

CSE 3318

Week of 07/24/2023

Instructor : Donna French

MST – Prim's Algorithm

Let's start with something a little smaller....

Prim's Algorithm tell us to pick a starting vertex.

Let's pick 0 and add it to our MST.

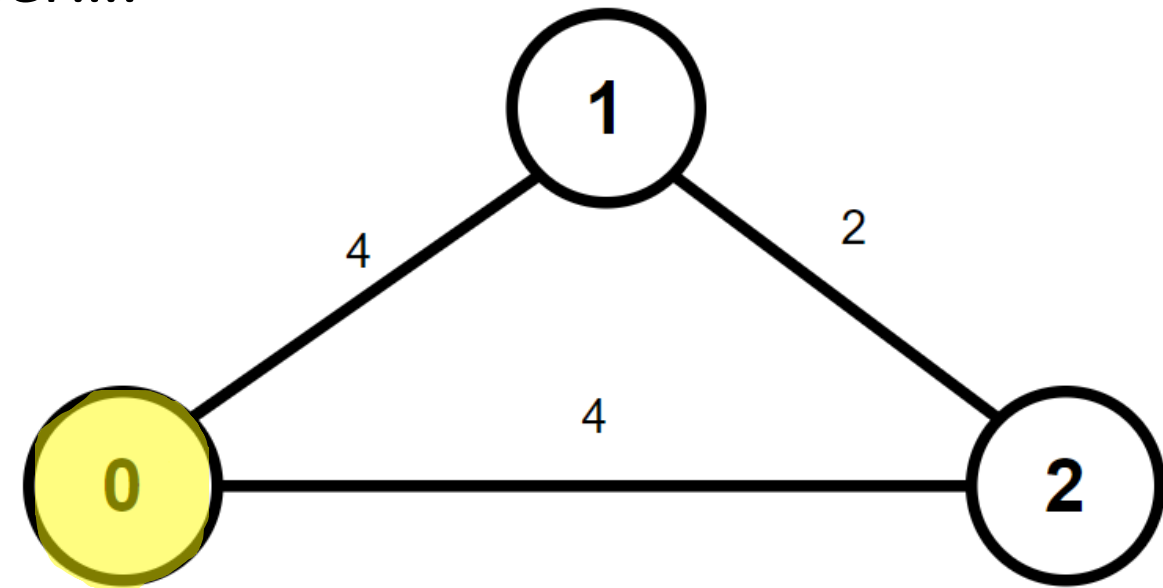
The edges incident to 0 are

the edge to Vertex 1 with a weight of 4 – (0,1,4)

the edge to Vertex 2 with a weight of 4 – (0,2,4)

Notice that I am denoting the edge using (from-vertex, to-vertex, weight)

Now I need to sort these by weight and then by to-vertex...



(0,1,4)

(0,2,4)

MST – Prim's Algorithm

We pick the edge with the lowest/minimum weight/cost.

In this case, both edges have a weight of 4 so we pick $(0,1,4)$ before $(0,2,4)$ because of our sorting.

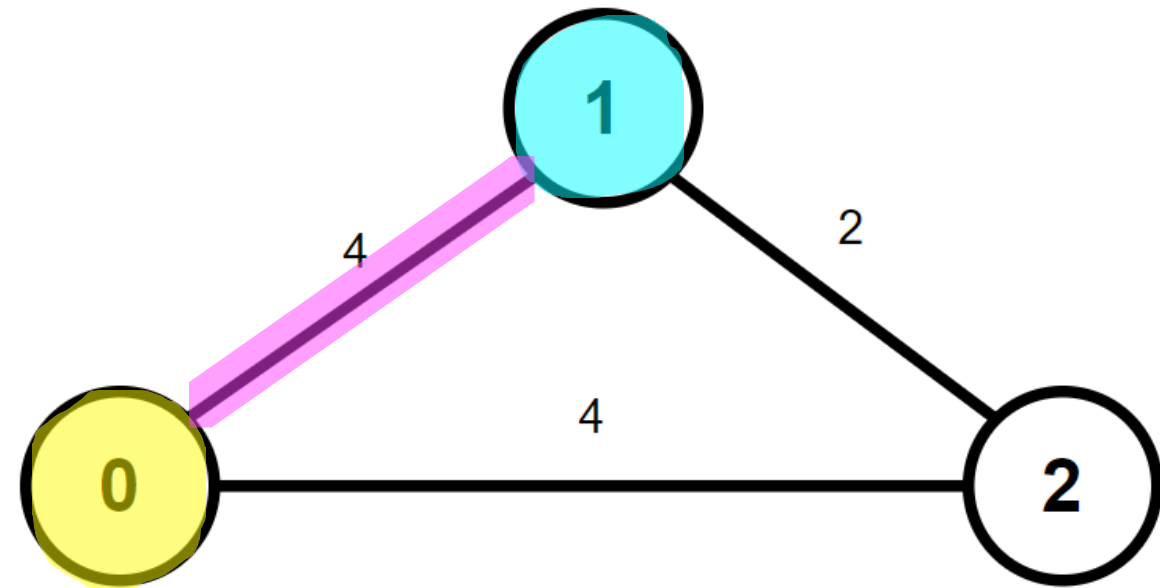
We remove $(0,1,4)$ from our list.

So now we add Vertex 1 to our MST.

Which edges are incident to Vertex 1?

We don't consider Vertex 0 because Vertex 0 is already part of our MST.

We add $(1,2,2)$ – the edge to Vertex 2 with a weight of 2 – we add it to our sorted list.



~~$(0,1,4)$~~
 $(1,2,2)$
 $(0,2,4)$

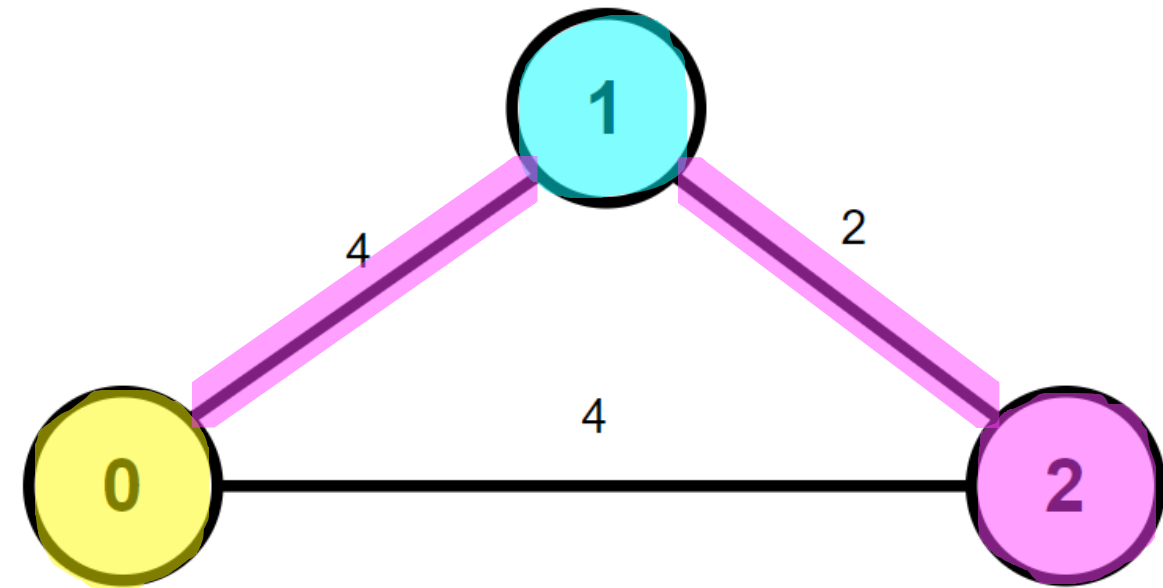
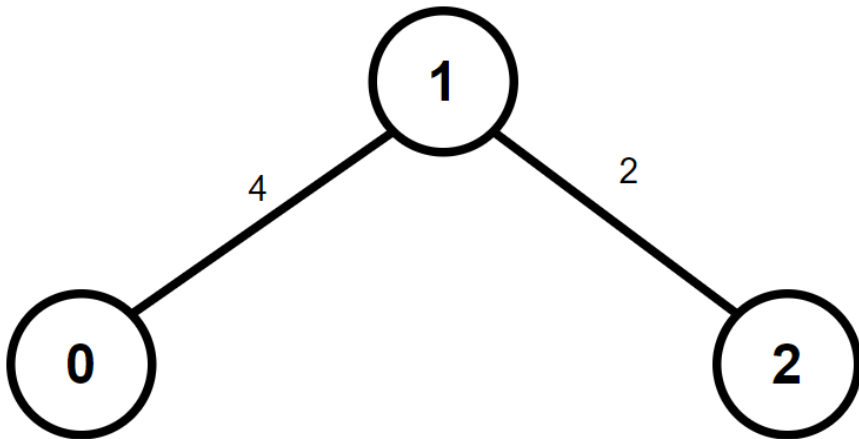
MST – Prim's Algorithm

We pick the edge with the lowest/minimum weight/cost.

Now that is (1,2,2)

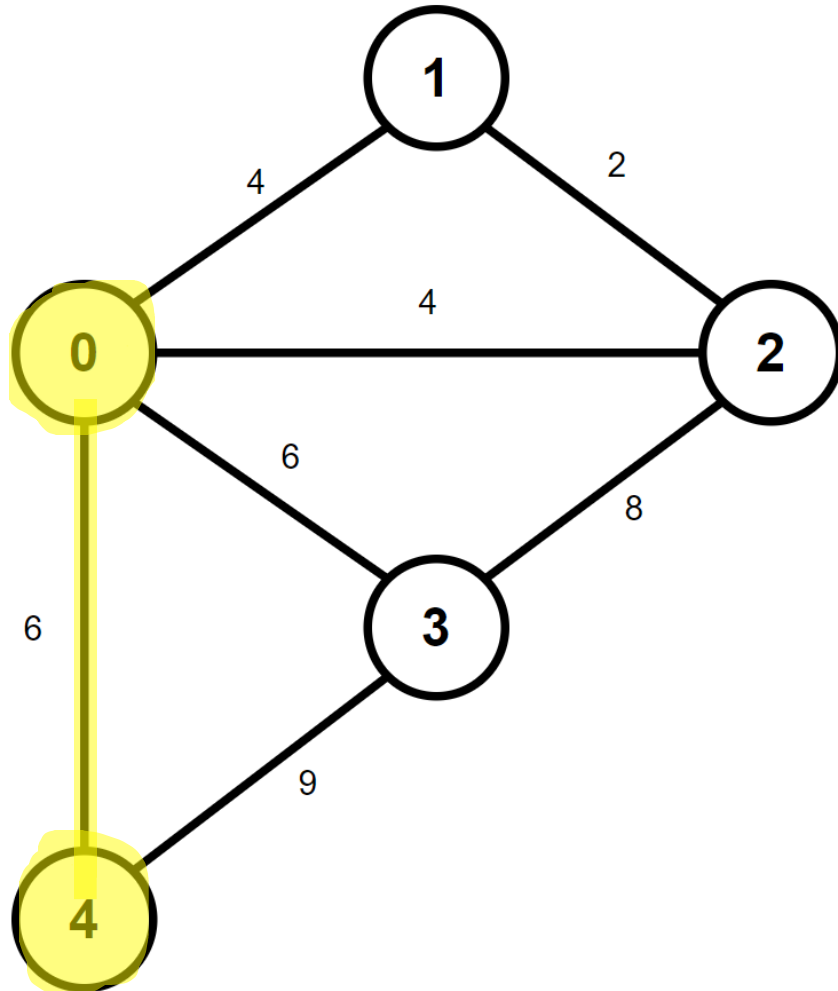
We add Vertex 2 to our MST.

