Hacettepe University
Computer Engineering Department

BBM204 Software Laboratory II

Week II
Recitation I

Spring 2019

Give the order of growth (as a function of *N*) of the running times of each of the following code fragments:

typical code framework	description	example
`a = b + c;`	*statement*	*add two numbers*

`double max = a[0];`
`for (int i = 1; i < N; i++)`
`   if (a[i] > max) max = a[i];`
	loop	*find the maximum*

`for (int i = 0; i < N; i++)`
`   for (int j = i+1; j < N; j++)`
`      if (a[i] + a[j] == 0)`
`         cnt++;`
	double loop	*check all pairs*

`for (int i = 0; i < N; i++)`
`   for (int j = i+1; j < N; j++)`
`      for (int k = j+1; k < N; k++)`
`         if (a[i] + a[j] + a[k] == 0)`
`            cnt++;`
	triple loop	*check all triples*

Solution

order of growth	typical code framework	description	example
1	`a = b + c;`	*statement*	*add two numbers*

N	`double max = a[0];`
	`for (int i = 1; i < N; i++)`
	`   if (a[i] > max) max = a[i];`
		loop	*find the maximum*

N^2	`for (int i = 0; i < N; i++)`
	`   for (int j = i+1; j < N; j++)`
	`      if (a[i] + a[j] == 0)`
	`         cnt++;`
		double loop	*check all pairs*

N^3	`for (int i = 0; i < N; i++)`
	`   for (int j = i+1; j < N; j++)`
	`      for (int k = j+1; k < N; k++)`
	`         if (a[i] + a[j] + a[k] == 0)`
	`            cnt++;`
		triple loop	*check all triples*

Find the complexity of the below program:

```
void function(int n)
{
    int count = 0;
    for (int i=n/2; i<=n; i++)
        for (int j=1; j<=n; j = 2 * j)
            for (int k=1; k<=n; k = k * 2)
                count++;
}
```

Solution

```
void function(int n)
{
    int count = 0;
    for (int i=n/2; i<=n; i++)

        // Executes O(Log n) times
        for (int j=1; j<=n; j = 2 * j)

            // Executes O(Log n) times
            for (int k=1; k<=n; k = k * 2)
                count++;
}
```

- Time complexity of this program **O(n log²n).**

Find the complexity of the below program:

```
void function(int n)
{
    int count = 0;
    for (int i=n/2; i<=n; i++)
        for (int j=1; j+n/2<=n; j = j++)
            for (int k=1; k<=n; k = k * 2)
                count++;
}
```

Solution

```
void function(int n)
{
    int count = 0;

    // outer loop executes n/2 times
    for (int i=n/2; i<=n; i++)

        // middle loop executes  n/2 times
        for (int j=1; j+n/2<=n; j = j++)

            // inner loop executes logn times
            for (int k=1; k<=n; k = k * 2)
                count++;
}
```

- Time Complexity of the this function **O(n^2logn).**

Find the complexity of the below program:

```
void function(int n)
{
    int count = 0;
    for (int i=0; i<n; i++)
        for (int j=i; j< i*i; j++)
            if (j%i == 0)
            {
                for (int k=0; k<j; k++)
                    printf("*");
            }
}
```

Solution

```c
void function(int n)
{
    int count = 0;

    // executes n times
    for (int i=0; i<n; i++)

        // executes O(n*n) times.
        for (int j=i; j< i*i; j++)
            if (j%i == 0)
            {
                // executes j times = O(n*n) times
                for (int k=0; k<j; k++)
                    printf("*");
            }
}
```

- Time Complexity of the this function **O(n^5).**

Hacettepe University
Computer Engineering Department

BBM204 Software Laboratory II

Week III
Recitation II

Spring 2019

1) Show in the style of the example trace with selection sort, how selection sort sorts the array: E A S Y Q U E S T I O N

		a[ ]											
i	min	0	1	2	3	4	5	6	7	8	9	10	11
		E	A	S	Y	Q	U	E	S	T	I	O	N
0	1	E	A	S	Y	Q	U	E	S	T	I	O	N
1	1	A	E	S	Y	Q	U	E	S	T	I	O	N
2	6	A	E	S	Y	Q	U	E	S	T	I	O	N
3	9	A	E	E	Y	Q	U	S	S	T	I	O	N
4	11	A	E	E	I	Q	U	S	S	T	Y	O	N
5	10	A	E	E	I	N	U	S	S	T	Y	O	Q
6	11	A	E	E	I	N	O	S	S	T	Y	U	Q
7	7	A	E	E	I	N	O	Q	S	T	Y	U	S
8	11	A	E	E	I	N	O	Q	S	T	Y	U	S
9	11	A	E	E	I	N	O	Q	S	S	Y	U	T
10	10	A	E	E	I	N	O	Q	S	S	T	U	Y
11	11	A	E	E	I	N	O	Q	S	S	T	U	Y
		A	E	E	I	N	O	Q	S	S	T	U	Y

Selection Sort Example

```java
public class SelectionSortExample
{
    public static void sort(Comparable[] a)
    {   //Sort a[] into increasing order.
        int N = a.length; // array length
        for (int i = 0; i < N; i++)
        { // Exchange a[i] with smallest entry in a[i+1...]
         int min = i; // index of minimal entr.
         for (int j = i+1; j < N; j++)
             if (less(a[j], a[min]))
                min = j;
         exch(a, i, min);
        }
    }

    private static boolean less(Comparable v, Comparable w)
    {
        return v.compareTo(w) < 0;

    }
    private static void exch(Comparable[] a, int i, int j)
    {
        Comparable t = a[i]; a[i] = a[j]; a[j] = t;
    }
```

```java
private static void show(Comparable[] a)
{ // Print the array, on a single line.
    for (int i = 0; i < a.length; i++)
        StdOut.print(a[i] + " ");
    StdOut.println();
}

public static boolean isSorted(Comparable[] a)
{ // Test whether the array entries are in order.
    for (int i = 1; i < a.length; i++)
        if (less(a[i], a[i-1])) return false;
            return true;
}

public static void main(String[] args)
{ // Read strings from standard input, sort them, and print.
    String[] a = In.readStrings();
    sort(a);
    assert isSorted(a);
    show(a);
}
}
```

What is the maximum number of exchanges involving any particular item during selection sort? What is the average number of exchanges involving an item?

		a[ ]											
i	min	0	1	2	3	4	5	6	7	8	9	10	11
		Z	A	B	C	D	E	F	G	H	I	J	K
0	1	Z	A	B	C	D	E	F	G	H	I	J	K
1	2	A	Z	B	C	D	E	F	G	H	I	J	K
2	3	A	B	Z	C	D	E	F	G	H	I	J	K
3	4	A	B	C	Z	D	E	F	G	H	I	J	K
4	5	A	B	C	D	Z	E	F	G	H	I	J	K
5	6	A	B	C	D	E	Z	F	G	H	I	J	K
6	7	A	B	C	D	E	F	Z	G	H	I	J	K
7	8	A	B	C	D	E	F	G	Z	H	I	J	K
8	9	A	B	C	D	E	F	G	H	Z	I	J	K
9	10	A	B	C	D	E	F	G	H	I	Z	J	K
10	11	A	B	C	D	E	F	G	H	I	J	Z	K
11	11	A	B	C	D	E	F	G	H	I	J	K	Z
		A	B	C	D	E	F	G	H	I	J	K	Z

Selection Sort

```
public static void sort(Comparable[] a)
{   //Sort a[] into increasing order.
    int N = a.length; // array length
    for (int i = 0; i < N; i++)
    { // Exchange a[i] with smallest entry in a[i+1...N).
     int min = i; // index of minimal entr.
     for (int j = i+1; j < N; j++)
         if (less(a[j], a[min]))
            min = j;
     exch(a, i, min);
    }
}
```

- The average number of exchanges is exactly 1 because there are exactly N exchanges and N items. The maximum number of exchanges is N, as in the following example.

Show in the style of the example trace with insertion sort, how insertion sort sorts the array : E A S Y Q U E S T I O N

a[ ]

i	j	0	1	2	3	4	5	6	7	8	9	10	11
		E	A	S	Y	Q	U	E	S	T	I	O	N
0	0	E	A	S	Y	Q	U	E	S	T	I	O	N
1	0	A	E	S	Y	Q	U	E	S	T	I	O	N
2	2	A	E	S	Y	Q	U	E	S	T	I	O	N
3	3	A	E	S	Y	Q	U	E	S	T	I	O	N
4	2	A	E	Q	S	Y	U	E	S	T	I	O	N
5	4	A	E	Q	S	U	Y	E	S	T	I	O	N
6	2	A	E	E	Q	S	U	Y	S	T	I	O	N
7	5	A	E	E	Q	S	S	U	Y	T	I	O	N
8	6	A	E	E	Q	S	S	T	U	Y	I	O	N
9	3	A	E	E	I	Q	S	S	T	U	Y	O	N
10	4	A	E	E	I	O	Q	S	S	T	U	Y	N
11	4	A	E	E	I	N	O	Q	S	S	T	U	Y
		A	E	E	I	N	O	Q	S	S	T	U	Y

Insertion Sort Example

```java
public class InsertionSortExample
{
    public static void sort(Comparable[] a)
    {   // Sort a[] into increasing order.
        int N = a.length;
        for (int i = 1; i < N; i++)
        { // Insert a[i] among a[i-1], a[i-2], a[i-3]... ..
            for (int j = i; j > 0 && less(a[j], a[j-1]); j--)
                exch(a, j, j-1);
        }
    }

    private static boolean less(Comparable v, Comparable w)
    {
        return v.compareTo(w) < 0;

    }
    private static void exch(Comparable[] a, int i, int j)
    {
        Comparable t = a[i]; a[i] = a[j]; a[j] = t;
    }

    private static void show(Comparable[] a)
    { // Print the array, on a single line.
        for (int i = 0; i < a.length; i++)
            StdOut.print(a[i] + " ");
        StdOut.println();
    }

    public static boolean isSorted(Comparable[] a)
    { // Test whether the array entries are in order.
        for (int i = 1; i < a.length; i++)
            if (less(a[i], a[i-1])) return false;
                return true;
    }

    public static void main(String[] args)
    { // Read strings from standard input, sort them, and print.
        String[] a = In.readStrings();
        sort(a);
        assert isSorted(a);
        show(a);
    }
}
```

Which method runs fastest for an array with all keys identical, selection sort or insertion sort?

Selection Sort vs Insertion Sort

```
public static void sort(Comparable[] a)
{   //Sort a[] into increasing order.
    int N = a.length; // array length
    for (int i = 0; i < N; i++)
    { // Exchange a[i] with smallest entry in a[i+1...N).
     int min = i; // index of minimal entr.
     for (int j = i+1; j < N; j++)
         if (less(a[j], a[min]))
            min = j;
     exch(a, i, min);
    }
}
```

```
public static void sort(Comparable[] a)
{   // Sort a[] into increasing order.
    int N = a.length;
    for (int i = 1; i < N; i++)
    { // Insert a[i] among a[i-1], a[i-2], a[i-3]... ..
       for (int j = i; j > 0 && less(a[j], a[j-1]); j--)
          exch(a, j, j-1);
    }
}
```

Insertion sort runs in linear time when all keys are equal.

- Suppose that we use insertion sort on a randomly ordered array where items have only one of three key values. Is the running time linear, quadratic, or something in between?

Quadratic.

Merge Sort

MergeSort(arr[], l, r)

If r > l

1. Find the middle point to divide the array into two halves:

    middle m = (l+r)/2

2. Call mergeSort for first half:

    Call mergeSort(arr, l, m)

3. Call mergeSort for second half:

    Call mergeSort(arr, m+1, r)

4. Merge the two halves sorted in step 2 and 3:

    Call merge(arr, l, m, r)

Given the following array as input, illustrate how the Mergesort algorithm performs. To illustrate theMergesort's behavior, start with the dividing of the array until the end condition of the recursive function is met and then show how the merge is performed.

3	8	4	10	1	5	6	9

3	8	4	10	1	5	6	9

3	8	4	10		1	5	6	9

3	8		4	10		1	5		6	9

3	8		4	10		1	5		6	9

3	4	8	10		1	5	6	9

3		8		4		10		1		5		6		9

1	3	4	5	6	8	9	10

Hacettepe University

Computer Engineering Department

BBM204 Software Laboratory II

Week IV

Recitation III

Spring 2019

Q1. What is the execution time (as a function of N) and complexity of each line (with big O notation) in the code fragments?

i=1;	O( )
while (i<=N) {	O( )
j = 1;	O( )
while (j<N) {	O( )
j=j+1; }	O( )
i=2*i;	O( )
}	

Solution

	time	O()
i=1;	1	O(1)
while (i<=N) {	Log (N) +2	O(Log (N))
j = 1;	Log (N) +1	O(Log (N))
while (j<N) {	(Log (N) +1) *(N)	O(N Log (N))
j=j+1; }	(Log (N) +1) * (N-1)	O(N Log (N))
i=2*i;	Log (N) +1	O(Log (N))
}		

Q2. Write the time complexities of Selection and Insertion Sort algorithms (Best Case and worst Case states for each algorithm).

- Selection Sort requires two nested for loops to complete itself. Hence for a given input size of n, following will be the time and space complexity for selection sort algorithm:
  i. Worst Case Time Complexity [Big-O]: **O(n^2)**
  ii. Best Case Time Complexity [Big-omega]: **O(n^2)**

```
void sort(int arr[])
{
    int n = arr.length;

    // One by one move boundary of unsorted subarray
    for (int i = 0; i < n-1; i++)
    {
        // Find the minimum element in unsorted array
        int min_idx = i;
        for (int j = i+1; j < n; j++)
            if (arr[j] < arr[min_idx])
                min_idx = j;

        // Swap the found minimum element with the first
        // element
        int temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
}
```

- If we provide an already sorted array to the insertion sort algorithm, it will still execute the outer for loop, thereby requiring **n** steps to sort an already sorted array of n elements, which makes its best case time complexity a linear function of n.
    i. Worst Case Time Complexity [Big-O]: **O(n²)**
    ii. Best Case Time Complexity [Big-O]: **O(n)**

```
/*Function to sort array using insertion sort*/
void sort(int arr[])
{
    int n = arr.length;
    for (int i = 1; i < n; ++i) {
        int key = arr[i];
        int j = i - 1;

        /* Move elements of arr[0..i-1], that are
           greater than key, to one position ahead
           of their current position */
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

b) Applying Insertion Sort algorithm on the array below, sort the elements in descending order. -show each step

56	9	31	77	90	17	4	6	28	4

step1	56	9	31	77	90	17	4	6	28	4
step2	56	31	9	77	90	17	4	6	28	4
step3	56	31	9	77	90	17	4	6	28	4
step4	56	31	77	9	90	17	4	6	28	4
step5	56	77	31	9	90	17	4	6	28	4
step6	77	56	31	9	90	17	4	6	28	4
step7	77	56	31	9	90	17	4	6	28	4
step8	77	56	31	90	9	17	4	6	28	4
step9	77	56	90	31	9	17	4	6	28	4
step10	77	90	56	31	9	17	4	6	28	4
step11	90	77	56	31	9	17	4	6	28	4
step12	90	77	56	31	17	9	4	6	28	4
step13	90	77	56	31	17	9	4	6	28	4
step14	90	77	56	31	17	9	6	4	28	4
step15	90	77	56	31	17	9	6	4	28	4
step16	90	77	56	31	17	9	6	28	4	4
step17	90	77	56	31	17	9	28	6	4	4
step18	90	77	56	31	17	28	9	6	4	4
step19	90	77	56	31	28	17	9	6	4	4
step20	90	77	56	31	28	17	9	6	4	4

Q3: Complete the Merge Sort Algorithm.

```
public class MergeSort {
    public void main(String[] args) {
        int[] a = { 5, 1, 6, 2, 3, 4 };
        mergeSort(a, a.length);
        for (int i = 0; i < a.length; i++)
            System.out.println(a[i]);
    }
    public void mergeSort(int[] a, int n) {


    }
    public void merge(int[] a, int[] l, int[] r, int left, int right) {


    }
}
```

```java
public class MergeSort {
    public void main(String[] args) {
        int[] a = { 5, 1, 6, 2, 3, 4 };
        mergeSort(a, a.length);
        for (int i = 0; i < a.length; i++)
            System.out.println(a[i]);
    }
    public void mergeSort(int[] a, int n) {
        if (n < 2)
            return;
        int mid = n / 2;
        int[] l = new int[mid];
        int[] r = new int[n - mid];
        for (int i = 0; i < mid; i++) {
            l[i] = a[i];
        }
        for (int i = mid; i < n; i++) {
            r[i - mid] = a[i];
        }
        mergeSort(l, mid);
        mergeSort(r, n - mid);
        merge(a, l, r, mid, n - mid);
    }
    public void merge(int[] a, int[] l, int[] r, int left, int right) {
        int i = 0, j = 0, k = 0;
        while (i < left && j < right) {
            if (l[i] <= r[j])
                a[k++] = l[i++];
            else
                a[k++] = r[j++];
        }
        while (i < left)
            a[k++] = l[i++];
        while (j < right)
            a[k++] = r[j++];
    }
}
```

Q. Like and binary search, ternary search is a searching technique that is used to determine the position of a specific value in an array. In binary search, the sorted array is divided into two parts while in ternary search, it is divided into three parts and then you determine in which part the element exists.

- a) Complete the ternary search function below bu using the definition above;

```
int ternary_search(int l,int r, int key)
{

        …

}
```

```
int ternary_search(int l,int r, int key)
{
    if(r>=l)
    {
        int mid1 = l + (r-l)/3;
        int mid2 = r -  (r-l)/3;
        if(ar[mid1] == key)
            return mid1;
        if(ar[mid2] == key)
            return mid2;
        if(key<ar[mid1])
            return ternary_search(l,mid1-1,key);
        else if(key>ar[mid2])
            return ternary_search(mid2+1,r,key);
        else
            return ternary_search(mid1+1,mid2-1,key);

    }
    return -1;
}
```

Hacettepe University

Computer Engineering Department

BBM204 Software Laboratory II

Week V

Recitation IV

Spring 2019

Binary Search Tree

```
class BinarySearchTree {

    /* Class containing left and right child of current node and key value*/
    class Node {

    // Root of BST
    Node root;

    // Constructor
    BinarySearchTree() {

    // This method mainly calls insertRec()
    void insert(int key) {

    /* A recursive function to insert a new key in BST */
    Node insertRec(Node root, int key) {

    // This method mainly calls InorderRec()
    void inorder() {

    // A utility function to do inorder traversal of BST
    void inorderRec(Node root) {

    // A utility function to search a given key in BST
    public Node search(Node root, int key)
    {


    // Driver Program to test above functions
    public static void main(String[] args) {
        BinarySearchTree tree = new BinarySearchTree();

        /* Let us create following BST
              50
          /     \
          30    70
          / \ / \
    20 40 60 80 */
        tree.insert(50);
        tree.insert(30);
        tree.insert(20);
        tree.insert(40);
        tree.insert(70);
        tree.insert(60);
        tree.insert(80);

        // print inorder traversal of the BST
        tree.inorder();
    }
}
```

Binary Search Tree

```java
class BinarySearchTree {

    /* Class containing left and right child of current node and key value*/
    class Node {

    // Root of BST
    Node root;

    // Constructor
    BinarySearchTree() {

    // This method mainly calls insertRec()
    void insert(int key) {

    /* A recursive function to insert a new key in BST */
    Node insertRec(Node root, int key) {

    // This method mainly calls InorderRec()
    void inorder() {

    // A utility function to do inorder traversal of BST
    void inorderRec(Node root) {

    // A utility function to search a given key in BST
    public Node search(Node root, int key)
    {


    // Driver Program to test above functions
    public static void main(String[] args) {
        BinarySearchTree tree = new BinarySearchTree();

        /* Let us create following BST
              50
           /     \
          30      70
         /  \    /  \
    20   40  60   80 */
        tree.insert(50);
        tree.insert(30);
        tree.insert(20);
        tree.insert(40);
        tree.insert(70);
        tree.insert(60);
        tree.insert(80);

        // print inorder traversal of the BST
        tree.inorder();
    }
}
```

```java
/* Class containing left and right child of current node and key value*/
class Node {
    int key;
    Node left, right;

    public Node(int item) {
        key = item;
        left = right = null;
    }
}

// Root of BST
Node root;

// Constructor
BinarySearchTree() {
    root = null;
}

// This method mainly calls insertRec()
void insert(int key) {
root = insertRec(root, key);
}
```

Binary Search Tree

```
class BinarySearchTree {

    /* Class containing left and right child of current node and key value*/
    class Node {

    // Root of BST
    Node root;

    // Constructor
    BinarySearchTree() {

    // This method mainly calls insertRec()
    void insert(int key) {

    /* A recursive function to insert a new key in BST */
    Node insertRec(Node root, int key) {

    // This method mainly calls InorderRec()
    void inorder() {

    // A utility function to do inorder traversal of BST
    void inorderRec(Node root) {

    // A utility function to search a given key in BST
    public Node search(Node root, int key)
    {


    // Driver Program to test above functions
    public static void main(String[] args) {
        BinarySearchTree tree = new BinarySearchTree();

        /* Let us create following BST
              50
           /     \
          30     70
         /  \  /  \
    20 40 60 80 */
        tree.insert(50);
        tree.insert(30);
        tree.insert(20);
        tree.insert(40);
        tree.insert(70);
        tree.insert(60);
        tree.insert(80);

        // print inorder traversal of the BST
        tree.inorder();
    }
}
```

```
/* A recursive function to insert a new key in BST */
Node insertRec(Node root, int key) {

    /* If the tree is empty, return a new node */
    if (root == null) {
        root = new Node(key);
        return root;
    }

    /* Otherwise, recur down the tree */
    if (key < root.key)
        root.left = insertRec(root.left, key);
    else if (key > root.key)
        root.right = insertRec(root.right, key);

    /* return the (unchanged) node pointer */
    return root;
}

// This method mainly calls InorderRec()
void inorder() {
inorderRec(root);
}
```

Binary Search Tree

```java
class BinarySearchTree {

    /* Class containing left and right child of current node and key value*/
    class Node {

    // Root of BST
    Node root;

    // Constructor
    BinarySearchTree() {

    // This method mainly calls insertRec()
    void insert(int key) {

    /* A recursive function to insert a new key in BST */
    Node insertRec(Node root, int key) {

    // This method mainly calls InorderRec()
    void inorder() {

    // A utility function to do inorder traversal of BST
    void inorderRec(Node root) {

    // A utility function to search a given key in BST
    public Node search(Node root, int key)
    {


    // Driver Program to test above functions
    public static void main(String[] args) {
        BinarySearchTree tree = new BinarySearchTree();

        /* Let us create following BST
              50
           /     \
          30      70
         /  \   /  \
    20 40 60 80 */
        tree.insert(50);
        tree.insert(30);
        tree.insert(20);
        tree.insert(40);
        tree.insert(70);
        tree.insert(60);
        tree.insert(80);

        // print inorder traversal of the BST
        tree.inorder();
    }
}
```

```java
// A utility function to do inorder traversal of BST
void inorderRec(Node root) {
    if (root != null) {
        inorderRec(root.left);
        System.out.println(root.key);
        inorderRec(root.right);
    }
}

// A utility function to search a given key in BST
public Node search(Node root, int key)
{
    // Base Cases: root is null or key is present at root
    if (root==null || root.key==key)
        return root;

    // val is greater than root's key
    if (root.key > key)
        return search(root.left, key);

    // val is less than root's key
    return search(root.right, key);
}
```

QuickSort

```java
class QuickSort
{
    /* This function takes last element as pivot,
    places the pivot element at its correct
    position in sorted array, and places all
    smaller (smaller than pivot) to left of
    pivot and all greater elements to right
    of pivot */
    int partition(int arr[], int low, int high)
    {

    /* The main function that implements QuickSort()
    arr[] --> Array to be sorted,
    low --> Starting index,
    high --> Ending index */
    void sort(int arr[], int low, int high)
    {

    /* A utility function to print array of size n */
    static void printArray(int arr[])
    {

    // Driver program
    public static void main(String args[])
    {
        int arr[] = {10, 7, 8, 9, 1, 5};
        int n = arr.length;

        QuickSort ob = new QuickSort();
        ob.sort(arr, 0, n-1);

        System.out.println("sorted array");
        printArray(arr);
    }
}
```

QuickSort

```java
class QuickSort
{
    /* This function takes last element as pivot,
    places the pivot element at its correct
    position in sorted array, and places all
    smaller (smaller than pivot) to left of
    pivot and all greater elements to right
    of pivot */
    int partition(int arr[], int low, int high)
    {

    /* The main function that implements QuickSort()
    arr[] --> Array to be sorted,
    low --> Starting index,
    high --> Ending index */
    void sort(int arr[], int low, int high)
    {

    /* A utility function to print array of size n */
    static void printArray(int arr[])
    {

    // Driver program
    public static void main(String args[])
    {
        int arr[] = {10, 7, 8, 9, 1, 5};
        int n = arr.length;

        QuickSort ob = new QuickSort();
        ob.sort(arr, 0, n-1);

        System.out.println("sorted array");
        printArray(arr);
    }
}
```

```java
/* The main function that implements QuickSort()
arr[] --> Array to be sorted,
low --> Starting index,
high --> Ending index */
void sort(int arr[], int low, int high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[pi] is
        now at right place */
        int pi = partition(arr, low, high);

        // Recursively sort elements before
        // partition and after partition
        sort(arr, low, pi-1);
        sort(arr, pi+1, high);
    }
}
```

QuickSort

```java
class QuickSort
{
    /* This function takes last element as pivot,
    places the pivot element at its correct
    position in sorted array, and places all
    smaller (smaller than pivot) to left of
    pivot and all greater elements to right
    of pivot */
    int partition(int arr[], int low, int high)
    {

    /* The main function that implements QuickSort()
    arr[] --> Array to be sorted,
    low --> Starting index,
    high --> Ending index */
    void sort(int arr[], int low, int high)
    {

    /* A utility function to print array of size n */
    static void printArray(int arr[])
    {

    // Driver program
    public static void main(String args[])
    {
        int arr[] = {10, 7, 8, 9, 1, 5};
        int n = arr.length;

        QuickSort ob = new QuickSort();
        ob.sort(arr, 0, n-1);

        System.out.println("sorted array");
        printArray(arr);
    }
}
```

```java
int partition(int arr[], int low, int high)
{
    int pivot = arr[high];
    int i = (low-1); // index of smaller element
    for (int j=low; j<high; j++)
    {
        // If current element is smaller than or
        // equal to pivot
        if (arr[j] <= pivot)
        {
            i++;

            // swap arr[i] and arr[j]
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }

    // swap arr[i+1] and arr[high] (or pivot)
    int temp = arr[i+1];
    arr[i+1] = arr[high];
    arr[high] = temp;

    return i+1;
}
```

QuickSort

```java
class QuickSort
{
    /* This function takes last element as pivot,
    places the pivot element at its correct
    position in sorted array, and places all
    smaller (smaller than pivot) to left of
    pivot and all greater elements to right
    of pivot */
    int partition(int arr[], int low, int high)
    {

    /* The main function that implements QuickSort()
    arr[] --> Array to be sorted,
    low --> Starting index,
    high --> Ending index */
    void sort(int arr[], int low, int high)
    {

    /* A utility function to print array of size n */
    static void printArray(int arr[])
    {

    // Driver program
    public static void main(String args[])
    {
        int arr[] = {10, 7, 8, 9, 1, 5};
        int n = arr.length;

        QuickSort ob = new QuickSort();
        ob.sort(arr, 0, n-1);

        System.out.println("sorted array");
        printArray(arr);
    }
}
```

```java
/* A utility function to print array of size n */
static void printArray(int arr[])
{
    int n = arr.length;
    for (int i=0; i<n; ++i)
        System.out.print(arr[i]+" ");
    System.out.println();
}
```

HeapSort

```java
// Java program for implementation of Heap Sort
public class HeapSort
{
    public void sort(int arr[])
    {

    // To heapify a subtree rooted with node i which is
    // an index in arr[]. n is size of heap
    void heapify(int arr[], int n, int i)
    {

    /* A utility function to print array of size n */
    static void printArray(int arr[])
    {

    // Driver program
    public static void main(String args[])
    {
        int arr[] = {12, 11, 13, 5, 6, 7};
        int n = arr.length;

        HeapSort ob = new HeapSort();
        ob.sort(arr);

        System.out.println("Sorted array is");
        printArray(arr);
    }
}
```

```java
public void sort(int arr[])
{
    int n = arr.length;

    // Build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // One by one extract an element from heap
    for (int i=n-1; i>=0; i--)
    {
        // Move current root to end
        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;

        // call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}
```

HeapSort

```java
// Java program for implementation of Heap Sort
public class HeapSort
{
    public void sort(int arr[])
    {

    // To heapify a subtree rooted with node i which is
    // an index in arr[]. n is size of heap
    void heapify(int arr[], int n, int i)
    {

    /* A utility function to print array of size n */
    static void printArray(int arr[])
    {

    // Driver program
    public static void main(String args[])
    {
        int arr[] = {12, 11, 13, 5, 6, 7};
        int n = arr.length;

        HeapSort ob = new HeapSort();
        ob.sort(arr);

        System.out.println("Sorted array is");
        printArray(arr);
    }
}
```

```java
// To heapify a subtree rooted with node i which is
// an index in arr[]. n is size of heap
void heapify(int arr[], int n, int i)
{
    int largest = i; // Initialize largest as root
    int l = 2*i + 1; // left = 2*i + 1
    int r = 2*i + 2; // right = 2*i + 2

    // If left child is larger than root
    if (l < n && arr[l] > arr[largest])
        largest = l;

    // If right child is larger than largest so far
    if (r < n && arr[r] > arr[largest])
        largest = r;

    // If largest is not root
    if (largest != i)
    {
        int swap = arr[i];
        arr[i] = arr[largest];
        arr[largest] = swap;

        // Recursively heapify the affected sub-tree
        heapify(arr, n, largest);
    }
}
```

HeapSort

```java
// Java program for implementation of Heap Sort
public class HeapSort
{
    public void sort(int arr[])
    {

    // To heapify a subtree rooted with node i which is
    // an index in arr[]. n is size of heap
    void heapify(int arr[], int n, int i)
    {

    /* A utility function to print array of size n */
    static void printArray(int arr[])
    {

    // Driver program
    public static void main(String args[])
    {
        int arr[] = {12, 11, 13, 5, 6, 7};
        int n = arr.length;

        HeapSort ob = new HeapSort();
        ob.sort(arr);

        System.out.println("Sorted array is");
        printArray(arr);
    }
}
```

```java
/* A utility function to print array of size n */
static void printArray(int arr[])
{
    int n = arr.length;
    for (int i=0; i<n; ++i)
        System.out.print(arr[i]+" ");
    System.out.println();
}
```

Symbol Table

```java
import java.util.Iterator;
import java.util.NoSuchElementException;
import java.util.TreeMap;

public class ST<Key extends Comparable<Key>, Value> implements Iterable<Key> {

    private TreeMap<Key, Value> st;

    /**
    public ST() {


    /**
    public Value get(Key key) {

    /**
    public void put(Key key, Value val) {

    /**
    public void delete(Key key) {

    /**
    public boolean contains(Key key) {

    /**
    public int size() {

    /**
    public boolean isEmpty() {

    /**
    public Iterable<Key> keys() {

    /**
    @Deprecated
    public Iterator<Key> iterator() {

    /**
    public Key min() {

    /**
    public Key max() {

    /**
    public Key ceiling(Key key) {

    /**
    public Key floor(Key key) {

    /**
    public static void main(String[] args) {
        ST<String, Integer> st = new ST<String, Integer>();
        for (int i = 0; !StdIn.isEmpty(); i++) {
            String key = StdIn.readString();
            st.put(key, i);
        }
        for (String s : st.keys())
            StdOut.println(s + " " + st.get(s));
    }
}
```

Symbol Table

```java
import java.util.Iterator;
import java.util.NoSuchElementException;
import java.util.TreeMap;

public class ST<Key extends Comparable<Key>, Value> implements Iterable<Key> {

    private TreeMap<Key, Value> st;

    /**
    public ST() {

    /**
    public Value get(Key key) {

    /**
    public void put(Key key, Value val) {

    /**
    public void delete(Key key) {

    /**
    public boolean contains(Key key) {

    /**
    public int size() {

    /**
    public boolean isEmpty() {

    /**
    public Iterable<Key> keys() {

    /**
    @Deprecated
    public Iterator<Key> iterator() {

    /**
    public Key min() {

    /**
    public Key max() {

    /**
    public Key ceiling(Key key) {

    /**
    public Key floor(Key key) {

    /**
    public static void main(String[] args) {
        ST<String, Integer> st = new ST<String, Integer>();
        for (int i = 0; !StdIn.isEmpty(); i++) {
            String key = StdIn.readString();
            st.put(key, i);
        }
        for (String s : st.keys())
            StdOut.println(s + " " + st.get(s));
    }
}
```

```java
/**
 * Initializes an empty symbol table.
 */
public ST() {
    st = new TreeMap<Key, Value>();
}


/**
 * Returns the value associated with the given key in this symbol table.
 *
 * @param  key the key
 * @return the value associated with the given key if the key is in this symbol table;
 *         {@code null} if the key is not in this symbol table
 * @throws IllegalArgumentException if {@code key} is {@code null}
 */
public Value get(Key key) {
    if (key == null) throw new IllegalArgumentException("calls get() with null key");
    return st.get(key);
}

/**
 * Inserts the specified key-value pair into the symbol table, overwriting the old
 * value with the new value if the symbol table already contains the specified key.
 * Deletes the specified key (and its associated value) from this symbol table
 * if the specified value is {@code null}.
 *
 * @param  key the key
 * @param  val the value
 * @throws IllegalArgumentException if {@code key} is {@code null}
 */
public void put(Key key, Value val) {
    if (key == null) throw new IllegalArgumentException("calls put() with null key");
    if (val == null) st.remove(key);
    else             st.put(key, val);
}
```

Symbol Table

```java
import java.util.Iterator;
import java.util.NoSuchElementException;
import java.util.TreeMap;

public class ST<Key extends Comparable<Key>, Value> implements Iterable<Key> {

    private TreeMap<Key, Value> st;

    /**
    public ST() {

    /**
    public Value get(Key key) {

    /**
    public void put(Key key, Value val) {

    /**
    public void delete(Key key) {

    /**
    public boolean contains(Key key) {

    /**
    public int size() {

    /**
    public boolean isEmpty() {

    /**
    public Iterable<Key> keys() {

    /**
    @Deprecated
    public Iterator<Key> iterator() {

    /**
    public Key min() {

    /**
    public Key max() {

    /**
    public Key ceiling(Key key) {

    /**
    public Key floor(Key key) {

    /**
    public static void main(String[] args) {
        ST<String, Integer> st = new ST<String, Integer>();
        for (int i = 0; !StdIn.isEmpty(); i++) {
            String key = StdIn.readString();
            st.put(key, i);
        }
        for (String s : st.keys())
            StdOut.println(s + " " + st.get(s));
    }
}
```

```java
/**
 * Removes the specified key and its associated value from this symbol table
 * (if the key is in this symbol table).
 *
 * @param  key the key
 * @throws IllegalArgumentException if {@code key} is {@code null}
 */
public void delete(Key key) {
    if (key == null) throw new IllegalArgumentException("calls delete() with null key");
    st.remove(key);
}

/**
 * Returns true if this symbol table contain the given key.
 *
 * @param  key the key
 * @return {@code true} if this symbol table contains {@code key} and
 *         {@code false} otherwise
 * @throws IllegalArgumentException if {@code key} is {@code null}
 */
public boolean contains(Key key) {
    if (key == null) throw new IllegalArgumentException("calls contains() with null key");
    return st.containsKey(key);
}

/**
 * Returns the number of key-value pairs in this symbol table.
 *
 * @return the number of key-value pairs in this symbol table
 */
public int size() {
    return st.size();
}
```

Symbol Table

```java
import java.util.Iterator;
import java.util.NoSuchElementException;
import java.util.TreeMap;

public class ST<Key extends Comparable<Key>, Value> implements Iterable<Key> {

    private TreeMap<Key, Value> st;

    /**
    public ST() {

    /**
    public Value get(Key key) {

    /**
    public void put(Key key, Value val) {

    /**
    public void delete(Key key) {

    /**
    public boolean contains(Key key) {

    /**
    public int size() {

    /**
    public boolean isEmpty() {

    /**
    public Iterable<Key> keys() {

    /**
    @Deprecated
    public Iterator<Key> iterator() {

    /**
    public Key min() {

    /**
    public Key max() {

    /**
    public Key ceiling(Key key) {

    /**
    public Key floor(Key key) {

    /**
    public static void main(String[] args) {
        ST<String, Integer> st = new ST<String, Integer>();
        for (int i = 0; !StdIn.isEmpty(); i++) {
            String key = StdIn.readString();
            st.put(key, i);
        }
        for (String s : st.keys())
            StdOut.println(s + " " + st.get(s));
    }
}
```

```java
/**
 * Returns true if this symbol table is empty.
 *
 * @return {@code true} if this symbol table is empty and {@code false} otherwise
 */
public boolean isEmpty() {
    return size() == 0;
}

/**
 * Returns all keys in this symbol table.
 * <p>
 * To iterate over all of the keys in the symbol table named {@code st},
 * use the foreach notation: {@code for (Key key : st.keys())}.
 *
 * @return all keys in this symbol table
 */
public Iterable<Key> keys() {
    return st.keySet();
}

/**
 * Returns all of the keys in this symbol table.
 * To iterate over all of the keys in a symbol table named {@code st}, use the
 * foreach notation: {@code for (Key key : st)}.
 * <p>
 * This method is provided for backward compatibility with the version from
 * <em>Introduction to Programming in Java: An Interdisciplinary Approach.</em>
 *
 * @return     an iterator to all of the keys in this symbol table
 * @deprecated Replaced by {@link #keys()}.
 */
@Deprecated
public Iterator<Key> iterator() {
    return st.keySet().iterator();
}
```

Symbol Table

```java
import java.util.Iterator;
import java.util.NoSuchElementException;
import java.util.TreeMap;

public class ST<Key extends Comparable<Key>, Value> implements Iterable<Key> {

    private TreeMap<Key, Value> st;

    /**
    public ST() {

    /**
    public Value get(Key key) {

    /**
    public void put(Key key, Value val) {

    /**
    public void delete(Key key) {

    /**
    public boolean contains(Key key) {

    /**
    public int size() {

    /**
    public boolean isEmpty() {

    /**
    public Iterable<Key> keys() {

    /**
    @Deprecated
    public Iterator<Key> iterator() {

    /**
    public Key min() {

    /**
    public Key max() {

    /**
    public Key ceiling(Key key) {

    /**
    public Key floor(Key key) {

    /**
    public static void main(String[] args) {
        ST<String, Integer> st = new ST<String, Integer>();
        for (int i = 0; !StdIn.isEmpty(); i++) {
            String key = StdIn.readString();
            st.put(key, i);
        }
        for (String s : st.keys())
            StdOut.println(s + " " + st.get(s));
    }
}
```

```java
/**
 * Returns the smallest key in this symbol table.
 *
 * @return the smallest key in this symbol table
 * @throws NoSuchElementException if this symbol table is empty
 */
public Key min() {
    if (isEmpty()) throw new NoSuchElementException("calls min() with empty symbol table");
    return st.firstKey();
}

/**
 * Returns the largest key in this symbol table.
 *
 * @return the largest key in this symbol table
 * @throws NoSuchElementException if this symbol table is empty
 */
public Key max() {
    if (isEmpty()) throw new NoSuchElementException("calls max() with empty symbol table");
    return st.lastKey();
}
```

Symbol Table

```java
import java.util.Iterator;
import java.util.NoSuchElementException;
import java.util.TreeMap;

public class ST<Key extends Comparable<Key>, Value> implements Iterable<Key> {

    private TreeMap<Key, Value> st;

    /**
    public ST() {

    /**
    public Value get(Key key) {

    /**
    public void put(Key key, Value val) {

    /**
    public void delete(Key key) {

    /**
    public boolean contains(Key key) {

    /**
    public int size() {

    /**
    public boolean isEmpty() {

    /**
    public Iterable<Key> keys() {

    /**
    @Deprecated
    public Iterator<Key> iterator() {

    /**
    public Key min() {

    /**
    public Key max() {

    /**
    public Key ceiling(Key key) {

    /**
    public Key floor(Key key) {

    /**
    public static void main(String[] args) {
        ST<String, Integer> st = new ST<String, Integer>();
        for (int i = 0; !StdIn.isEmpty(); i++) {
            String key = StdIn.readString();
            st.put(key, i);
        }
        for (String s : st.keys())
            StdOut.println(s + " " + st.get(s));
    }
}
```

```java
/**
 * Returns the smallest key in this symbol table greater than or equal to {@code key}.
 *
 * @param  key the key
 * @return the smallest key in this symbol table greater than or equal to {@code key}
 * @throws NoSuchElementException if there is no such key
 * @throws IllegalArgumentException if {@code key} is {@code null}
 */
public Key ceiling(Key key) {
    if (key == null) throw new IllegalArgumentException("argument to ceiling() is null");
    Key k = st.ceilingKey(key);
    if (k == null) throw new NoSuchElementException("all keys are less than " + key);
    return k;
}

