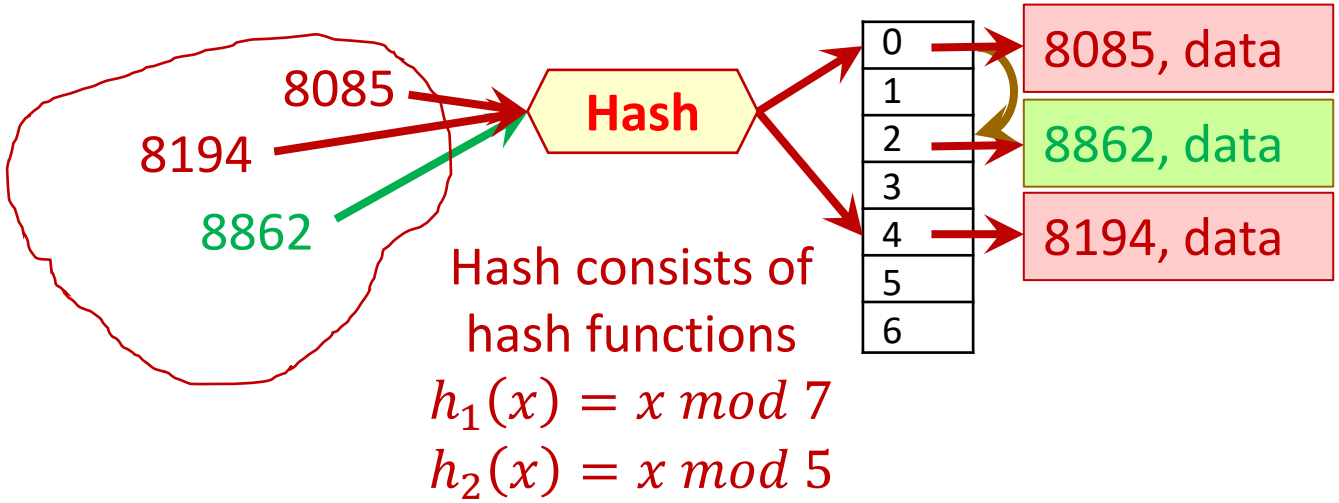


First hash probes to slot 0, which is occupied. First try of second hash probes to slot 2. → $Successful!$

What happen if the second time hash is also collide? For example, instead of inserting 8862, we insert 8099?

$$h(8099) = 8099 \ mod \ 7 = 0$$
$$h(8099) = 8099 \ mod \ 5 = 4$$

Step-by-step example: insert 8862



Hash consists of hash functions

$h_1(x) = x \bmod 7$

$h_2(x) = x \bmod 5$

First hash probes to slot 0, which is occupied. First try of second hash probes to slot 2. → $Successful!$

$h(x) = h_1(x) + \big(i \times h_2(x)\big) \bmod 7, \qquad where\ i = 0,1,2,3, \ldots$

$\big(8862 \bmod 7 + \ (1 \times 8862 \bmod 5)\big) \bmod 7$

$\big(0 + \ (2)\big) \bmod 7 = 2$

Instead of defining a third hash function and so on, an usual implementation is to continuously use the second hash function, but combine the first and second hash function together as follow:

$$h(x) = h_1(x) + \big(i \times h_2(x)\big)\ mod\ 7, \qquad where\ i = 0,1,2,3, \ldots$$

$$h(8862) = h_1(8862) + \big(0 \times h_2(8862)\big)\ mod\ 7, \qquad where\ i = 0$$
$$= (0 + 0)\ mod\ 7 = 0$$

$$h(8862) = h_1(8862) + \big(1 \times h_2(8862)\big)\ mod\ 7, \qquad where\ i = 1$$
$$= (0 + 2)\ mod\ 7 = 2$$

Taking the example of inserting 8099, we have

$$h(x) = h_1(x) + \big(i \times h_2(x)\big)\ mod\ 7, \qquad where\ i = 0,1,2,3, \ldots$$

$$h(8099) = h_1(8099) + \big(0 \times h_2(8099)\big)\ mod\ 7, \qquad where\ i = 0$$
$$= (0 + 0)\ mod\ 7 = 0$$