Our graph has 3 vertices and we have 3 vertices in our MST and 2 edges.



~~(0,1,4)~~
~~(1,2,2)~~
(0,2,4)

MST – Prim's Algorithm

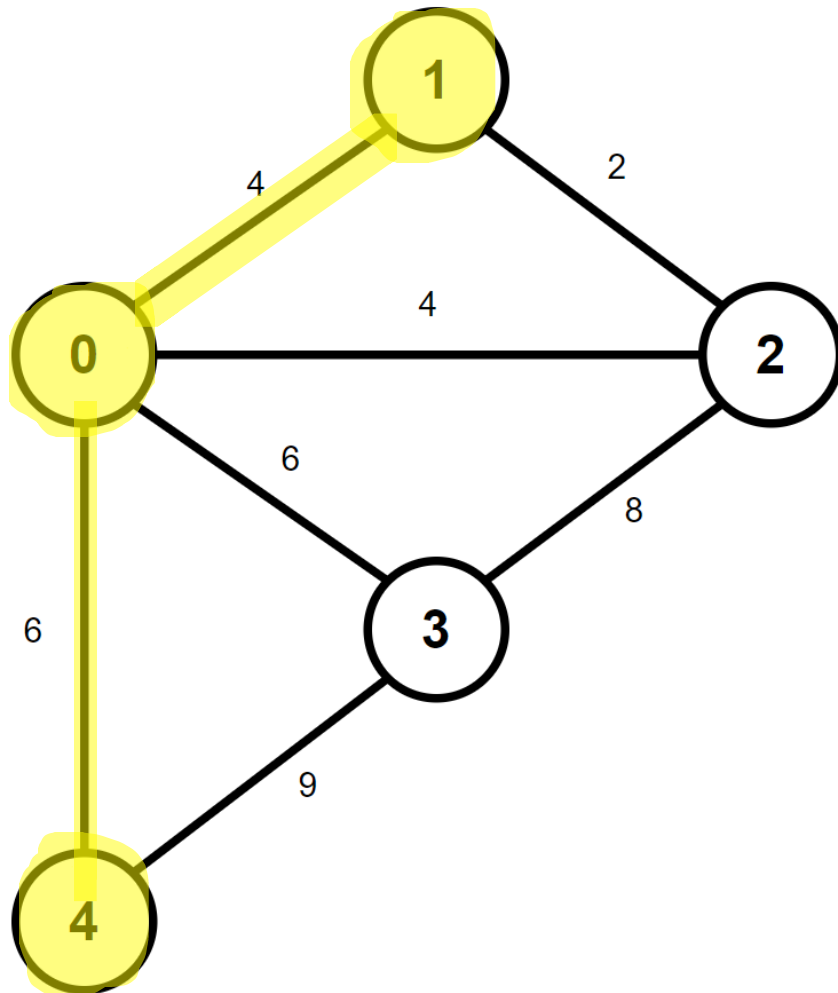


Pick a starting vertex.

Let's pick 4.

~~(4,0,6)~~
(4,3,9)

MST – Prim's Algorithm



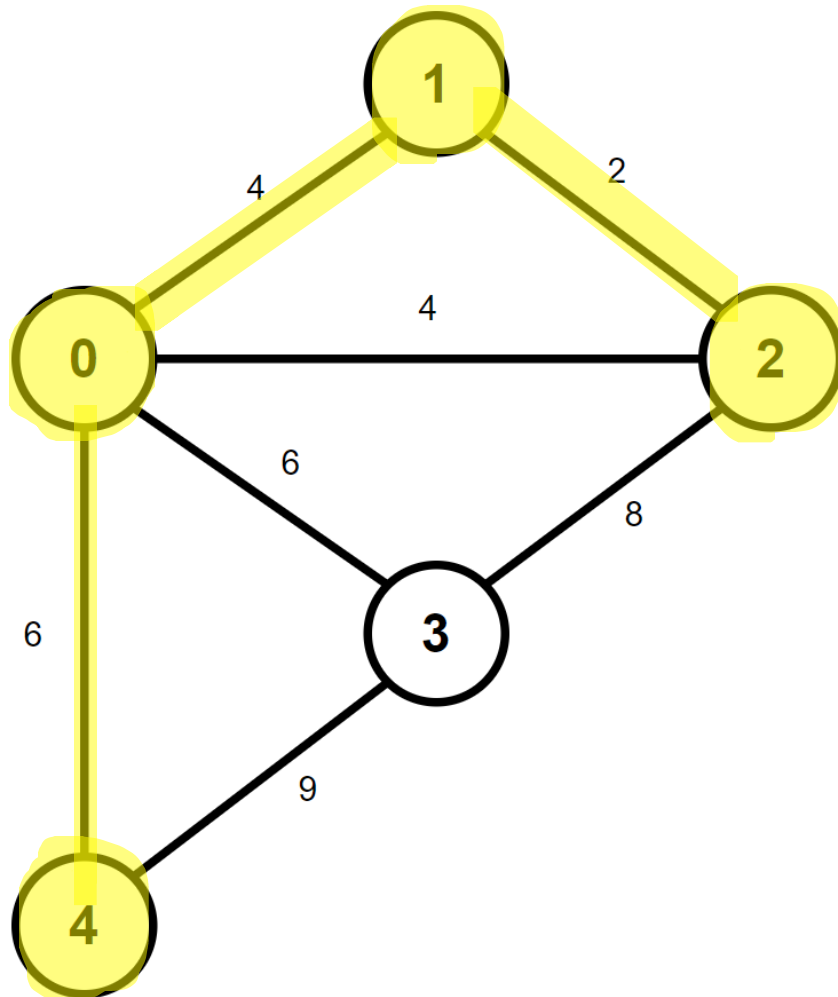
~~(4,0,6)~~
~~(1,0,4)~~
(0,3,6)
(0,2,4)
(0,3,6)
(4,3,9)

Now we look at the edges from Vertex 0.

(0,3,6)
(0,2,4)
(0,1,4)

We add them in order to our list.

MST – Prim's Algorithm



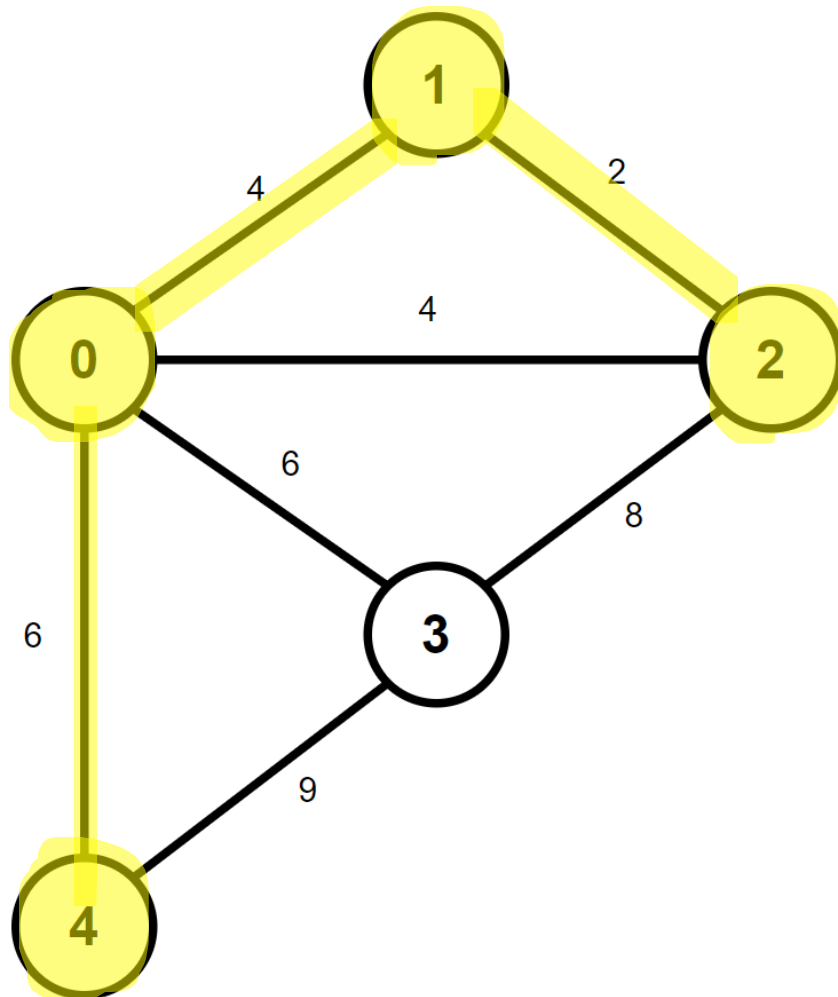
~~(1,0,6)~~
~~(0,1,4)~~
~~(1,2,2)~~
(0,2,4)
(0,3,6)
(4,3,9)

Now we look at the edges from Vertex 1.

(1,2,2)

We add to our ordered list.