/**
 * Returns the largest key in this symbol table less than or equal to {@code key}.
 *
 * @param  key the key
 * @return the largest key in this symbol table less than or equal to {@code key}
 * @throws NoSuchElementException if there is no such key
 * @throws IllegalArgumentException if {@code key} is {@code null}
 */
public Key floor(Key key) {
    if (key == null) throw new IllegalArgumentException("argument to floor() is null");
    Key k = st.floorKey(key);
    if (k == null) throw new NoSuchElementException("all keys are greater than " + key);
    return k;
}
```

Hacettepe University

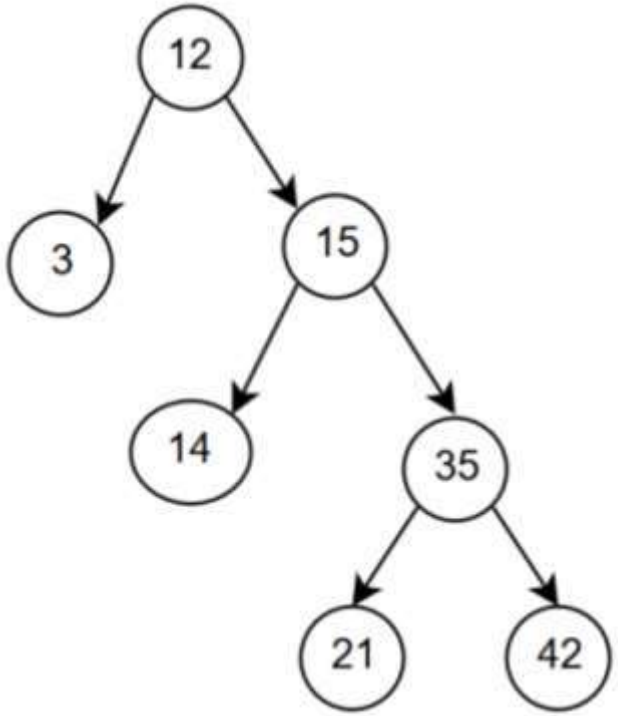
Computer Engineering Department

BBM204 Software Laboratory II

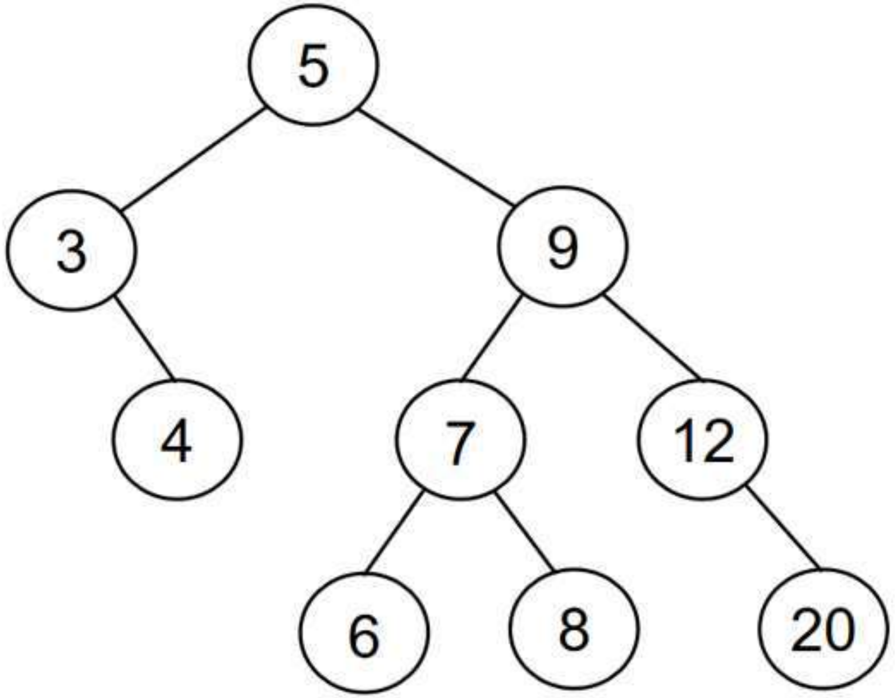
Recitation V

Spring 2019

Q1)Draw the binary search tree that is created if the following numbers are inserted in the tree in the given order. 12 15 3 35 21 42 14



Q2) The binary search tree shown below was constructed by inserting a sequence of items into an empty tree. Write an input sequences will produce this binary search tree.



A. 5 3 4 9 12 7 8 6 20

B. 5 9 3 7 6 8 4 12 20

C. 5 9 7 8 6 12 20 3 4

D. 5 9 7 3 8 12 6 4 20

E. 5 9 3 6 7 8 4 12 20

This sequence will not produce this binary search tree.

Q3. Given a sequence of numbers:
7, 15, 13, 12, 9, 10

a) Draw a binary max-heap (in a tree form) by inserting the above numbers reading them from left to right

b) Show a tree that can be the result after the call to deleteMax() on the above heap

c) Show a tree after another call to deleteMax()

b) Give nonrecursive implementations of get() for BST.

public Value get(Key key){ }

```
public Value get(Key key){
    Node x = root;
    while (x != null){
        int cmp = key.compareTo(x.key);
        if (cmp == 0)
            return x.val;
        else if (cmp < 0)
            x = x.left;
        else if (cmp > 0)
            x = x.right;
    }
    return null;
}
```

Hacettepe University

Computer Engineering Department

BBM204 Software Laboratory II

Recitation VI

Spring 2019

2-3 Trees

2-3 tree is a tree data structure in which every internal node (non-leaf node) has either one data element and two children or two data elements and three children. If a node contains one data element **leftVal**, it has two subtrees (children) namely **left** and **middle**. Whereas if a node contains two data elements **leftVal** and **rightVal**, it has three subtrees namely **left**, **middle** and **right**.

Search: To search a key **K** in given 2-3 tree **T**, we follow the following procedure:

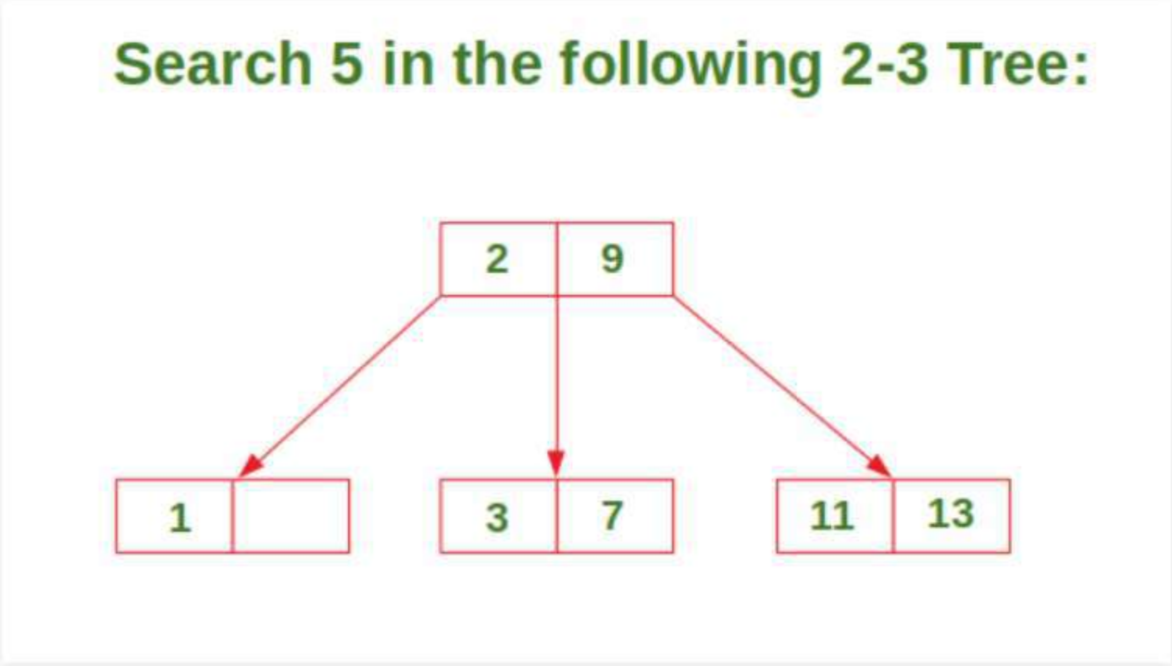
Base cases:

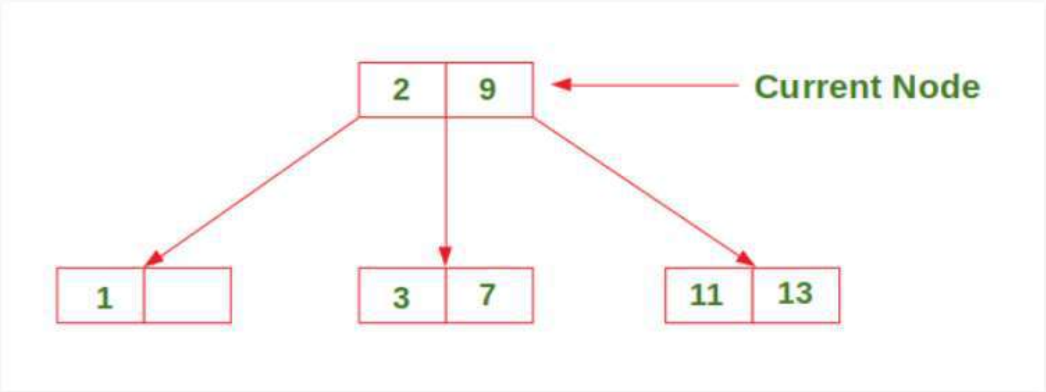
1. If **T** is empty, return False (key cannot be found in the tree).
2. If current node contains data value which is equal to **K**, return True.
3. If we reach the leaf-node and it doesn't contain the required key value **K**, return False.

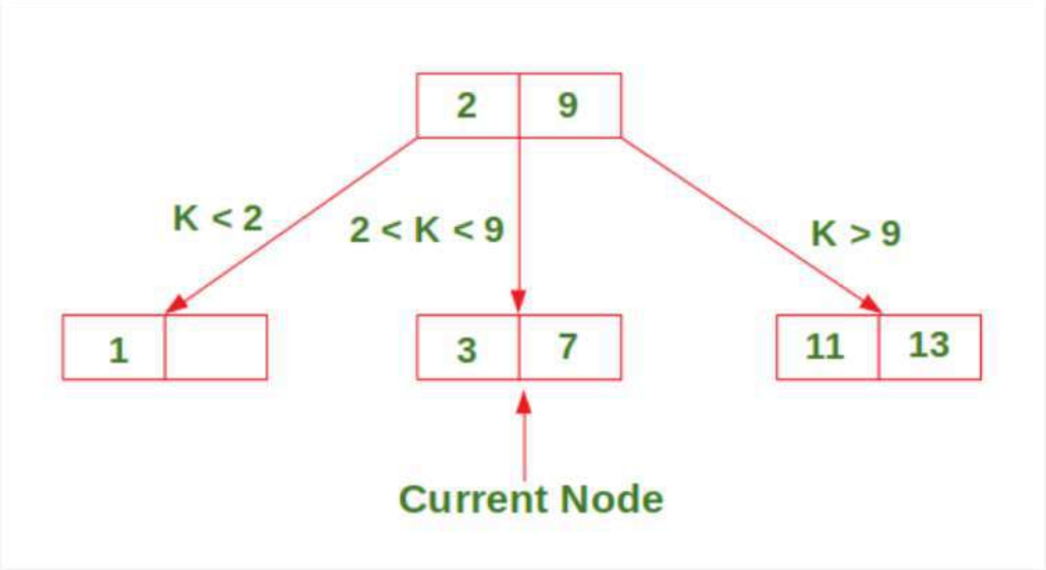
Recursive Calls:

1. If **K** < currentNode.leftVal, we explore the left subtree of the current node.
2. Else if currentNode.leftVal < **K** currentNode.rightVal, we explore the right subtree of the current node.
3. Else if **K** > currentNode.rightVal, we explore the right subtree of the current node.

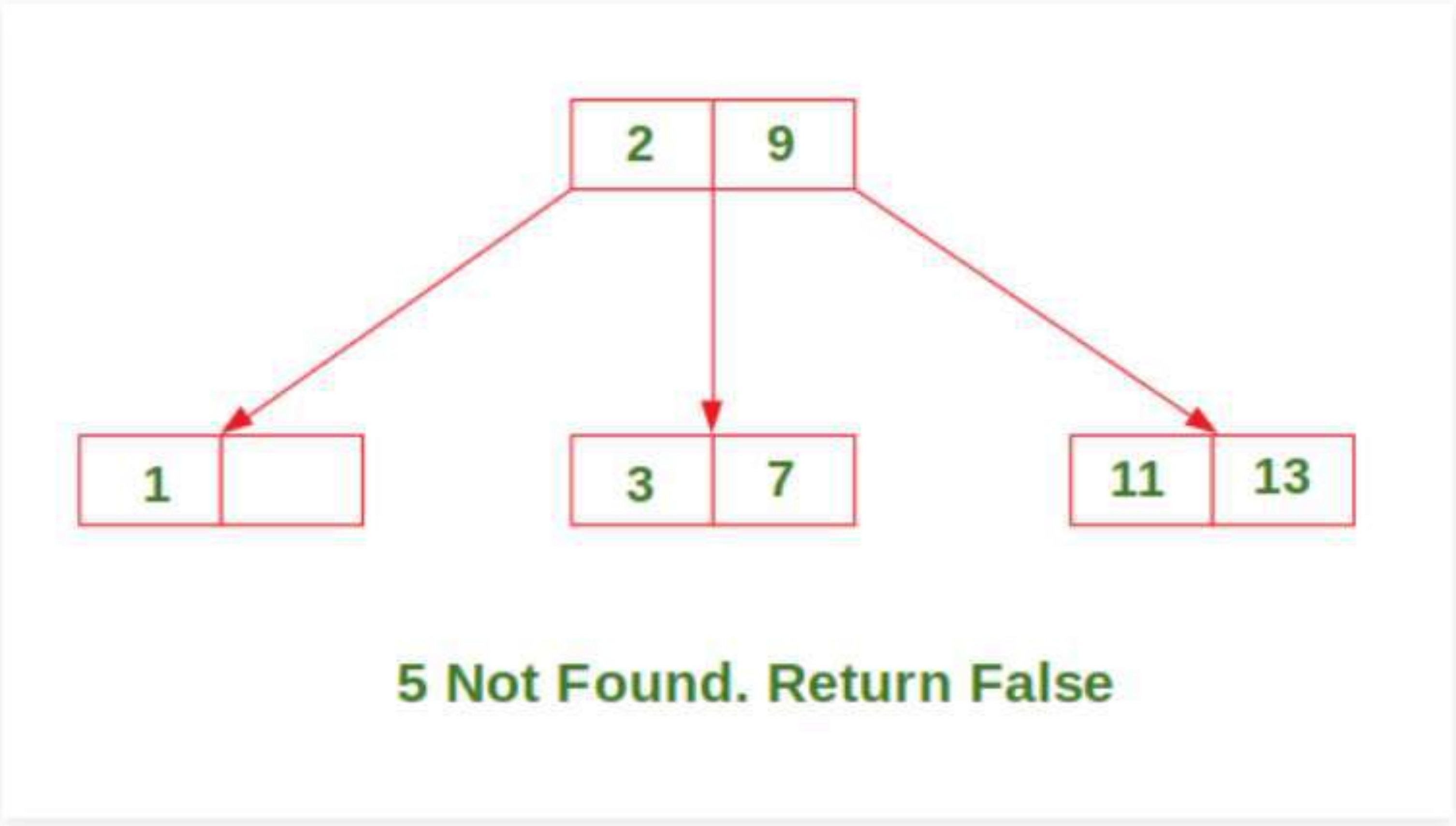
2-3 Trees (Search)





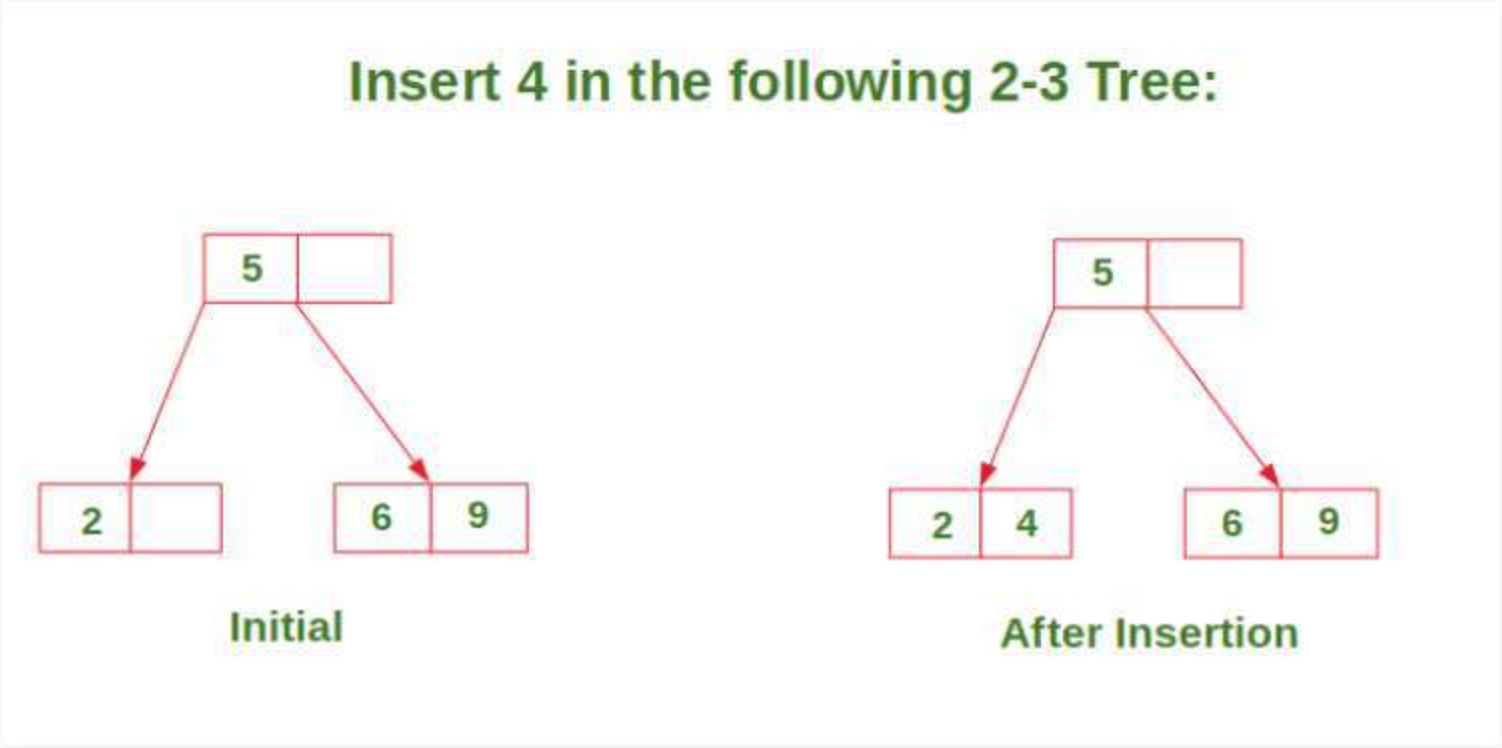


2-3 Trees (Search)



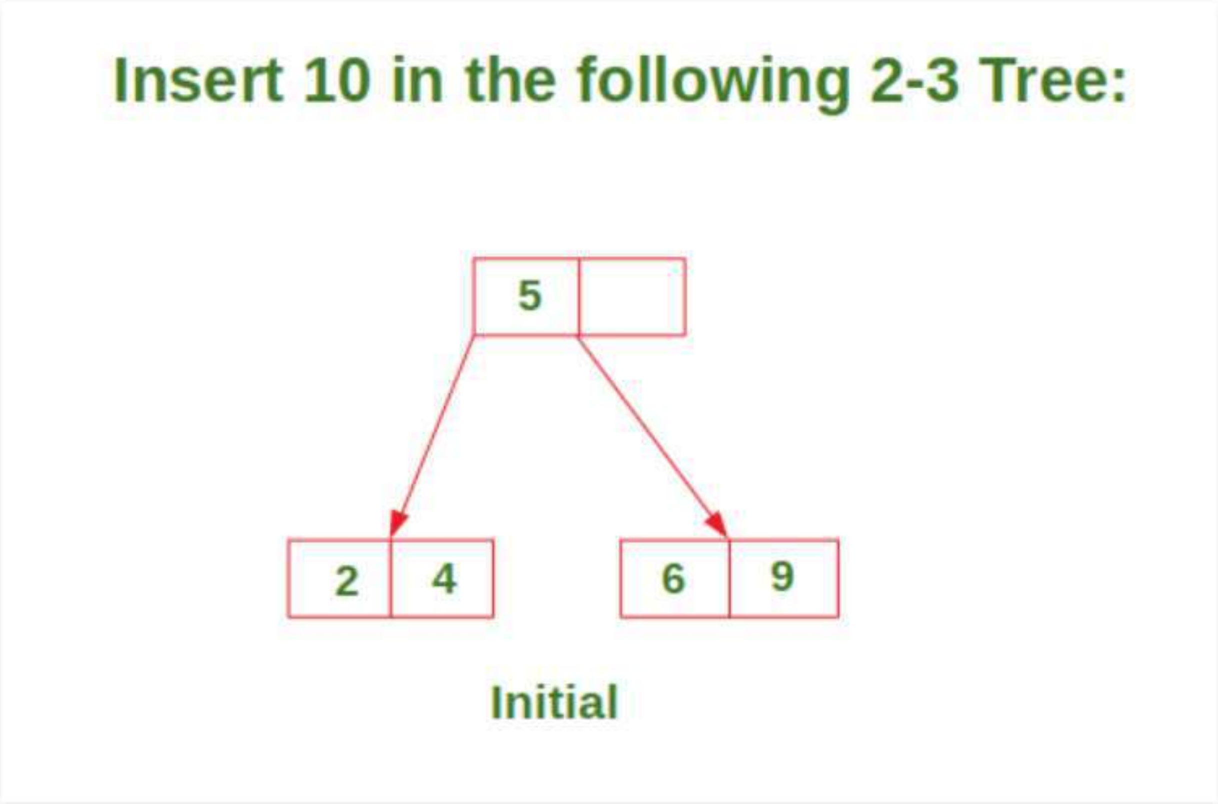
2-3 Trees (Insertion)

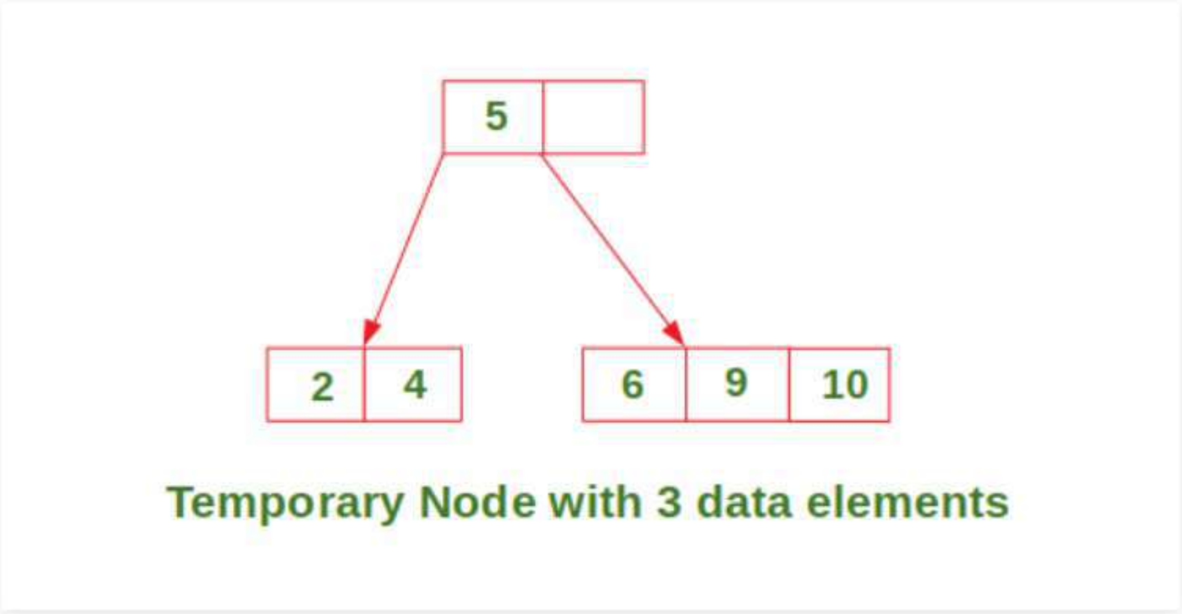
Case 1: Insert in a node with only one data element



2-3 Trees (Insertion)

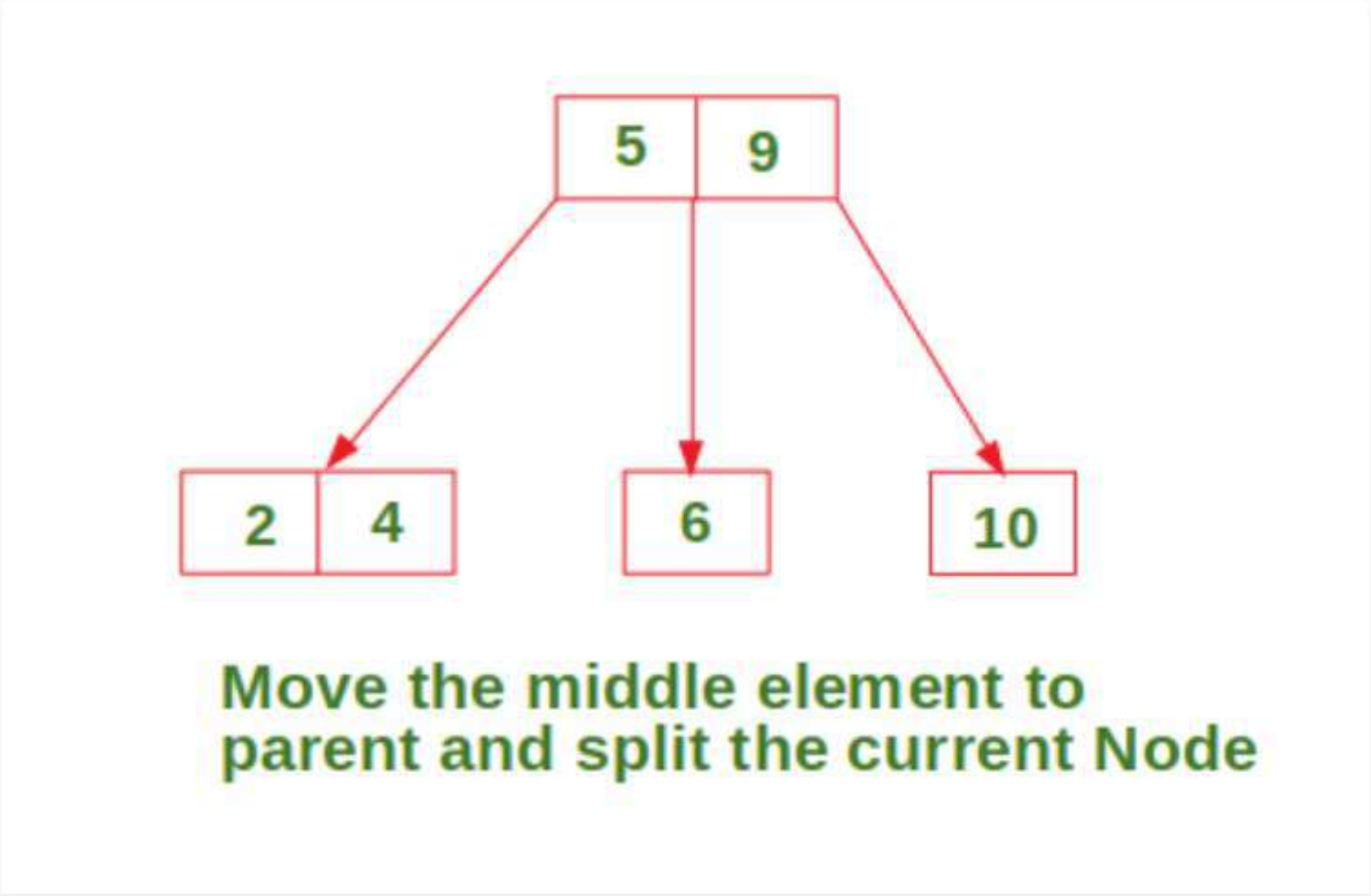
Case 2: Insert in a node with two data elements whose parent contains only one data element.





2-3 Trees (Insertion)

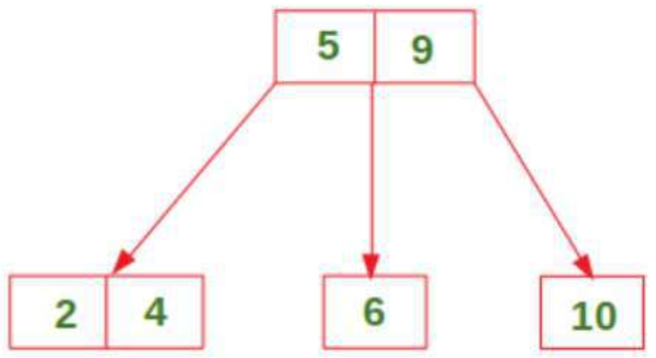
Case 2: Insert in a node with two data elements whose parent contains only one data element.



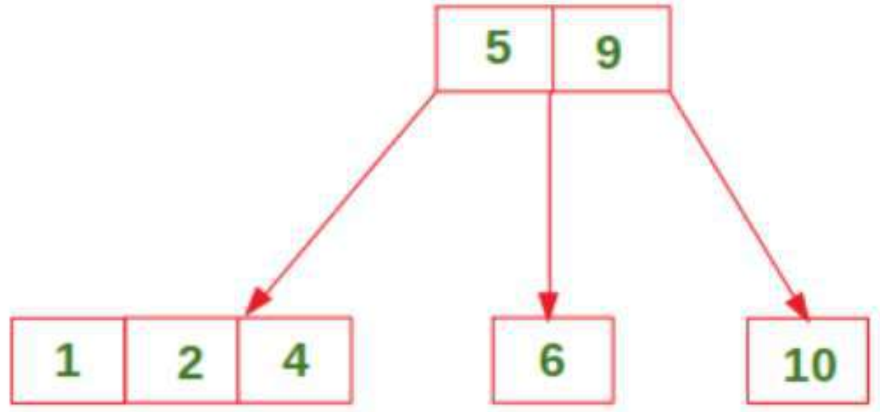
2-3 Trees (Insertion)

Case 3: Insert in a node with two data elements whose parent also contains two data elements.

Insert 1 in the following 2-3 Tree:



Initial

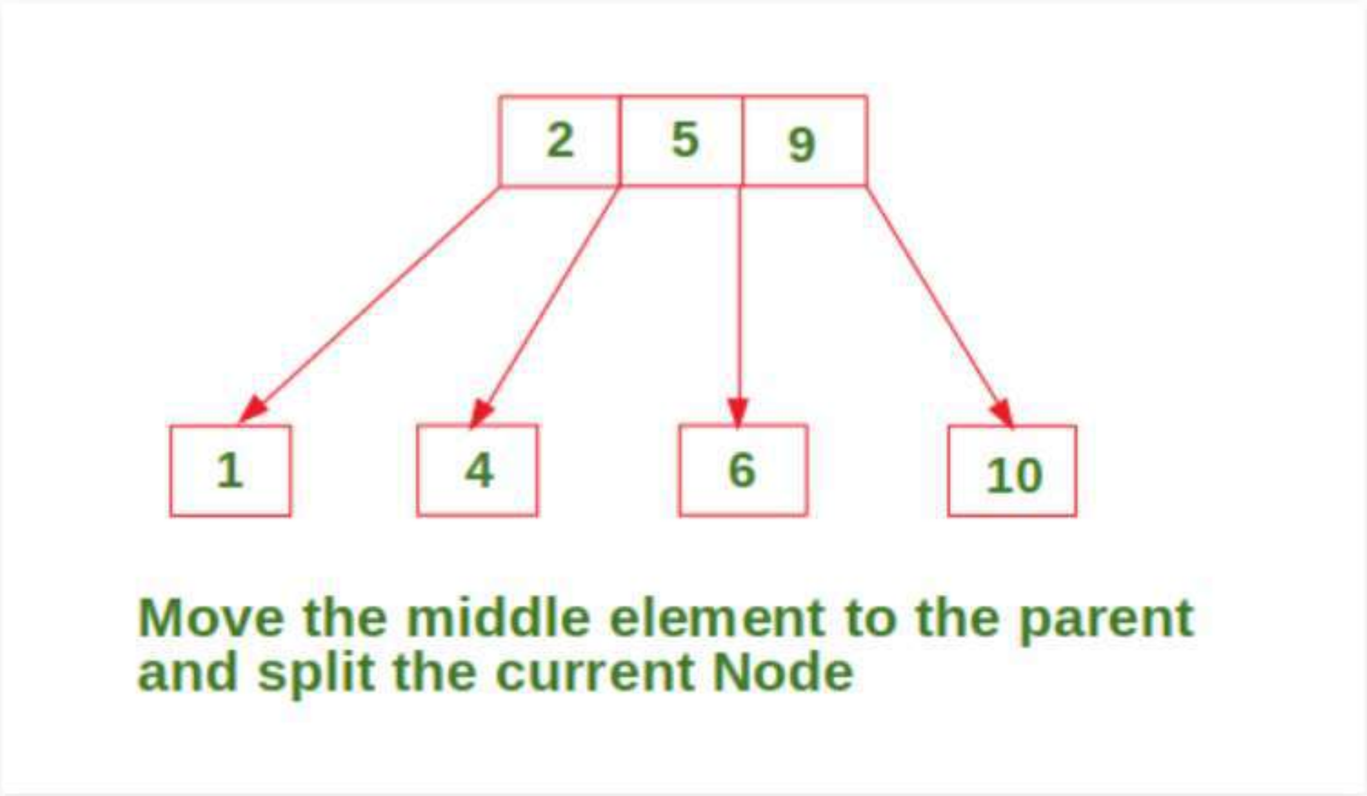


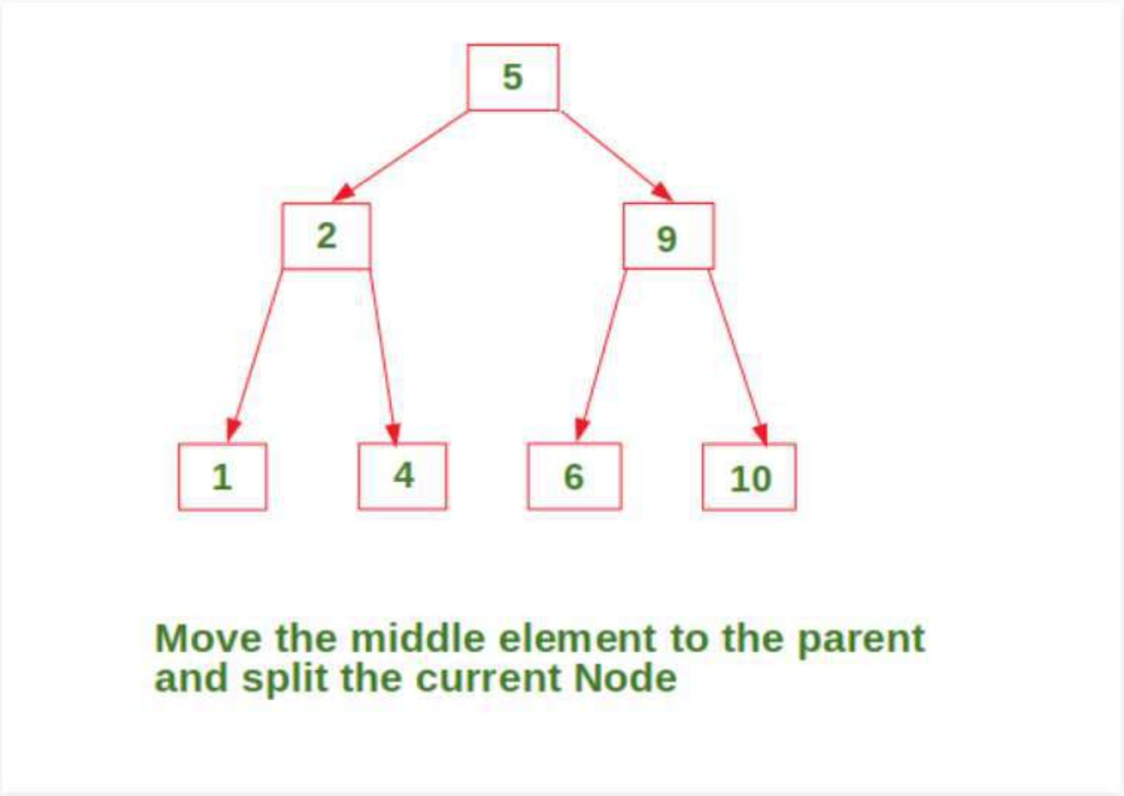
Temporary Node with 3 data elements

2-3 Trees (Insertion)

Case 3: Insert in a node with two data elements

whose parent also contains two data elements.

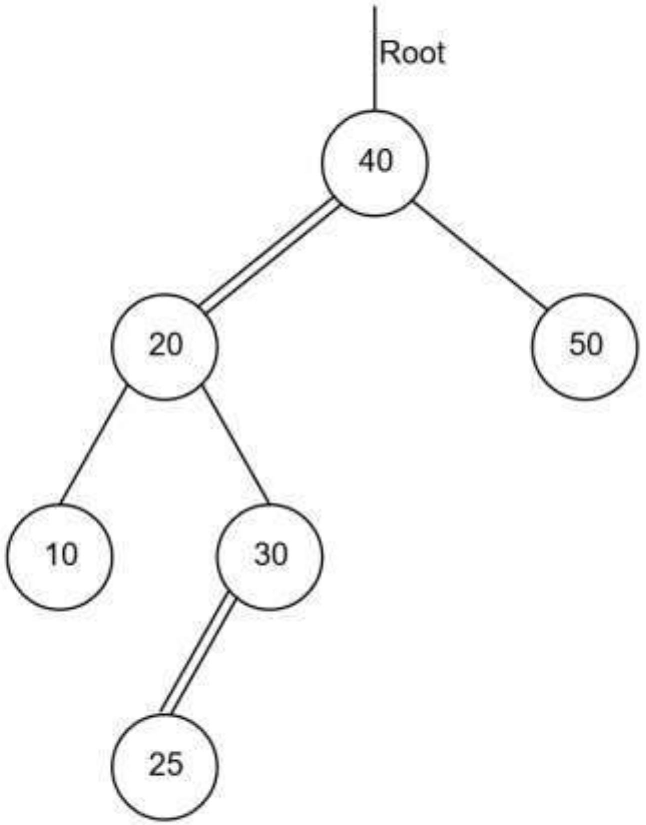




Characteristics of LLRB

1. Root node is Always BLACK in color.
2. Every new Node inserted is always RED in color.
3. Every NULL child of a node is considered as BLACK in color.
4. There should not be a node which has RIGHT RED child and LEFT BLACK child(or NULL child as all NULLS are BLACK), if present Left rotate the node, and swap the colors of current node and its **LEFT** child so as to maintain consistency for rule 2 i.e., new node must be RED in color.
5. There should not be a node which has LEFT RED child and LEFT RED grandchild, if present Right Rotate the node and swap the colors between node and it's **RIGHT** child to follow rule 2.
6. There should not be a node which has LEFT RED child and RIGHT RED child, if present Invert the colors of all nodes i. e., current_node, LEFT child, and RIGHT child.

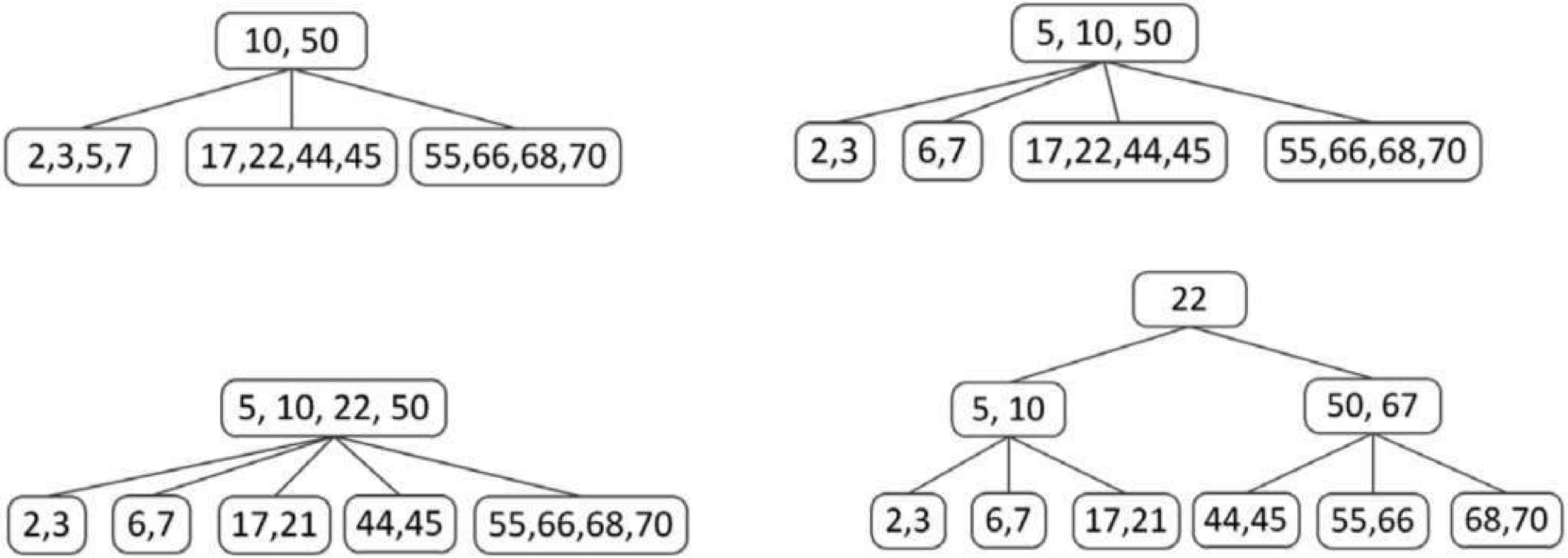
Construct a red-black tree by inserting the keys in the following sequence into an initially empty red-black tree: **10, 20, 30, 40, 50, 25.** Show each step. (A double edge indicates a red pointer and single edge indicates a black pointer.)



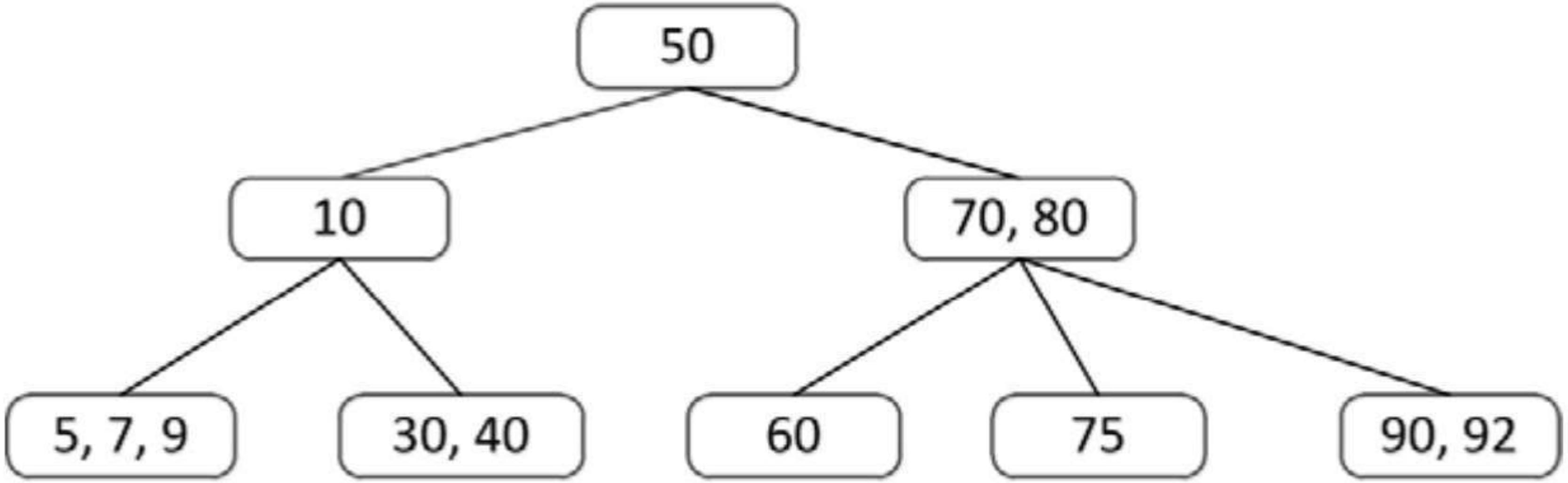
B-Trees

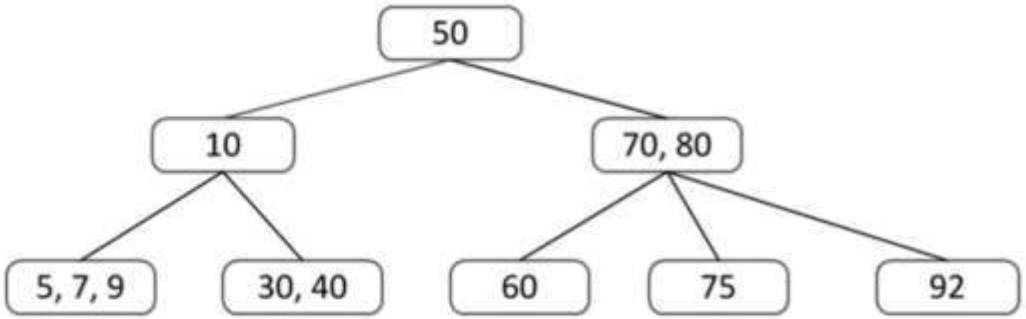
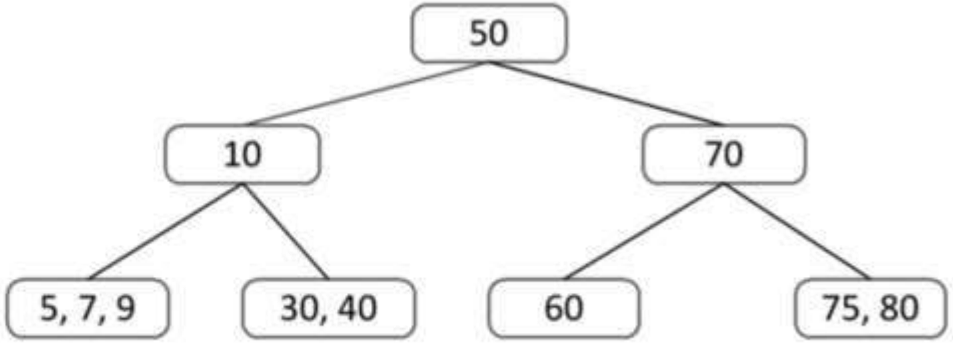
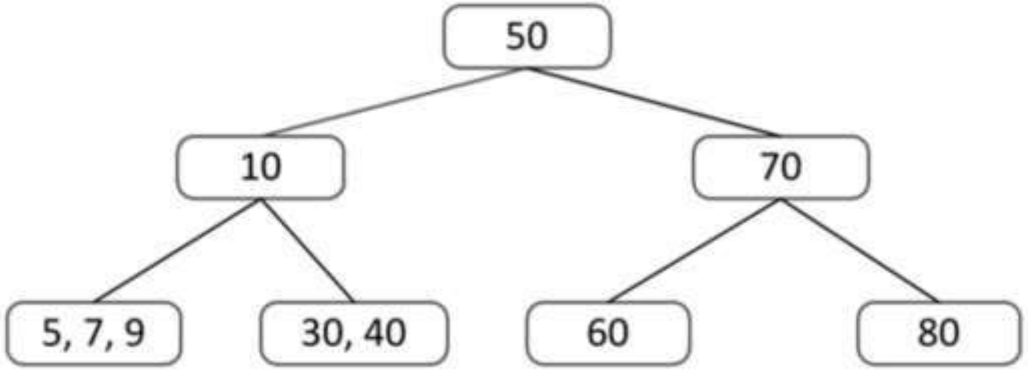
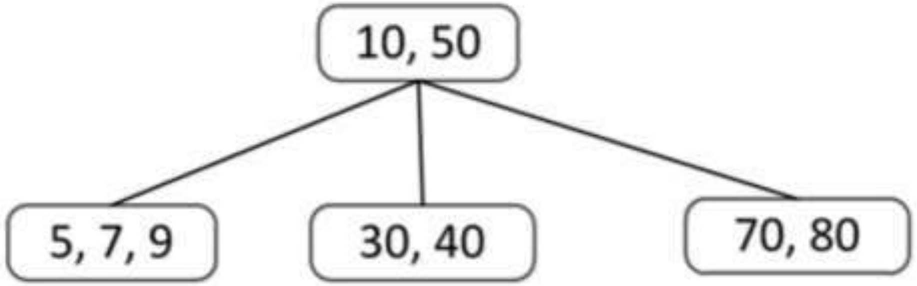
- **Properties of B-Tree**
  1) All leaves are at same level.
  2) A B-Tree is defined by the term *minimum degree* ‘t’. The value of t depends upon disk block size.
  3) Every node except root must contain at least t-1 keys. Root may contain minimum 1 key.
  4) All nodes (including root) may contain at most 2t – 1 keys.
  5) Number of children of a node is equal to the number of keys in it plus 1.
  6) All keys of a node are sorted in increasing order. The child between two keys k1 and k2 contains all keys in the range from k1 and k2.

Given the following B-tree of order m=5, show each corresponding B-tree after insertion of 17,6,21,67, in this order. Use commas (,) to separate the data in a node.


10, 50
2,3,5,7
17,22,44,45
55,66,68,70
5, 10, 50
2,3
6,7
17,22,44,45
55,66,68,70
5, 10, 22, 50
2,3
6,7
17,21
44,45
55,66,68,70
22
5, 10
50, 67
2,3
6,7
17,21
44,45
55,66
68,70

Given the following B-tree of order m=4, show each corresponding B-tree after deletion of 90,92,75,60, in this order. Use commas(,) to separate the data in a node.




50
10
70, 80
5, 7, 9
30, 40
60
75
92

50
10
70
5, 7, 9
30, 40
60
75, 80

50
10
70
5, 7, 9
30, 40
60
80

10, 50
5, 7, 9
30, 40
70, 80

Hacettepe University

Computer Engineering Department

BBM204 Software Laboratory II

Recitation VII

Spring 2019

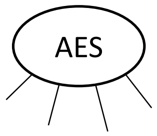
Q1. Insert each element in an array given below to a 2-3 search tree (*show every rotation and color flip operation*)

S, E, A, R, C, H, X, M, P, L

Order (Ascending, Left to Right): A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z

Q1. Insert each element in an array given below to a 2-3 search tree (*show every rotation and color flip operation*)

S, E, A, R, C, H, X, M, P, L



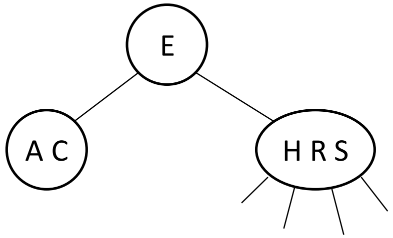
split 4-node
(move E to parent)

insert S
insert E
insert A

Q1. Insert each element in an array given below to a 2-3 search tree (*show every rotation and color flip operation*)

S, E, A, R, C, H, X, M, P, L

insert R
insert C
insert H

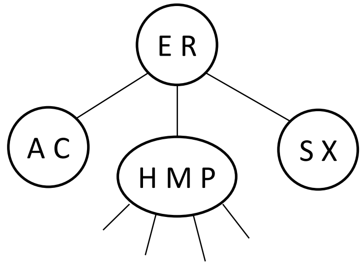


split 4-node
(move R to parent)

Q1. Insert each element in an array given below to a 2-3 search tree (*show every rotation and color flip operation*)

S, E, A, R, C, H, X, M, P, L

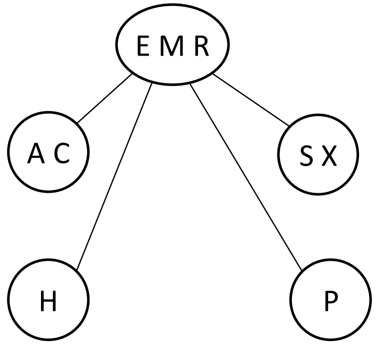
insert X
insert M
insert P



split 4-node
(move M to parent)

Q1. Insert each element in an array given below to a 2-3 search tree (*show every rotation and color flip operation*)

S, E, A, R, C, H, X, M, P, L

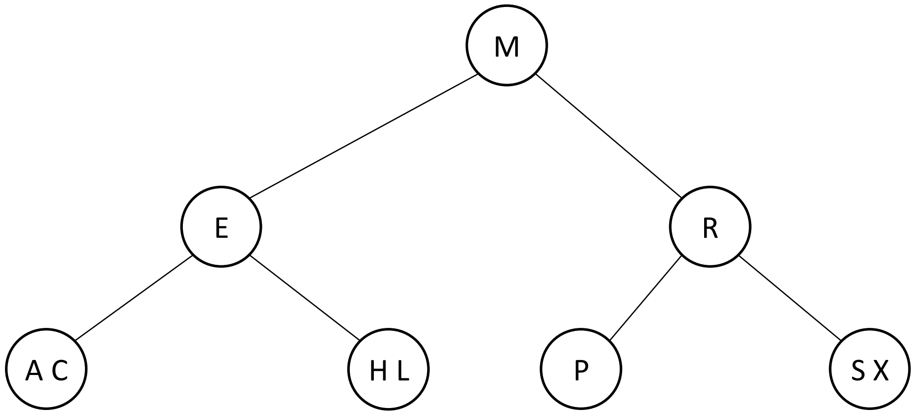


split 4-node
(move M to parent again)

Q1. Insert each element in an array given below to a 2-3 search tree (*show every rotation and color flip operation*)

S, E, A, R, C, H, X, M, P, L

insert L



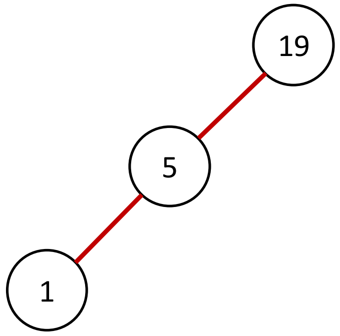
Q2. Insert each element in an array given below to a Red-Black BST (*show every rotation and color flip operation*)

19, 5, 1, 18, 3, 8, 24, 13, 16, 12

Q2. Insert each element in an array given below to a Red-Black BST (*show every rotation and color flip operation*)

19, 5, 1, 18, 3, 8, 24, 13, 16, 12

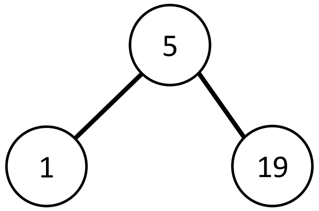
insert 19
insert 5
insert 1



two left reds in a row!

Q2. Insert each element in an array given below to a Red-Black BST (*show every rotation and color flip operation*)

19, 5, 1, 18, 3, 8, 24, 13, 16, 12

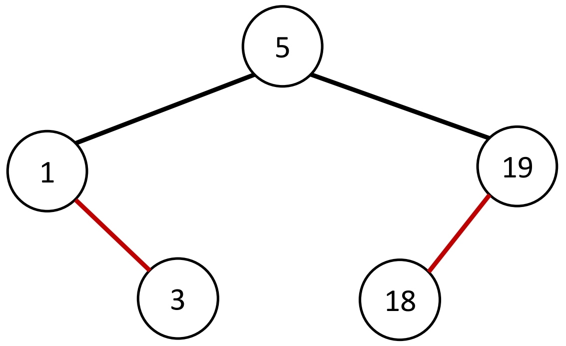


both children red
(flip colors)

Q2. Insert each element in an array given below to a Red-Black BST (*show every rotation and color flip operation*)

19, 5, 1, 18, 3, 8, 24, 13, 16, 12

right link red
(rotate 1 left)

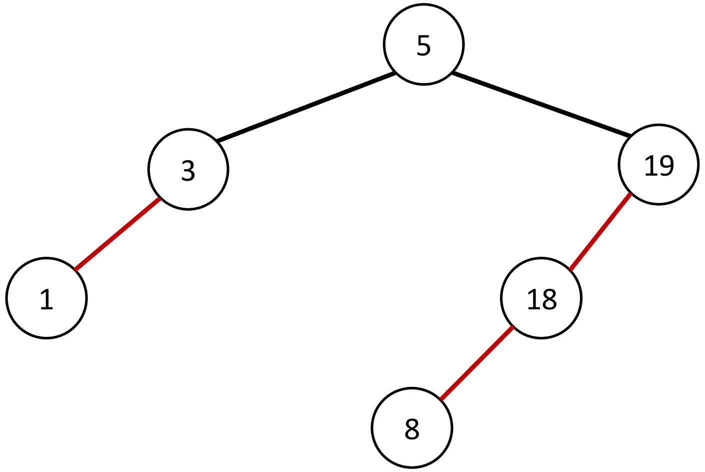


insert 18
insert 3

Q2. Insert each element in an array given below to a Red-Black BST (*show every rotation and color flip operation*)

19, 5, 1, 18, 3, 8, 24, 13, 16, 12

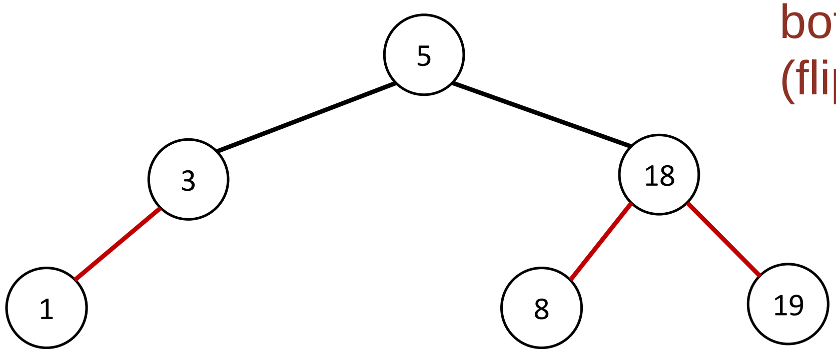
insert 8



two left reds in a row
(rotate 19 right)

Q2. Insert each element in an array given below to a Red-Black BST (*show every rotation and color flip operation*)

19, 5, 1, 18, 3, 8, 24, 13, 16, 12

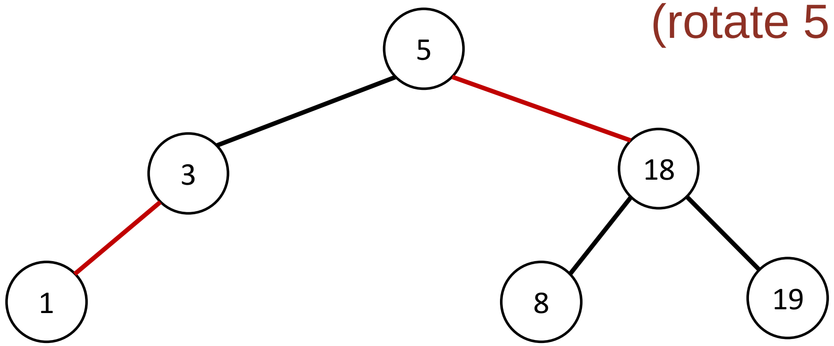


both children red
(flip colors)

Q2. Insert each element in an array given below to a Red-Black BST (*show every rotation and color flip operation*)

19, 5, 1, 18, 3, 8, 24, 13, 16, 12

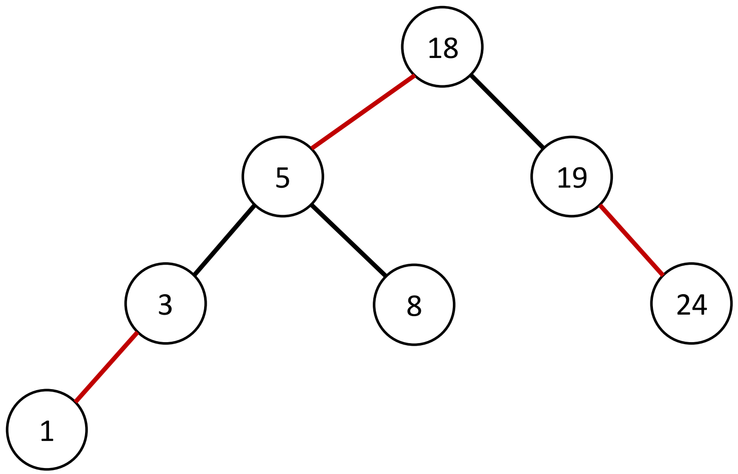
right link red
(rotate 5 left)



Q2. Insert each element in an array given below to a Red-Black BST (*show every rotation and color flip operation*)

19, 5, 1, 18, 3, 8, 24, 13, 16, 12

insert 24

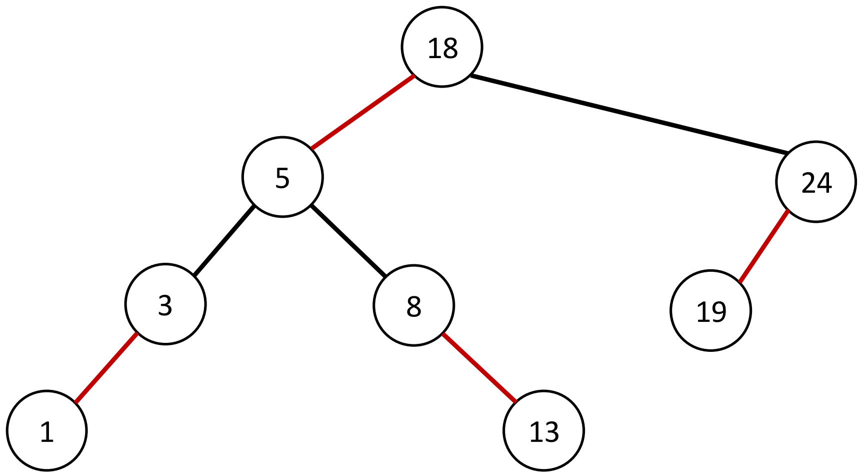


right link red
(rotate 19 left)

Q2. Insert each element in an array given below to a Red-Black BST (*show every rotation and color flip operation*)

19, 5, 1, 18, 3, 8, 24, 13, 16, 12

insert 13

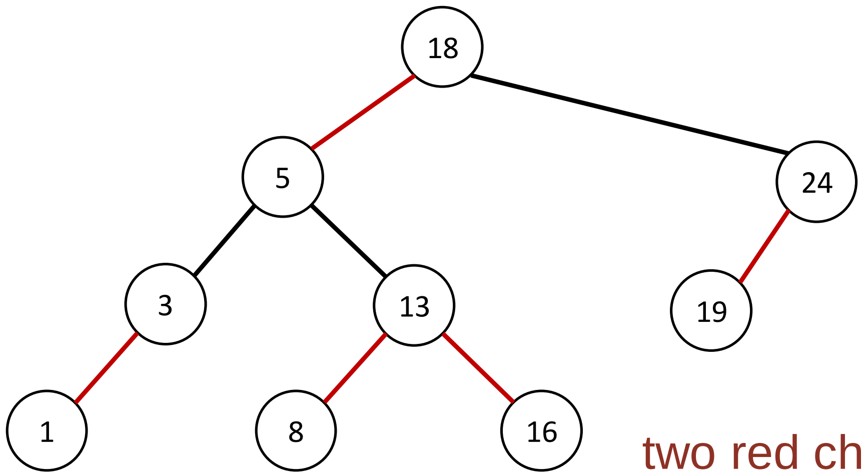


right link red
(rotate 13 left)

Q2. Insert each element in an array given below to a Red-Black BST (*show every rotation and color flip operation*)

19, 5, 1, 18, 3, 8, 24, 13, 16, 12

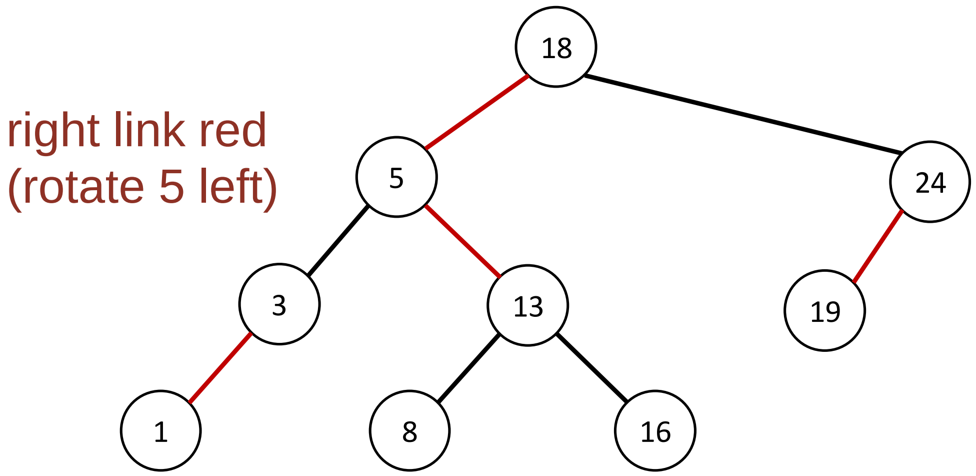
insert 16



two red children
(flip colors)

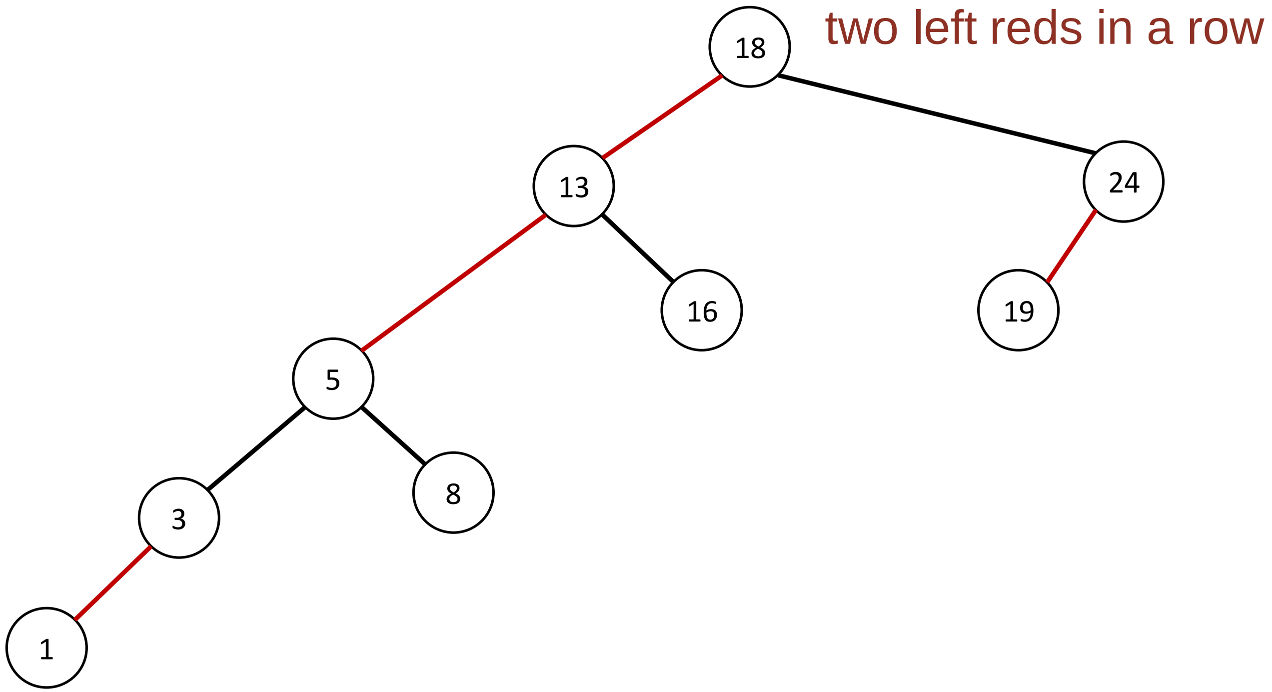
Q2. Insert each element in an array given below to a Red-Black BST (*show every rotation and color flip operation*)

19, 5, 1, 18, 3, 8, 24, 13, 16, 12



Q2. Insert each element in an array given below to a Red-Black BST (*show every rotation and color flip operation*)

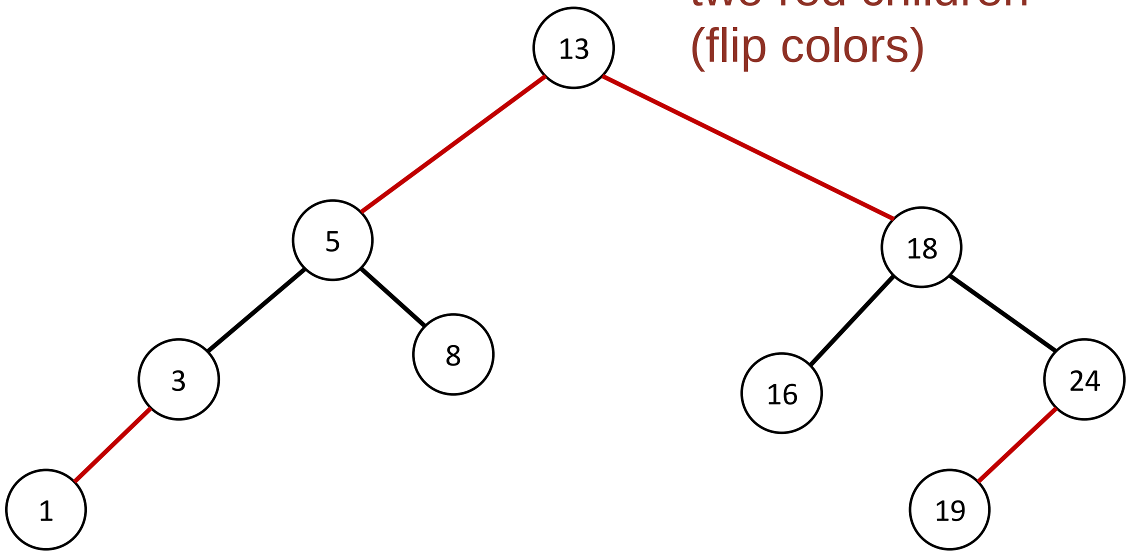
19, 5, 1, 18, 3, 8, 24, 13, 16, 12



Q2. Insert each element in an array given below to a Red-Black BST (*show every rotation and color flip operation*)

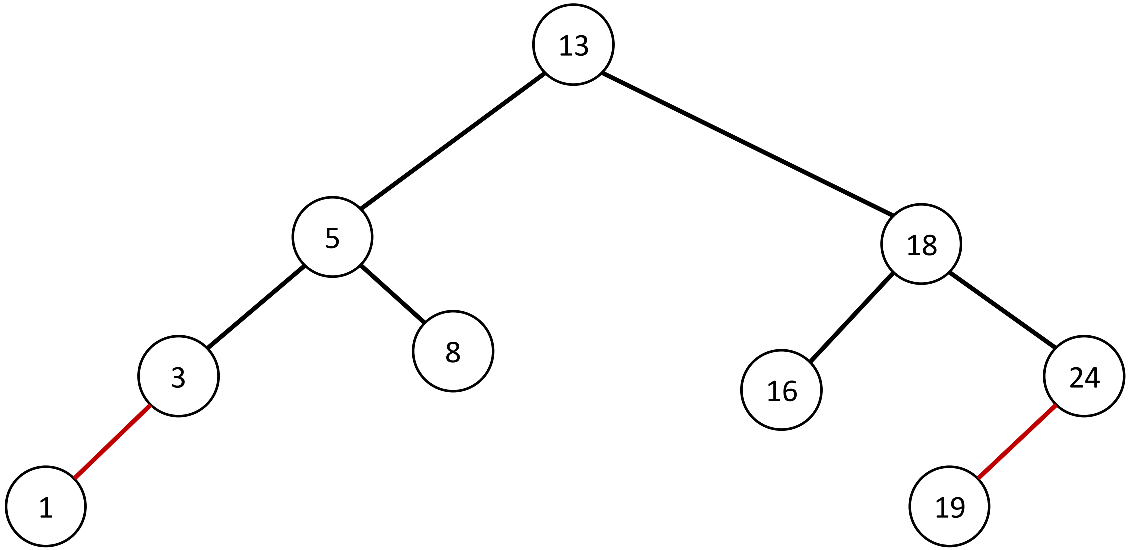
19, 5, 1, 18, 3, 8, 24, 13, 16, 12

two red children
(flip colors)



Q2. Insert each element in an array given below to a Red-Black BST (*show every rotation and color flip operation*)

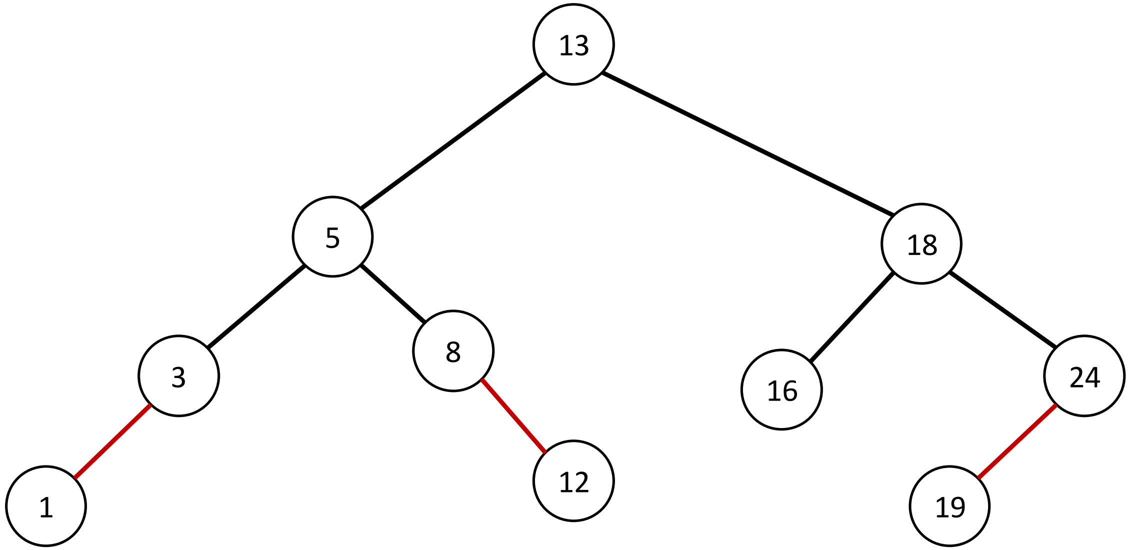
19, 5, 1, 18, 3, 8, 24, 13, 16, 12



Q2. Insert each element in an array given below to a Red-Black BST (*show every rotation and color flip operation*)

19, 5, 1, 18, 3, 8, 24, 13, 16, 12

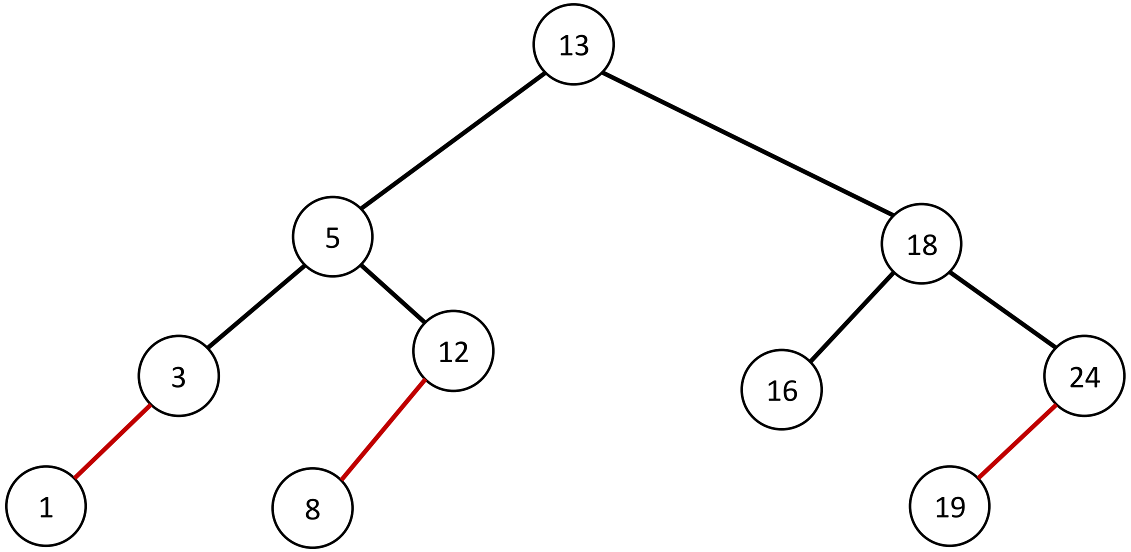
insert 12



right link red
(rotate 8 left)

Q2. Insert each element in an array given below to a Red-Black BST (*show every rotation and color flip operation*)

19, 5, 1, 18, 3, 8, 24, 13, 16, 12



Q3. **Mark the following true or false**.

Red-Black Tree is a self-balancing Binary Search Tree (BST) where every node follows following rules:

Root of tree is always black. (......)

Root of tree is always red. (......)

There are no two adjacent red nodes. (......)

There are no two adjacent black nodes. (......)

All the nodes in the red-black tree can be black. (......)

All the nodes in the red-black tree can be red. (......)

Q3. **Mark the following true or false**.

Red-Black Tree is a self-balancing Binary Search Tree (BST) where every node follows following rules:

Root of tree is always black. (T)

Root of tree is always red. (……)

There are no two adjacent red nodes. (……)

There are no two adjacent black nodes. (……)

All the nodes in the red-black tree can be black. (……)

All the nodes in the red-black tree can be red. (……)

Q3. **Mark the following true or false**.

Red-Black Tree is a self-balancing Binary Search Tree (BST) where every node follows following rules:

Root of tree is always black. (T)

Root of tree is always red. (F)

There are no two adjacent red nodes. (……)

There are no two adjacent black nodes. (……)

All the nodes in the red-black tree can be black. (……)

All the nodes in the red-black tree can be red. (……)

Q3. **Mark the following true or false.**

Red-Black Tree is a self-balancing Binary Search Tree (BST) where every node follows following rules:

Root of tree is always black. (T)

Root of tree is always red. (F)

There are no two adjacent red nodes. (T)

There are no two adjacent black nodes. (……)

All the nodes in the red-black tree can be black. (……)

All the nodes in the red-black tree can be red. (……)

Q3. **Mark the following true or false.**

Red-Black Tree is a self-balancing Binary Search Tree (BST) where every node follows following rules:

Root of tree is always black. (T)

Root of tree is always red. (F)

There are no two adjacent red nodes. (T)

There are no two adjacent black nodes. (F)

All the nodes in the red-black tree can be black. (……)

All the nodes in the red-black tree can be red. (……)

Q3. **Mark the following true or false**.

Red-Black Tree is a self-balancing Binary Search Tree (BST) where every node follows following rules:

Root of tree is always black. (T)

Root of tree is always red. (F)

There are no two adjacent red nodes. (T)

There are no two adjacent black nodes. (F)

All the nodes in the red-black tree can be black. (T)

All the nodes in the red-black tree can be red. (......)

Q3. **Mark the following true or false.**

Red-Black Tree is a self-balancing Binary Search Tree (BST) where every node follows following rules:

Root of tree is always black. (T)

Root of tree is always red. (F)

There are no two adjacent red nodes. (T)

There are no two adjacent black nodes. (F)

All the nodes in the red-black tree can be black. (T)

All the nodes in the red-black tree can be red. (F)

Hacettepe University
Computer Engineering Department

BBM204 Software Laboratory II

Week IV
Recitation III

Spring 2018

Q1.

Give the contents of the hash table that results when you insert items with the keys E A S Y Q U T I O N in that order into an initially empty table of M = 5 lists, using separate chaining with unordered lists. Use the hash function 11 k mod M to transform the kth letter of the alphabet into a table index, e.g., hash(I) = hash(9) = 99 % 5 = 4.

Solution:Separate chaining means that we are mapping our keys to our values as normal, but when we have a collision we just create a list. So let's say we have a hash map of size 3. The keys are numbers, but the values at each key are lists.

So what is being asked here is fairly simple. We have the keys (letters) and we want to map them to values (the numbers). There will be 5 lists in 5 buckets, numbered from 0 to 4, so our table looks like this:

A	B	C	D	E	F	G	H	I	J
1	2	3	4	5	6	7	8	9	10
K	L	M	N	O	P	Q	R	S	T
11	12	13	14	15	16	17	18	19	20
U	V	W	X	Y	Z				
21	22	23	24	25	26				

key	hash	value
E	0	0

0	E0
1	null
2	null
3	null
4	null

key	hash	value
A	1	1

0	E0
1	A1
2	null
3	null
4	null

key	hash	value
S	4	2

0	E0
1	A1
2	null
3	null
4	S2

key	hash	value
Y	0	3