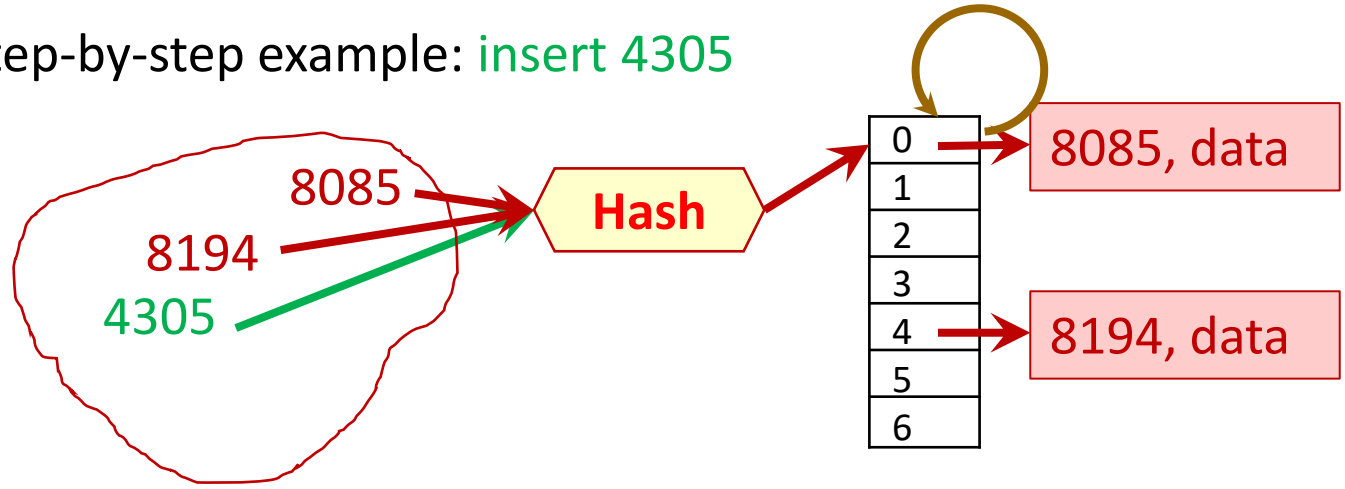
$$h(8099) = h_1(8099) + \big(1 \times h_2(8099)\big)\ mod\ 7, \qquad where\ i = 1$$
$$= (0 + 4)\ mod\ 7 = 4$$

$$h(8099) = h_1(8099) + \big(2 \times h_2(8099)\big)\ mod\ 7, \qquad where\ i = 2$$
$$= (0 + 8)\ mod\ 7 = 1$$

Double Hashing

- Under what situation, double hashing is not acceptable?
- For example, try to insert key 4305.

Step-by-step example: insert 4305



$$h(x) = h_1(x) + \big(i \times h_2(x)\big)\ mod\ 7, \qquad where\ i = 0,1,2,3, \ldots$$

$$\big(4305\ mod\ 7 + (0 \times 4305\ mod\ 5)\big)\ mod\ 7 = 0$$

$$\big(4305\ mod\ 7 + (1 \times 4305\ mod\ 5)\big)\ mod\ 7 = 0$$

$$\big(4305\ mod\ 7 + (2 \times 4305\ mod\ 5)\big)\ mod\ 7 = 0$$

$$\ldots$$

Double hashing

- With double hashing, $hash_2(key)$ must not be 0.
- To avoid $hash_2(key) = 0$, we try to meet the following conditions:
  - The size of hash table must be a prime m
  - Define $hash_2$ function as $hash_2(key) = (key \ mod \ s) + 1$, where
    - $s < m$ but s need not be prime,
    - Usually, $s = m - 1$

Expected Number of Probes

- For both the quadratic and double hashing, the expected (average) number of successful probes and unsuccessful probes in searching an object (key) in a hash table are the same, and are defined as follow:

Successful probe	Unsuccessful probe
$\frac{1}{\alpha}\left(\ln\frac{1}{1-\alpha}\right)$	$\frac{1}{1-\alpha}$

$where\ \alpha\ is\ the\ load\ factor\ of\ the\ hash\ table.$

Example

- Given a hash table using double hashing collision resolution with load factor $\frac{7}{11}$, what is the expected number of probes in an unsuccessful search and the expected number of probes in a successful search?
- Unsuccessful search:

$$\frac{1}{1-\alpha} = \frac{1}{1-\frac{7}{11}} = \frac{1}{1-0.636} = \frac{1}{0.36} = 2.75 \text{ probes}$$

Example

- Successful search:

$$\frac{1}{\alpha}\left(\ln\frac{1}{1-\alpha}\right) = \frac{1}{\frac{7}{11}}\left(\ln\frac{1}{1-\frac{7}{11}}\right) = \frac{11}{7}(\ln 2.75)$$
$$= 1.59 \text{ probes.}$$

We will look at the computation of average number of probes with empirical case during tutorial sessions.

Good Collision Resolution Method

- Minimize clustering
- Always find an empty slot if it exists
- Give different prove sequences when 2 keys collide, that is, no secondary clustering
- Efficient with $O(1)$ run-time.

When To Rehash?

- Time to rehash
  - When the table is getting full, the operations are getting slow.
  - For quadratic probing, insertions might fail when the table is more than half full.

When To Rehash?

- Rehash operation
  - Build another table about twice as big with a new hash function.
  - Scan the original table, or each key, compute the new hash value and insert the data into the new hash table
  - Delete the original table

When To Rehash?

- Guideline: Use the load factor to decide when to rehash.
  - For open addressing: 0.5
  - For closed addressing: 1

Associative tables - Efficiency

- Insertion into an associative table is efficient; Θ(1)
- Finding an entry in an associative table is efficient; Θ(1)
- Deletion from an associative table is efficient; Θ(1), although some care must be taken in handling deletions from the second array based implementation
- Listing of entries (especially ordered listing) is not efficient.

Summary

- Array
- Stack
- Queue
- Linked Lists
- Record (structures)
- Graphs
  - Directed/undirected graph
  - Graph representation: adjacency matrix/ edge list

Summary

- Trees
  - k-ary tree
  - Tree terminology: leaf, internal node, depth, height, level
  - Binary tree: full binary tree, complete binary tree
  - Binary Search Tree
  - Heap

Summary

- Associative tables
  - Implementation using linked list
  - Implementation using array: 2 alternatives