MST – Prim's Algorithm



~~(4,0,6)~~
~~(0,1,4)~~
~~(1,2,2)~~
(0,2,4)
(0,3,6)
(2,3,8)
(4,3,9)

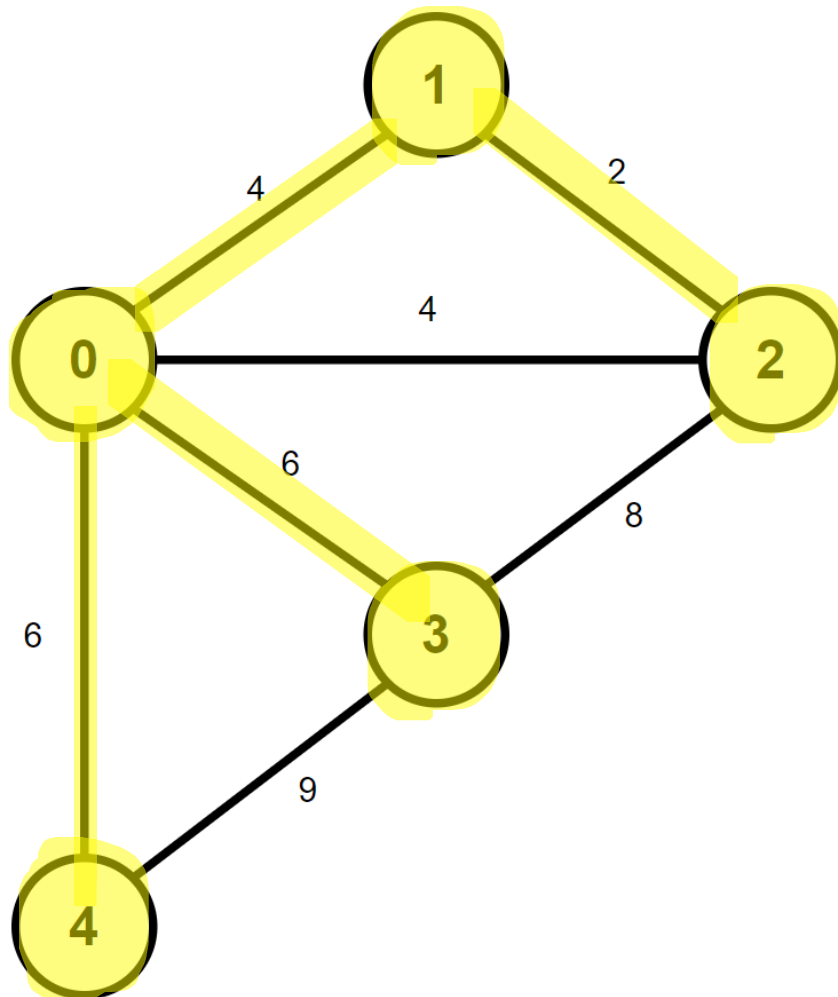
Vertex 2 has an edge with a weight of 4 going to Vertex 0.

Vertex 0 is already in our MST so we don't add that edge.

Vertex 2 also has an edge of weight 8 going to Vertex 3.

We add (2,3,8) to the list.

MST – Prim's Algorithm



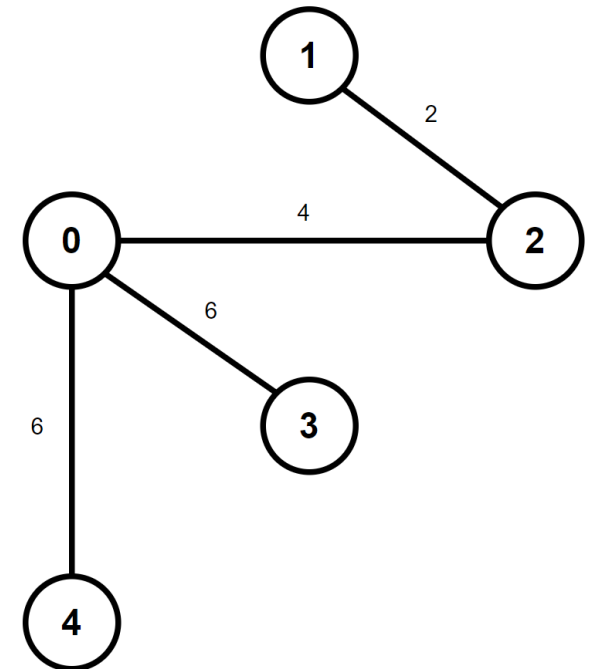
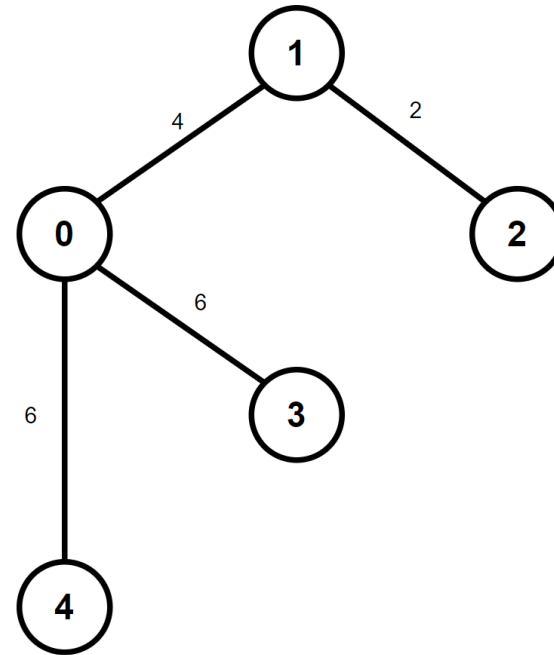
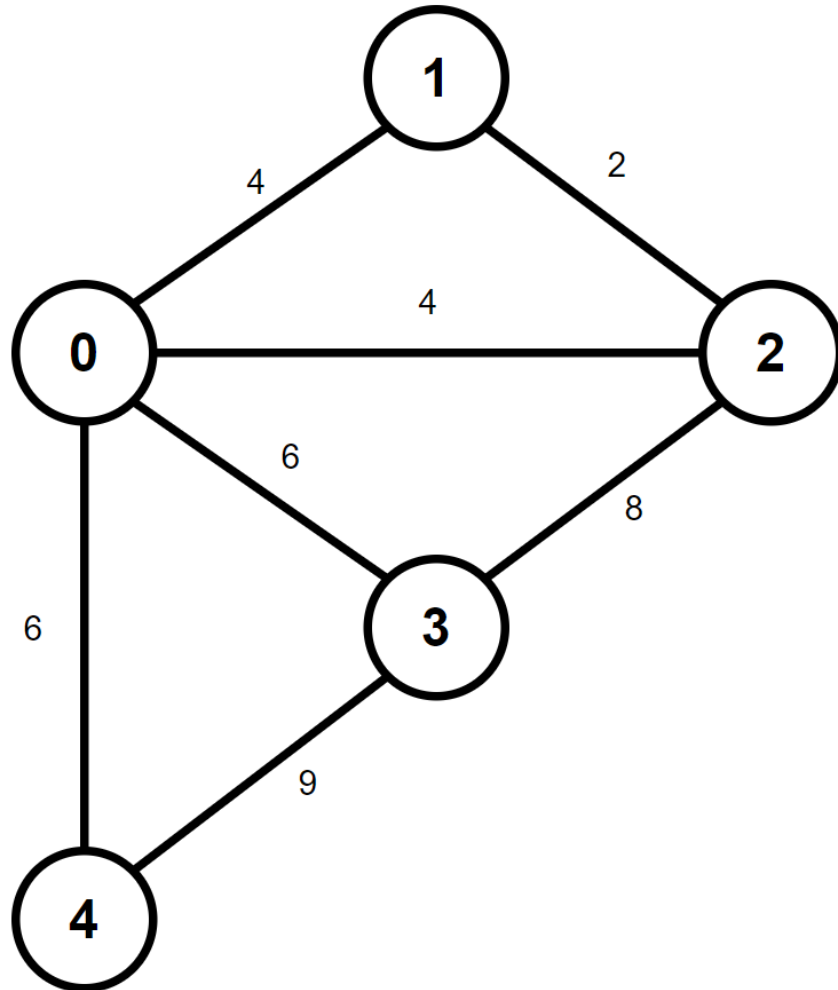
~~(1,0,6)~~
~~(0,1,4)~~
~~(1,2,2)~~
~~(0,2,4)~~
~~(0,3,6)~~
(2,3,8)
(4,3,9)

The next item in our list is (0,2,4).

Since Vertex 2 has been added to our MST, we eliminate that choice from our list.

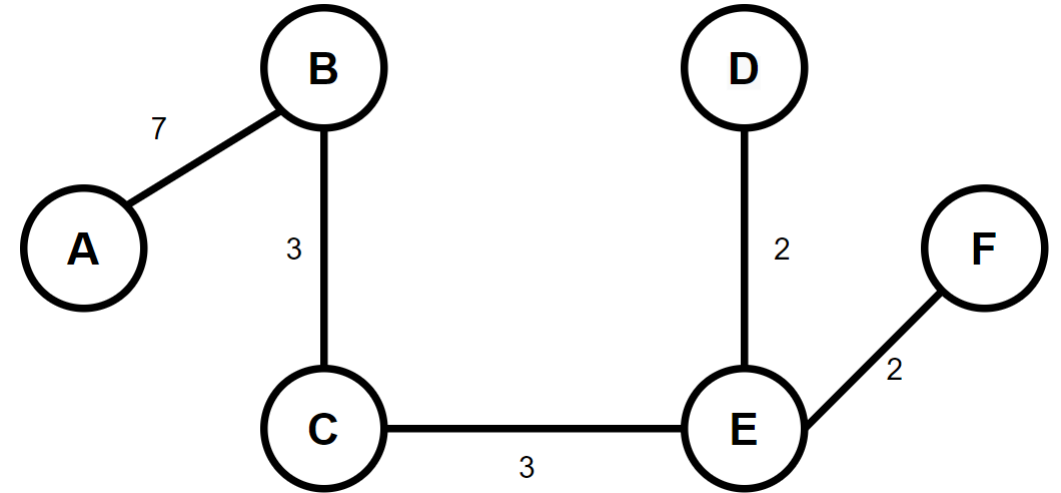
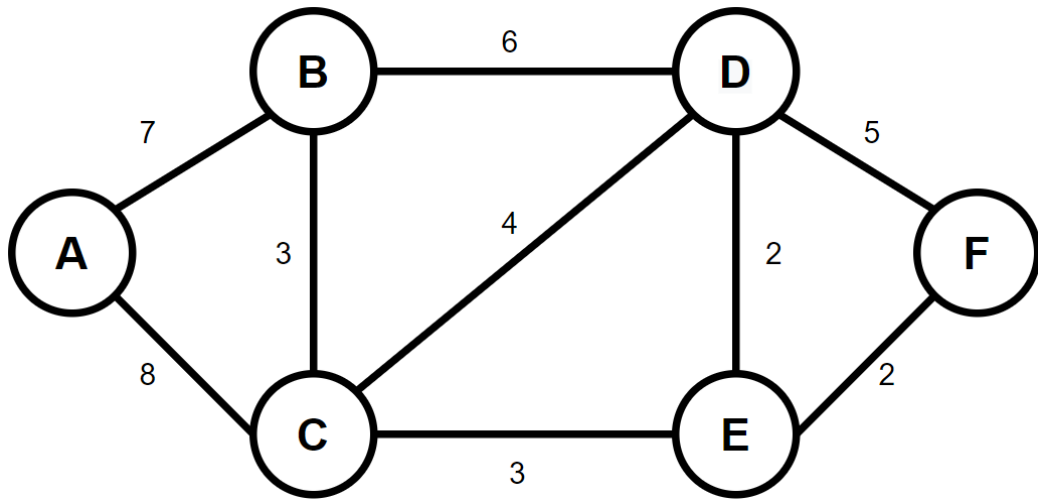
(0,3,6) is next and we have not been to Vertex 3 so we use (0,3,6) to add Vertex 3 to our MST.