```
0 E0 -> Y3
1 A1
2 null
3 null
4 S2
```

key	hash	value
Q	2	4

```
0 E0 -> Y3
1 A1
2 Q4
3 null
4 S2
```

key	hash	value
U	1	5

```
0 E0 -> Y3
1 A1 -> U5
2 Q4
3 null
4 S2
```

key	hash	value
T	0	6

```
0 E0 -> Y3 -> T6
1 A1 -> U5
2 Q4
3 null
4 S2
```

key	hash	value
I	4	7

```
0 E0 -> Y3 -> T6
1 A1 -> U5
2 Q4
3 null
4 S2 -> I7
```

key	hash	value
O	0	8

```
0 E0 -> Y3 -> T6 -> O8
1 A1 -> U5
2 Q4
3 null
4 S2 -> I7
```

```
key hash value
  N   4    9
```

```
0 E0 -> Y3 -> T6 -> O8
1 A1 -> U5
2 Q4
3 null
4 S2 -> I7 -> N9
```

Q2.

Give the contents of the hash table that results when you insert items with the keys E A S Y Q U T I O N in that order into an initially empty table of size M = 16 using linear probing. Use the hash function 11k mod M to transform the kth letter of the alphabet into a table index.

key	hash	value
E	7	0

0	
1	
2	
3	
4	
5	
6	
7	**E0**
8	
9	
10	
11	
12	
13	
14	
15	

key	hash	value
A	11	1

0	
1	
2	
3	
4	
5	
6	
7	E0
8	
9	
10	
11	**A1**
12	
13	
14	
15	

key	hash	value
S	1	2

0	
1	S2
2	
3	
4	
5	
6	
7	E0
8	
9	
10	
11	A1
12	
13	
14	
15	

key	hash	value
Y	3	3

0	
1	S2
2	
3	Y3
4	
5	
6	
7	E0
8	
9	
10	
11	A1
12	
13	
14	
15	

key	hash	value
Q	11	4

0	
1	S2
2	
3	Y3
4	
5	
6	
7	E0
8	
9	
10	
11	A1
12	**Q4**
13	
14	
15	

key	hash	value
U	7	5

0	
1	S2
2	
3	Y3
4	
5	
6	
7	E0
8	**U5**
9	
10	
11	A1
12	Q4
13	
14	
15	

key	hash	value
T	12	6

0	
1	S2
2	
3	Y3
4	
5	
6	
7	E0
8	U5
9	
10	
11	A1
12	Q4
13	**T6**
14	
15	

key	hash	value
I	3	7

0	
1	S2
2	
3	Y3
4	**I7**
5	
6	
7	E0
8	U5
9	
10	
11	A1
12	Q4
13	T6
14	
15	

key hash value

O 11 8

0	
1	S2
2	
3	Y3
4	I7
5	**O8**
6	
7	E0
8	U5
9	
10	
11	A1
12	Q4
13	T6
14	
15	

key hash value

N 10 9

0	
1	S2
2	
3	Y3
4	I7
5	
6	
7	E0
8	U5
9	
10	**N9**
11	A1
12	Q4
13	T6
14	O8
15	

Sparse Vector - Dense Vector

Indices: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,12,13,14

Dense Vector: [0,0,0,0,0,0,1,0,0,0,0,2,0,0,3]

```
int vector[15] ={0,0,0,0,0,0,1,0,0,0,0,2,0,0,3};
```

Sparse Vector: [6:1,11:2,14:3]

```
HashMap<Integer, Double> vector = new HashMap<Integer, Double>();
vector.put(6, 1);
vector.put(11, 2);
vector.put(14, 3);
```

SparseVector.java

```java
import java.util.HashMap;

public class SparseVector {
    private int d;                     // dimension
    private HashMap<Integer, Double> st;  // the vector, represented by index-value pairs

    /**
     * Initializes a d-dimensional zero vector.
     * @param d the dimension of the vector
     */
    public SparseVector(int d) {
        this.d  = d;
        this.st = new HashMap<Integer, Double>();
    }

    /**
     * Sets the ith coordinate of this vector to the specified value.
     *
     * @param  i the index
     * @param  value the new value
     * @throws IllegalArgumentException unless i is between 0 and d-1
     */
    public void put(int i, double value) {
        if (i < 0 || i >= d) throw new IllegalArgumentException("Illegal index");
        if (value == 0.0) st.remove(i);
        else              st.put(i, value);
    }

    /**
     * Returns the ith coordinate of this vector.
     *
     * @param  i the index
     * @return the value of the ith coordinate of this vector
     * @throws IllegalArgumentException unless i is between 0 and d-1
     */
    public double get(int i) {
        if (i < 0 || i >= d) throw new IllegalArgumentException("Illegal index");
        if (st.containsKey(i)) return st.get(i);
        else                   return 0.0;
    }

    /**
     * Returns the number of nonzero entries in this vector.
     *
     * @return the number of nonzero entries in this vector
     */
    public int nnz() {
        return st.size();
    }

    /**
     * Returns the dimension of this vector.
     *
     * @return the dimension of this vector
     * @deprecated Replaced by {@link #dimension()}.
     */
    @Deprecated
    public int size() {
        return d;
    }

    /**
     * Returns the dimension of this vector.
     *
     * @return the dimension of this vector
     */
    public int dimension() {
        return d;
    }

    /**
     * Returns the inner product of this vector with the specified vector.
     *
     * @param  that the other vector
     * @return the dot product between this vector and that vector
     * @throws IllegalArgumentException if the lengths of the two vectors are not equal
     */
    public double dot(SparseVector that) {
        if (this.d != that.d) throw new IllegalArgumentException("Vector lengths disagree");
        double sum = 0.0;

        // iterate over the vector with the fewest nonzeros
        if (this.st.size() <= that.st.size()) {
            for (int i : this.st.keySet())
                if (that.st.containsKey(i)) sum += this.get(i) * that.get(i);
        }
        else  {
            for (int i : that.st.keySet())
                if (this.st.containsKey(i)) sum += this.get(i) * that.get(i);
        }
        return sum;
    }
```

SparseVector.java

```java
    /**
     * Returns the inner product of this vector with the specified array.
     *
     * @param  that the array
     * @return the dot product between this vector and that array
     * @throws IllegalArgumentException if the dimensions of the vector and the array are not equal
     */
    public double dot(double[] that) {
        double sum = 0.0;
        for (int i : st.keySet())
            sum += that[i] * this.get(i);
        return sum;
    }

    /**
     * Returns the magnitude of this vector.
     * This is also known as the L2 norm or the Euclidean norm.
     *
     * @return the magnitude of this vector
     */
    public double magnitude() {
        return Math.sqrt(this.dot(this));
    }


    /**
     * Returns the Euclidean norm of this vector.
     *
     * @return the Euclidean norm of this vector
     * @deprecated Replaced by {@link #magnitude()}.
     */
    @Deprecated
    public double norm() {
        return Math.sqrt(this.dot(this));
    }

    /**
     * Returns the scalar-vector product of this vector with the specified scalar.
     *
     * @param  alpha the scalar
     * @return the scalar-vector product of this vector with the specified scalar
     */
    public SparseVector scale(double alpha) {
        SparseVector c = new SparseVector(d);
        for (int i : this.st.keySet()) c.put(i, alpha * this.get(i));
        return c;
    }

    /**
     * Returns the sum of this vector and the specified vector.
     *
     * @param  that the vector to add to this vector
     * @return the sum of this vector and that vector
     * @throws IllegalArgumentException if the dimensions of the two vectors are not equal
     */
    public SparseVector plus(SparseVector that) {
        if (this.d != that.d) throw new IllegalArgumentException("Vector lengths disagree"
        SparseVector c = new SparseVector(d);
        for (int i : this.st.keySet()) c.put(i, this.get(i));                // c = this
        for (int i : that.st.keySet()) c.put(i, that.get(i) + c.get(i));     // c = c + th
        return c;
    }

   /**
     * Returns a string representation of this vector.
     * @return a string representation of this vector, which consists of the
     *         the vector entries, separates by commas, enclosed in parentheses
     */
    public String toString() {
        StringBuilder s = new StringBuilder();
        for (int i : st.keySet()) {
            s.append("(" + i + ", " + st.get(i) + ") ");
        }
        return s.toString();
    }