MST – Prim's Algorithm

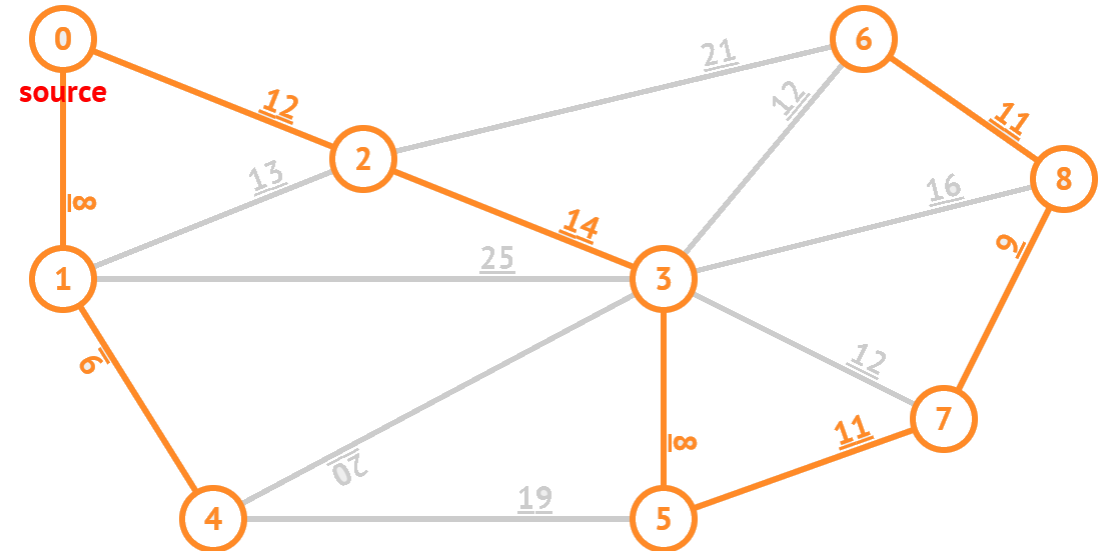
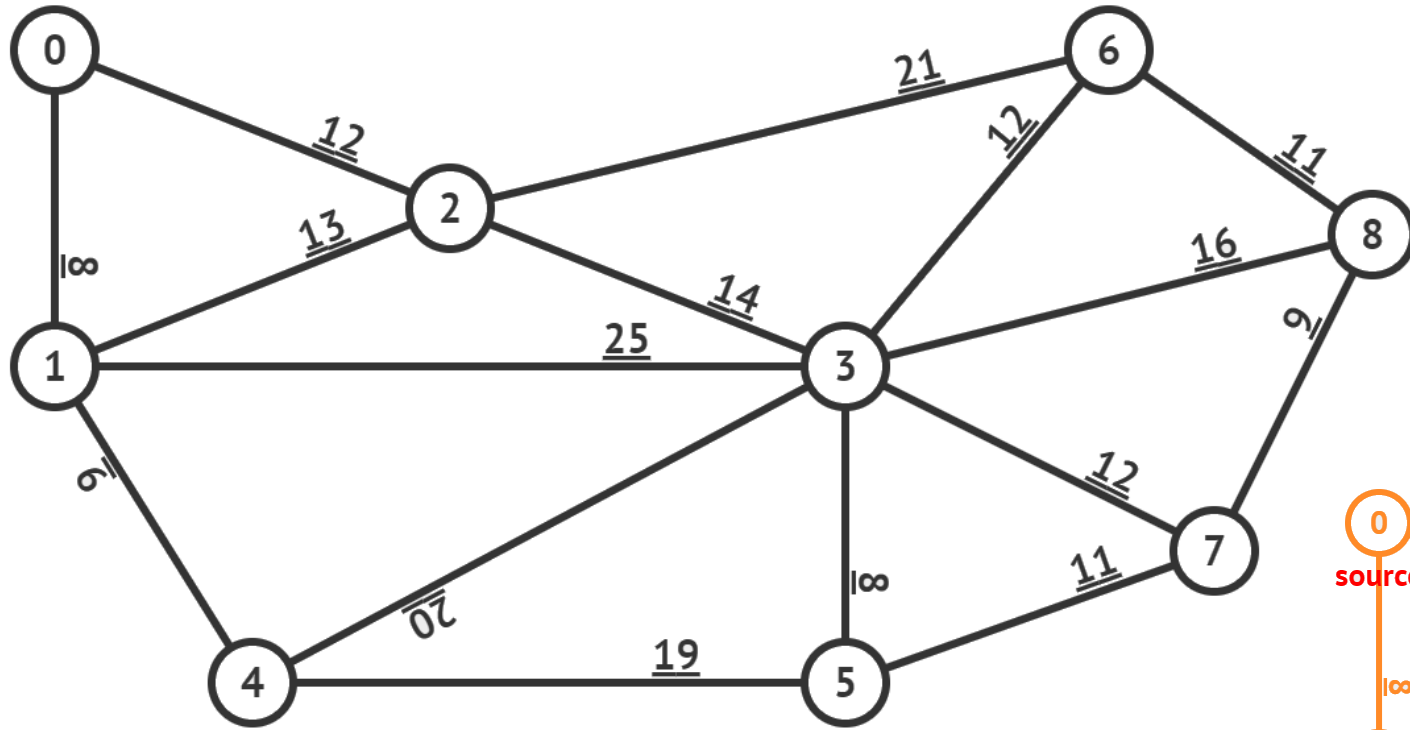


MST – Prim's Algorithm

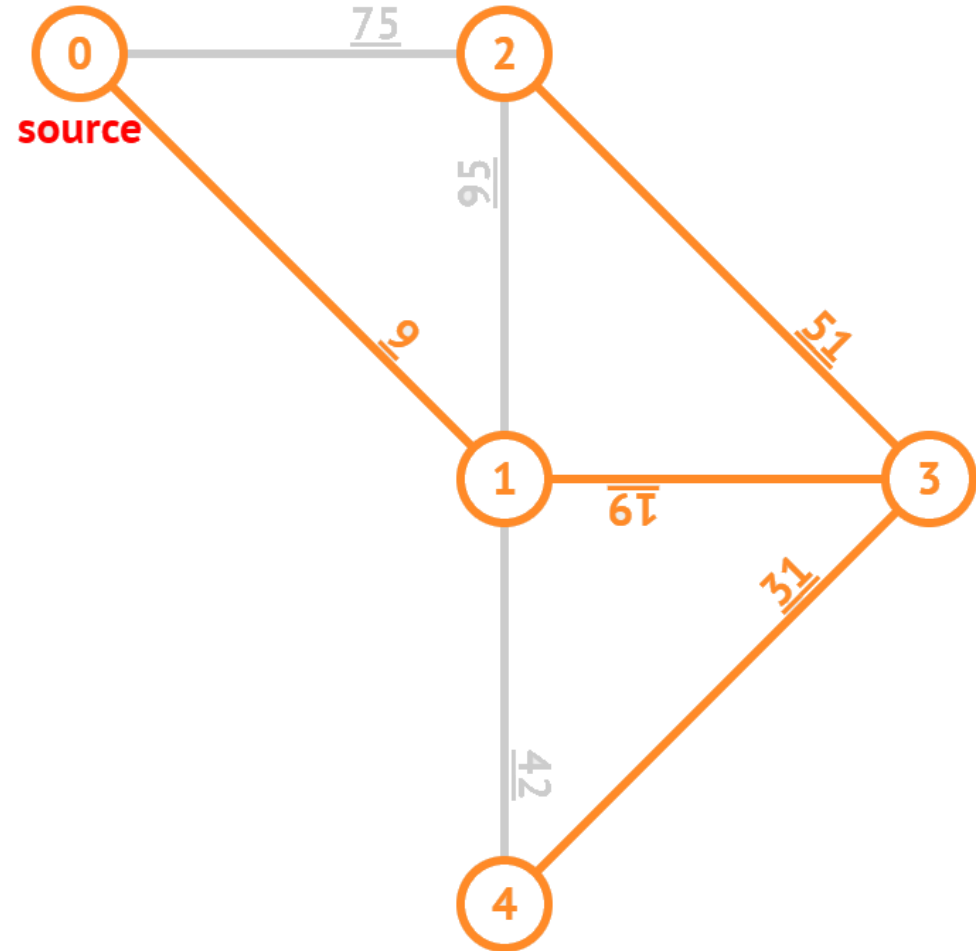
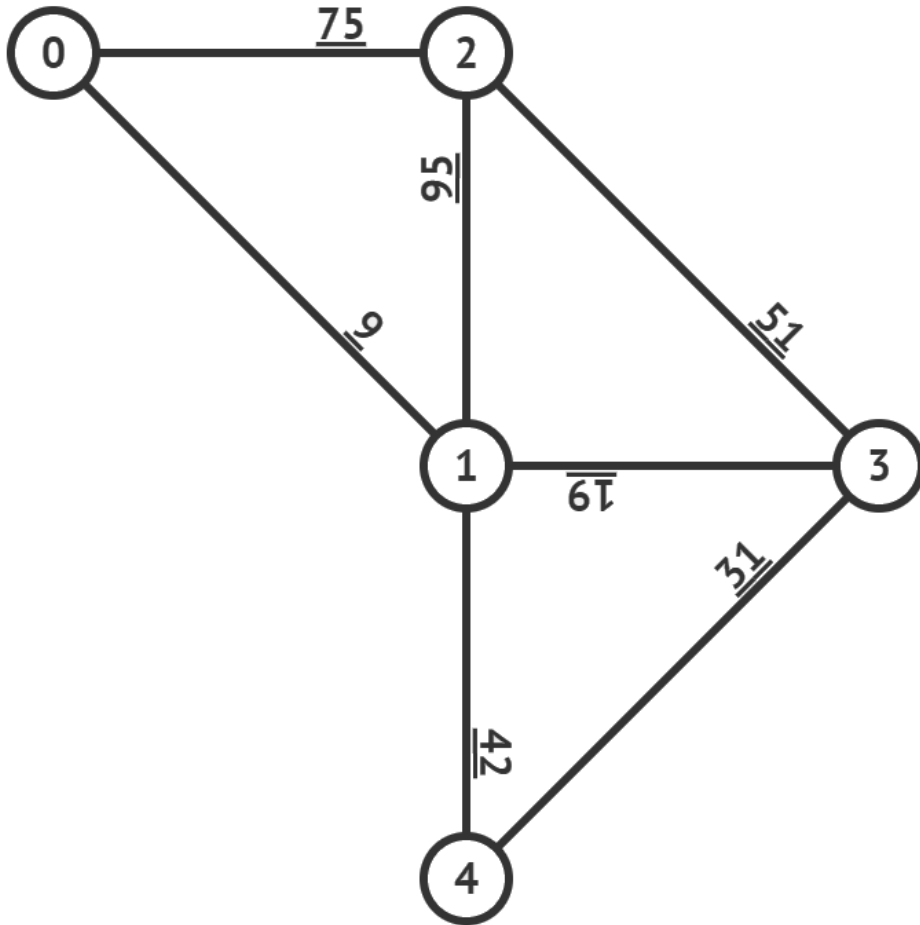
What is the Minimum Spanning Tree for this graph?