    /**
     * Unit tests the {@code SparseVector} data type.
     *
     * @param args the command-line arguments
     */
    public static void main(String[] args) {
        SparseVector a = new SparseVector(10);
        SparseVector b = new SparseVector(10);
        a.put(3, 0.50);
        a.put(9, 0.75);
        a.put(6, 0.11);
        a.put(6, 0.00);
        b.put(3, 0.60);
        b.put(4, 0.90);
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("a dot b = " + a.dot(b));
        System.out.println("a + b   = " + a.plus(b));
    }
}
```

SparseMatrix.java

```java
public class SparseMatrix {
    private int n;                 // n-by-n matrix
    private SparseVector[] rows;   // the rows, each row is a sparse vector

    // initialize an n-by-n matrix of all 0s
    public SparseMatrix(int n) {
        this.n = n;
        rows = new SparseVector[n];
        for (int i = 0; i < n; i++)
            rows[i] = new SparseVector(n);
    }

    // put A[i][j] = value
    public void put(int i, int j, double value) {
        if (i < 0 || i >= n) throw new IllegalArgumentException("Illegal index");
        if (j < 0 || j >= n) throw new IllegalArgumentException("Illegal index");
        rows[i].put(j, value);
    }

    // return A[i][j]
    public double get(int i, int j) {
        if (i < 0 || i >= n) throw new IllegalArgumentException("Illegal index");
        if (j < 0 || j >= n) throw new IllegalArgumentException("Illegal index");
        return rows[i].get(j);
    }

    // return the number of nonzero entries (not the most efficient implementation)
    public int nnz() {
        int sum = 0;
        for (int i = 0; i < n; i++)
            sum += rows[i].nnz();
        return sum;
    }

    // return the matrix-vector product b = Ax
    public SparseVector times(SparseVector x) {
        if (n != x.size()) throw new IllegalArgumentException("Dimensions disagree");
        SparseVector b = new SparseVector(n);
        for (int i = 0; i < n; i++)
            b.put(i, rows[i].dot(x));
        return b;
    }


    // return this + that
    public SparseMatrix plus(SparseMatrix that) {
        if (this.n != that.n) throw new RuntimeException("Dimensions disagree");
        SparseMatrix result = new SparseMatrix(n);
        for (int i = 0; i < n; i++)
            result.rows[i] = this.rows[i].plus(that.rows[i]);
        return result;
    }


    // return a string representation
    public String toString() {
        String s = "n = " + n + ", nonzeros = " + nnz() + "\n";
        for (int i = 0; i < n; i++) {
            s += i + ": " + rows[i] + "\n";
        }
        return s;
    }


    // test client
    public static void main(String[] args) {
        SparseMatrix A = new SparseMatrix(5);
        SparseVector x = new SparseVector(5);
        A.put(0, 0, 1.0);
        A.put(1, 1, 1.0);
        A.put(2, 2, 1.0);
        A.put(3, 3, 1.0);
        A.put(4, 4, 1.0);
        A.put(2, 4, 0.3);
        x.put(0, 0.75);
        x.put(2, 0.11);
        System.out.println("x     : " + x);
        System.out.println("A     : " + A);
        System.out.println("Ax    : " + A.times(x));
        System.out.println("A + A : " + A.plus(A));
    }

}
```

Hacettepe University
Computer Engineering Department

BBM204 Software Laboratory II

Week X
Recitation IX

Spring 2019

Q1. Consider a hash table of size **3** with hash function $\boldsymbol{h(x)=(5x+2)\ mod\ 3}$. Draw the table that results after inserting given keys below in the given order when collisions are handled by **separate chaining algorithm**.

5, 4, 3, 11, 7, 10, 8, 9, 23, 40

Hash	**Value**
5	(5*5+2) mod 3 = 0

0	5
1	
2	

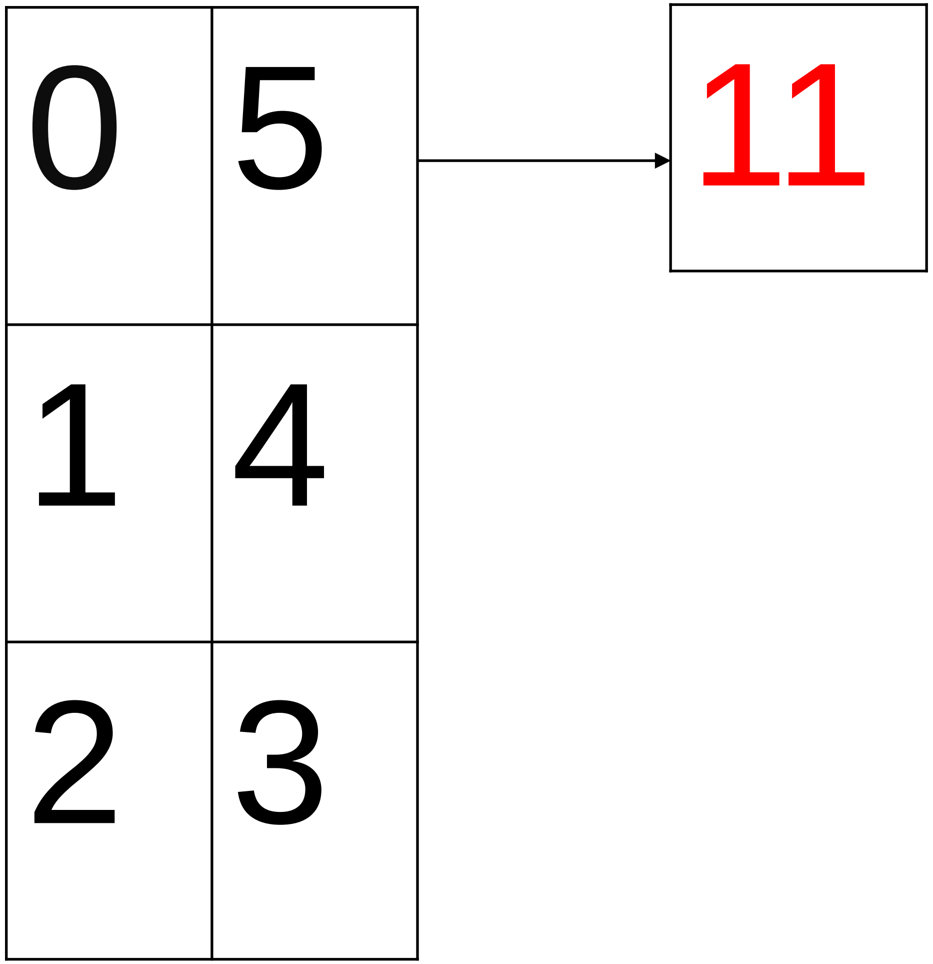
Hash	**Value**
4	(5*4+2) mod 3 = 1

0	5
1	4
2	

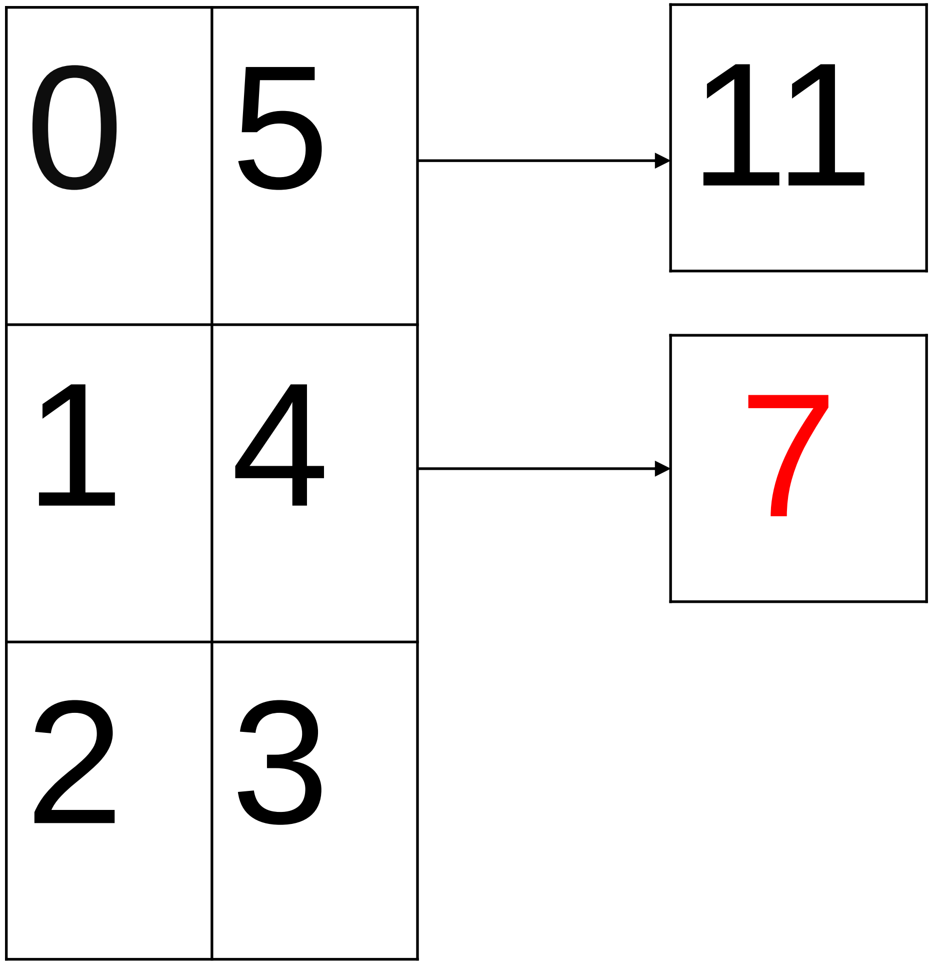
Hash	**Value**
3	(5*3+2) mod 3 = 2

0	5
1	4
2	3

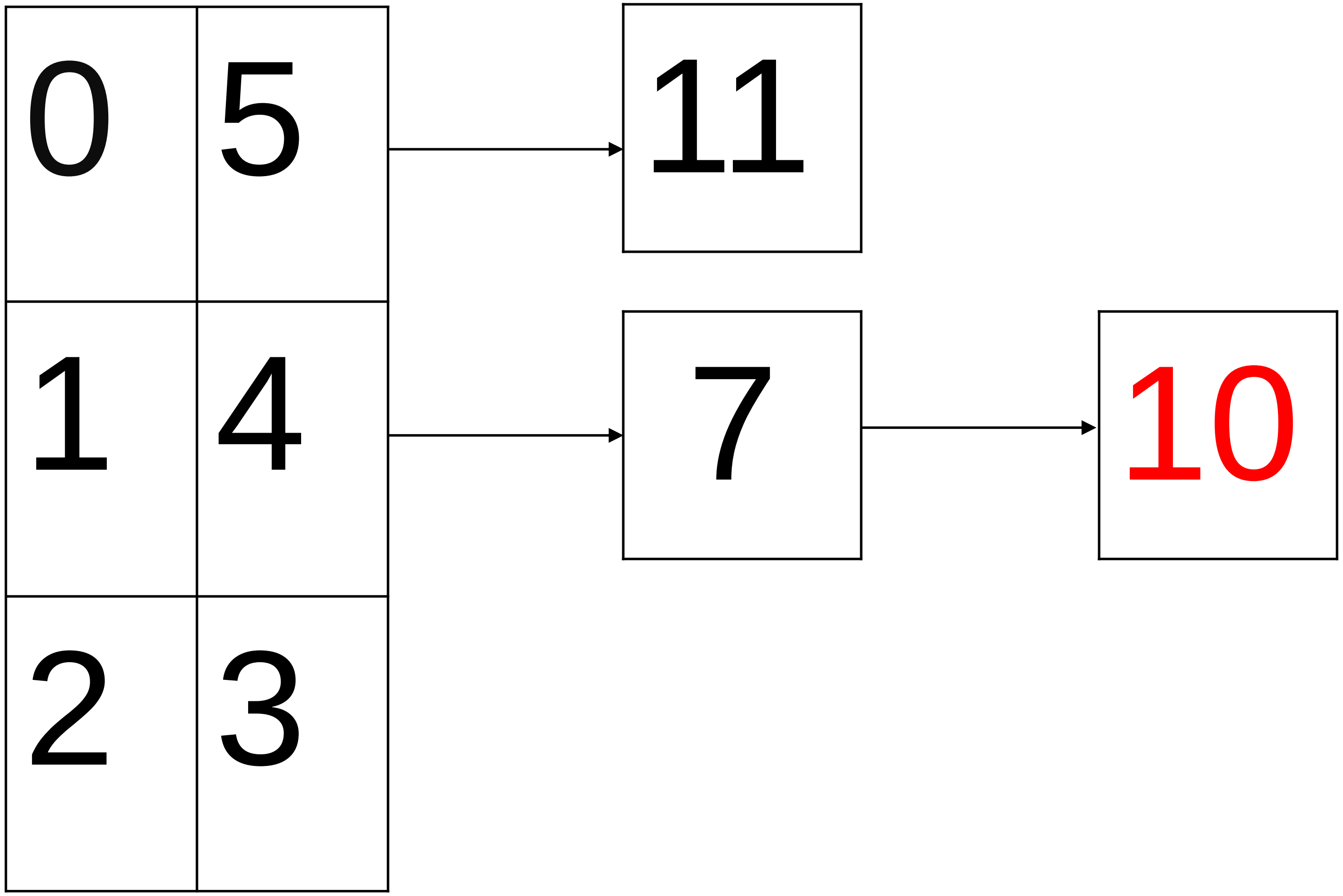
Hash	**Value**
11	(5*11+2) mod 3 = 0



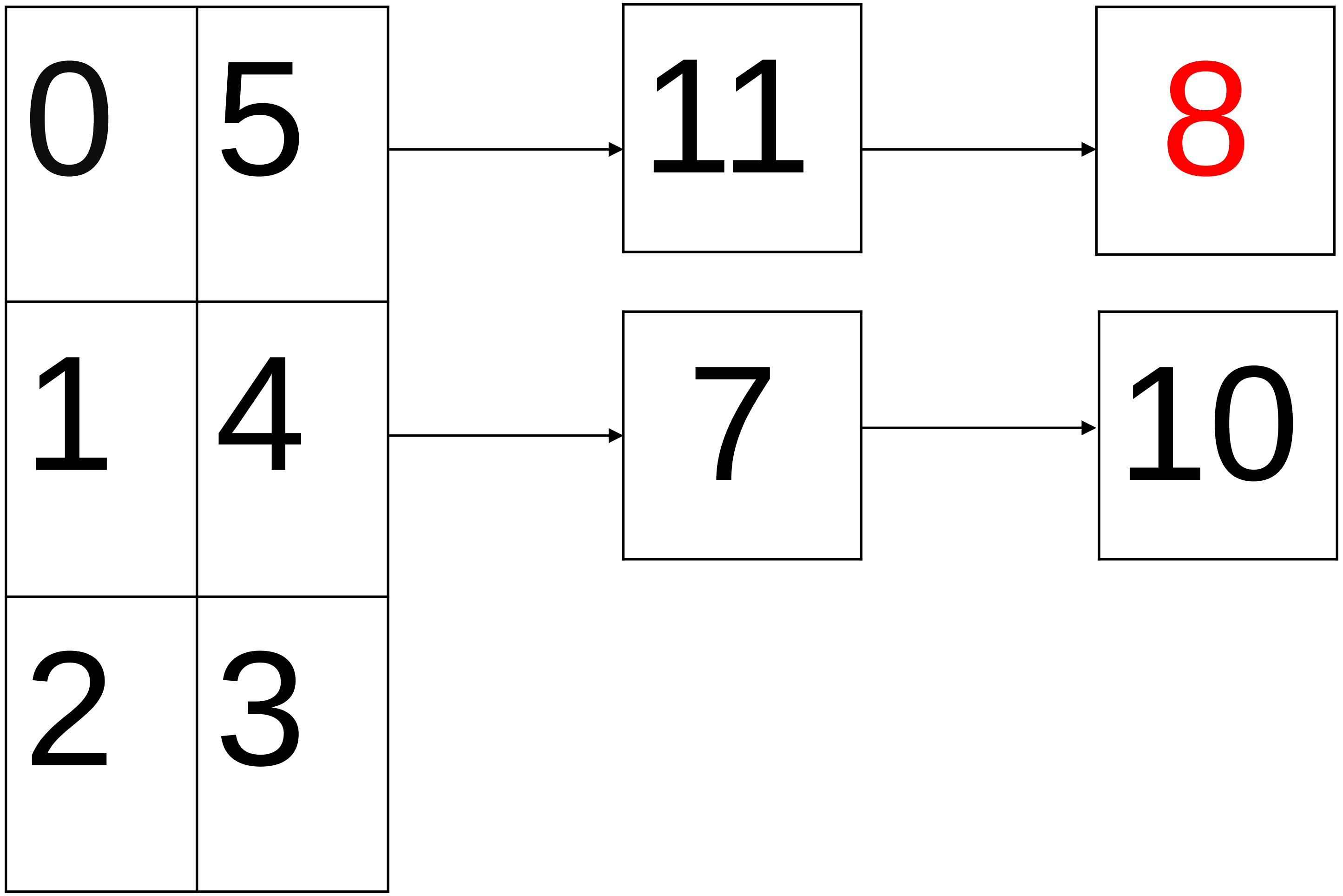
Hash	**Value**
7	(5*7+2) mod 3 = 1



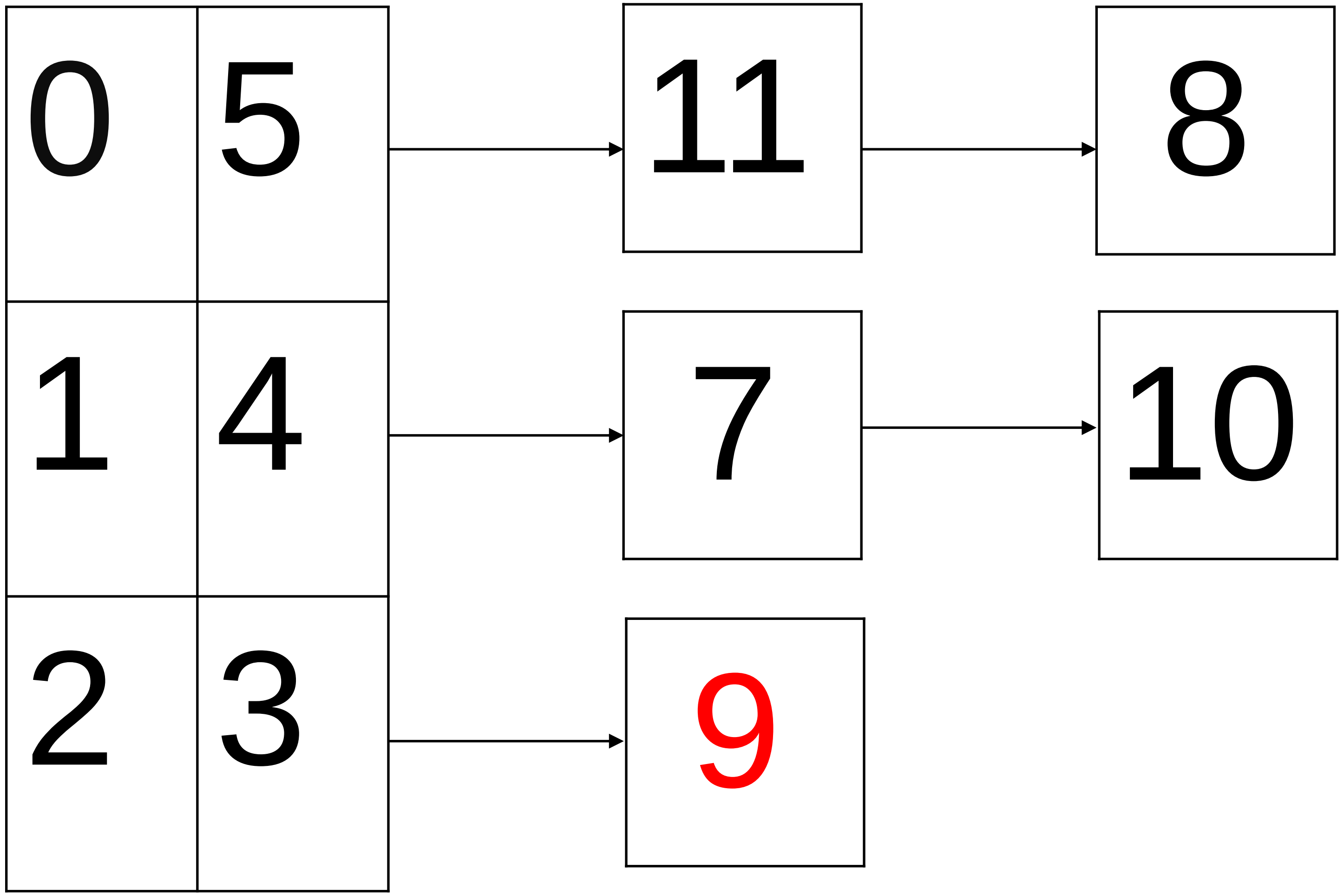
Hash	**Value**
10	(5*10+2) mod 3 = 1



Hash	**Value**
8	(5*8+2) mod 3 = 0

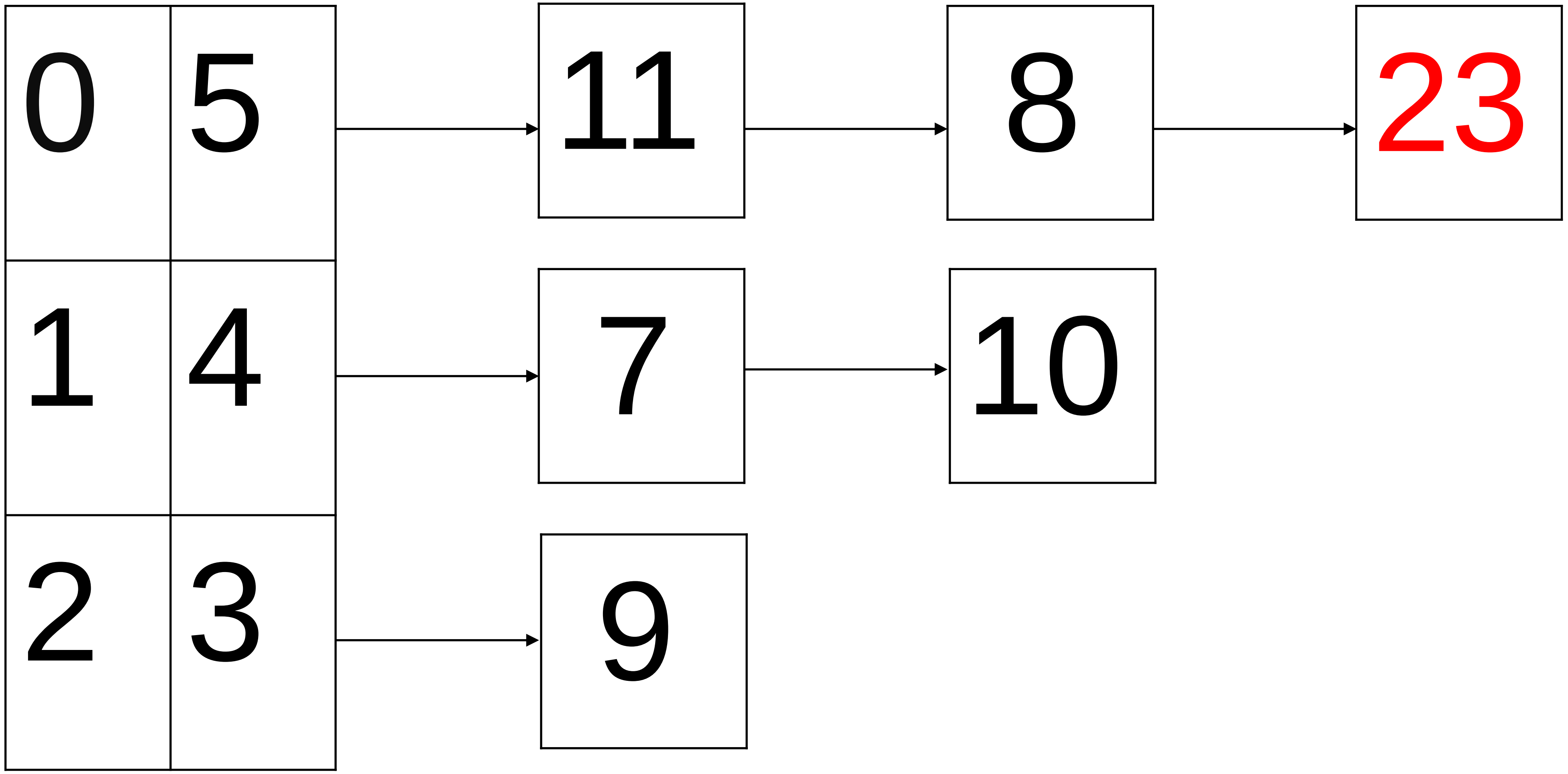


Hash	**Value**
9	(5*9+2) mod 3 = 2

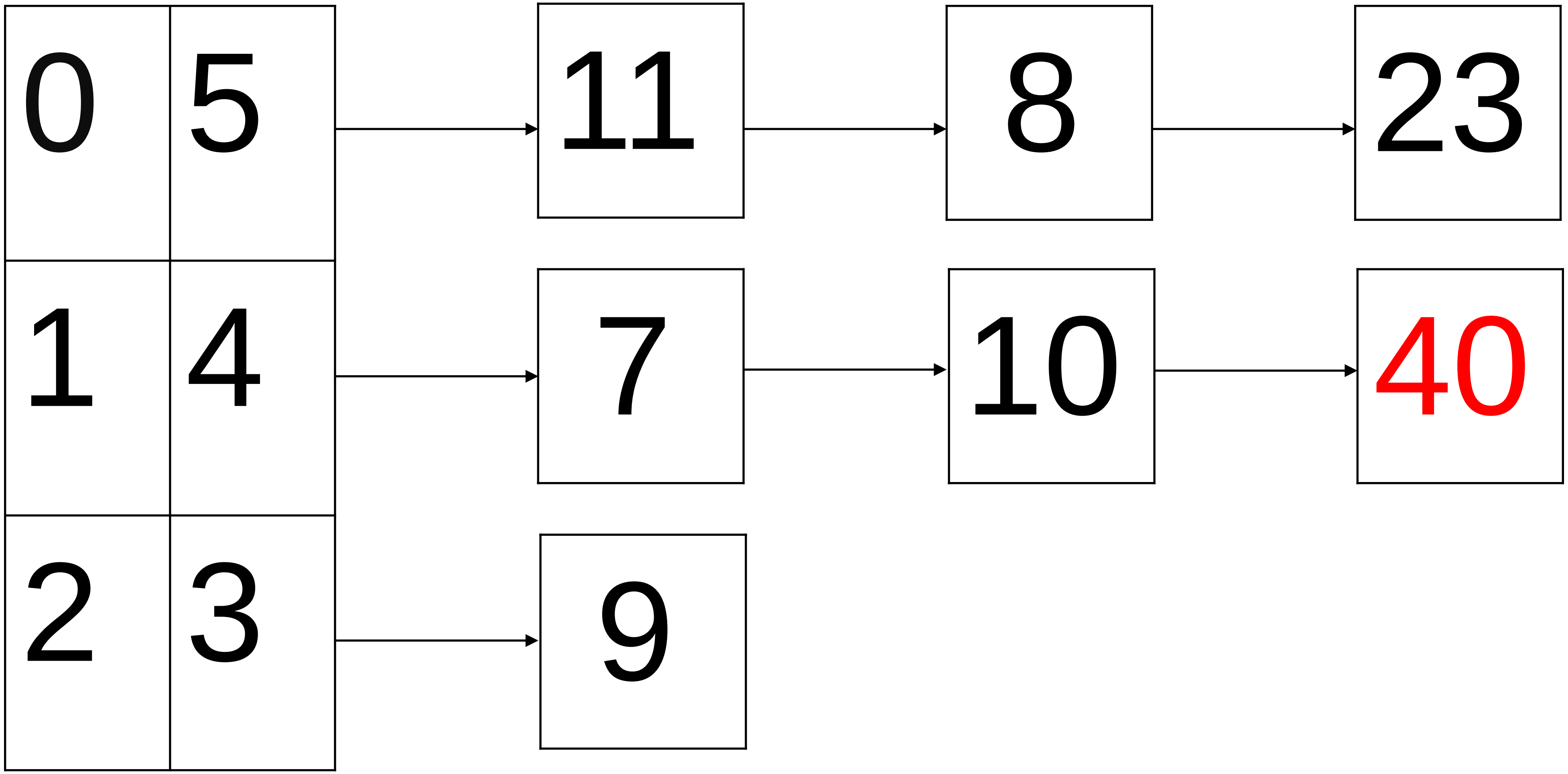


Hash **Value**

23 $(5*23+2) \bmod 3 = 0$



Hash	**Value**
40	(5*40+2) mod 3 = 1



Q2. Consider a hash table of size **10** with hash function $\boldsymbol{h(x) = (9 - (x + 4))\ mod\ 10}$. Draw the table that results after inserting given keys below in the given order when collisions are handled by **linear probing algorithm**.

12, 11, 7, 6, 3, 13, 10, 4, 2, 9

0	
1	
2	
3	12
4	
5	
6	
7	
8	
9	

Hash	**Value**
12	(9-(12+4)) mod 10 = 3

0	
1	
2	
3	12
4	11
5	
6	
7	
8	
9	

Hash	**Value**
11	(9-(11+4)) mod 10 = 4

0	
1	
2	
3	12
4	11
5	
6	
7	
8	7
9	

Hash	**Value**
7	(9-(7+4)) mod 10 = 8

0	
1	
2	
3	12
4	11
5	
6	
7	
8	7
9	6

Hash	**Value**
6	(9-(6+4)) mod 10 = 9

0	
1	
2	3
3	12
4	11
5	
6	
7	
8	7
9	6

Hash	**Value**
3	(9-(3+4)) mod 10 = 2

Hash	**Value**
13	(9-(13+4)) mod 10 = 2

0	
1	
2	3
3	12
4	11
5	13
6	
7	
8	7
9	6

0	
1	
2	3
3	12
4	11
5	13
6	10
7	
8	7
9	6

Hash	**Value**
10	(9-(10+4)) mod 10 = 5

0	
1	4
2	3
3	12
4	11
5	13
6	10
7	
8	7
9	6

Hash	**Value**
4	(9-(4+4)) mod 10 = 1

0	
1	4
2	3
3	12
4	11
5	13
6	10
7	2
8	7
9	6

Hash	**Value**
2	(9-(2+4)) mod 10 = 3

0	9
1	4
2	3
3	12
4	11
5	13
6	10
7	2
8	7
9	6

Hash	**Value**
9	(9-(9+4)) mod 10 = 6

Depth First Search is equivalent to which of the traversal in the Binary Trees?

a) Pre-order Traversal

b) Post-order Traversal

c) Level-order Traversal

d) In-order Traversal

Answer: a

Explanation: In Depth First Search, we explore all the nodes aggressively to one path and then backtrack to the node. Hence, it is equivalent to the pre-order traversal of a Binary Tree.

The Depth First Search traversal of a graph will result into?

a) Linked List

b) Tree

c) Graph with back edges

d) None of the mentioned

Answer: b

Explanation: The Depth First Search will make a graph which don’t have back edges (a tree) which is known as Depth First Tree.

What can be the applications of Depth First Search?
a) For generating topological sort of a graph
b) For generating Strongly Connected Components of a directed graph
c) Detecting cycles in the graph
d) All of the mentioned
Answer: d
Explanation: Depth First Search can be applied to all of the mentioned problems.

Hacettepe University

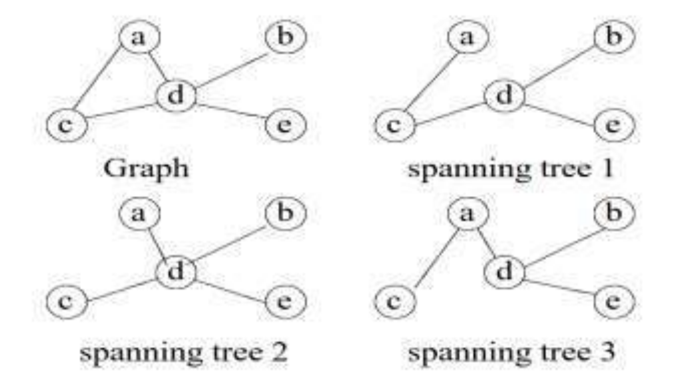
Computer Engineering Department

BBM204 Software Laboratory II

Minimum Spanning Trees: Prim and Kruskal

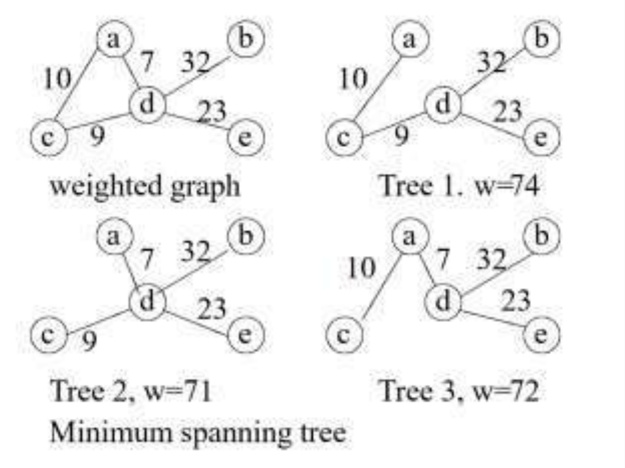
Spanning Trees

Spanning Trees: A subgraph T of a undirected graph G=(V,E) is a spanning tree of G if it is a tree and contains every vertex of G.



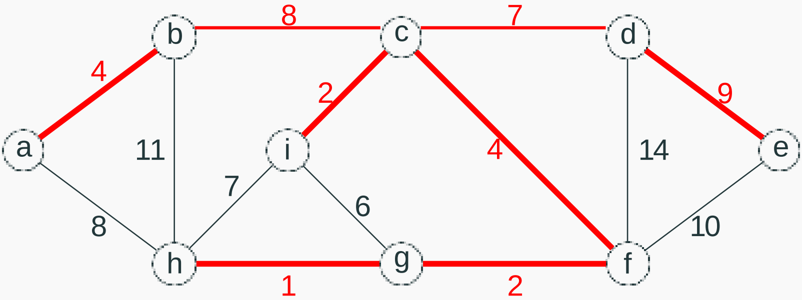
Minimum spanning trees

A Minimum SpanningTree in an undirected connected weighted graph is a spanning tree of minimum weight (among all spanning trees).



Minimum spanning trees: example

An example of an minimum spanning tree (MST):



Minimum spanning trees: properties

Important properties:

- A valid MST cannot contain a cycle

Minimum spanning trees: properties

Important properties:

- A valid MST cannot contain a cycle
- If we add or remove an edge from an MST, it's no longer a valid MST for that graph.
  Adding an edge introduces a cycle; removing an edge means vertices are no longer connected.

Minimum spanning trees: properties

Important properties:

- A valid MST cannot contain a cycle
- If we add or remove an edge from an MST, it's no longer a valid MST for that graph.
  Adding an edge introduces a cycle; removing an edge means vertices are no longer connected.
- If there are $|V|$ vertices, the MST contains exactly $|V| - 1$ edges.

Minimum spanning trees: properties

Important properties:

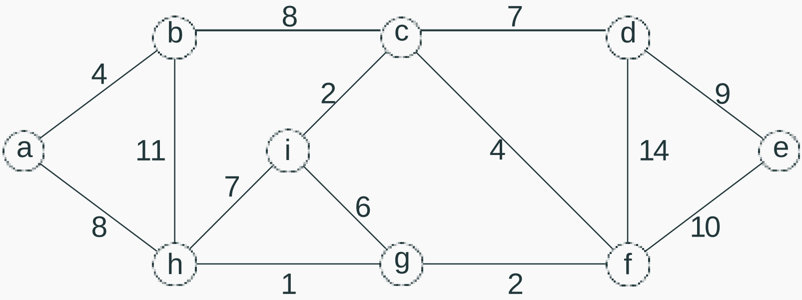
- A valid MST cannot contain a cycle
- If we add or remove an edge from an MST, it's no longer a valid MST for that graph.
  Adding an edge introduces a cycle; removing an edge means vertices are no longer connected.
- If there are $|V|$ vertices, the MST contains exactly $|V| - 1$ edges.
- An MST is always a tree.

Minimum spanning trees: properties

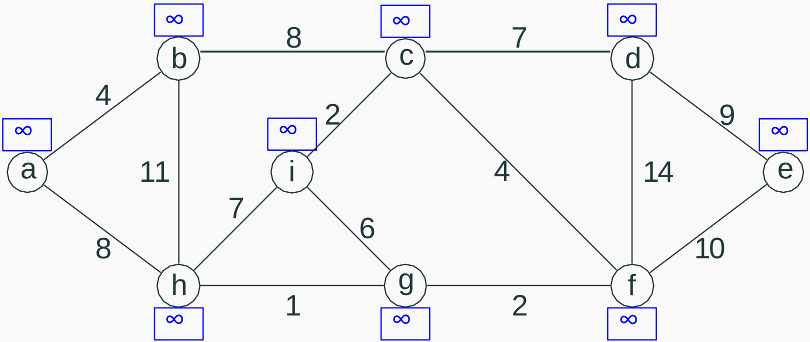
Important properties:

- A valid MST cannot contain a cycle
- If we add or remove an edge from an MST, it's no longer a valid MST for that graph.
  Adding an edge introduces a cycle; removing an edge means vertices are no longer connected.
- If there are $|V|$ vertices, the MST contains exactly $|V| - 1$ edges.
- An MST is always a tree.
- If every edge has a unique weight, there exists a unique MST.

Prim's algorithm: an example

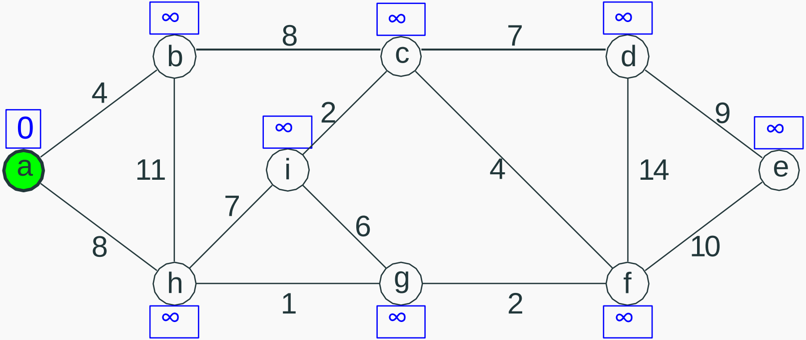


Prim's algorithm: an example



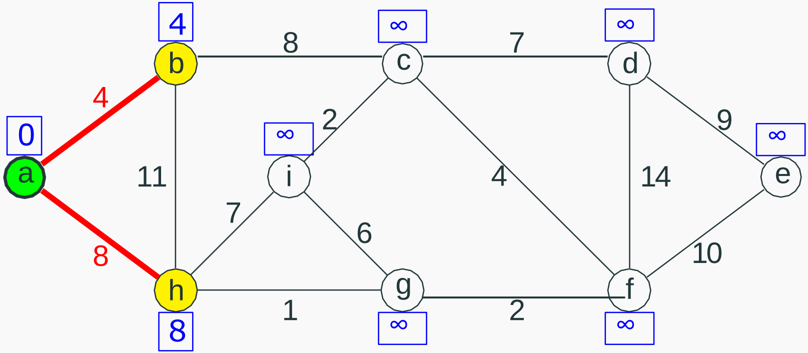
We initially set all costs to ∞, just like with Dijkstra.

Prim's algorithm: an example



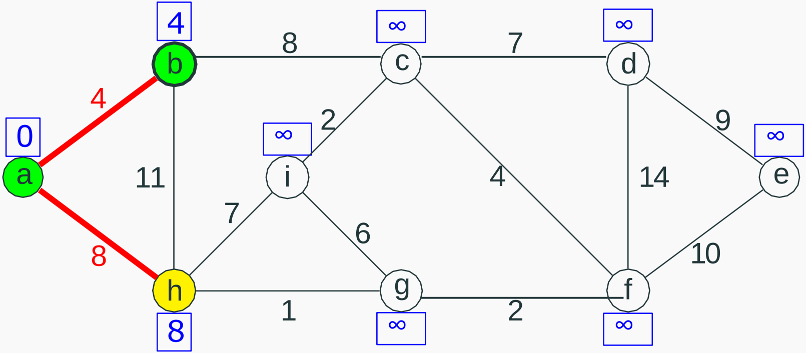
We pick an arbitrary node to start.

Prim’s algorithm: an example



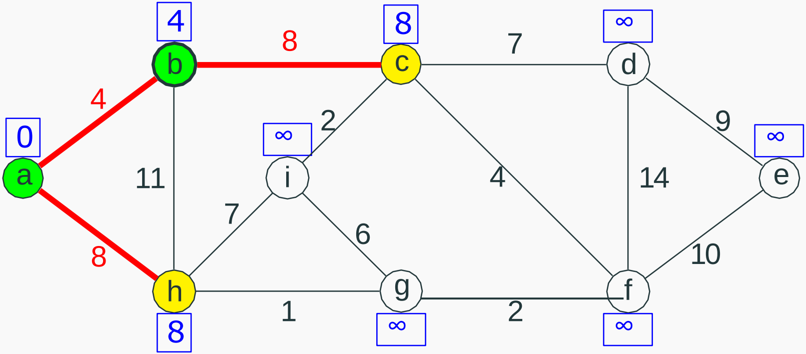
We update the adjacent nodes.

Prim’s algorithm: an example



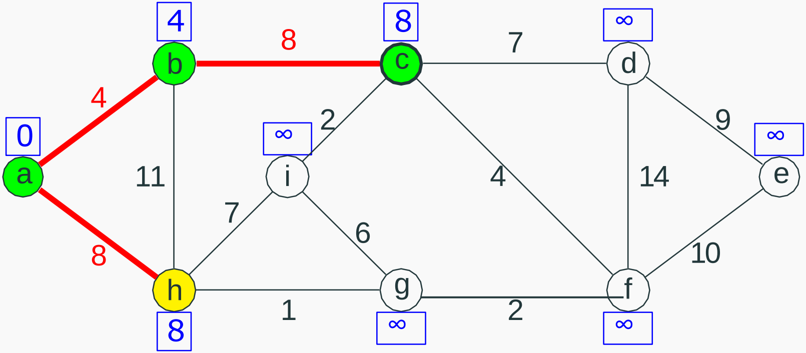
We select the one with the smallest cost.

Prim’s algorithm: an example



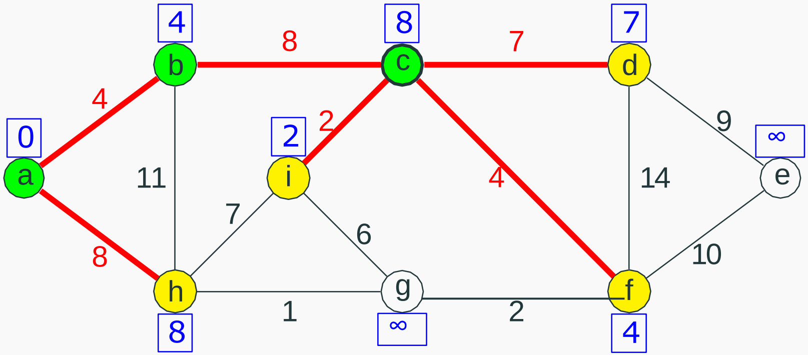
We potentially need to update h and c, but only c changes.

Prim’s algorithm: an example



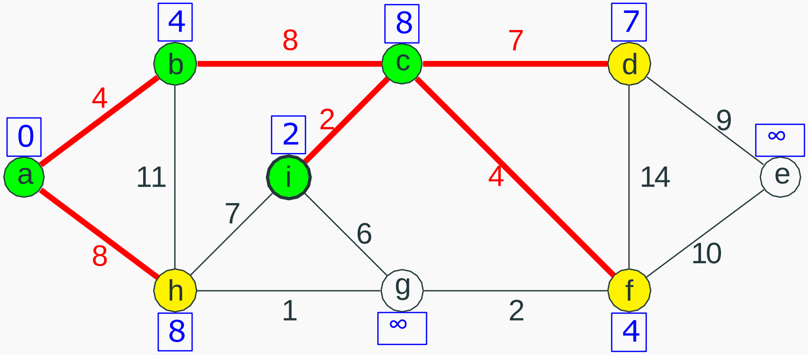
We (arbitrarily) pick c.

Prim's algorithm: an example



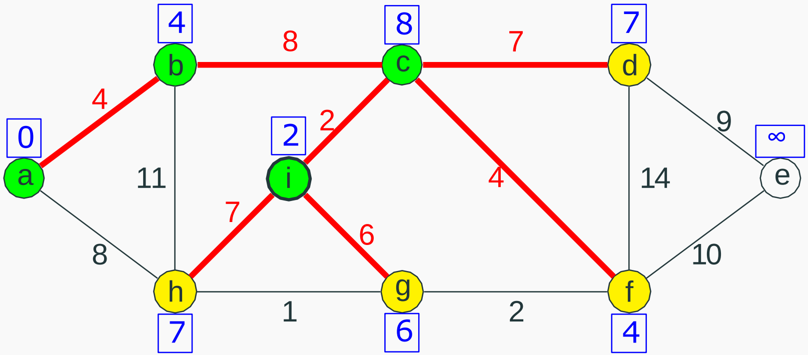
...and update the adjacent nodes. Note that we don't add the cumulative cost: the cost represents the shortest path to *any* green node, not to the start.

Prim’s algorithm: an example



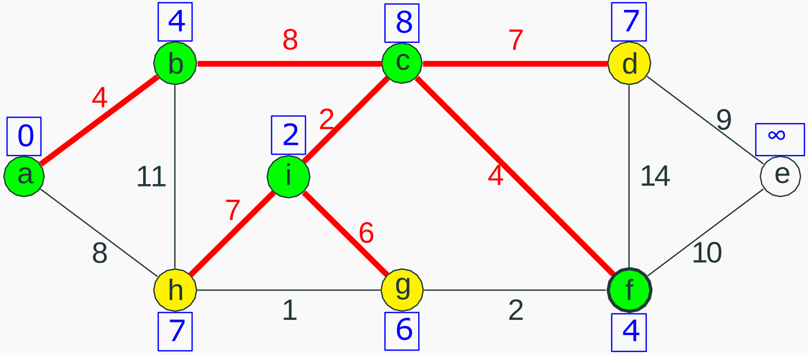
i has the smallest cost.

Prim's algorithm: an example



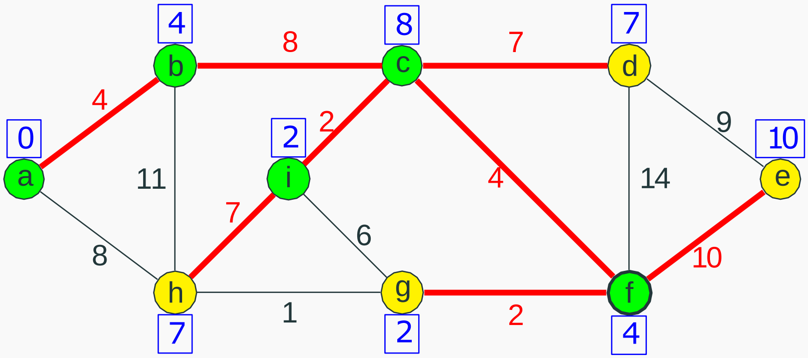
We update both unvisited nodes, and modify the edge to *h* since we now have a better option.

Prim's algorithm: an example



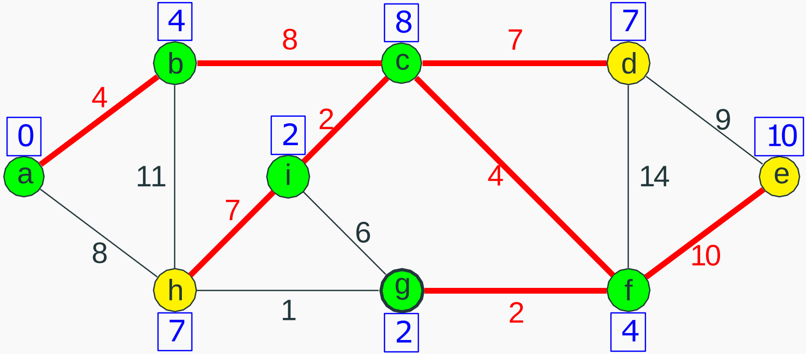
f has the smallest cost.

Prim’s algorithm: an example



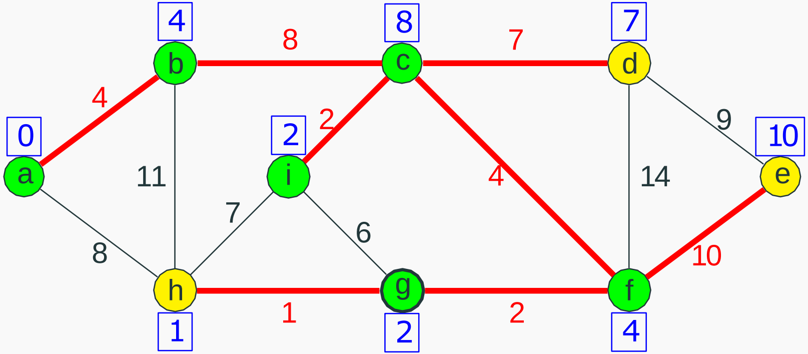
Again, we update the adjacent unvisited nodes.

Prim's algorithm: an example



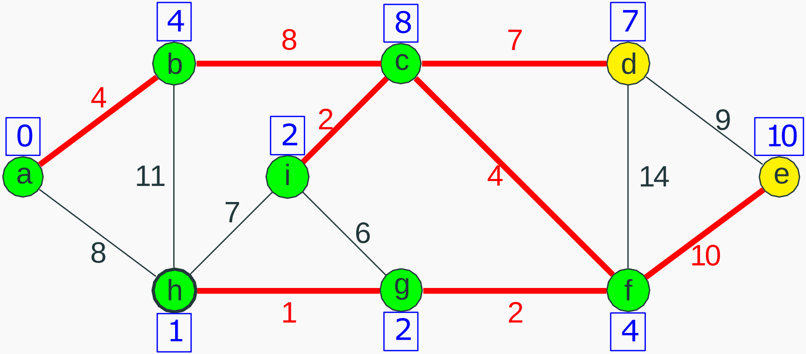
g has the smallest cost.

Prim's algorithm: an example



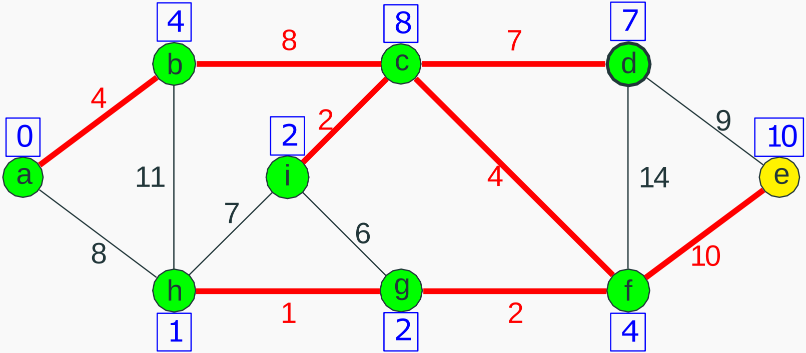
We update *h* again.

Prim's algorithm: an example



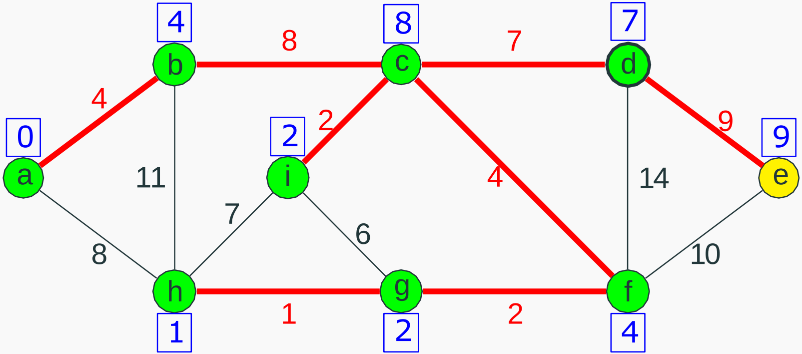
h has the smallest cost. Note that there nothing to update here.

Prim’s algorithm: an example



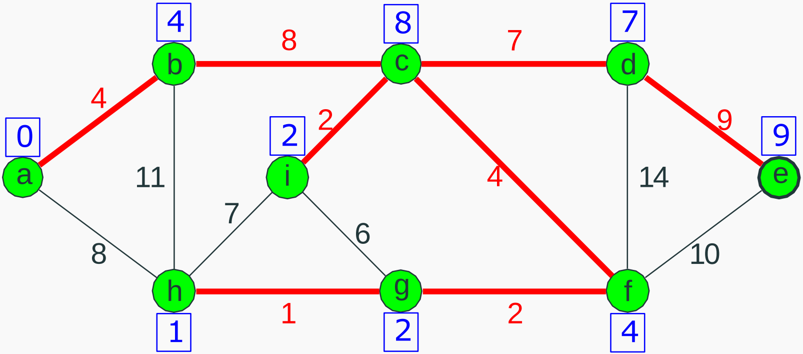
d has the smallest cost.

Prim's algorithm: an example



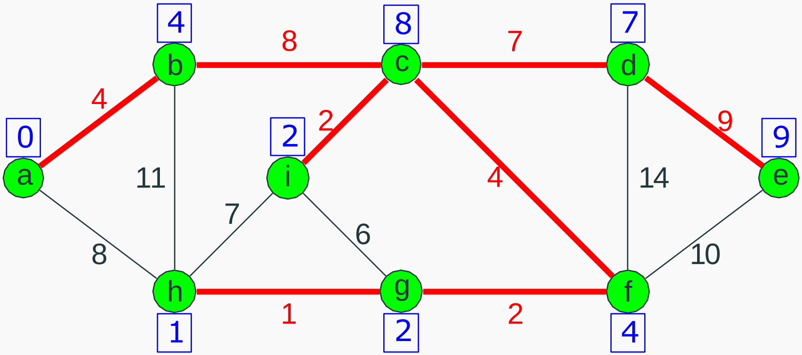
We can update e.

Prim's algorithm: an example



e has the smallest cost.

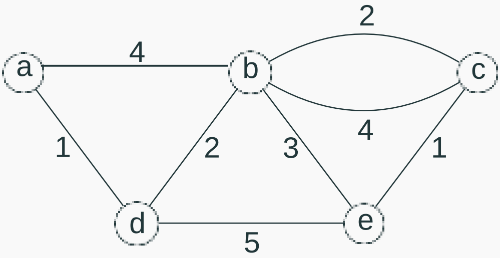
Prim's algorithm: an example



There are no more nodes left, so we're done.

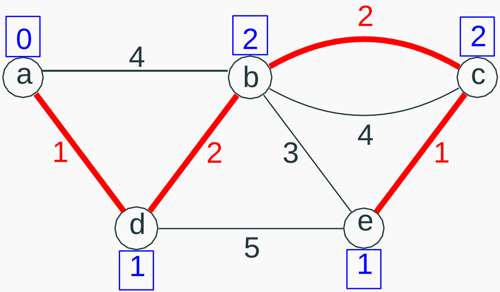
Prim's algorithm: another example

Now you try. Start on node a.



Prim's algorithm: another example

Now you try. Start on node a.



Prim's algorithm: Pseudo Code

```
ReachSet = {0};                  // start any node
UnReachSet = {1, 2, ..., N-1};
SpanningTree = {};

while ( UnReachSet ≠ empty )
{
  Find edge e = (x, y) such that:
     x ∈ ReachSet
     y ∈ UnReachSet
     e has smallest cost

  SpanningTree = SpanningTree ∪ {e};

  ReachSet   = ReachSet ∪ {y};
  UnReachSet = UnReachSet - {y};
}
```

Minimum spanning trees, approach 2

Recap: Prim's algorithm works similarly to Dijkstra's – we start with a single node, and "grow" our MST.

Minimum spanning trees, approach 2

Recap: Prim's algorithm works similarly to Dijkstra's – we start with a single node, and "grow" our MST.

A second approach: instead of "growing" our MST, we...

- Initially place each node into its own MST of size 1 – so, we start with $|V|$ MSTs in total.

Minimum spanning trees, approach 2

Recap: Prim's algorithm works similarly to Dijkstra's – we start with a single node, and "grow" our MST.

A second approach: instead of "growing" our MST, we...

- Initially place each node into its own MST of size 1 – so, we start with $|V|$ MSTs in total.
- Steadily combine together different MSTs until we have just one left

Minimum spanning trees, approach 2

Recap: Prim's algorithm works similarly to Dijkstra's – we start with a single node, and "grow" our MST.

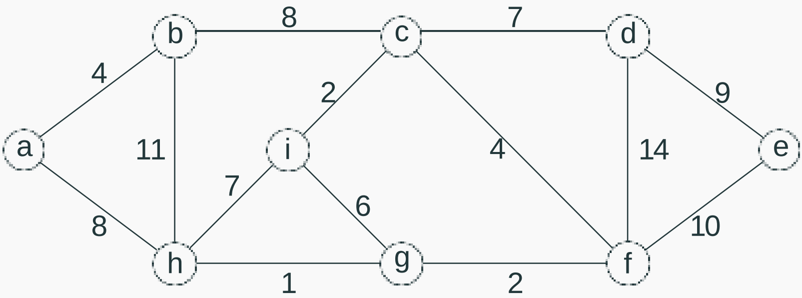
A second approach: instead of "growing" our MST, we...

- Initially place each node into its own MST of size 1 – so, we start with $|V|$ MSTs in total.
- Steadily combine together different MSTs until we have just one left
- How? Loop through every single edge, see if we can use it to join two different MSTs together.

This algorithm is called **Kruskal's algorithm**

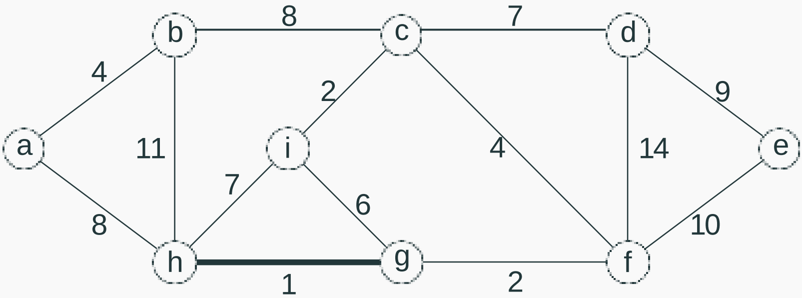
Kruskal’s algorithm: example with a weighted graph

Now you try:



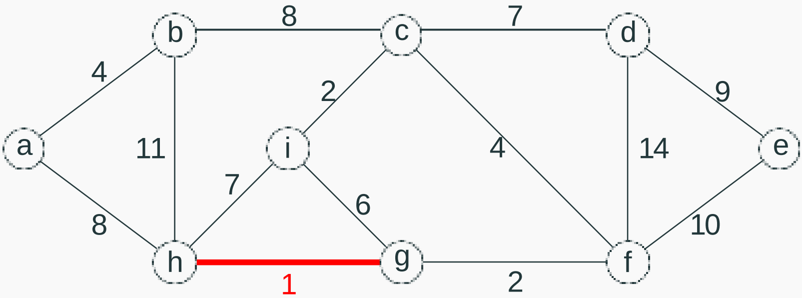
Kruskal's algorithm: example with a weighted graph

Now you try:



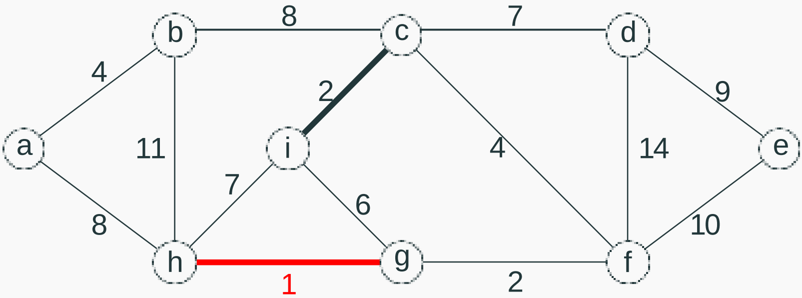
Kruskal's algorithm: example with a weighted graph

Now you try:



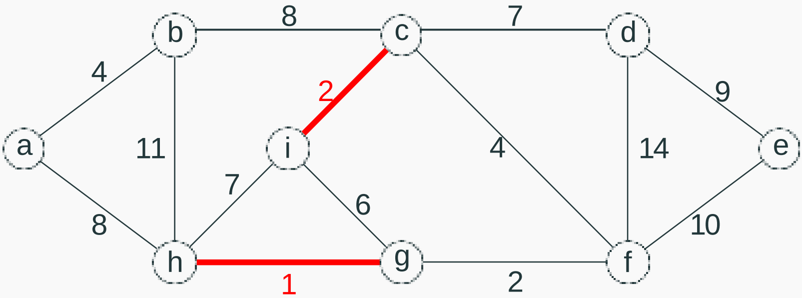
Kruskal's algorithm: example with a weighted graph

Now you try:



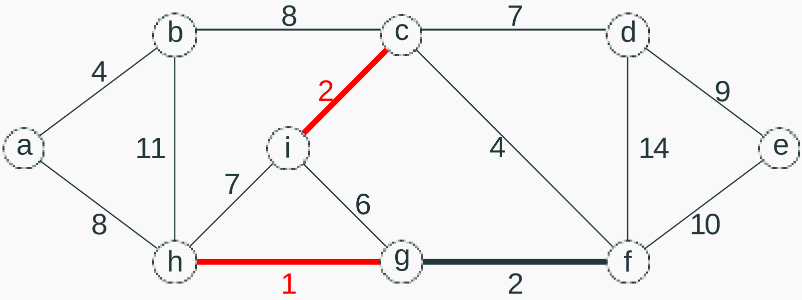
Kruskal's algorithm: example with a weighted graph

Now you try:



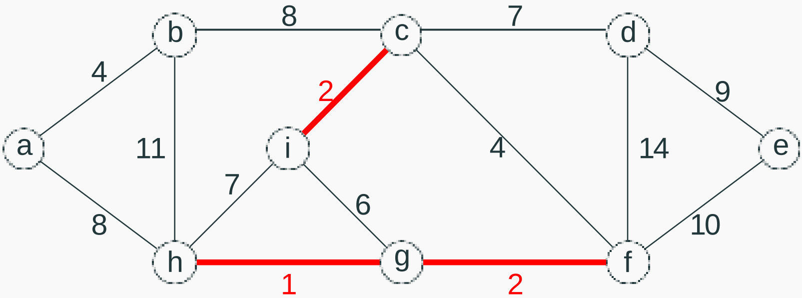
Kruskal's algorithm: example with a weighted graph

Now you try:



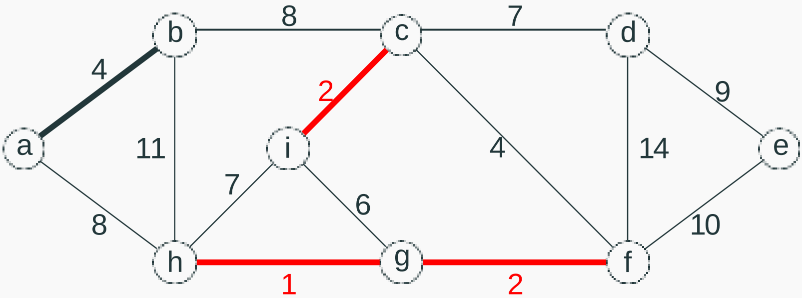
Kruskal's algorithm: example with a weighted graph

Now you try:



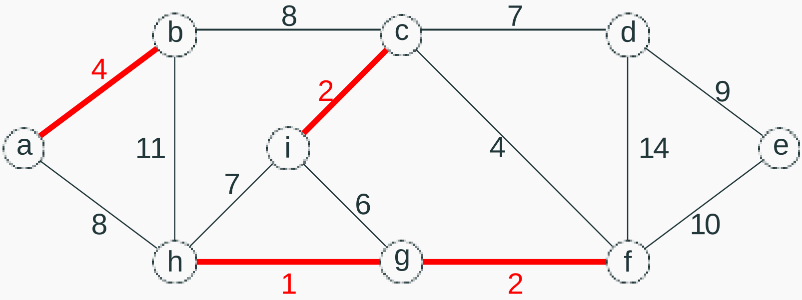
Kruskal's algorithm: example with a weighted graph

Now you try:



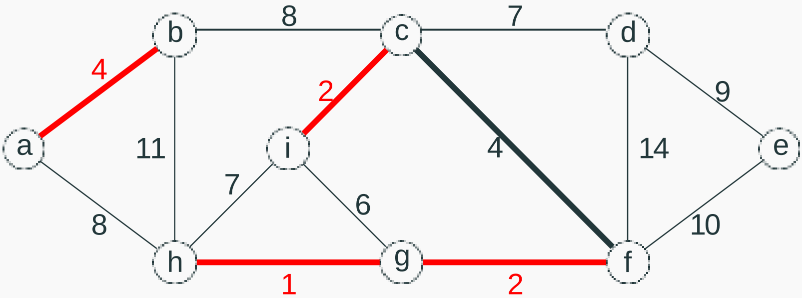
Kruskal’s algorithm: example with a weighted graph

Now you try:



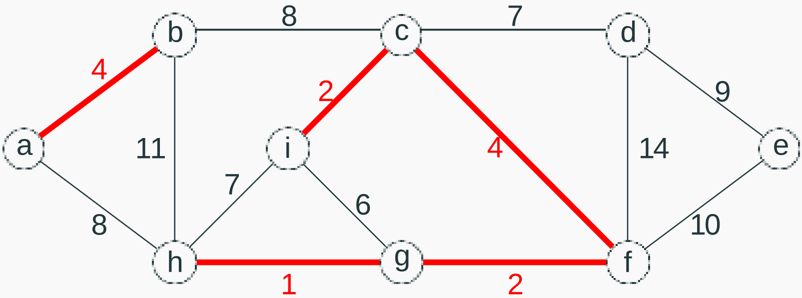
Kruskal's algorithm: example with a weighted graph

Now you try:



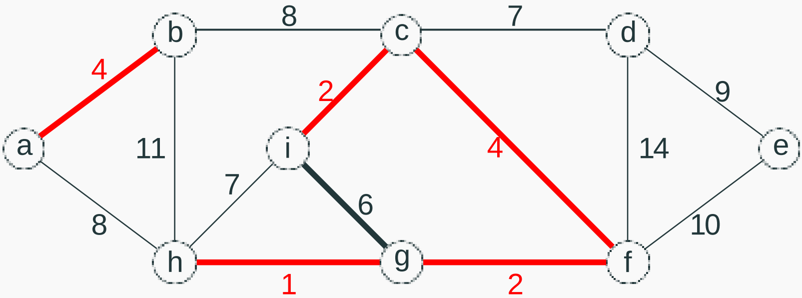
Kruskal’s algorithm: example with a weighted graph

Now you try:



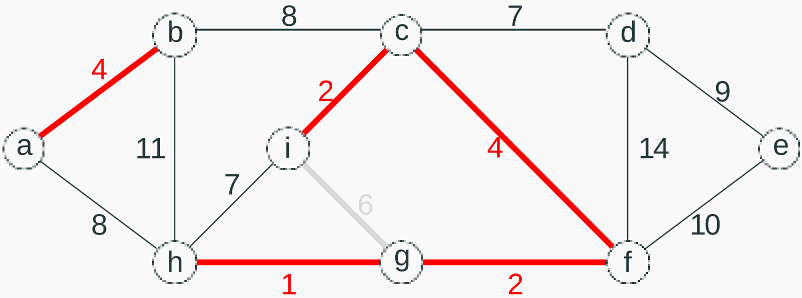
Kruskal's algorithm: example with a weighted graph

Now you try:



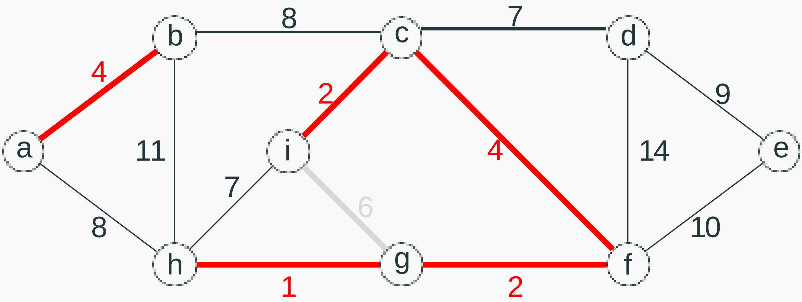
Kruskal's algorithm: example with a weighted graph

Now you try:



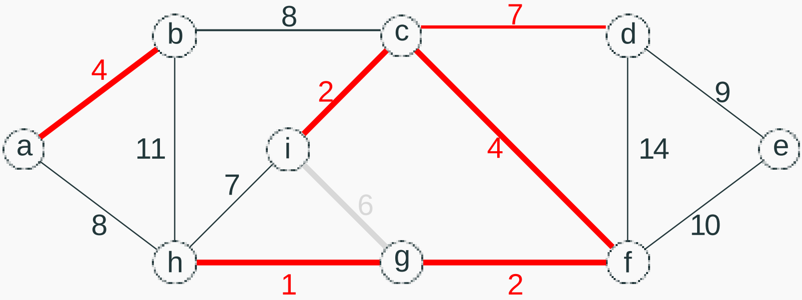
Kruskal's algorithm: example with a weighted graph

Now you try:



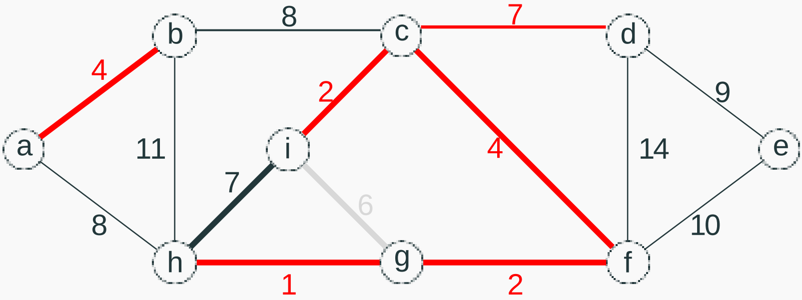
Kruskal's algorithm: example with a weighted graph

Now you try:



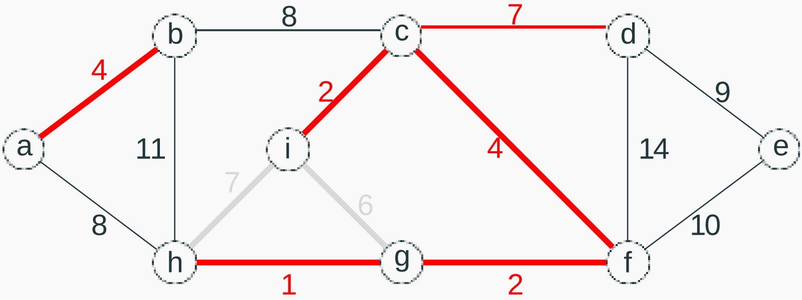
Kruskal's algorithm: example with a weighted graph

Now you try:



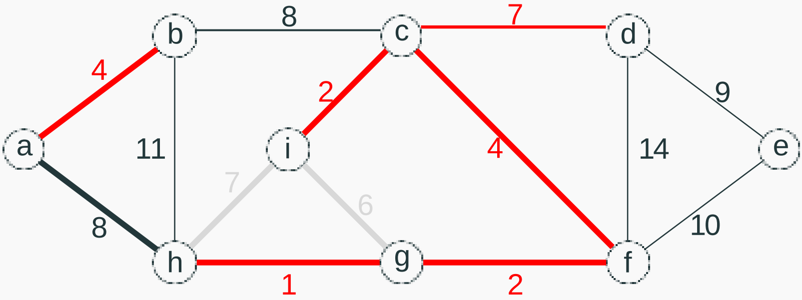
Kruskal's algorithm: example with a weighted graph

Now you try:



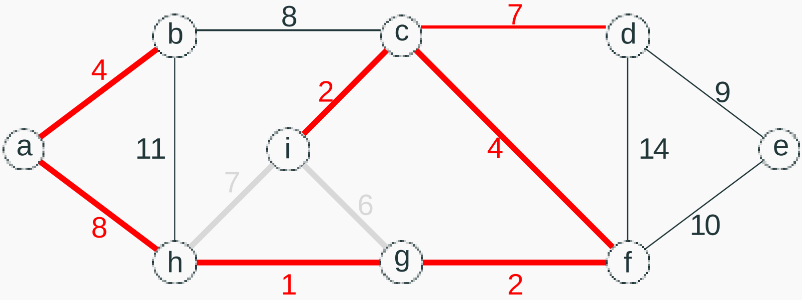
Kruskal's algorithm: example with a weighted graph

Now you try:



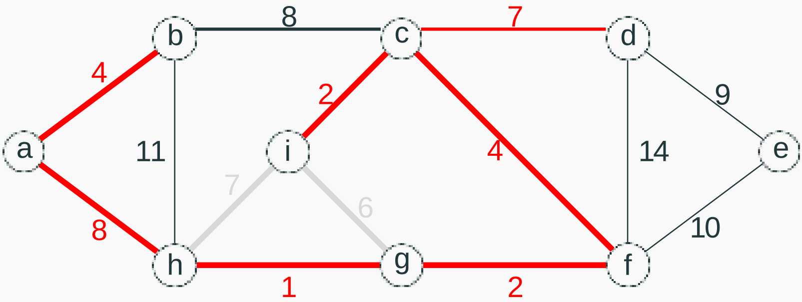
Kruskal's algorithm: example with a weighted graph

Now you try:



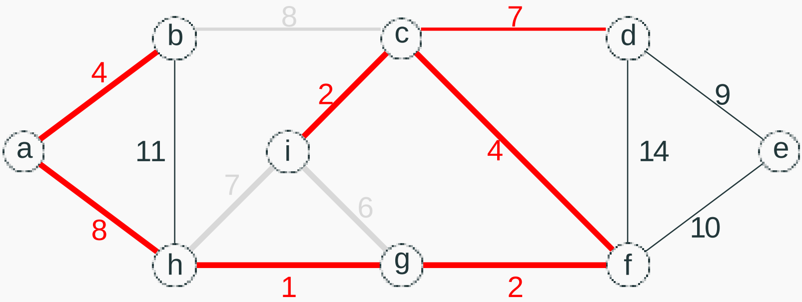
Kruskal's algorithm: example with a weighted graph

Now you try:



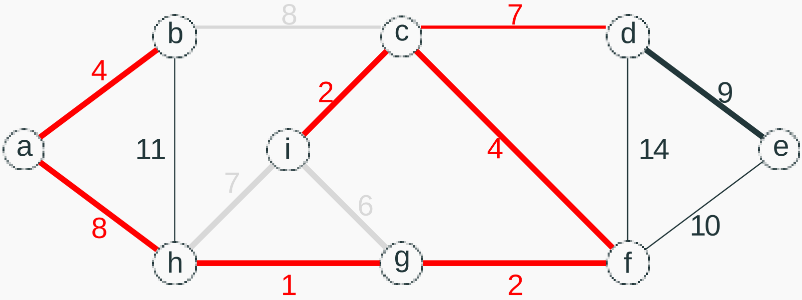
Kruskal's algorithm: example with a weighted graph

Now you try:



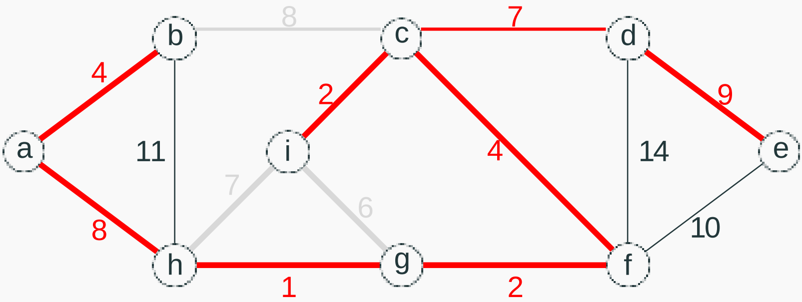
Kruskal's algorithm: example with a weighted graph

Now you try:



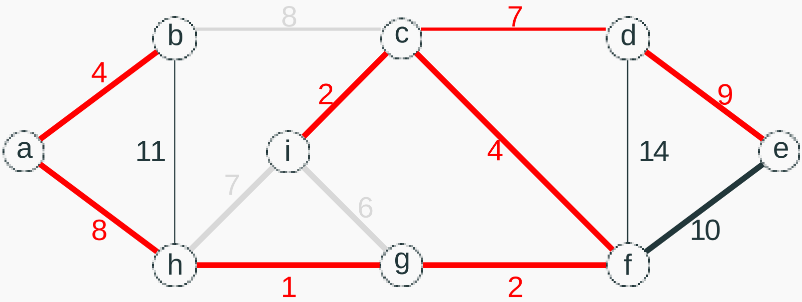
Kruskal's algorithm: example with a weighted graph

Now you try:



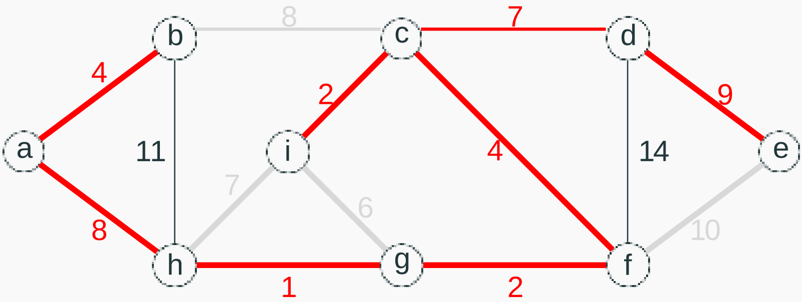
Kruskal’s algorithm: example with a weighted graph

Now you try:



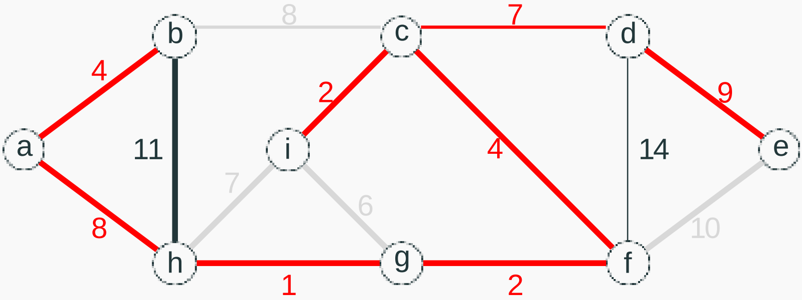
Kruskal's algorithm: example with a weighted graph

Now you try:



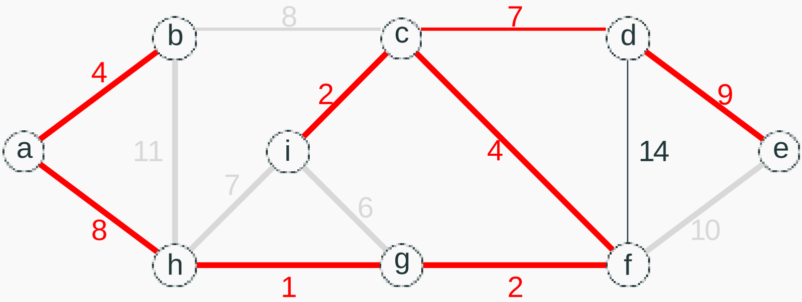
Kruskal's algorithm: example with a weighted graph

Now you try:



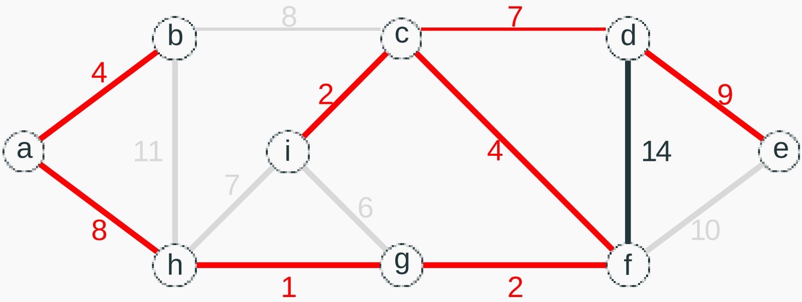
Kruskal's algorithm: example with a weighted graph

Now you try:



Kruskal's algorithm: example with a weighted graph

Now you try:



Kruskal's algorithm: Pseudo Code

```
KRUSKAL(G):
A = ∅
foreach v ∈ G.V:
   MAKE-SET(v)
foreach (u, v) in G.E ordered by weight(u, v), increasing:
   if FIND-SET(u) ≠ FIND-SET(v):
       A = A ∪ {(u, v)}
       UNION(u, v)
return A
```

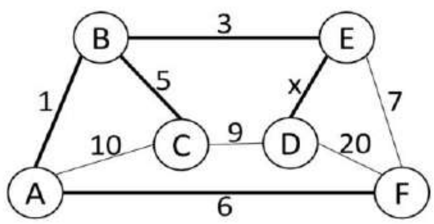
BBM 204 QUIZ 5
13.05.2019

Number:
Name :

1) Answer the following questions with either true or false.

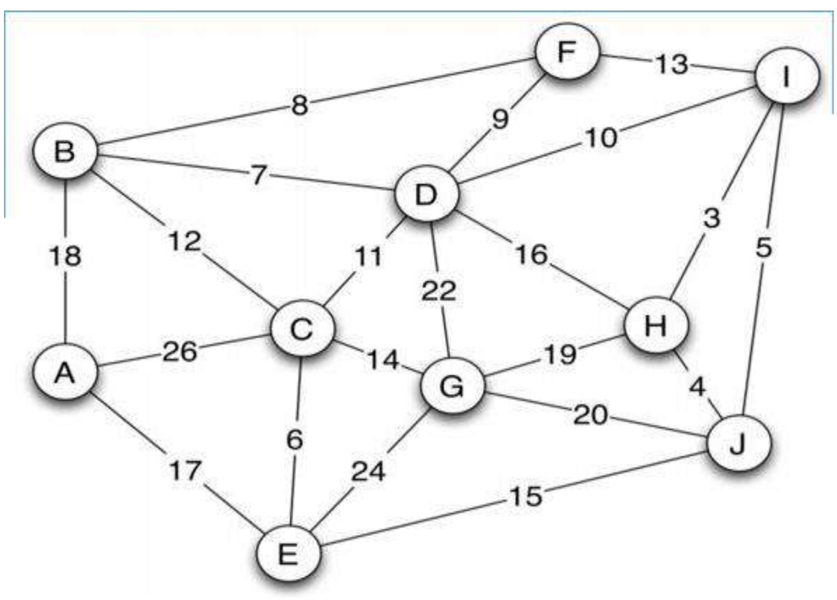
( ) Prim's and Kruskal's algorithms will always return the same Minimum Spanning tree (MST).
( ) Prim's algorithm for computing the MST only work if the weights are positive.
( ) An MST for a connected graph has exactly V-1 edges, V being the number of vertices in the graph.
( ) A graph where every edge weight is unique (there are no two edges with the same weight) has a unique MST.
Solution: false,false,true,true

2) For the following graph the bold edges form a Minimum Spanning Tree. What can you tell about the range of values for x?



Solution : $x \leq 9$

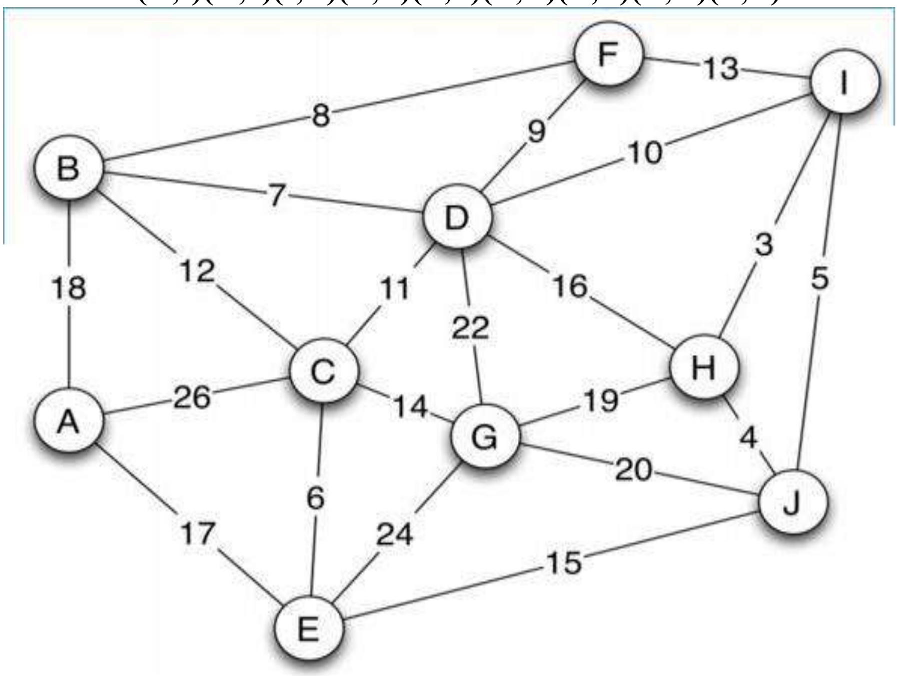
3) Use Kruskal's algorithm to compute the Minimum Spanning Tree (MST) of the following graph. Write down the edges of the MST in the order in which Kruskal's algorithm adds them to the MST. Use the format (node1, node2) to denote an edge.



(H,I)(H,J)(E,C)(B,D)(B,F)(D,I)(C,D)(C,G)(A,E)

4) Use Prim's algorithm starting at node H to compute the Minimum Spanning Tree (MST) of the graph given above. In particular, write down the edges of the MST in the order in which Prim's algorithm adds them to the MST. Use the format (node1, node2) to denote an edge.

Solution: (H,I)(H,J)(I,D)(D,B)(B,F)(D,C)(C,E)(C,G)(E,A)



Hacettepe University
Computer Engineering Department

BBM204 Software Laboratory II

Week IX
Recitation IV

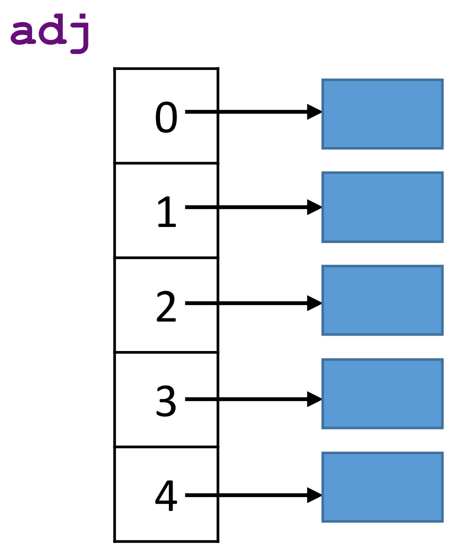
Spring 2018

Q1.

Write a java implementation that finds out if a directed graph ***G*** is <u>cyclic</u>.