MST – Prim's Algorithm



MST – Prim's Algorithm



Prim's Algorithm

What is the run time of Prim's Algorithm?

The big O of Prim's Algorithm will depend on how the algorithm is implemented

Which data structures are used and how those data structures are used.

IT DEPENDS



Prim's Algorithm

Implementing Prim using an array as the priority queue

$O(V^2)$

Implementing Prim using a binary heap as the priority queue

$O(E \log_2 V)$

Implementing Prim using a Fibonacci heap as the priority queue

$O(V \log_2 V + E)$

MST – Kruskal's Algorithm

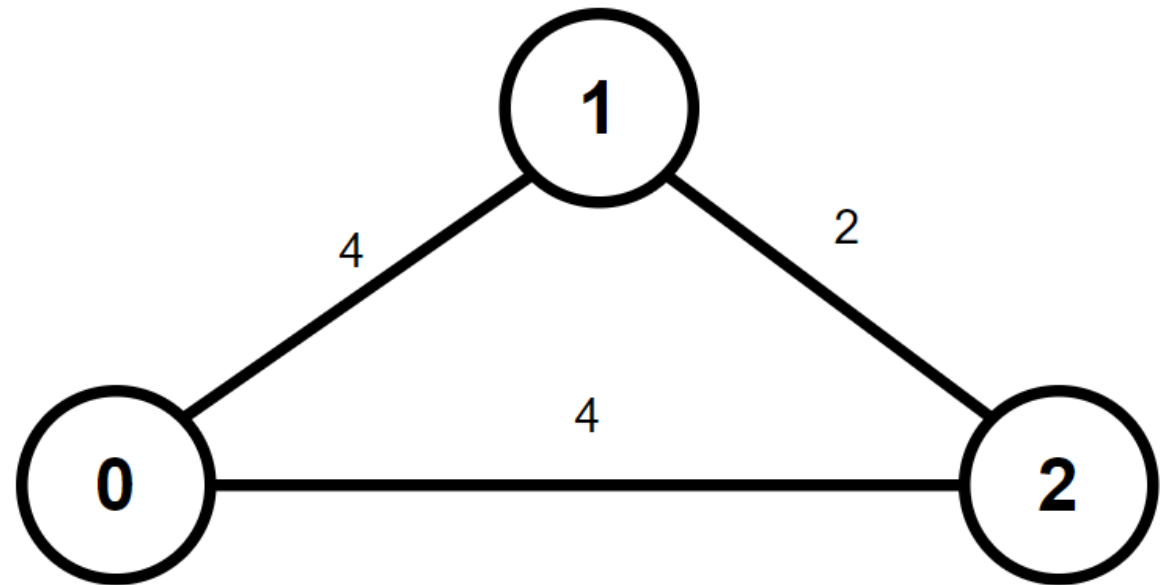
Kruskal's Algorithm is another Greedy Algorithm for MST.

Kruskal's Algorithm tell us to order all the vertices by their edge weights.

(1,2,2)

(0,1,4)

(0,2,4)



MST – Kruskal's Algorithm

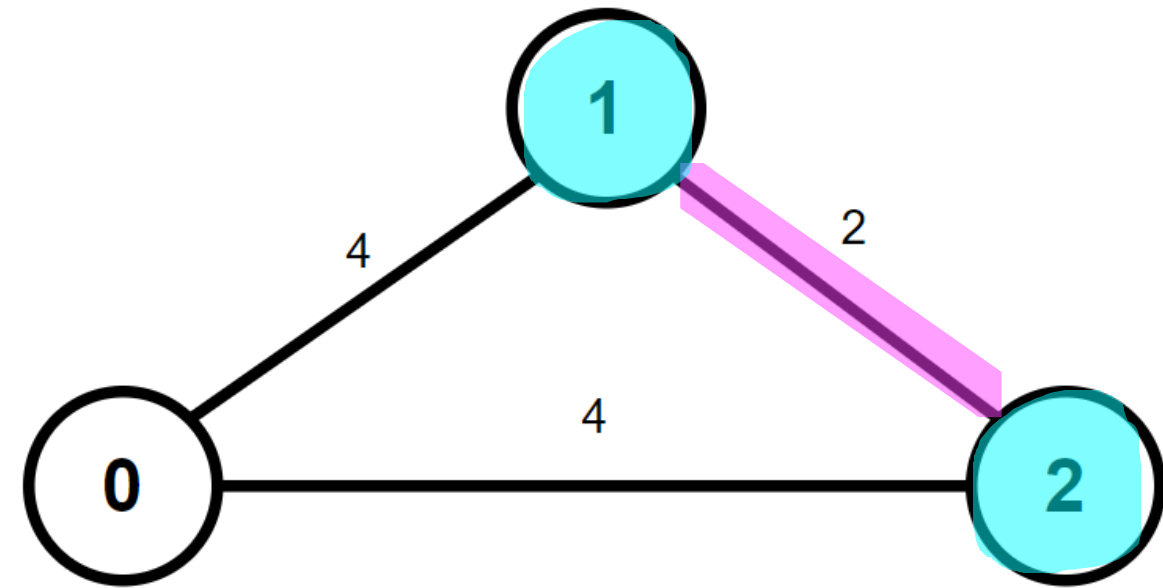
~~(1,2,2)~~

(0,1,4)

(0,2,4)

Kruskal says to pick the edge with the lowest value/weight.

(1,2,2)



MST – Kruskal's Algorithm

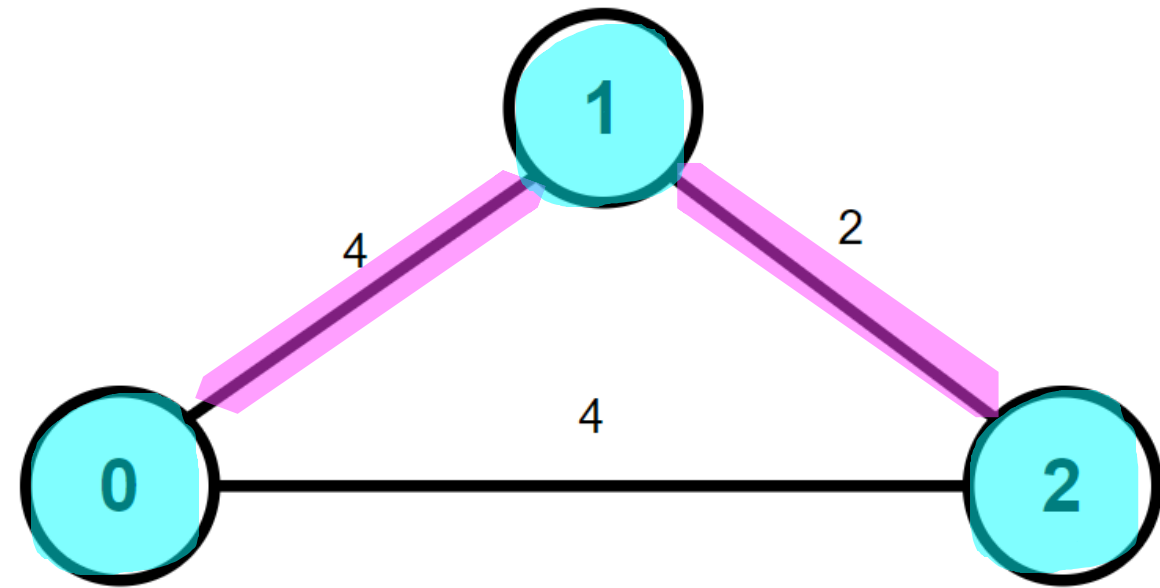
~~(1,2,2)~~

~~(0,1,4)~~

(0,2,4)

Kruskal says to pick the pair with the lowest value.