```java
public class Graph
    private int V;
    private HashMap<Integer, ArrayList> adj;

    public Graph(int v) {
        V = v;
        adj = new HashMap<>();

        for (int i=0; i<v; ++i)
            adj.put(i,new ArrayList());
        }
```

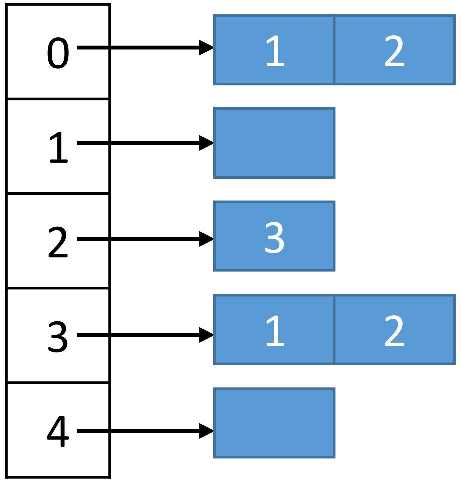
```java
Graph grph = new Graph(5);

grph.addEdge(0,1);
grph.addEdge(0,2);
grph.addEdge(2,3);
grph.addEdge(3,1);
grph.addEdge(3,2);

System.out.println(grph.isCyclic());
```



```
Graph grph = new Graph(5);

grph.addEdge(0,1);
grph.addEdge(0,2);
grph.addEdge(2,3);
grph.addEdge(3,1);
grph.addEdge(3,2);

System.out.println(grph.isCyclic());
```

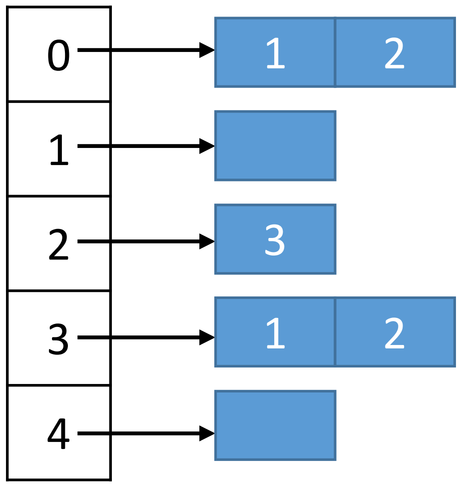
```
public class Graph
    private int V;
    private HashMap<Integer, ArrayList> adj;

    public void addEdge(int v, int w) {
        adj.get(v).add(w);
```

adj



```java
System.out.println(grph.isCyclic());
```

adj

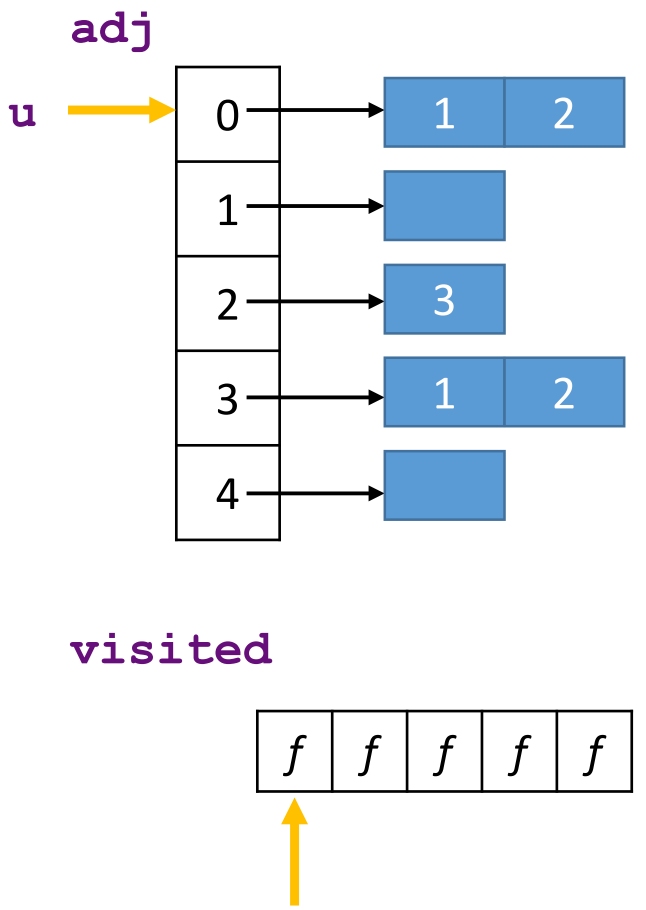


visited

f	*f*	*f*	*f*	*f*

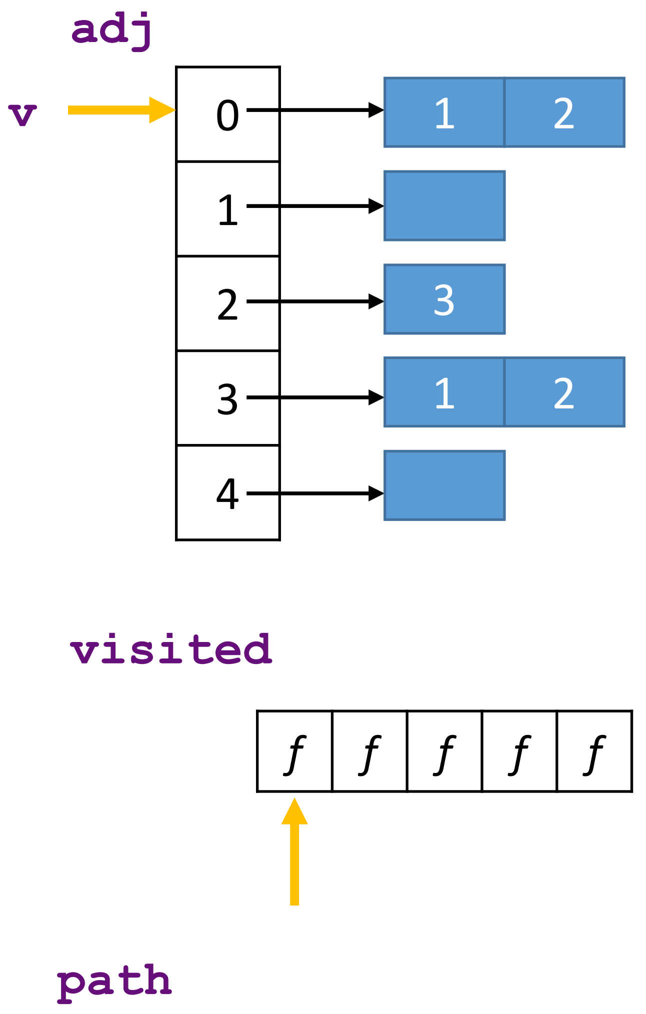
```java
public boolean isCyclic() {
    Boolean visited[] = new Boolean[V];

    for (int i = 0; i < V; i++)
        visited[i] = false;

    for (int u = 0; u < V; u++)
        if (!visited[u])
            if (isCyclic(u, visited , new ArrayList()))
                return true;

    return false;
}


public boolean isCyclic(int v, Boolean visited[], ArrayList path) {
    visited[v] = true;
    path.add(v);
    Integer neighbor;

    Iterator<Integer> it = adj.get(v).iterator();
    while (it.hasNext()) {

        neighbor = it.next();
        if (path.contains(neighbor))
            return true;
        if (!visited[neighbor]) {
            if (isCyclic(neighbor, visited, path))
                return true;
        }
    }
    path.remove(new Integer(v));
    return false;
}
```

```
System.out.println(grph.isCyclic());
```



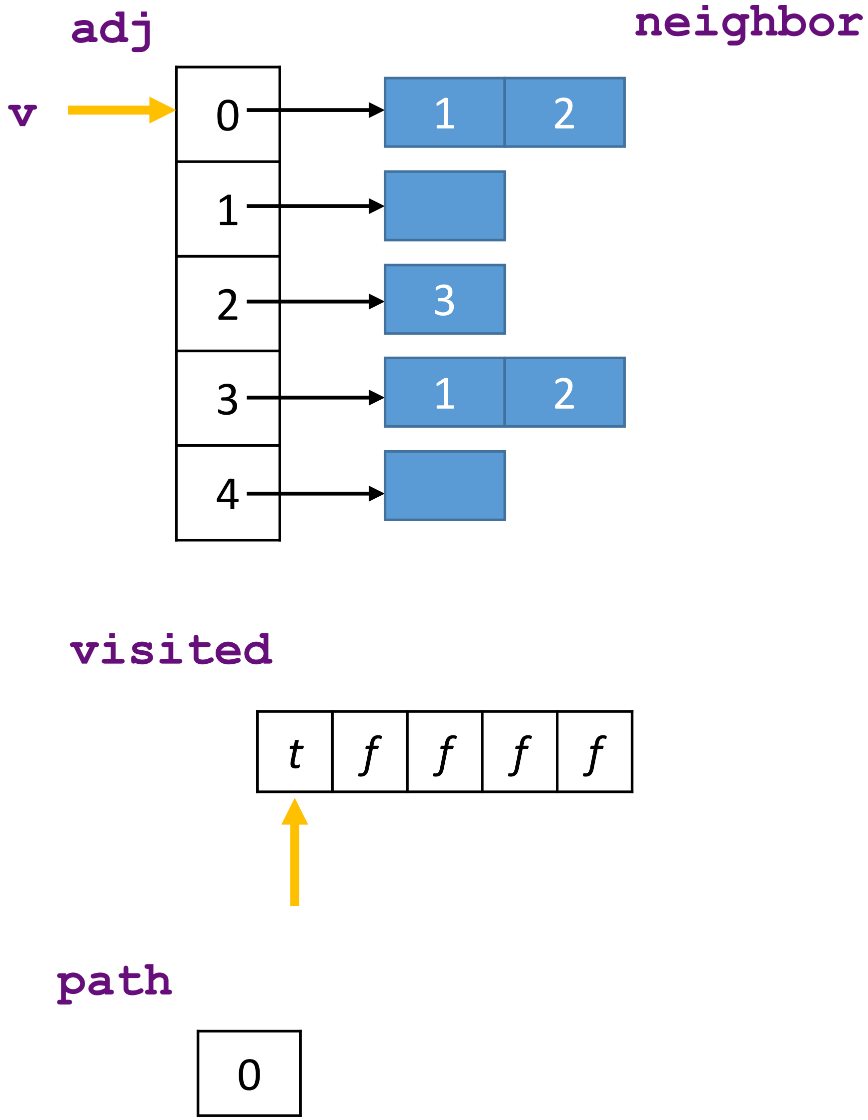

```
public boolean isCyclic() {
    Boolean visited[] = new Boolean[V];

    for (int i = 0; i < V; i++)
        visited[i] = false;

    for (int u = 0; u < V; u++)
        if (!visited[u])
            if (isCyclic(u, visited , new ArrayList()))
                return true;

    return false;
}


public boolean isCyclic(int v, Boolean visited[], ArrayList path) {
    visited[v] = true;
    path.add(v);
    Integer neighbor;

    Iterator<Integer> it = adj.get(v).iterator();
    while (it.hasNext()) {

        neighbor = it.next();
        if (path.contains(neighbor))
            return true;
        if (!visited[neighbor]) {
            if (isCyclic(neighbor, visited, path))
                return true;
        }
    }
    path.remove(new Integer(v));
    return false;
}
```

```
System.out.println(grph.isCyclic());
```



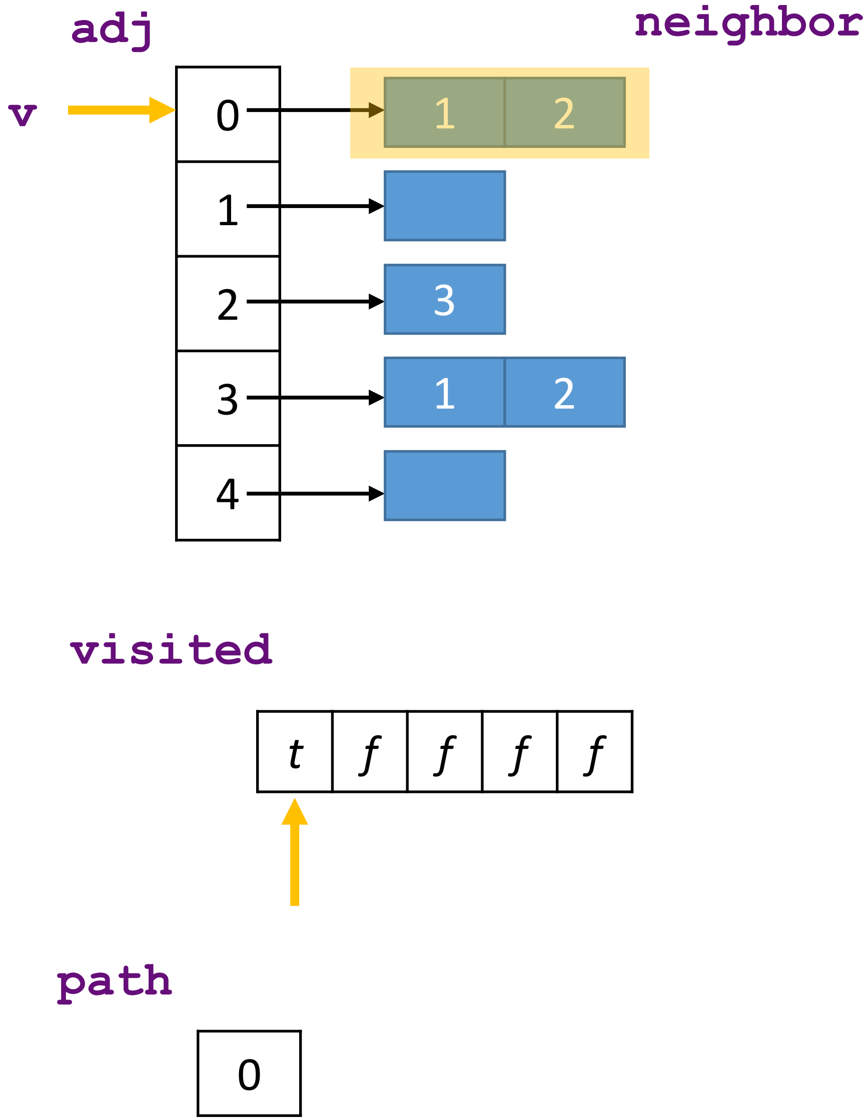

```
public boolean isCyclic() {
    Boolean visited[] = new Boolean[V];

    for (int i = 0; i < V; i++)
        visited[i] = false;

    for (int u = 0; u < V; u++)
        if (!visited[u])
            if (isCyclic(u, visited , new ArrayList()))
                return true;

    return false;
}


public boolean isCyclic(int v, Boolean visited[], ArrayList path) {
    visited[v] = true;
    path.add(v);
    Integer neighbor;

    Iterator<Integer> it = adj.get(v).iterator();
    while (it.hasNext()) {

        neighbor = it.next();
        if (path.contains(neighbor))
            return true;
        if (!visited[neighbor]) {
            if (isCyclic(neighbor, visited, path))
                return true;
        }
    }
    path.remove(new Integer(v));
    return false;
}
```

```
System.out.println(grph.isCyclic());
```



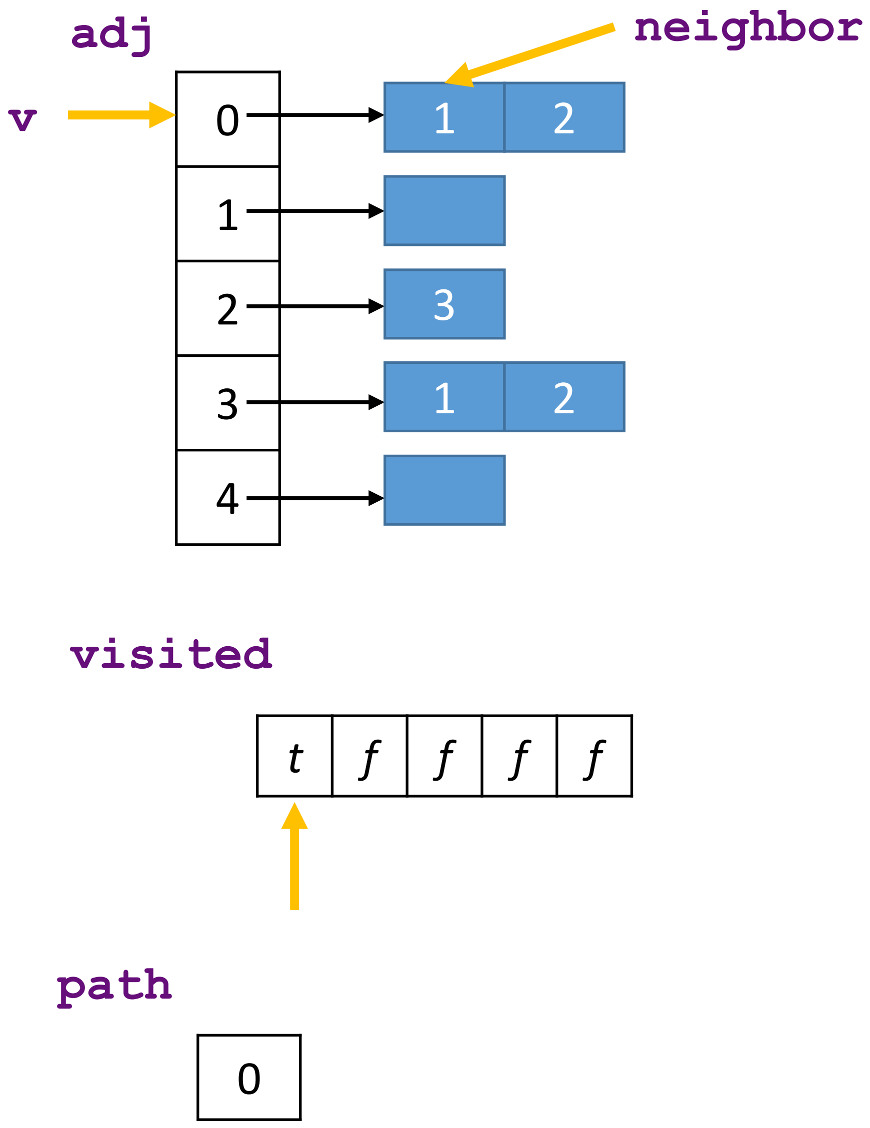

```
public boolean isCyclic() {
    Boolean visited[] = new Boolean[V];

    for (int i = 0; i < V; i++)
        visited[i] = false;

    for (int u = 0; u < V; u++)
        if (!visited[u])
            if (isCyclic(u, visited , new ArrayList()))
                return true;

    return false;
}


public boolean isCyclic(int v, Boolean visited[], ArrayList path) {
    visited[v] = true;
    path.add(v);
    Integer neighbor;

    Iterator<Integer> it = adj.get(v).iterator();
    while (it.hasNext()) {

        neighbor = it.next();
        if (path.contains(neighbor))
            return true;
        if (!visited[neighbor]) {
            if (isCyclic(neighbor, visited, path))
                return true;
        }
    }
    path.remove(new Integer(v));
    return false;
}
```

```
System.out.println(grph.isCyclic());
```



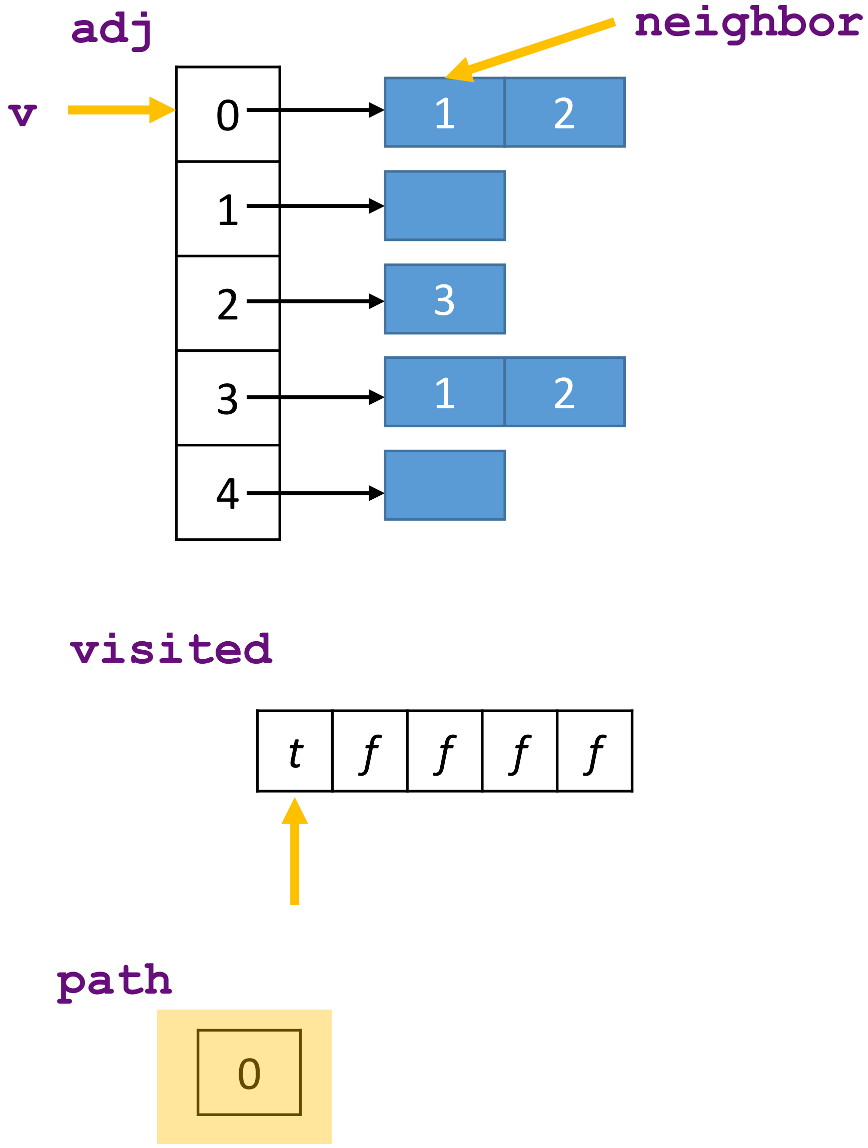

```
public boolean isCyclic() {
    Boolean visited[] = new Boolean[V];

    for (int i = 0; i < V; i++)
        visited[i] = false;

    for (int u = 0; u < V; u++)
        if (!visited[u])
            if (isCyclic(u, visited , new ArrayList()))
                return true;

    return false;
}


public boolean isCyclic(int v, Boolean visited[], ArrayList path) {
    visited[v] = true;
    path.add(v);
    Integer neighbor;

    Iterator<Integer> it = adj.get(v).iterator();
    while (it.hasNext()) {

        neighbor = it.next();
        if (path.contains(neighbor))
            return true;
        if (!visited[neighbor]) {
            if (isCyclic(neighbor, visited, path))
                return true;
        }
    }
    path.remove(new Integer(v));
    return false;
}
```

```
System.out.println(grph.isCyclic());
```



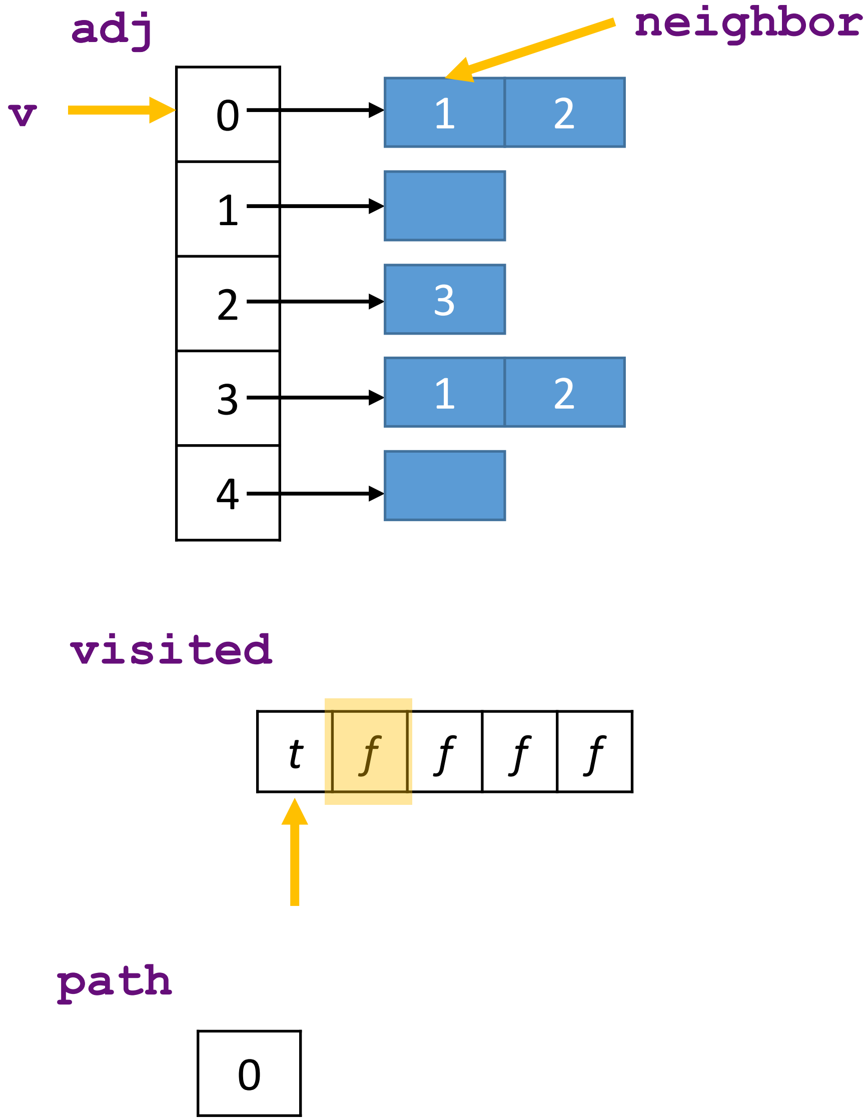

```
public boolean isCyclic() {
    Boolean visited[] = new Boolean[V];

    for (int i = 0; i < V; i++)
        visited[i] = false;

    for (int u = 0; u < V; u++)
        if (!visited[u])
            if (isCyclic(u, visited , new ArrayList()))
                return true;

    return false;
}


public boolean isCyclic(int v, Boolean visited[], ArrayList path) {
    visited[v] = true;
    path.add(v);
    Integer neighbor;

    Iterator<Integer> it = adj.get(v).iterator();
    while (it.hasNext()) {

        neighbor = it.next();
        if (path.contains(neighbor))
            return true;
        if (!visited[neighbor]) {
            if (isCyclic(neighbor, visited, path))
                return true;
        }
    }
    path.remove(new Integer(v));
    return false;
}
```

```
System.out.println(grph.isCyclic());
```



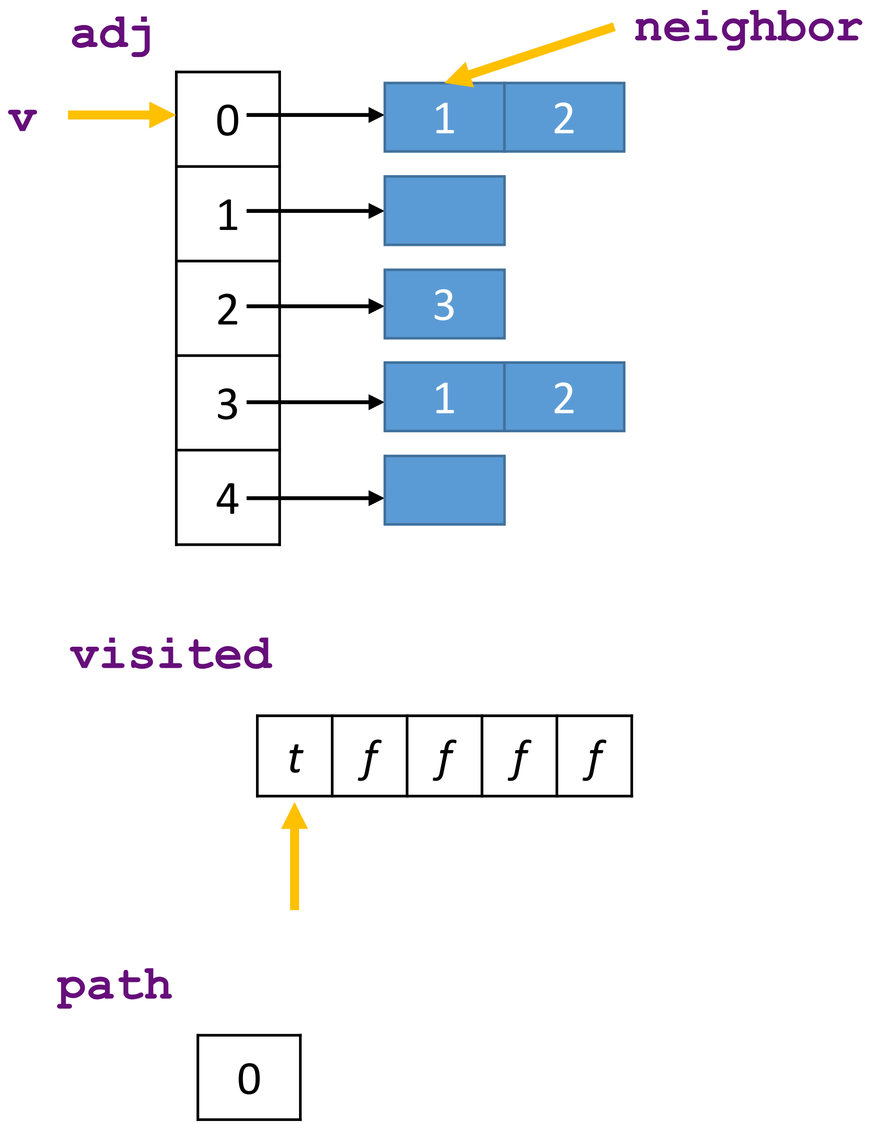

```
public boolean isCyclic() {
    Boolean visited[] = new Boolean[V];

    for (int i = 0; i < V; i++)
        visited[i] = false;

    for (int u = 0; u < V; u++)
        if (!visited[u])
            if (isCyclic(u, visited , new ArrayList()))
                return true;

    return false;
}


public boolean isCyclic(int v, Boolean visited[], ArrayList path) {
    visited[v] = true;
    path.add(v);
    Integer neighbor;

    Iterator<Integer> it = adj.get(v).iterator();
    while (it.hasNext()) {

        neighbor = it.next();
        if (path.contains(neighbor))
            return true;
        if (!visited[neighbor]) {
            if (isCyclic(neighbor, visited, path))
                return true;
        }
    }
    path.remove(new Integer(v));
    return false;
}
```

```
System.out.println(grph.isCyclic());
```



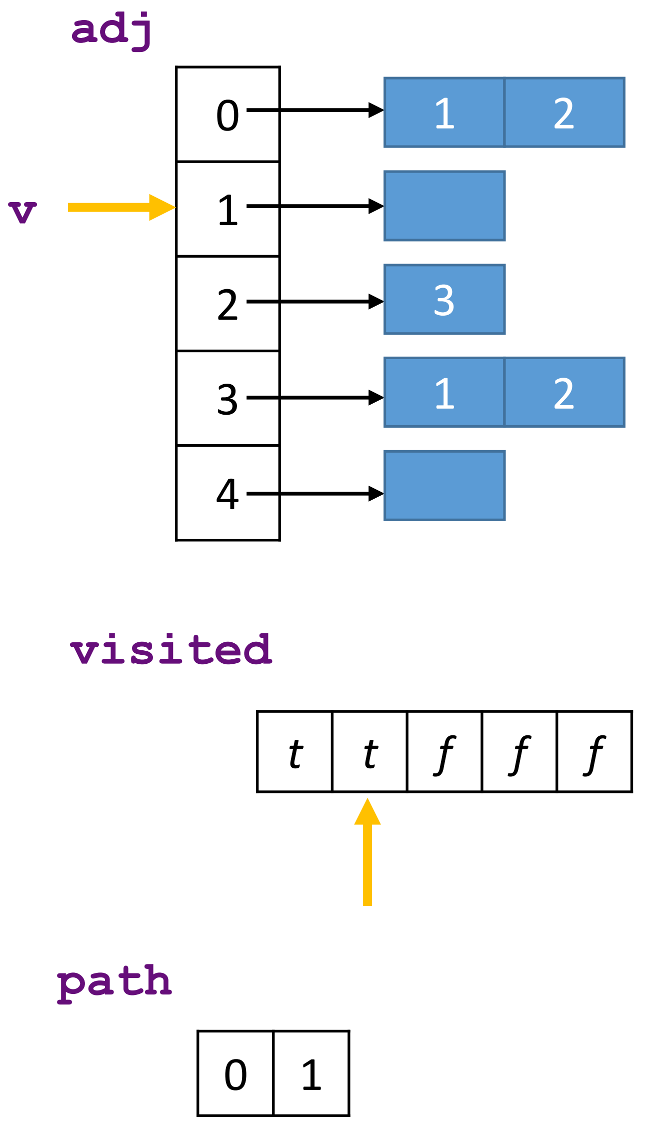

```
public boolean isCyclic() {
    Boolean visited[] = new Boolean[V];

    for (int i = 0; i < V; i++)
        visited[i] = false;

    for (int u = 0; u < V; u++)
        if (!visited[u])
            if (isCyclic(u, visited , new ArrayList()))
                return true;

    return false;
}


public boolean isCyclic(int v, Boolean visited[], ArrayList path) {
    visited[v] = true;
    path.add(v);
    Integer neighbor;

    Iterator<Integer> it = adj.get(v).iterator();
    while (it.hasNext()) {

        neighbor = it.next();
        if (path.contains(neighbor))
            return true;
        if (!visited[neighbor]) {
            if (isCyclic(neighbor, visited, path))
                return true;
        }
    }
    path.remove(new Integer(v));
    return false;
}
```

```
System.out.println(grph.isCyclic());
```



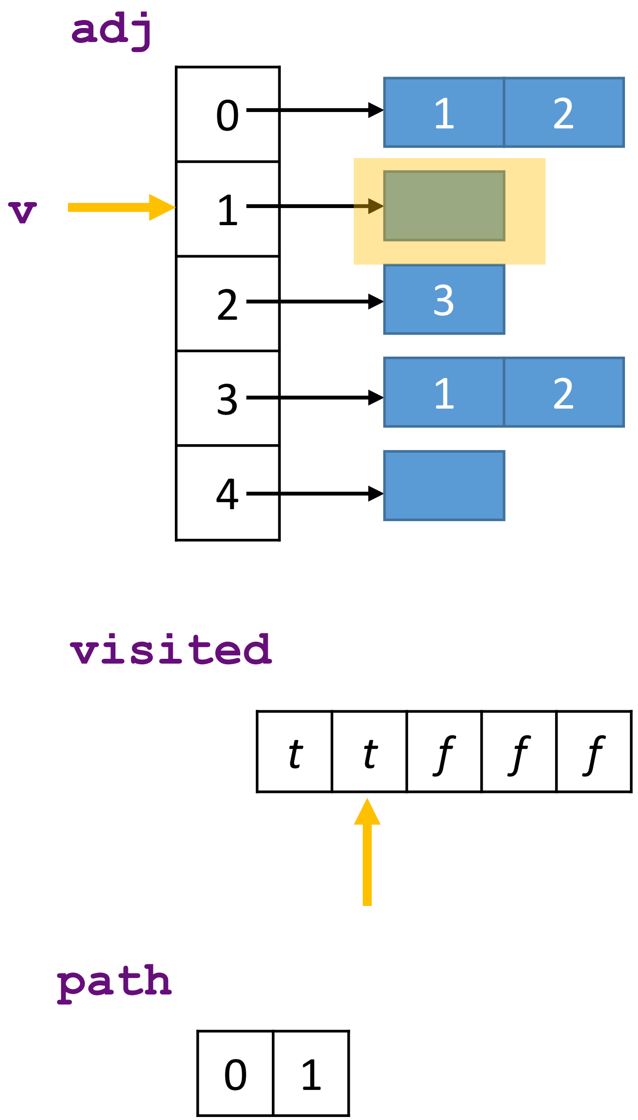

```
public boolean isCyclic() {
    Boolean visited[] = new Boolean[V];

    for (int i = 0; i < V; i++)
        visited[i] = false;

    for (int u = 0; u < V; u++)
        if (!visited[u])
            if (isCyclic(u, visited , new ArrayList()))
                return true;

    return false;
}


public boolean isCyclic(int v, Boolean visited[], ArrayList path) {
    visited[v] = true;
    path.add(v);
    Integer neighbor;

    Iterator<Integer> it = adj.get(v).iterator();
    while (it.hasNext()) {

        neighbor = it.next();
        if (path.contains(neighbor))
            return true;
        if (!visited[neighbor]) {
            if (isCyclic(neighbor, visited, path))
                return true;
        }
    }
    path.remove(new Integer(v));
    return false;
}
```

```java
System.out.println(grph.isCyclic());
```



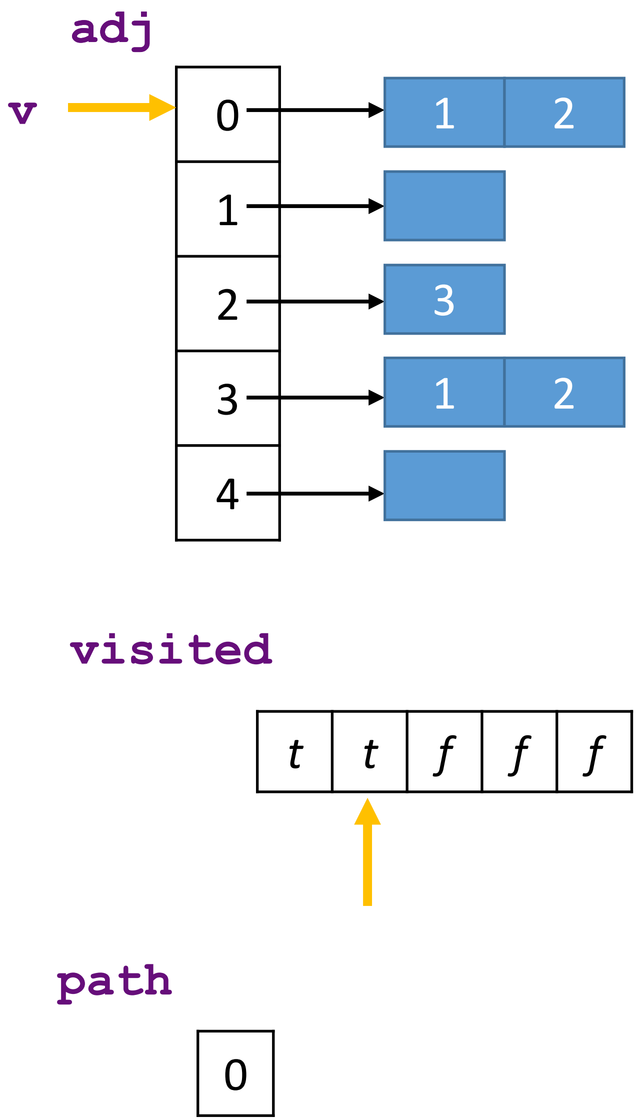

```java
public boolean isCyclic() {
    Boolean visited[] = new Boolean[V];

    for (int i = 0; i < V; i++)
        visited[i] = false;

    for (int u = 0; u < V; u++)
        if (!visited[u])
            if (isCyclic(u, visited , new ArrayList()))
                return true;

    return false;
}


public boolean isCyclic(int v, Boolean visited[], ArrayList path) {
    visited[v] = true;
    path.add(v);
    Integer neighbor;

    Iterator<Integer> it = adj.get(v).iterator();
    while (it.hasNext()) {

        neighbor = it.next();
        if (path.contains(neighbor))
            return true;
        if (!visited[neighbor]) {
            if (isCyclic(neighbor, visited, path))
                return true;
        }
    }
    path.remove(new Integer(v));
    return false;
}
```

```
System.out.println(grph.isCyclic());
```



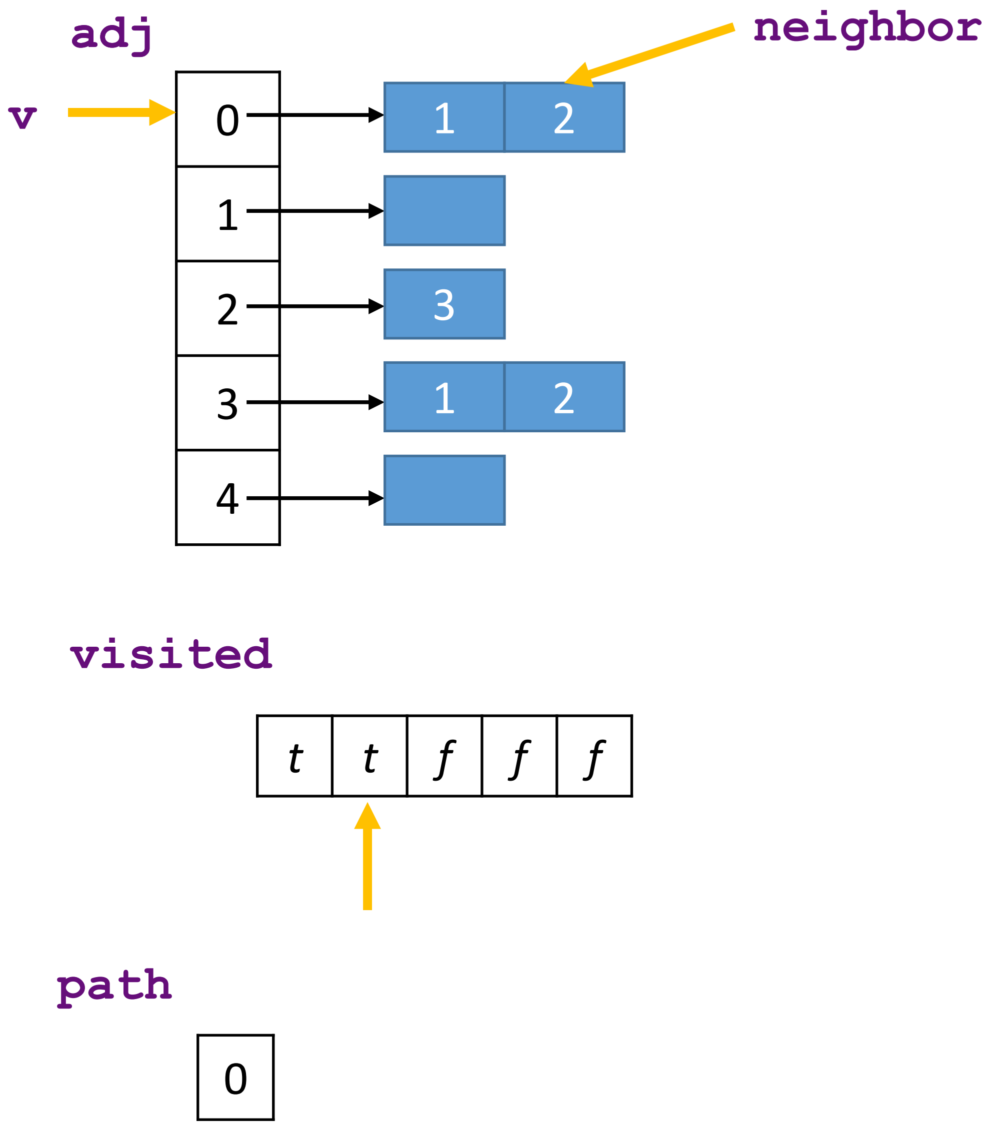