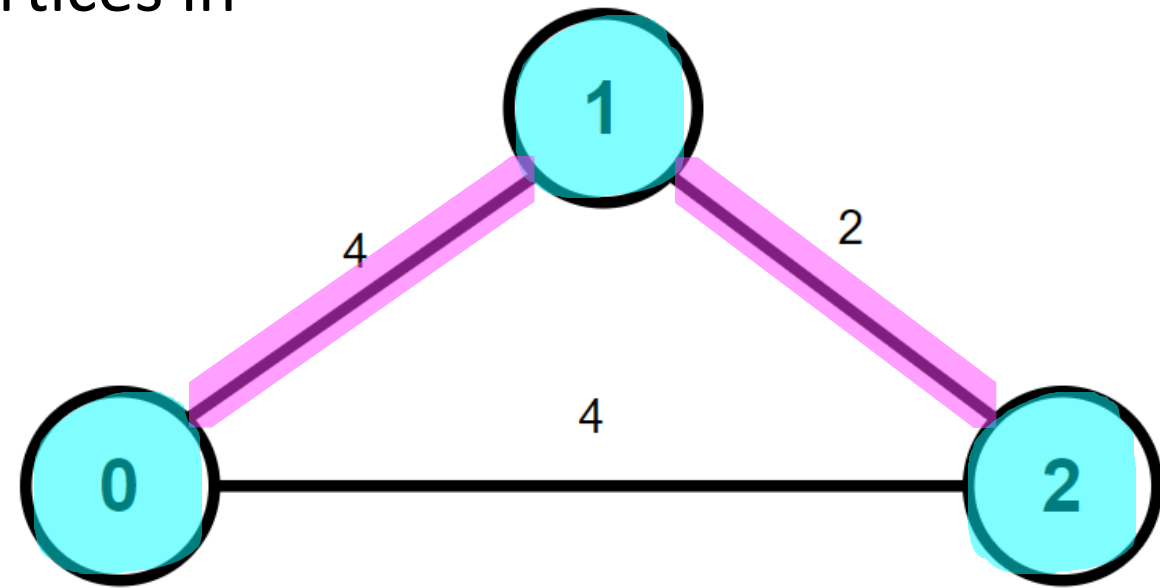
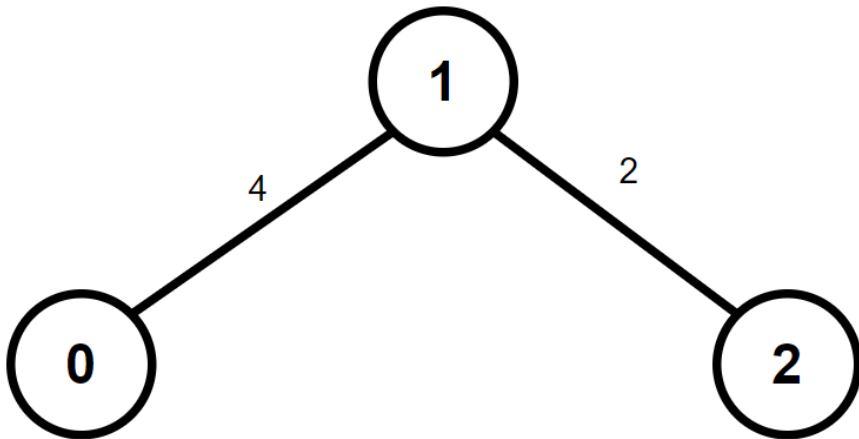
(0,1,4)



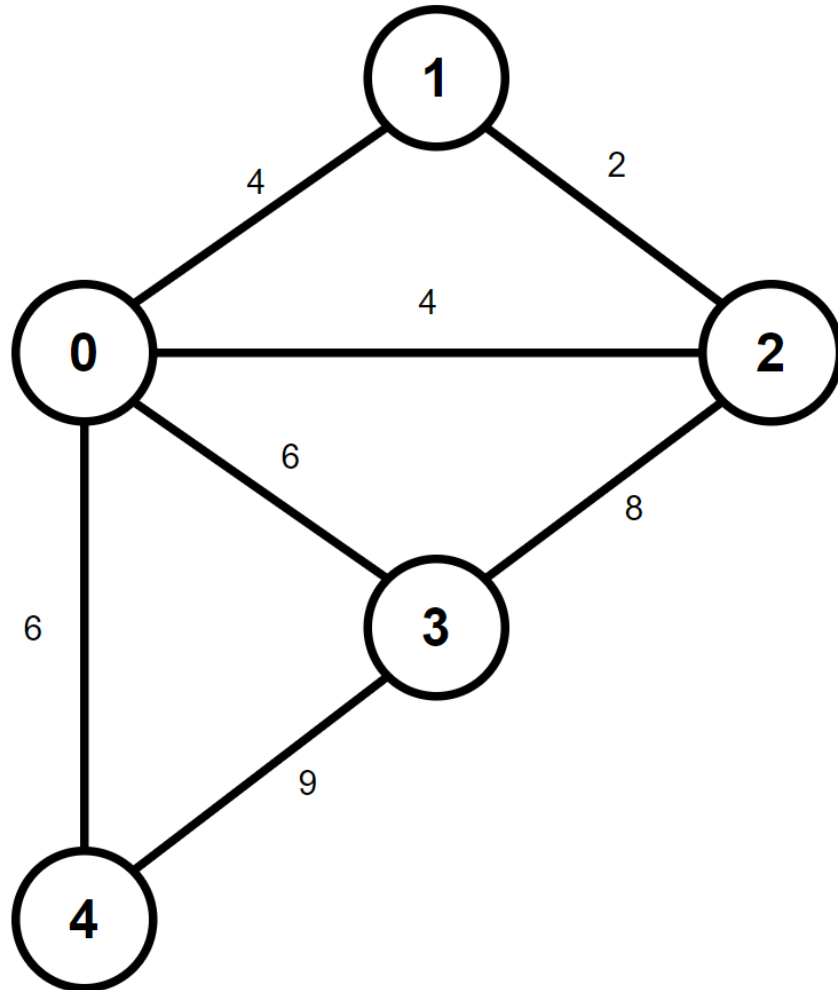
MST – Kruskal's Algorithm

Our graph has 3 vertices and we have 3 vertices in our MST and 2 edges.

We are done.



MST – Kruskal's Algorithm



Order all of the vertex,edge pairs by edge weight.

(0,1,4)

(0,2,4)

(0,3,6)

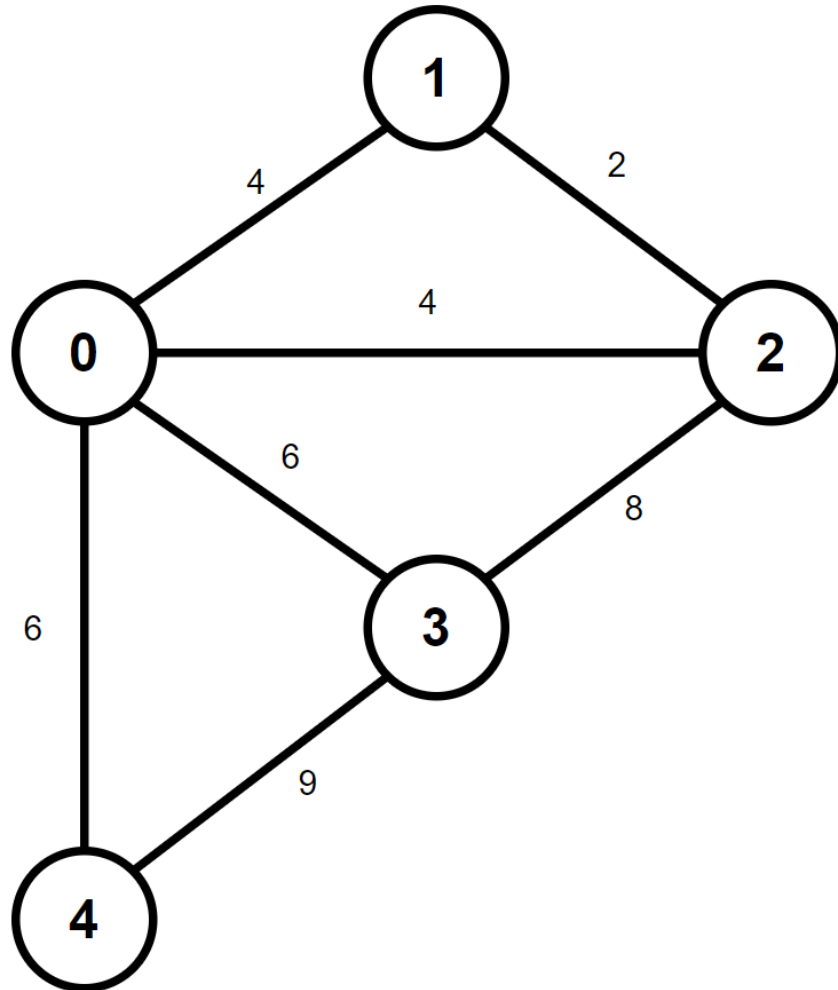
(0,4,6)

(1,2,2)

(2,3,8)

(3,4,9)

MST – Kruskal's Algorithm



Order all of the vertex,edge pairs by edge weight.

(1,2,2)

(0,1,4)

(0,2,4)

(0,3,6)

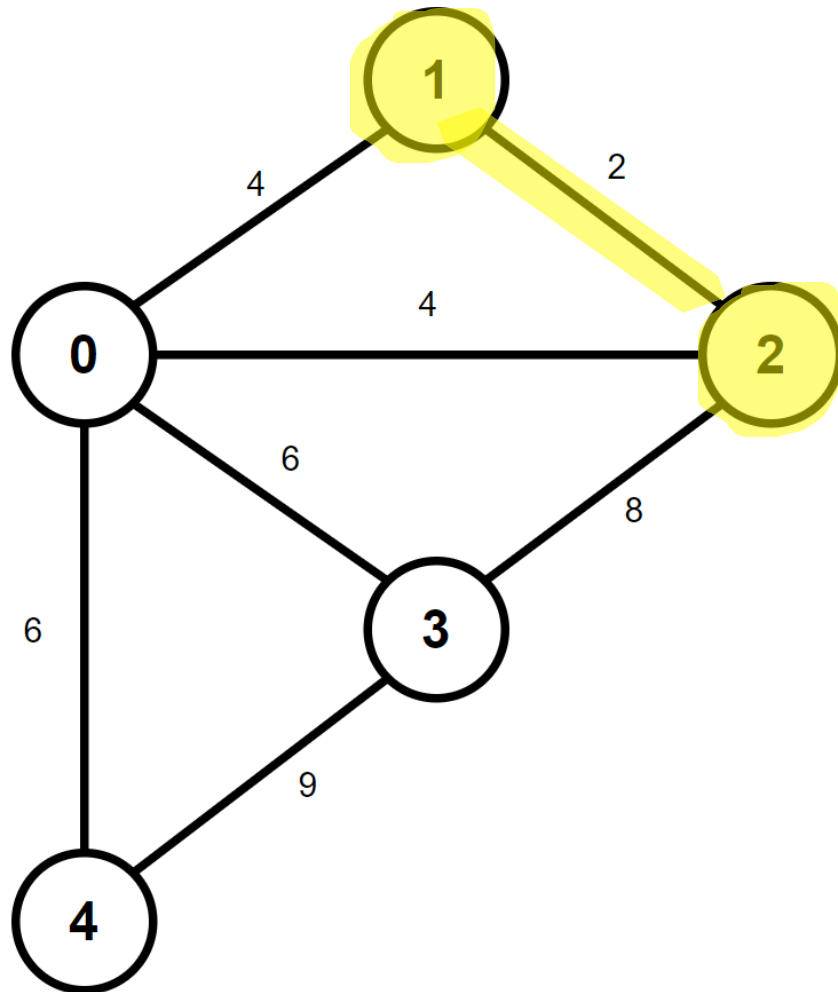
(0,4,6)

(2,3,8)

(3,4,9)

MST – Kruskal's Algorithm

Choose the minimum...



~~(1,2,2)~~

(0,1,4)

(0,2,4)

(0,3,6)

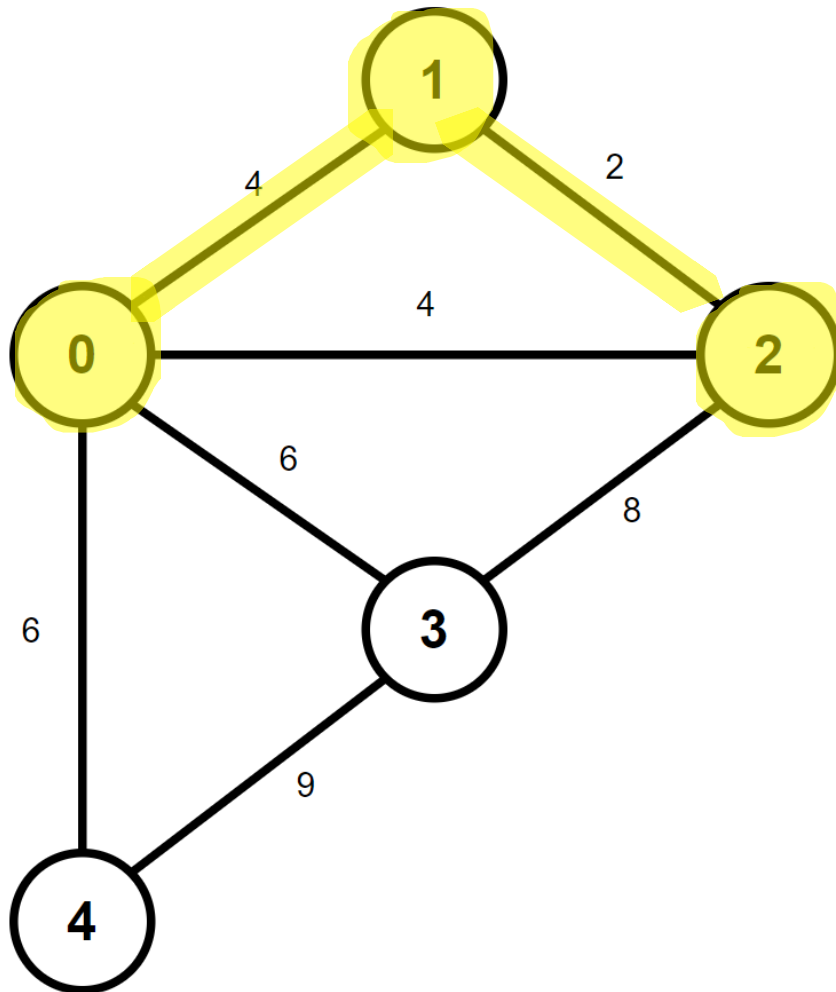
(0,4,6)

(2,3,8)

(3,4,9)

MST – Kruskal's Algorithm

Choose the minimum...



~~(1,2,2)~~

~~(0,1,4)~~

(0,2,4)

(0,3,6)

(0,4,6)

(2,3,8)

(3,4,9)

MST – Kruskal's Algorithm

Choose the minimum...

The next choice is

~~(1,2,2)~~

~~(0,1,4)~~

~~(0,2,4)~~

(0,3,6)

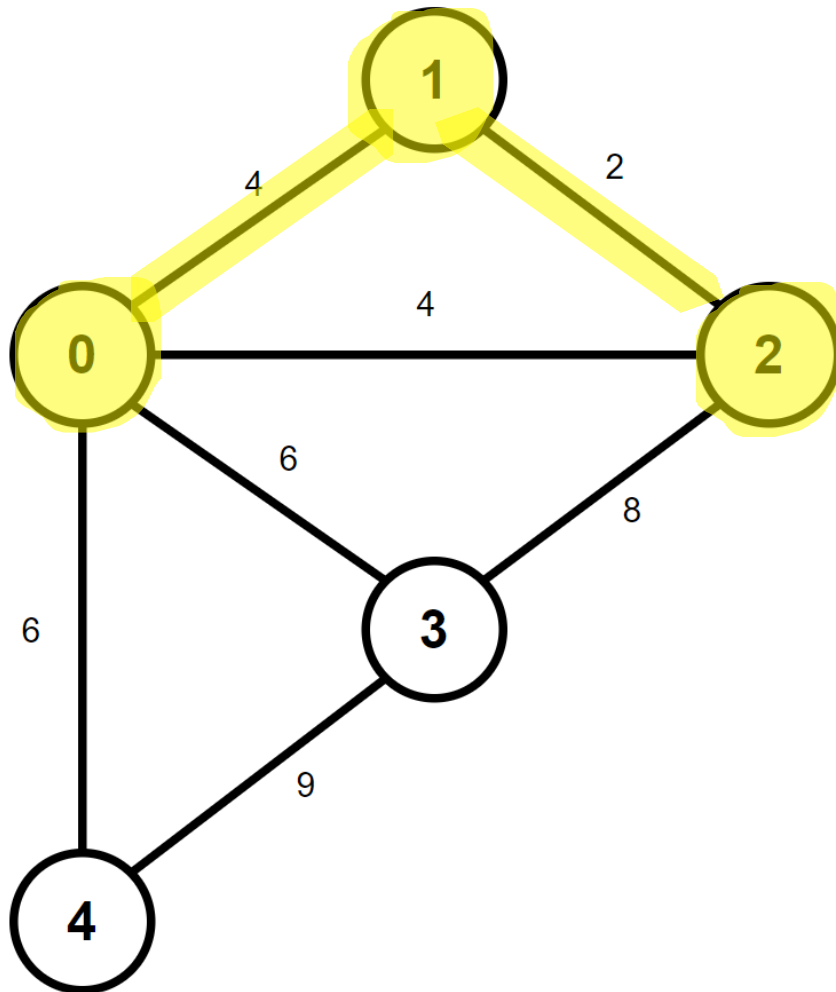
(0,4,6)

(2,3,8)

(3,4,9)

(0,2,4)

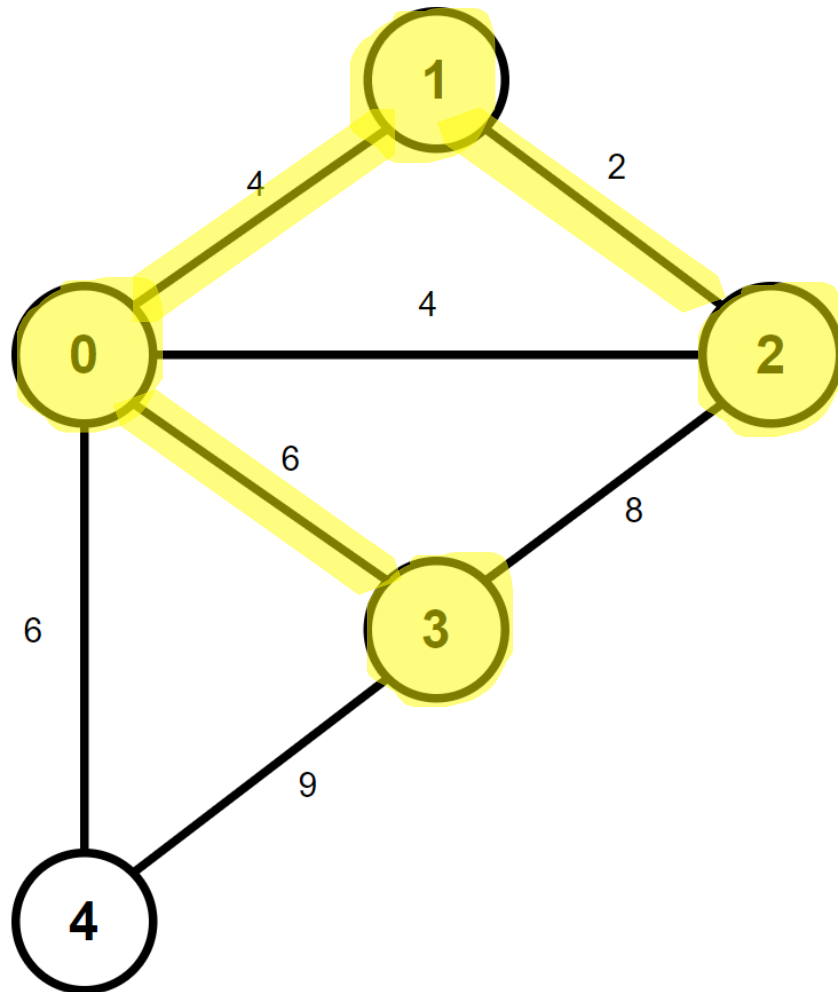
Vertex 0 and Vertex 2 are both already in our MST and adding an edge between them would create a cycle which is not allowed.



MST – Kruskal's Algorithm

Choose the minimum...