```
public boolean isCyclic() {
    Boolean visited[] = new Boolean[V];

    for (int i = 0; i < V; i++)
        visited[i] = false;

    for (int u = 0; u < V; u++)
        if (!visited[u])
            if (isCyclic(u, visited , new ArrayList()))
                return true;

    return false;
}


public boolean isCyclic(int v, Boolean visited[], ArrayList path) {
    visited[v] = true;
    path.add(v);
    Integer neighbor;

    Iterator<Integer> it = adj.get(v).iterator();
    while (it.hasNext()) {

        neighbor = it.next();
        if (path.contains(neighbor))
            return true;
        if (!visited[neighbor]) {
            if (isCyclic(neighbor, visited, path))
                return true;
        }
    }
    path.remove(new Integer(v));
    return false;
}
```

```java
System.out.println(grph.isCyclic());
```



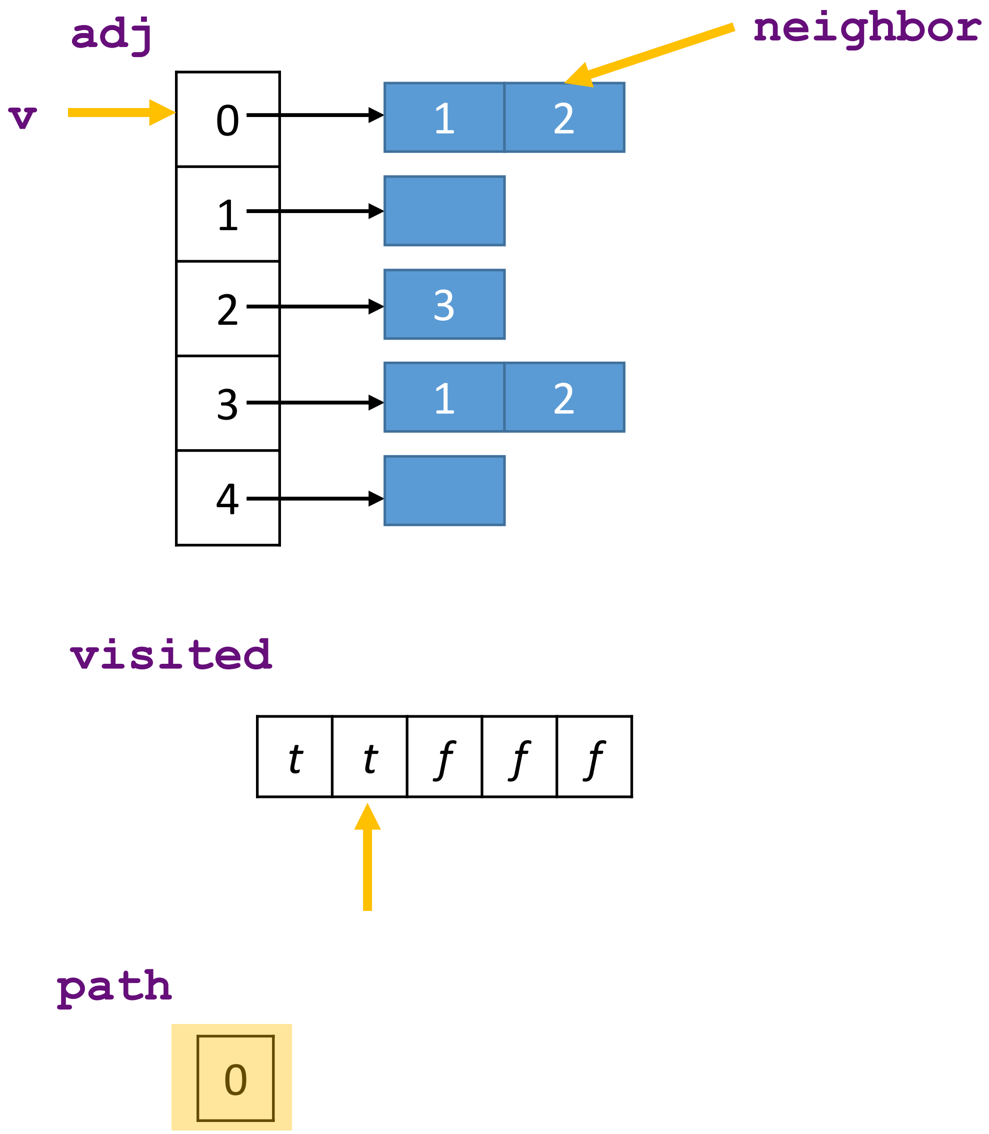

```java
public boolean isCyclic() {
    Boolean visited[] = new Boolean[V];

    for (int i = 0; i < V; i++)
        visited[i] = false;

    for (int u = 0; u < V; u++)
        if (!visited[u])
            if (isCyclic(u, visited , new ArrayList()))
                return true;

    return false;
}


public boolean isCyclic(int v, Boolean visited[], ArrayList path) {
    visited[v] = true;
    path.add(v);
    Integer neighbor;

    Iterator<Integer> it = adj.get(v).iterator();
    while (it.hasNext()) {

        neighbor = it.next();
        if (path.contains(neighbor))
            return true;
        if (!visited[neighbor]) {
            if (isCyclic(neighbor, visited, path))
                return true;
        }
    }
    path.remove(new Integer(v));
    return false;
}
```

```
System.out.println(grph.isCyclic());
```



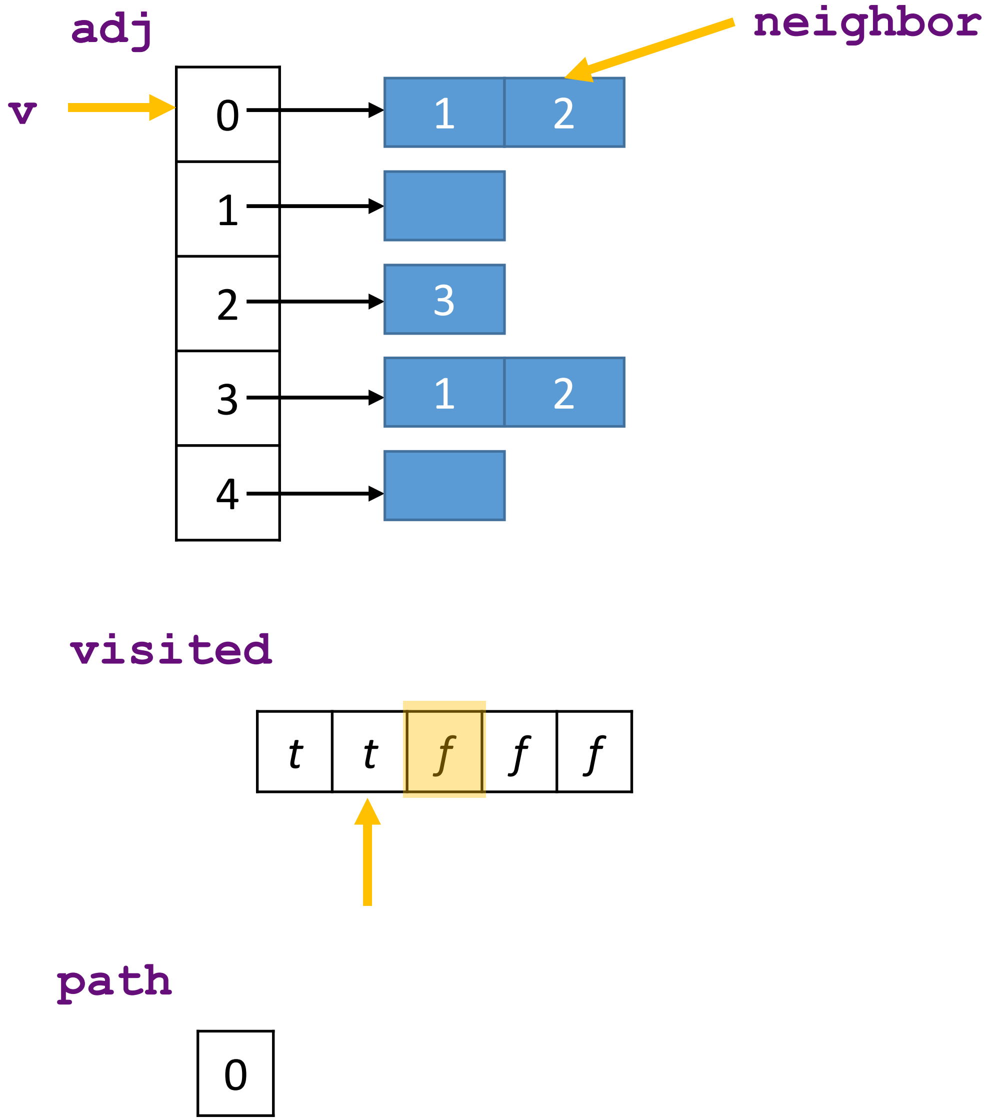

```
public boolean isCyclic() {
    Boolean visited[] = new Boolean[V];

    for (int i = 0; i < V; i++)
        visited[i] = false;

    for (int u = 0; u < V; u++)
        if (!visited[u])
            if (isCyclic(u, visited , new ArrayList()))
                return true;

    return false;
}


public boolean isCyclic(int v, Boolean visited[], ArrayList path) {
    visited[v] = true;
    path.add(v);
    Integer neighbor;

    Iterator<Integer> it = adj.get(v).iterator();
    while (it.hasNext()) {

        neighbor = it.next();
        if (path.contains(neighbor))
            return true;
        if (!visited[neighbor]) {
            if (isCyclic(neighbor, visited, path))
                return true;
        }
    }
    path.remove(new Integer(v));
    return false;
}
```

```java
System.out.println(grph.isCyclic());
```



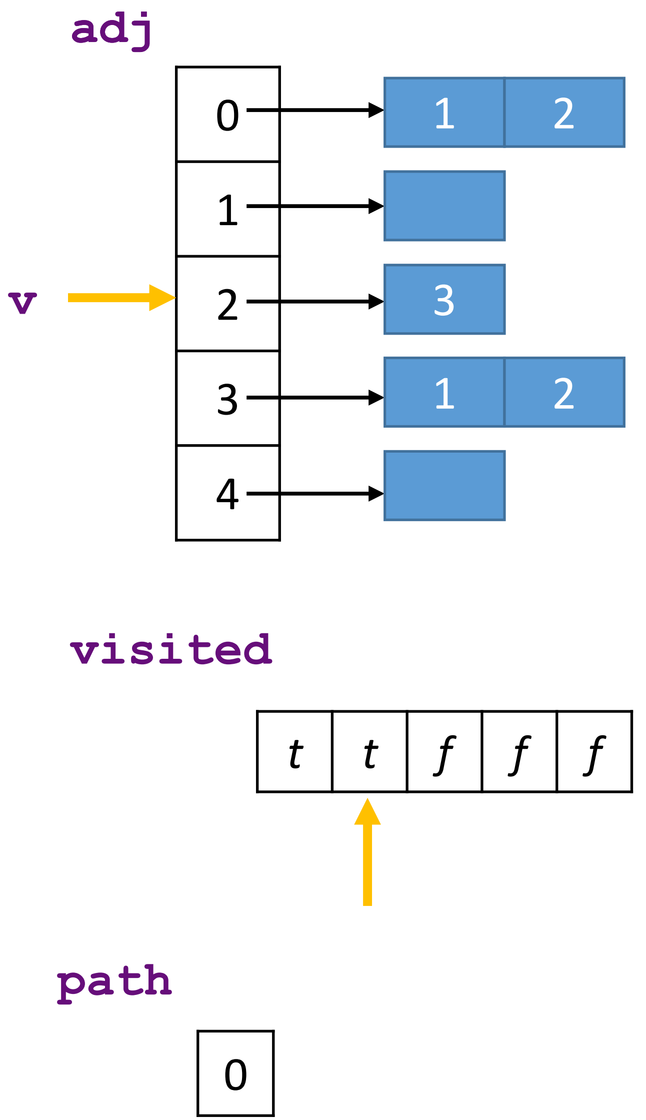

```java
public boolean isCyclic() {
    Boolean visited[] = new Boolean[V];

    for (int i = 0; i < V; i++)
        visited[i] = false;

    for (int u = 0; u < V; u++)
        if (!visited[u])
            if (isCyclic(u, visited , new ArrayList()))
                return true;

    return false;
}


public boolean isCyclic(int v, Boolean visited[], ArrayList path) {
    visited[v] = true;
    path.add(v);
    Integer neighbor;

    Iterator<Integer> it = adj.get(v).iterator();
    while (it.hasNext()) {

        neighbor = it.next();
        if (path.contains(neighbor))
            return true;
        if (!visited[neighbor]) {
            if (isCyclic(neighbor, visited, path))
                return true;
        }
    }
    path.remove(new Integer(v));
    return false;
}
```

```java
System.out.println(grph.isCyclic());
```



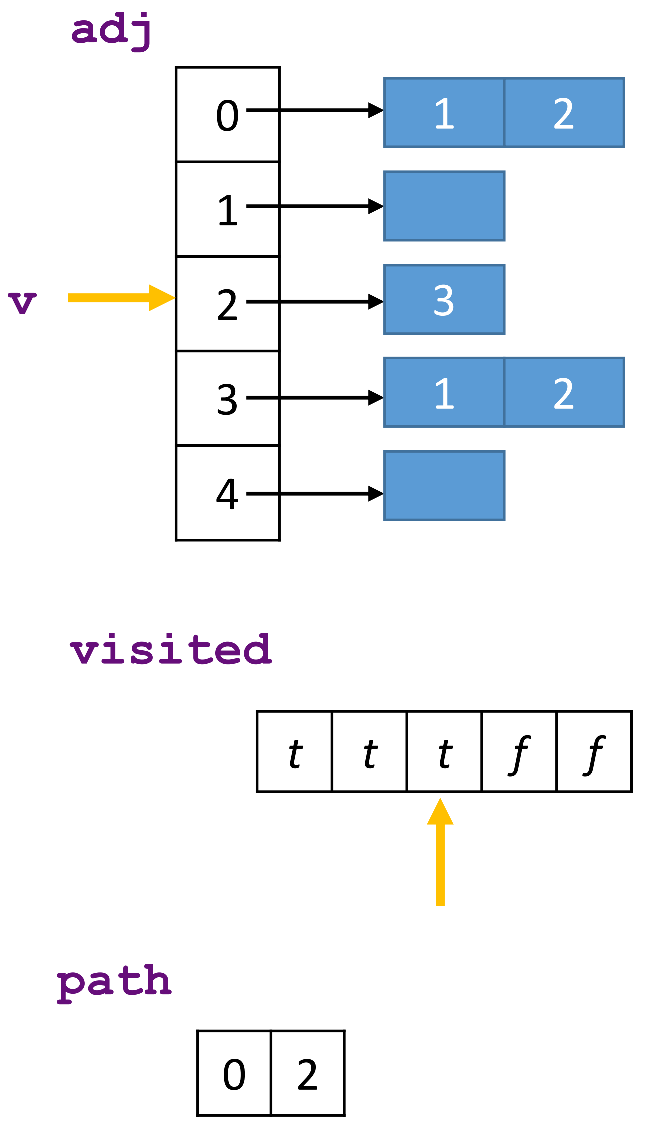

```java
public boolean isCyclic() {
    Boolean visited[] = new Boolean[V];

    for (int i = 0; i < V; i++)
        visited[i] = false;

    for (int u = 0; u < V; u++)
        if (!visited[u])
            if (isCyclic(u, visited , new ArrayList()))
                return true;

    return false;
}


public boolean isCyclic(int v, Boolean visited[], ArrayList path) {
    visited[v] = true;
    path.add(v);
    Integer neighbor;

    Iterator<Integer> it = adj.get(v).iterator();
    while (it.hasNext()) {

        neighbor = it.next();
        if (path.contains(neighbor))
            return true;
        if (!visited[neighbor]) {
            if (isCyclic(neighbor, visited, path))
                return true;
        }
    }
    path.remove(new Integer(v));
    return false;
}
```

```java
System.out.println(grph.isCyclic());
```



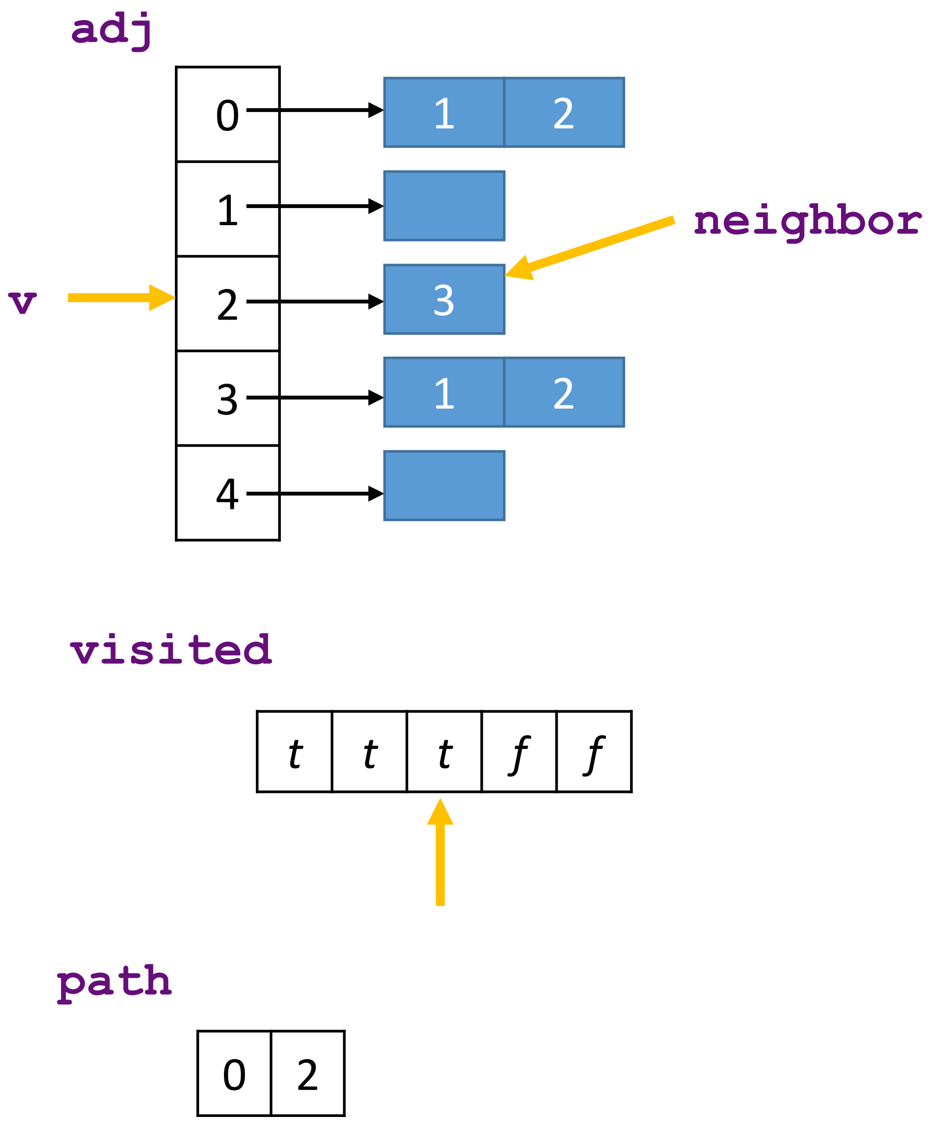

```java
public boolean isCyclic() {
    Boolean visited[] = new Boolean[V];

    for (int i = 0; i < V; i++)
        visited[i] = false;

    for (int u = 0; u < V; u++)
        if (!visited[u])
            if (isCyclic(u, visited , new ArrayList()))
                return true;

    return false;
}


public boolean isCyclic(int v, Boolean visited[], ArrayList path) {
    visited[v] = true;
    path.add(v);
    Integer neighbor;

    Iterator<Integer> it = adj.get(v).iterator();
    while (it.hasNext()) {

        neighbor = it.next();
        if (path.contains(neighbor))
            return true;
        if (!visited[neighbor]) {
            if (isCyclic(neighbor, visited, path))
                return true;
        }
    }
    path.remove(new Integer(v));
    return false;
}
```

```java
System.out.println(grph.isCyclic());
```



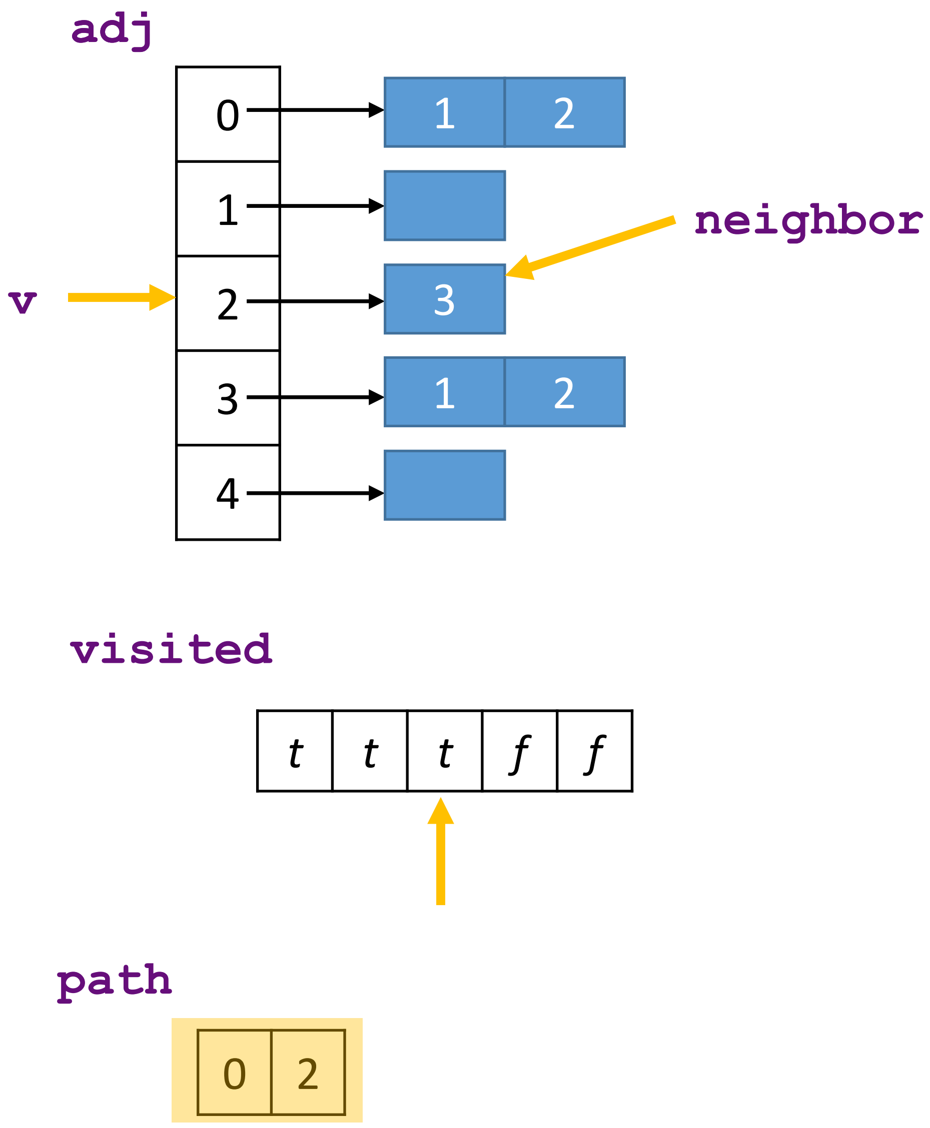

```java
public boolean isCyclic() {
    Boolean visited[] = new Boolean[V];

    for (int i = 0; i < V; i++)
        visited[i] = false;

    for (int u = 0; u < V; u++)
        if (!visited[u])
            if (isCyclic(u, visited , new ArrayList()))
                return true;

    return false;
}


public boolean isCyclic(int v, Boolean visited[], ArrayList path) {
    visited[v] = true;
    path.add(v);
    Integer neighbor;

    Iterator<Integer> it = adj.get(v).iterator();
    while (it.hasNext()) {

        neighbor = it.next();
        if (path.contains(neighbor))
            return true;
        if (!visited[neighbor]) {
            if (isCyclic(neighbor, visited, path))
                return true;
        }
    }
    path.remove(new Integer(v));
    return false;
}
```

```
System.out.println(grph.isCyclic());
```



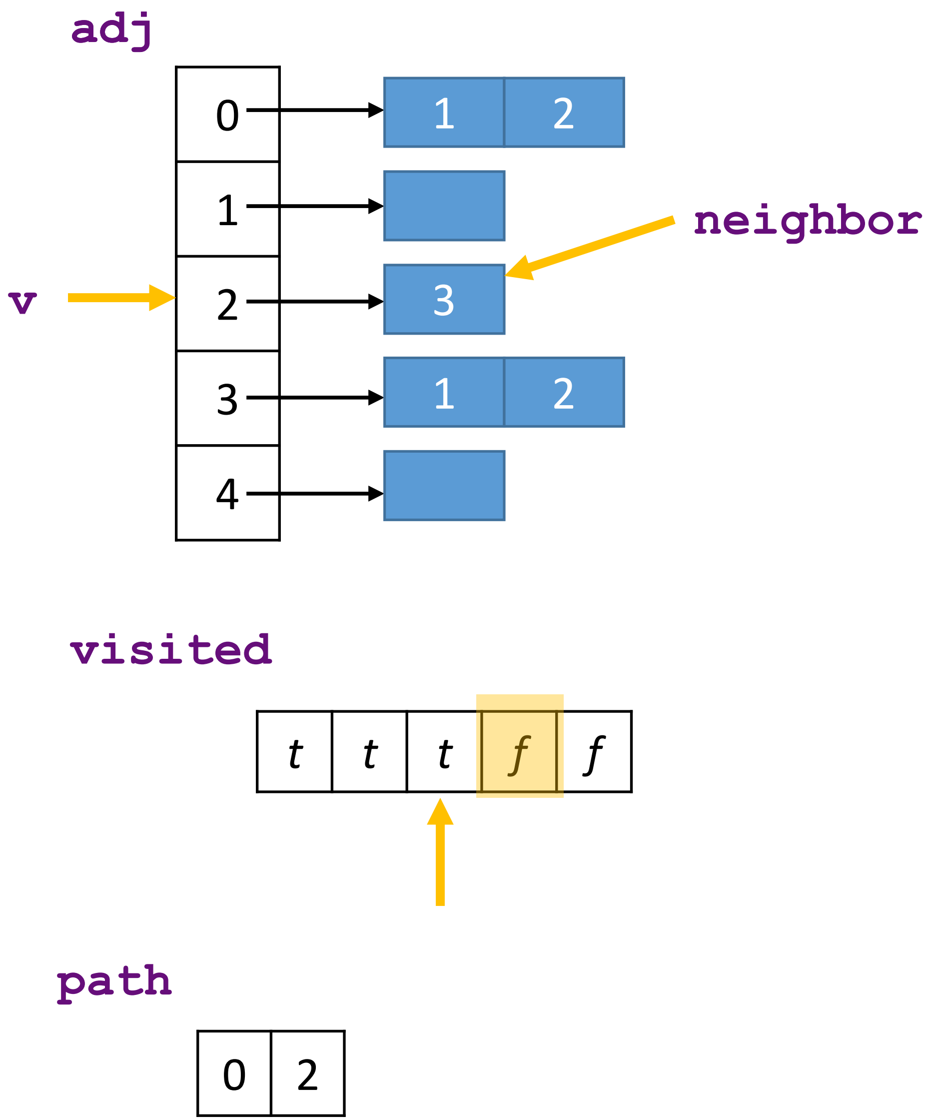

```
public boolean isCyclic() {
    Boolean visited[] = new Boolean[V];

    for (int i = 0; i < V; i++)
        visited[i] = false;

    for (int u = 0; u < V; u++)
        if (!visited[u])
            if (isCyclic(u, visited , new ArrayList()))
                return true;

    return false;
}


public boolean isCyclic(int v, Boolean visited[], ArrayList path) {
    visited[v] = true;
    path.add(v);
    Integer neighbor;

    Iterator<Integer> it = adj.get(v).iterator();
    while (it.hasNext()) {

        neighbor = it.next();
        if (path.contains(neighbor))
            return true;
        if (!visited[neighbor]) {
            if (isCyclic(neighbor, visited, path))
                return true;
        }
    }
    path.remove(new Integer(v));
    return false;
}
```

```java
System.out.println(grph.isCyclic());
```



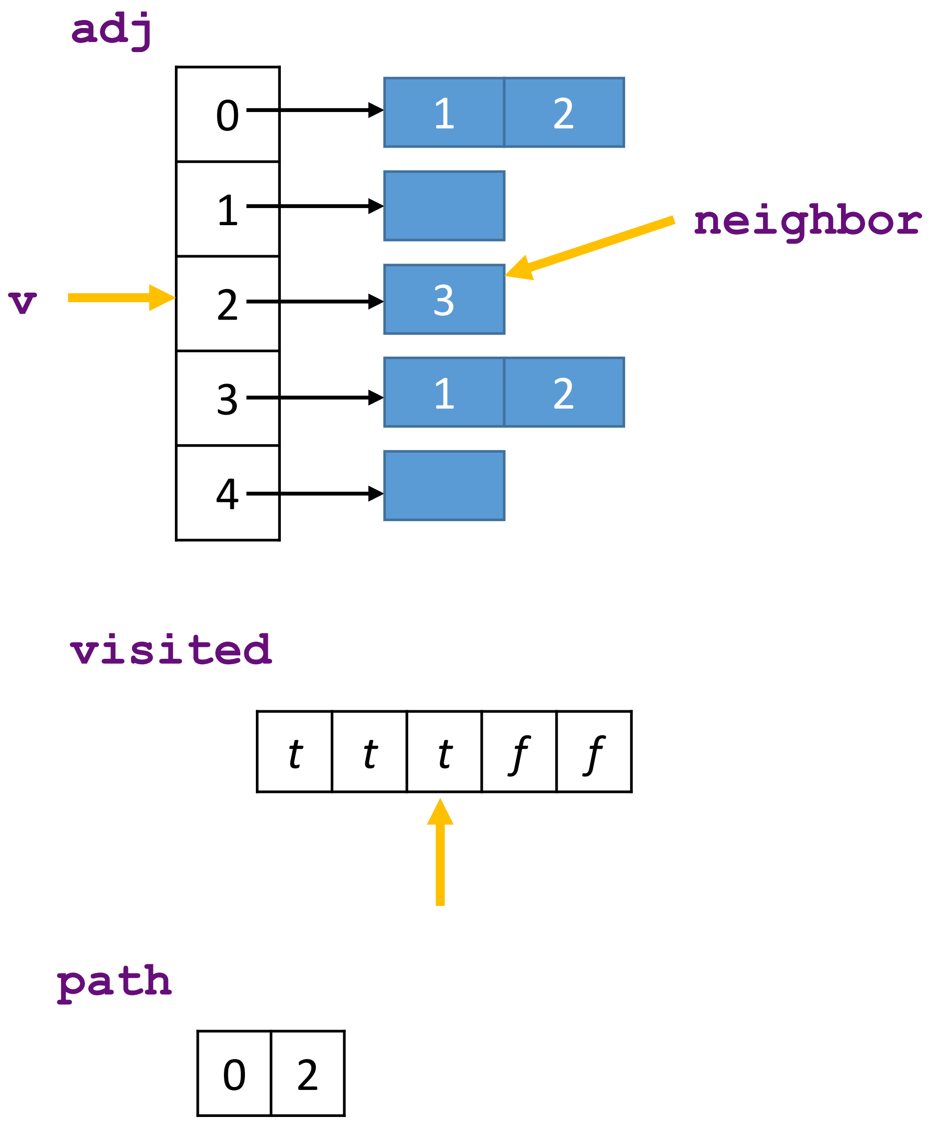

```java
public boolean isCyclic() {
    Boolean visited[] = new Boolean[V];

    for (int i = 0; i < V; i++)
        visited[i] = false;

    for (int u = 0; u < V; u++)
        if (!visited[u])
            if (isCyclic(u, visited , new ArrayList()))
                return true;

    return false;
}


public boolean isCyclic(int v, Boolean visited[], ArrayList path) {
    visited[v] = true;
    path.add(v);
    Integer neighbor;

    Iterator<Integer> it = adj.get(v).iterator();
    while (it.hasNext()) {

        neighbor = it.next();
        if (path.contains(neighbor))
            return true;
        if (!visited[neighbor]) {
            if (isCyclic(neighbor, visited, path))
                return true;
        }
    }
    path.remove(new Integer(v));
    return false;
}
```

```java
System.out.println(grph.isCyclic());
```



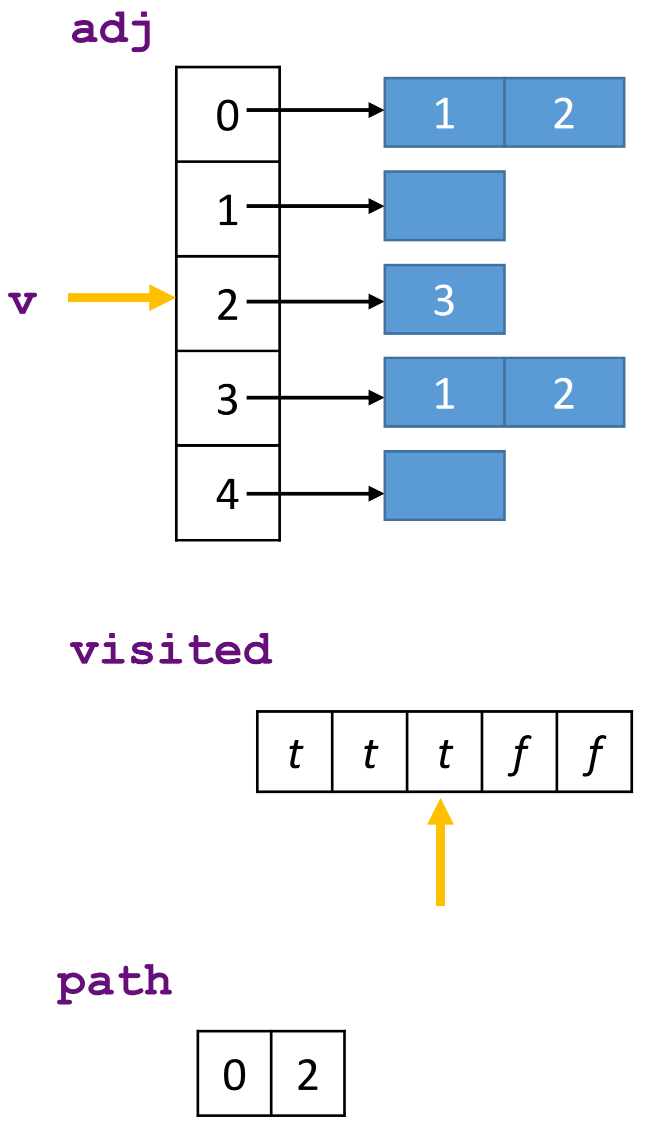

```java
public boolean isCyclic() {
    Boolean visited[] = new Boolean[V];

    for (int i = 0; i < V; i++)
        visited[i] = false;

    for (int u = 0; u < V; u++)
        if (!visited[u])
            if (isCyclic(u, visited , new ArrayList()))
                return true;

    return false;
}


public boolean isCyclic(int v, Boolean visited[], ArrayList path) {
    visited[v] = true;
    path.add(v);
    Integer neighbor;

    Iterator<Integer> it = adj.get(v).iterator();
    while (it.hasNext()) {

        neighbor = it.next();
        if (path.contains(neighbor))
            return true;
        if (!visited[neighbor]) {
            if (isCyclic(neighbor, visited, path))
                return true;
        }
    }
    path.remove(new Integer(v));
    return false;
}
```

```java
System.out.println(grph.isCyclic());
```



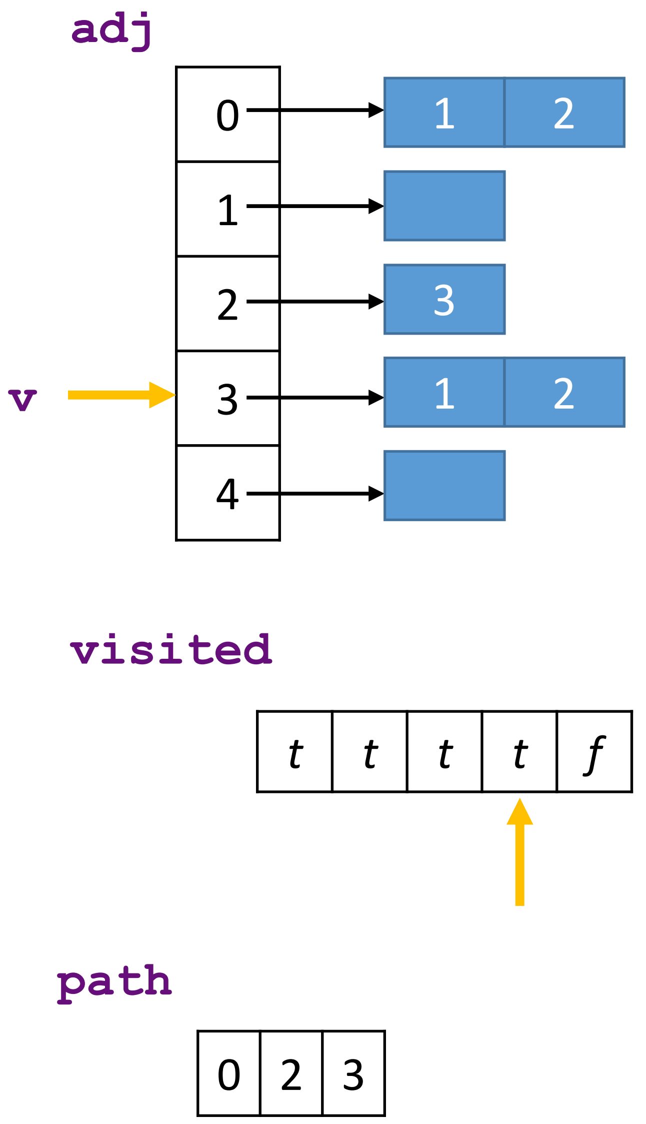

```java
public boolean isCyclic() {
    Boolean visited[] = new Boolean[V];

    for (int i = 0; i < V; i++)
        visited[i] = false;

    for (int u = 0; u < V; u++)
        if (!visited[u])
            if (isCyclic(u, visited , new ArrayList()))
                return true;

    return false;
}


public boolean isCyclic(int v, Boolean visited[], ArrayList path) {
    visited[v] = true;
    path.add(v);
    Integer neighbor;

    Iterator<Integer> it = adj.get(v).iterator();
    while (it.hasNext()) {

        neighbor = it.next();
        if (path.contains(neighbor))
            return true;
        if (!visited[neighbor]) {
            if (isCyclic(neighbor, visited, path))
                return true;
        }
    }
    path.remove(new Integer(v));
    return false;
}
```

```
System.out.println(grph.isCyclic());
```



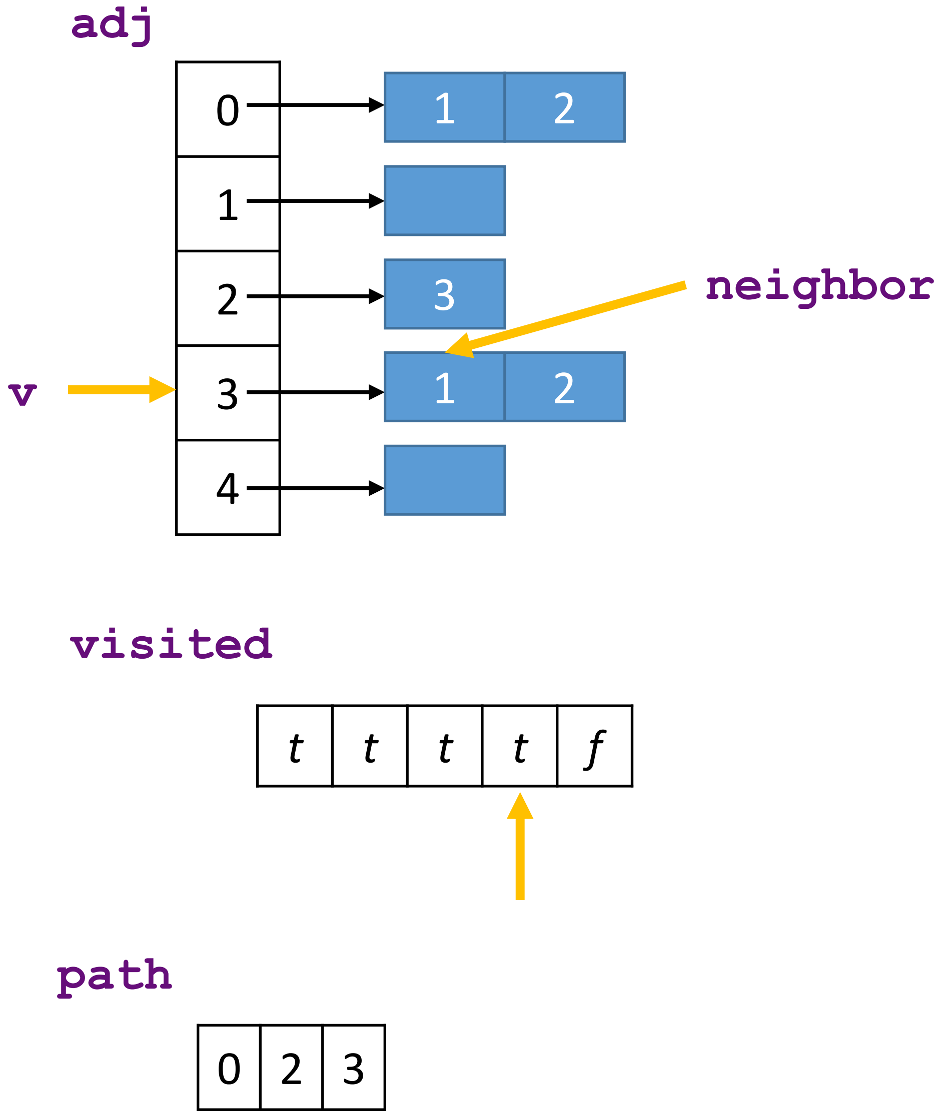

```
public boolean isCyclic() {
    Boolean visited[] = new Boolean[V];

    for (int i = 0; i < V; i++)
        visited[i] = false;

    for (int u = 0; u < V; u++)
        if (!visited[u])
            if (isCyclic(u, visited , new ArrayList()))
                return true;

    return false;
}


public boolean isCyclic(int v, Boolean visited[], ArrayList path) {
    visited[v] = true;
    path.add(v);
    Integer neighbor;

    Iterator<Integer> it = adj.get(v).iterator();
    while (it.hasNext()) {

        neighbor = it.next();
        if (path.contains(neighbor))
            return true;
        if (!visited[neighbor]) {
            if (isCyclic(neighbor, visited, path))
                return true;
        }
    }
    path.remove(new Integer(v));
    return false;
}
```

```
System.out.println(grph.isCyclic());
```



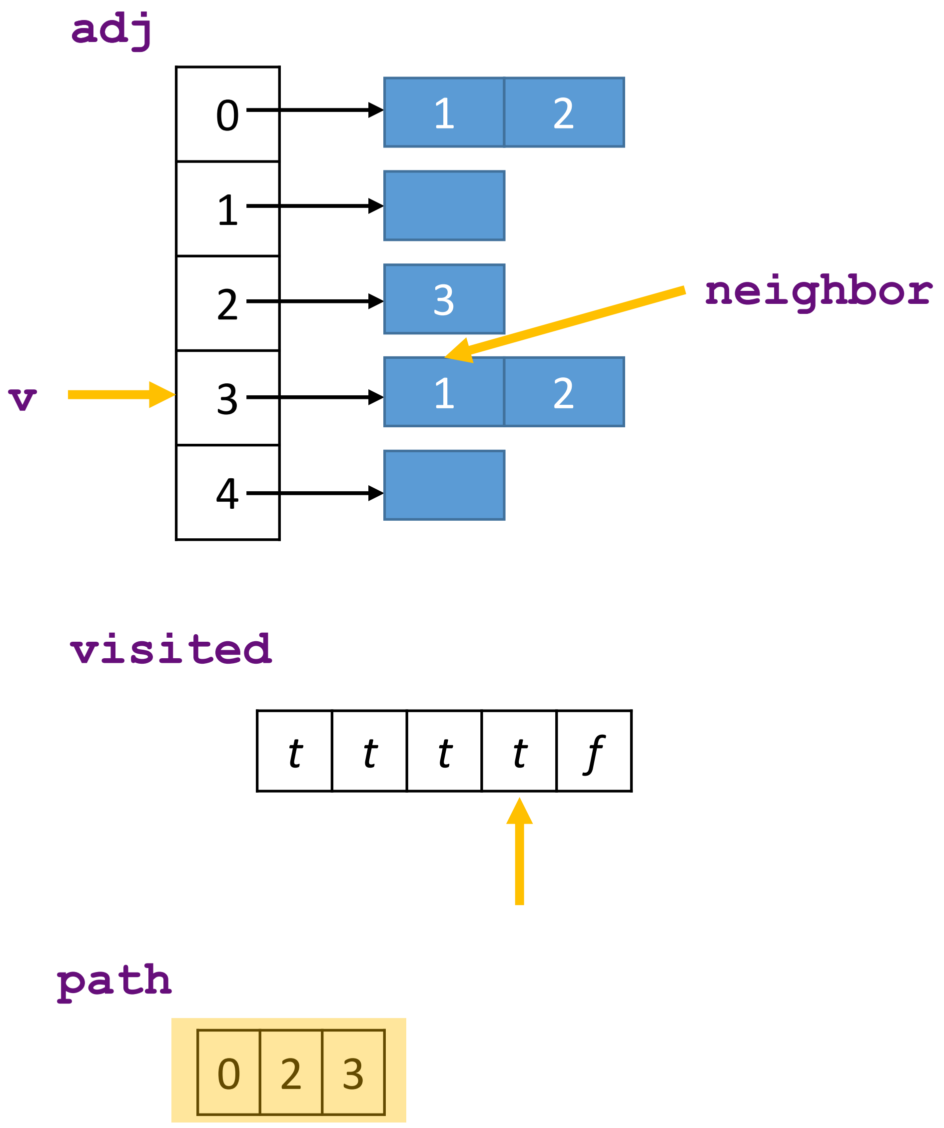

```
public boolean isCyclic() {
    Boolean visited[] = new Boolean[V];

    for (int i = 0; i < V; i++)
        visited[i] = false;

    for (int u = 0; u < V; u++)
        if (!visited[u])
            if (isCyclic(u, visited , new ArrayList()))
                return true;

    return false;
}


public boolean isCyclic(int v, Boolean visited[], ArrayList path) {
    visited[v] = true;
    path.add(v);
    Integer neighbor;

    Iterator<Integer> it = adj.get(v).iterator();
    while (it.hasNext()) {

        neighbor = it.next();
        if (path.contains(neighbor))
            return true;
        if (!visited[neighbor]) {
            if (isCyclic(neighbor, visited, path))
                return true;
        }
    }
    path.remove(new Integer(v));
    return false;
}
```

```java
System.out.println(grph.isCyclic());
```



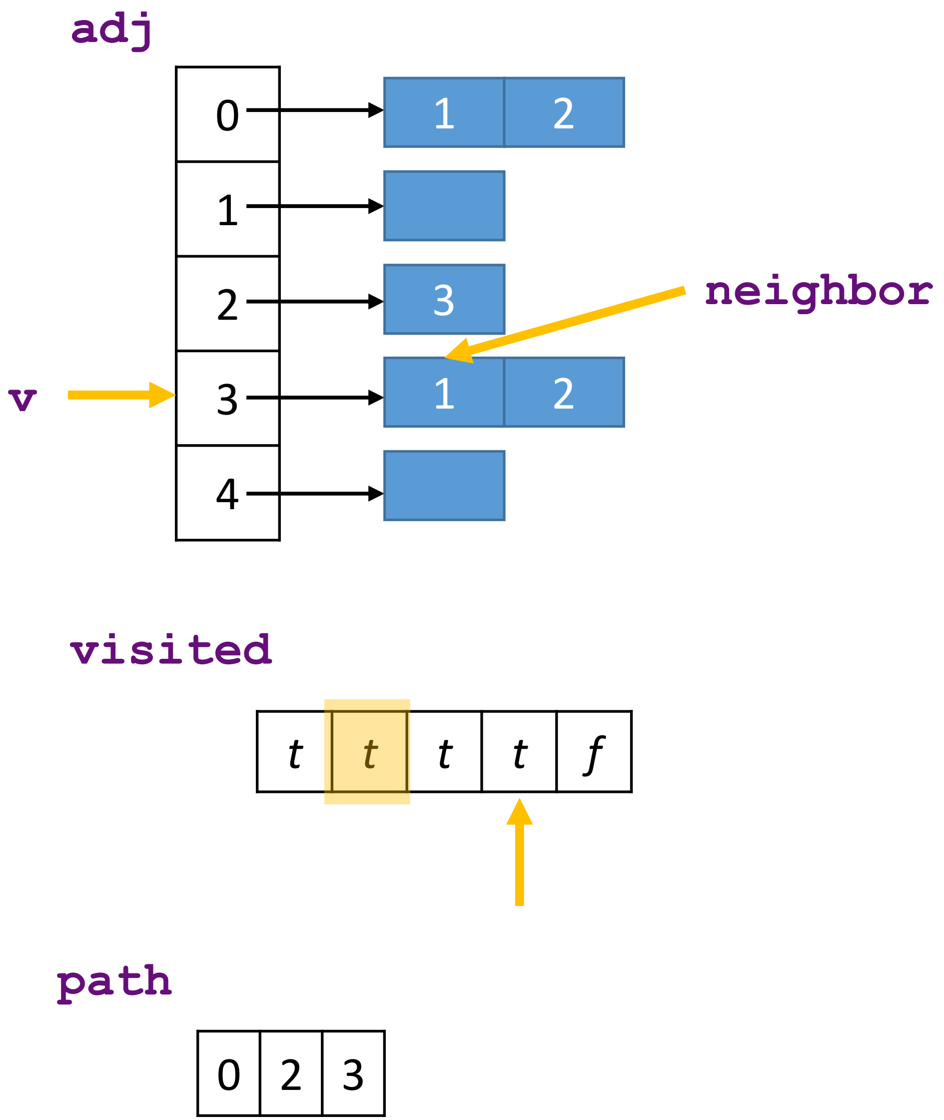

```java
public boolean isCyclic() {
    Boolean visited[] = new Boolean[V];

    for (int i = 0; i < V; i++)
        visited[i] = false;

    for (int u = 0; u < V; u++)
        if (!visited[u])
            if (isCyclic(u, visited , new ArrayList()))
                return true;

    return false;
}


public boolean isCyclic(int v, Boolean visited[], ArrayList path) {
    visited[v] = true;
    path.add(v);
    Integer neighbor;

    Iterator<Integer> it = adj.get(v).iterator();
    while (it.hasNext()) {

        neighbor = it.next();
        if (path.contains(neighbor))
            return true;
        if (!visited[neighbor]) {
            if (isCyclic(neighbor, visited, path))
                return true;
        }
    }
    path.remove(new Integer(v));
    return false;
}
```

```
System.out.println(grph.isCyclic());
```



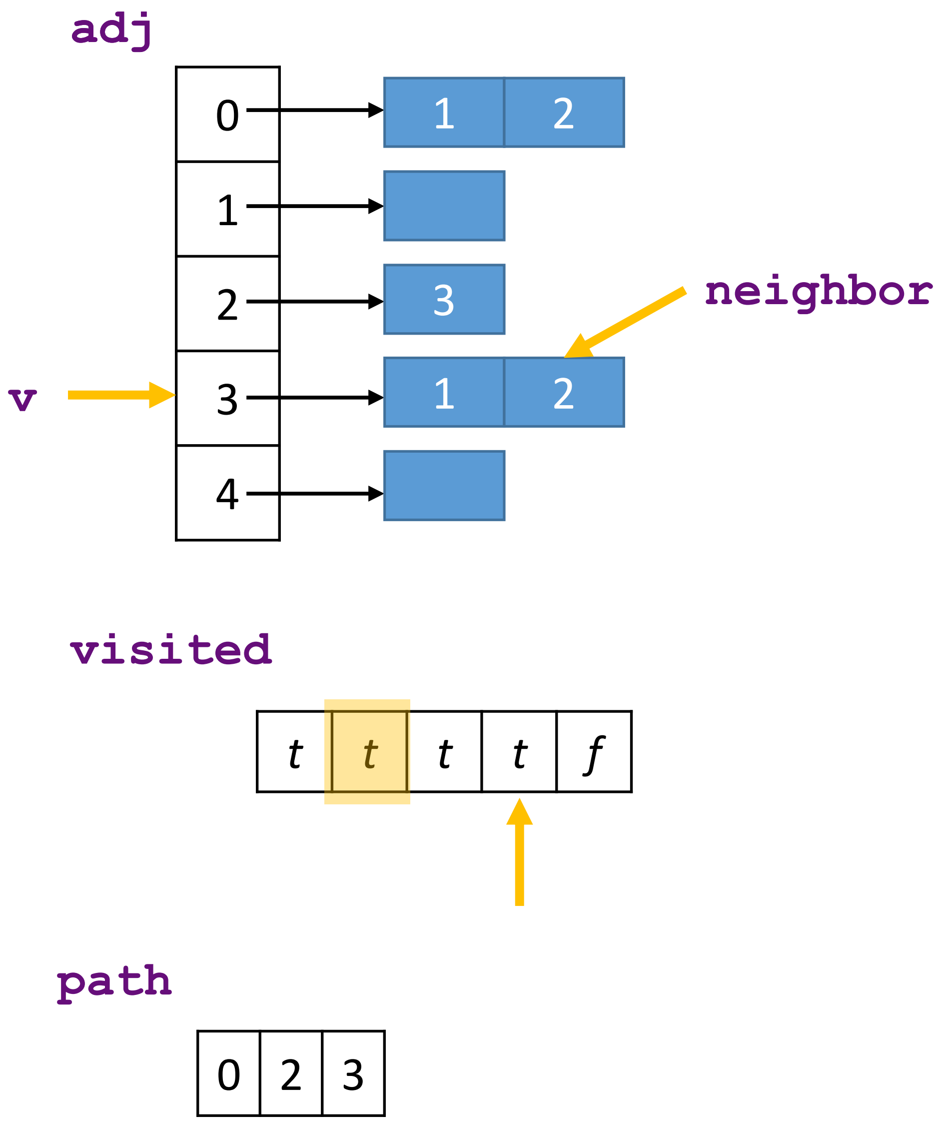

```
public boolean isCyclic() {
    Boolean visited[] = new Boolean[V];

    for (int i = 0; i < V; i++)
        visited[i] = false;

    for (int u = 0; u < V; u++)
        if (!visited[u])
            if (isCyclic(u, visited , new ArrayList()))
                return true;

    return false;
}


public boolean isCyclic(int v, Boolean visited[], ArrayList path) {
    visited[v] = true;
    path.add(v);
    Integer neighbor;

    Iterator<Integer> it = adj.get(v).iterator();
    while (it.hasNext()) {

        neighbor = it.next();
        if (path.contains(neighbor))
            return true;
        if (!visited[neighbor]) {
            if (isCyclic(neighbor, visited, path))
                return true;
        }
    }
    path.remove(new Integer(v));
    return false;
}
```

```
System.out.println(grph.isCyclic());
```



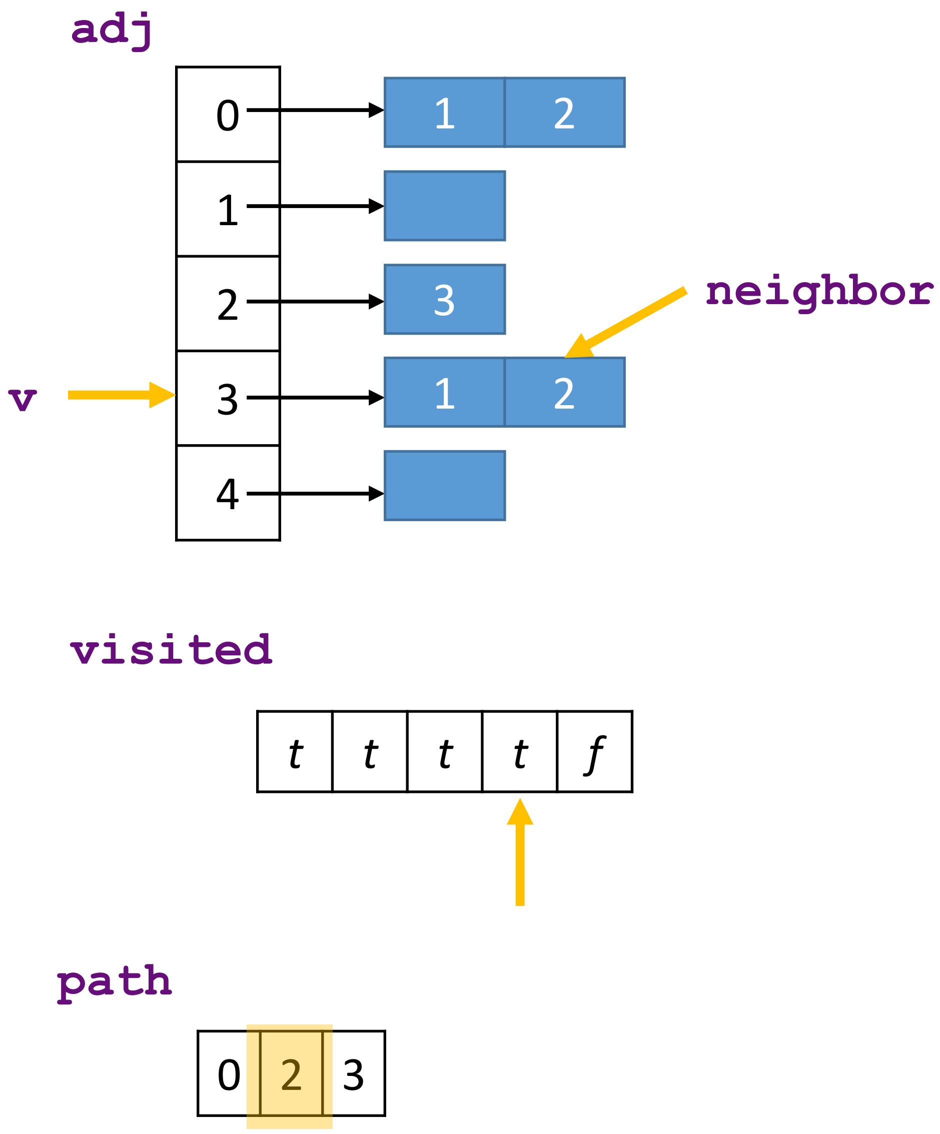

```
public boolean isCyclic() {
    Boolean visited[] = new Boolean[V];

    for (int i = 0; i < V; i++)
        visited[i] = false;

    for (int u = 0; u < V; u++)
        if (!visited[u])
            if (isCyclic(u, visited , new ArrayList()))
                return true;

    return false;
}


public boolean isCyclic(int v, Boolean visited[], ArrayList path) {
    visited[v] = true;
    path.add(v);
    Integer neighbor;

    Iterator<Integer> it = adj.get(v).iterator();
    while (it.hasNext()) {

        neighbor = it.next();
        if (path.contains(neighbor))
            return true;
        if (!visited[neighbor]) {
            if (isCyclic(neighbor, visited, path))
                return true;
        }
    }
    path.remove(new Integer(v));
    return false;
}
```

```
System.out.println(grph.isCyclic());
```



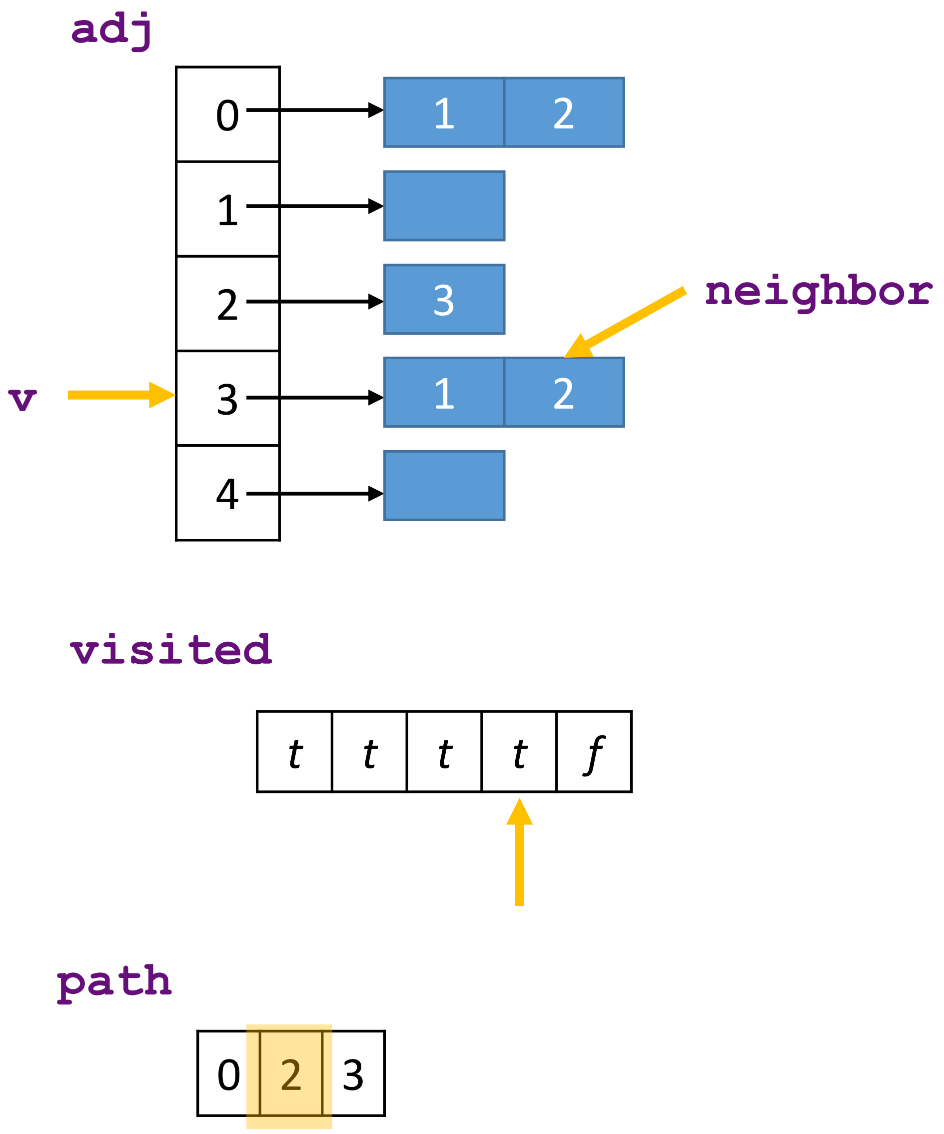

```
public boolean isCyclic() {
    Boolean visited[] = new Boolean[V];

    for (int i = 0; i < V; i++)
        visited[i] = false;

    for (int u = 0; u < V; u++)
        if (!visited[u])
            if (isCyclic(u, visited , new ArrayList()))
                return true;

    return false;
}


public boolean isCyclic(int v, Boolean visited[], ArrayList path) {
    visited[v] = true;
    path.add(v);
    Integer neighbor;

    Iterator<Integer> it = adj.get(v).iterator();
    while (it.hasNext()) {

        neighbor = it.next();
        if (path.contains(neighbor))
            return true;
        if (!visited[neighbor]) {
            if (isCyclic(neighbor, visited, path))
                return true;
        }
    }
    path.remove(new Integer(v));
    return false;
}
```

```java
System.out.println(grph.isCyclic());
```




```java
public boolean isCyclic() {
    Boolean visited[] = new Boolean[V];

    for (int i = 0; i < V; i++)
        visited[i] = false;

    for (int u = 0; u < V; u++)
        if (!visited[u])
            if (isCyclic(u, visited , new ArrayList()))
                return true;

    return false;
}


public boolean isCyclic(int v, Boolean visited[], ArrayList path) {
    visited[v] = true;
    path.add(v);
    Integer neighbor;

    Iterator<Integer> it = adj.get(v).iterator();
    while (it.hasNext()) {

        neighbor = it.next();
        if (path.contains(neighbor))
            return true;
        if (!visited[neighbor]) {
            if (isCyclic(neighbor, visited, path))
                return true;
        }
    }
    path.remove(new Integer(v));
    return false;
}
```

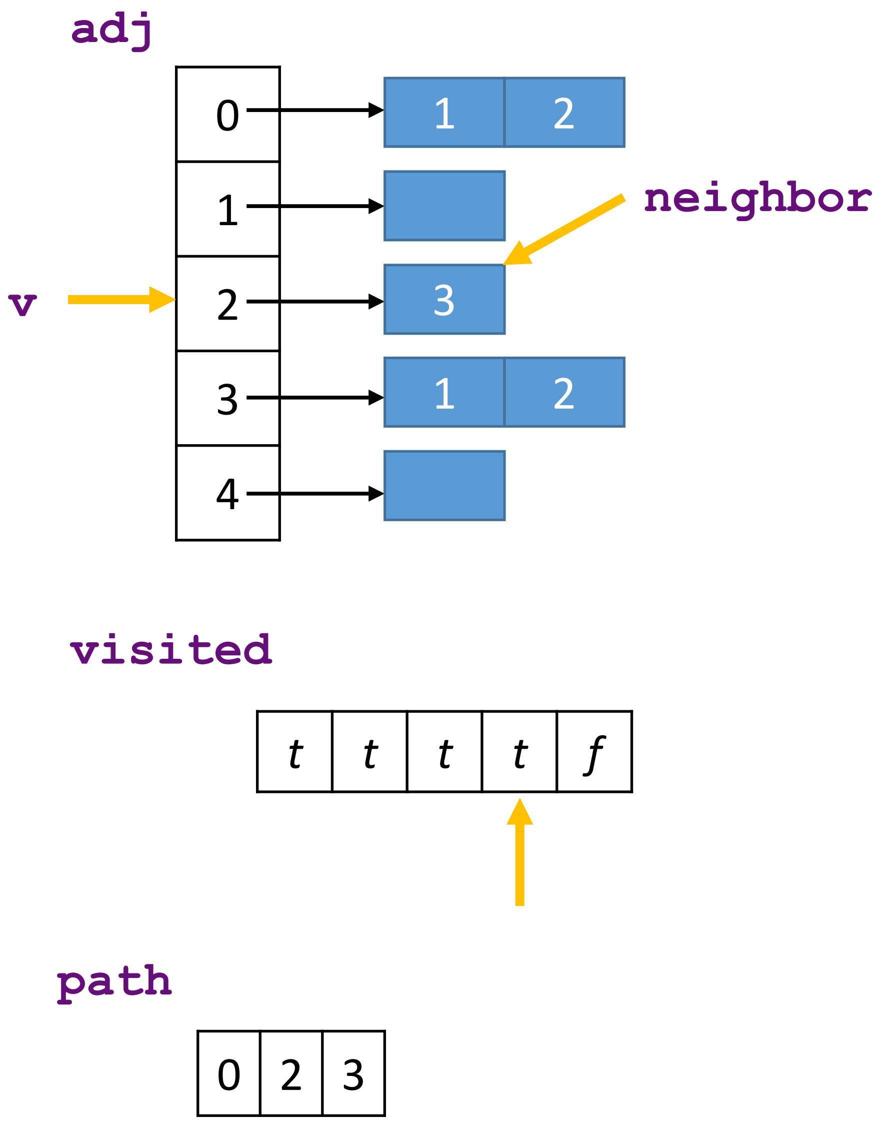
```
System.out.println(grph.isCyclic());
```



return true;

```java
public boolean isCyclic() {
    Boolean visited[] = new Boolean[V];

    for (int i = 0; i < V; i++)
        visited[i] = false;

    for (int u = 0; u < V; u++)
        if (!visited[u])
            if (isCyclic(u, visited , new ArrayList()))
                return true;

    return false;
}


public boolean isCyclic(int v, Boolean visited[], ArrayList path) {
    visited[v] = true;
    path.add(v);
    Integer neighbor;

    Iterator<Integer> it = adj.get(v).iterator();
    while (it.hasNext()) {

        neighbor = it.next();
        if (path.contains(neighbor))
            return true;
        if (!visited[neighbor]) {
            if (isCyclic(neighbor, visited, path))
                return true;
        }
    }
    path.remove(new Integer(v));
    return false;
}
```

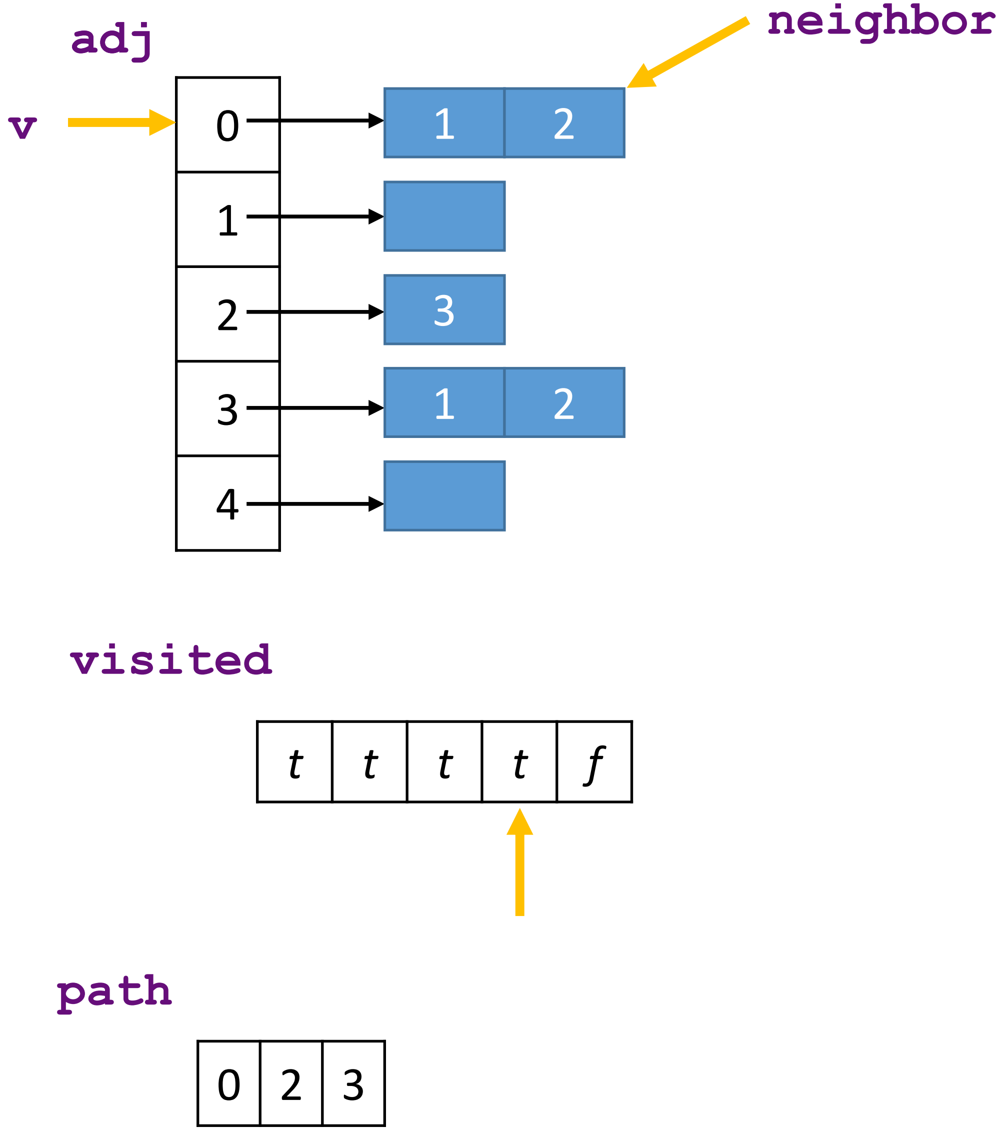
```
System.out.println(grph.isCyclic());
```




```
return true;
```

```
public boolean isCyclic() {
    Boolean visited[] = new Boolean[V];

    for (int i = 0; i < V; i++)
        visited[i] = false;

    for (int u = 0; u < V; u++)
        if (!visited[u])
            if (isCyclic(u, visited , new ArrayList()))
                return true;

    return false;
}


public boolean isCyclic(int v, Boolean visited[], ArrayList path) {
    visited[v] = true;
    path.add(v);
    Integer neighbor;

    Iterator<Integer> it = adj.get(v).iterator();
    while (it.hasNext()) {

        neighbor = it.next();
        if (path.contains(neighbor))
            return true;
        if (!visited[neighbor]) {
            if (isCyclic(neighbor, visited, path))
                return true;
        }
    }
    path.remove(new Integer(v));
    return false;
}
```

```java
System.out.println(grph.isCyclic());
```

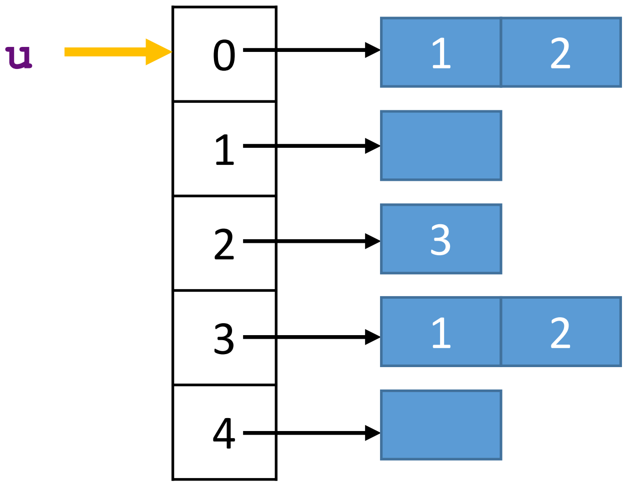



```java
return true;
```

```java
public boolean isCyclic() {
    Boolean visited[] = new Boolean[V];

    for (int i = 0; i < V; i++)
        visited[i] = false;

    for (int u = 0; u < V; u++)
        if (!visited[u])
            if (isCyclic(u, visited , new ArrayList()))
                return true;

    return false;
}


public boolean isCyclic(int v, Boolean visited[], ArrayList path) {
    visited[v] = true;
    path.add(v);
    Integer neighbor;

    Iterator<Integer> it = adj.get(v).iterator();
    while (it.hasNext()) {

        neighbor = it.next();
        if (path.contains(neighbor))
            return true;
        if (!visited[neighbor]) {
            if (isCyclic(neighbor, visited, path))
                return true;
        }
    }
    path.remove(new Integer(v));
    return false;
}
```

```
System.out.println(grph.isCyclic());
```

adj



visited

t	*t*	*t*	*t*	*f*

path

0	2	3

return true;

```
public boolean isCyclic() {
    Boolean visited[] = new Boolean[V];

    for (int i = 0; i < V; i++)
        visited[i] = false;

    for (int u = 0; u < V; u++)
        if (!visited[u])
            if (isCyclic(u, visited , new ArrayList()))
                return true;

    return false;
}


public boolean isCyclic(int v, Boolean visited[], ArrayList path) {
    visited[v] = true;
    path.add(v);
    Integer neighbor;

    Iterator<Integer> it = adj.get(v).iterator();
    while (it.hasNext()) {

        neighbor = it.next();
        if (path.contains(neighbor))
            return true;
        if (!visited[neighbor]) {
            if (isCyclic(neighbor, visited, path))
                return true;
        }
    }
    path.remove(new Integer(v));
    return false;
}
```

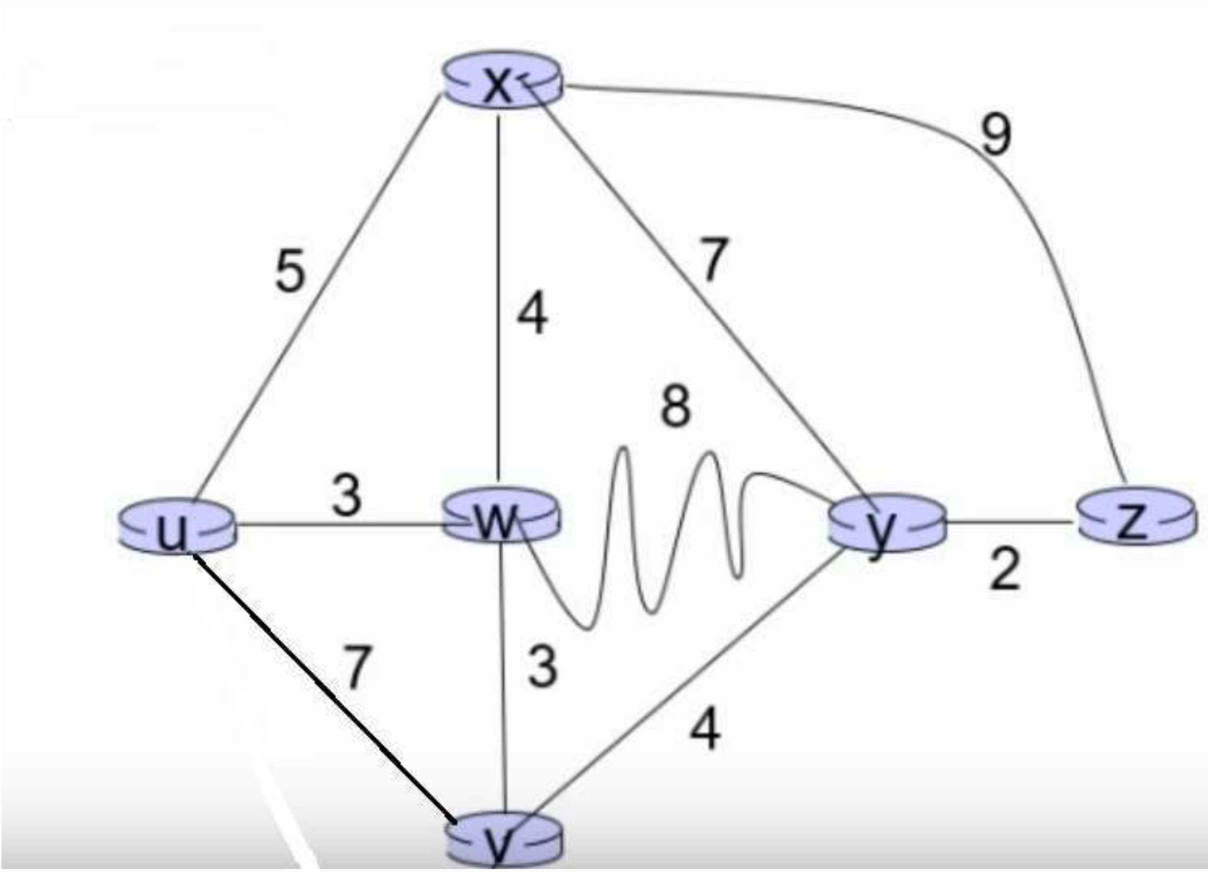
DIJKSTRA’S ALGORITHM

notation:

- **c(x,y):** link cost from node x to y; = ∞ if not direct neighbors
- **D(v):** current value of cost of path from source to dest. v
- **p(v):** predecessor node along path from source to v
- **N':** set of nodes whose least cost path definitively known

```
Initialization:
  N' = {u}
  for all nodes v
    if v adjacent to u
        then D(v) = c(u,v)
    else D(v) = ∞

Loop
  find w not in N' such that D(w) is a minimum
  add w to N'
  update D(v) for all v adjacent to w and not in N' :
     D(v) = min( D(v), D(w) + c(w,v) )
  /* new cost to v is either old cost to v or known
   shortest path cost to w plus cost from w to v */
until all nodes in N'
```

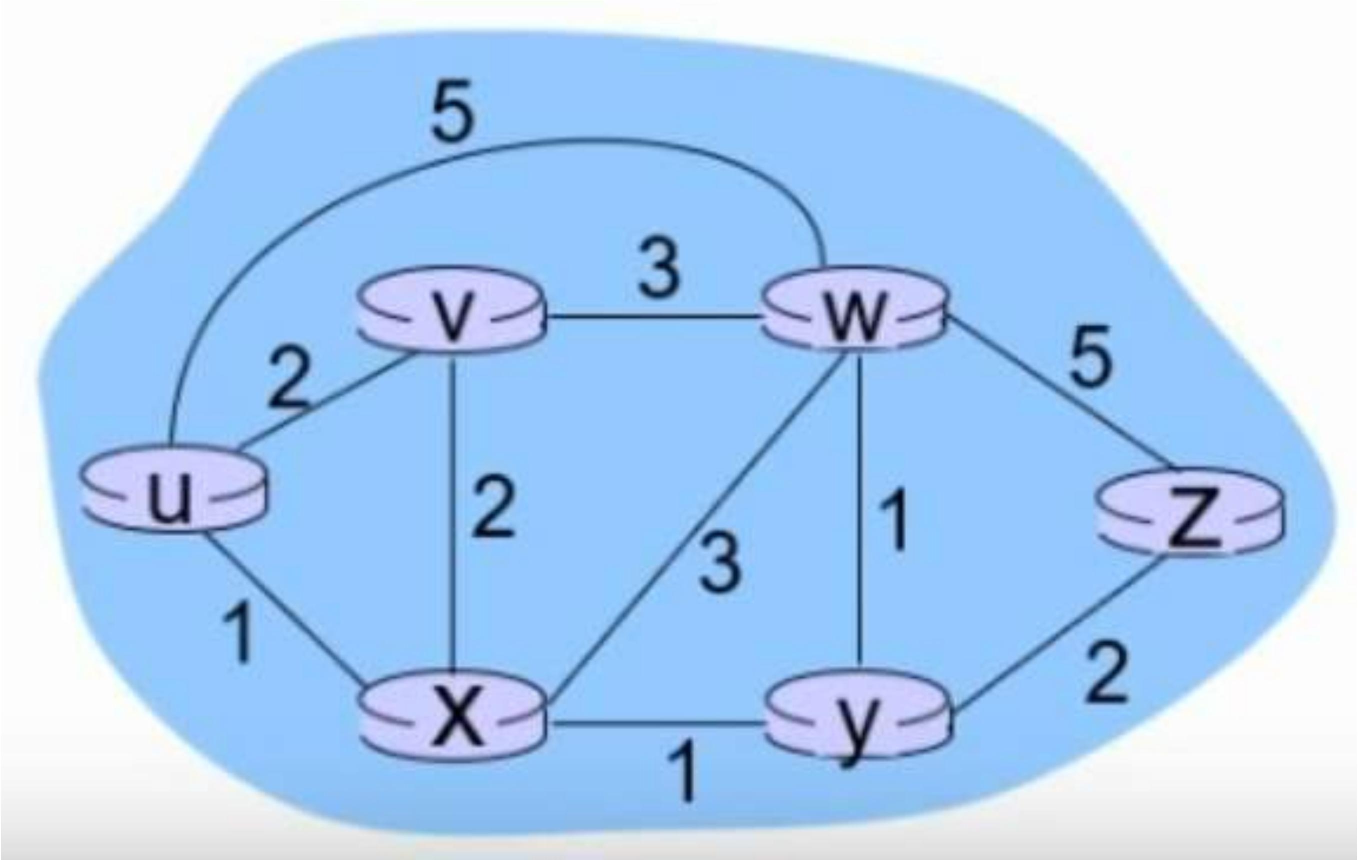
DIJKSTRA'S ALGORITHM



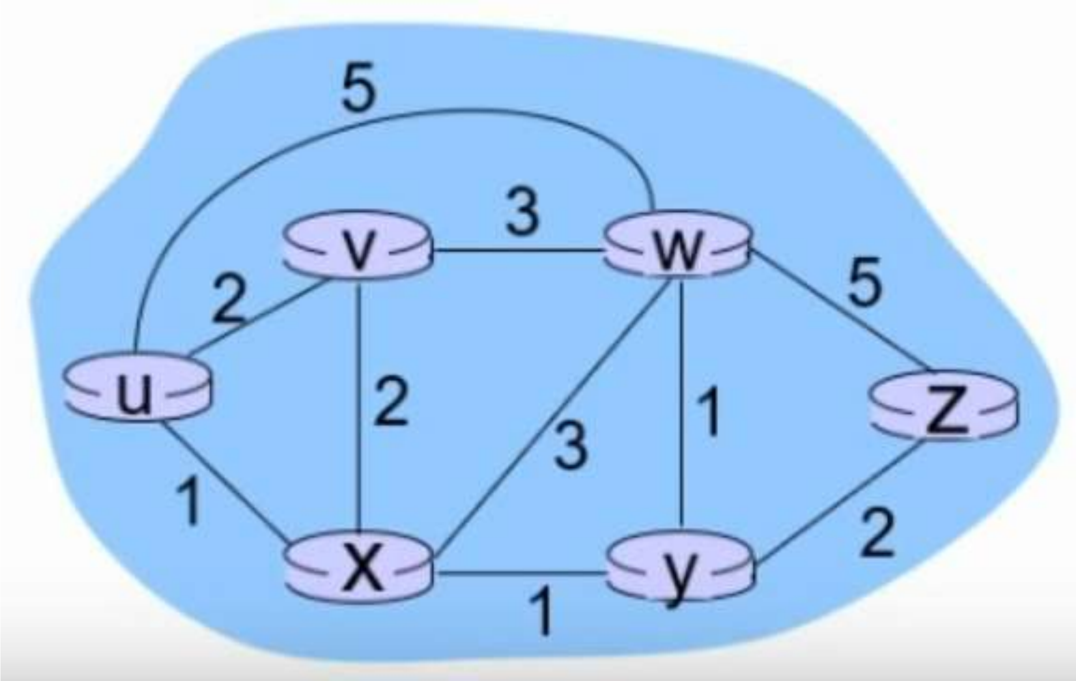
Step	N'	D(v) p(v)	D(w) p(w)	D(x) p(x)	D(y) p(y)	D(z) p(z)
0	u	7,u	3,u	5,u	∞	∞
1	uw	6,w		5,u	11,w	∞
2	uwx	6,w			11,w	14,x
3	uwxv				10,v	14,x
4	uwxvy					12,y
5	uwxvyz					

D(v) = min(D(v), D(w) + c(w,v))

DIJKSTRA’S ALGORITHM



DIJKSTRA'S ALGORITHM



$$D(v) = \min(D(v), D(w) + c(w,v))$$

Step	N'	D(v),p(v)	D(w),p(w)	D(x),p(x)	D(y),p(y)	D(z),p(z)
0	u	2,u	5,u	1,u	∞	∞
1	ux	2,u	4,x		2,x	∞
2	uxy	2,u	3,y			4,y
3	uxyv		3,y			4,y
4	uxyvw					4,y
5	uxyvwz					