The next choice is



~~(1,2,2)~~

~~(0,1,4)~~

~~(0,2,4)~~

~~(0,3,6)~~

(0,4,6)

(2,3,8)

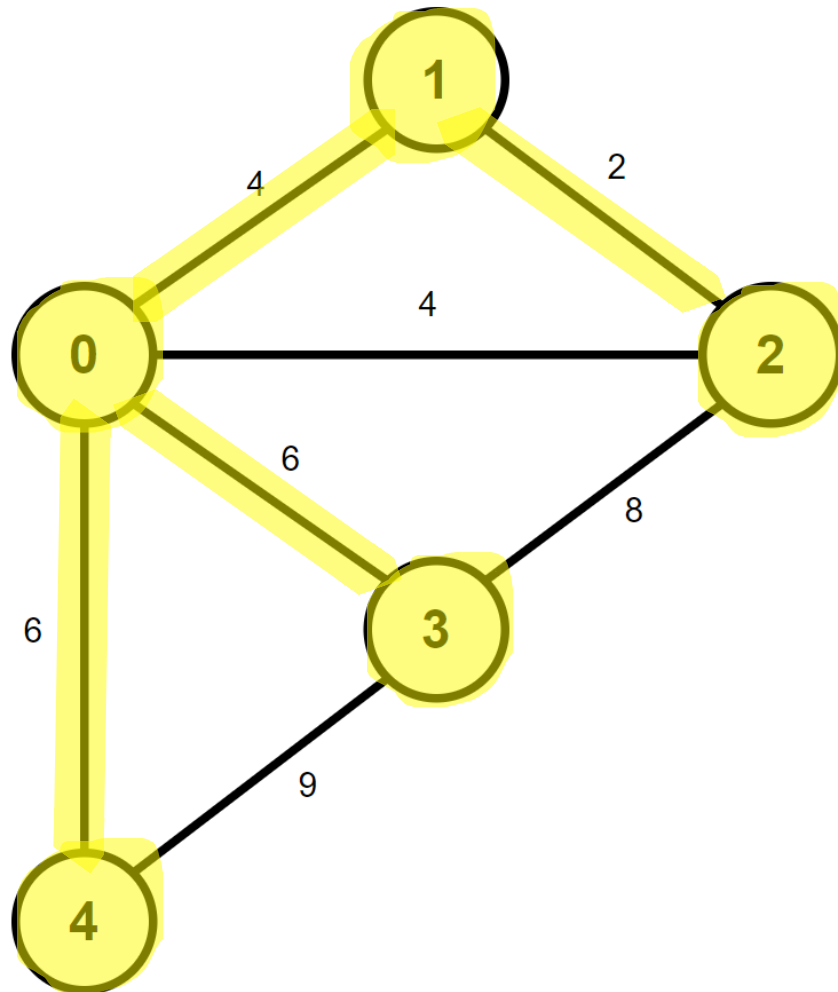
(3,4,9)

(0,3,6)

MST – Kruskal's Algorithm

Choose the minimum...

The next choice is



~~(1,2,2)~~

~~(0,1,4)~~

~~(0,2,4)~~

~~(0,3,6)~~

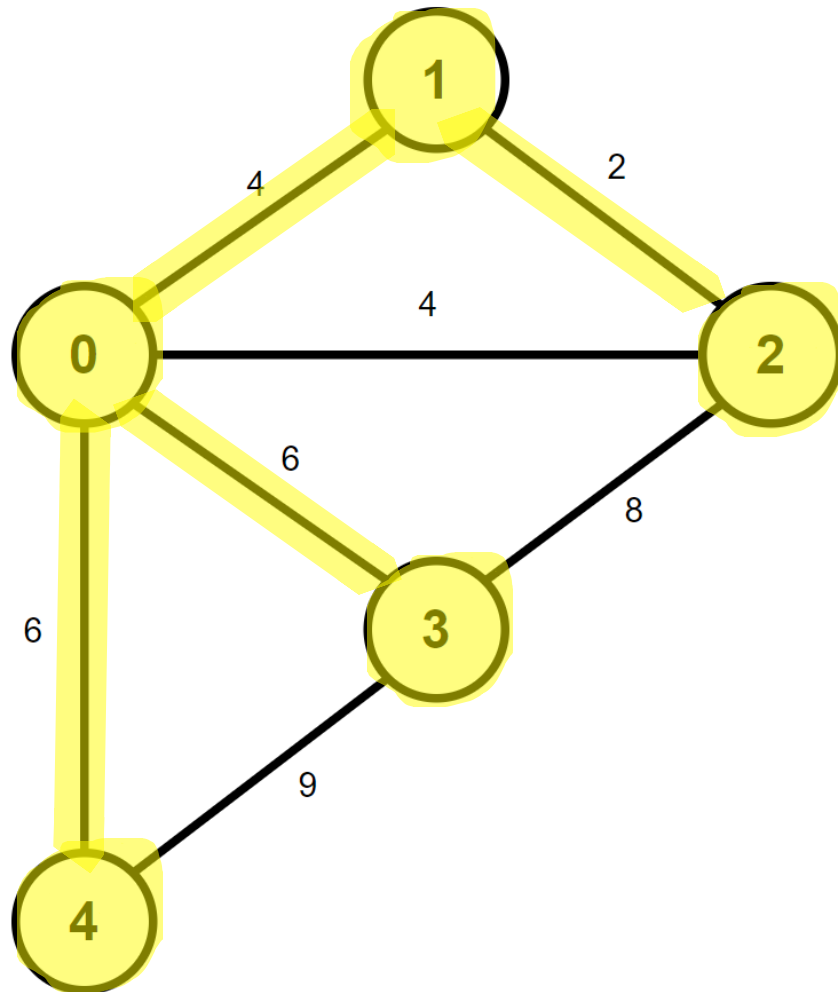
~~(0,4,6)~~

(2,3,8)

(3,4,9)

(0,4,6)

MST – Kruskal's Algorithm



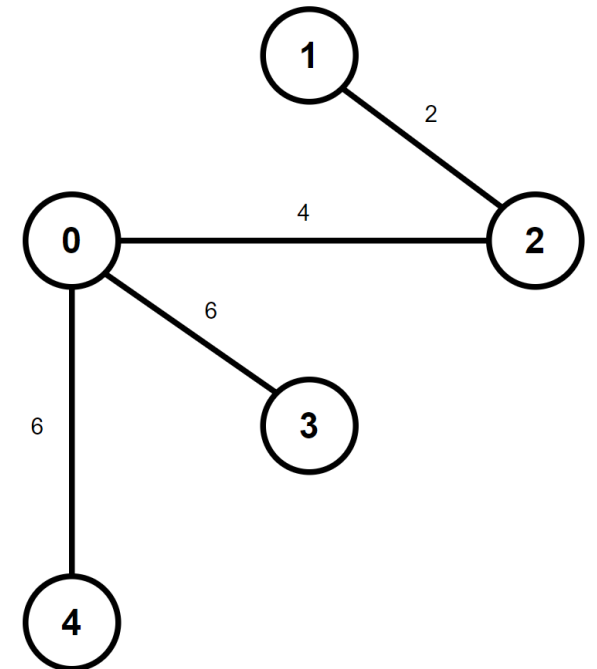
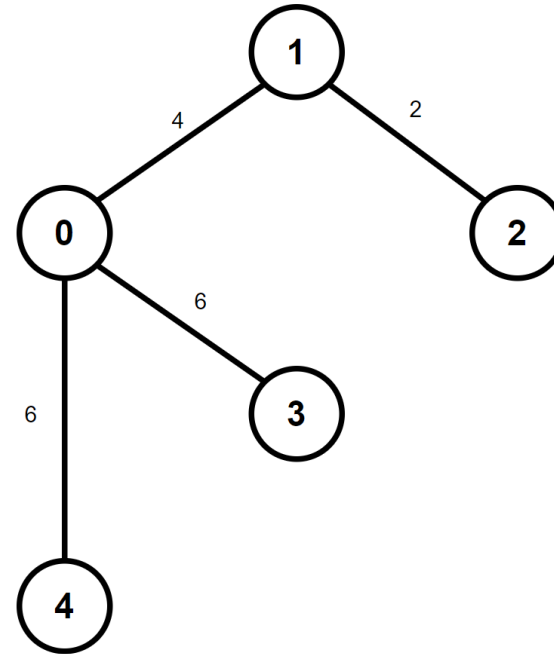
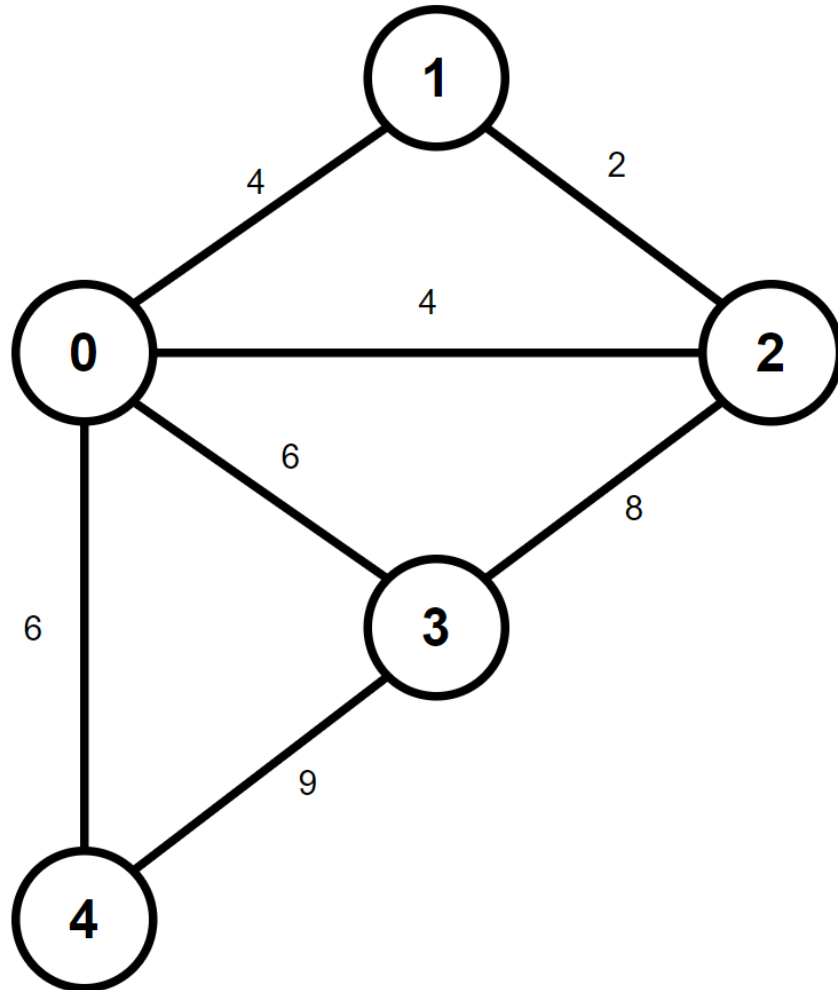
Choose the minimum...

~~(1,2,2)~~
~~(0,1,4)~~
~~(0,2,4)~~
~~(0,3,6)~~
~~(0,4,6)~~
(2,3,8)
(3,4,9)

Now we could either recognize that we have 5 vertices in our graph and 5 vertices (and 4 edges) in our MST so we must be done.

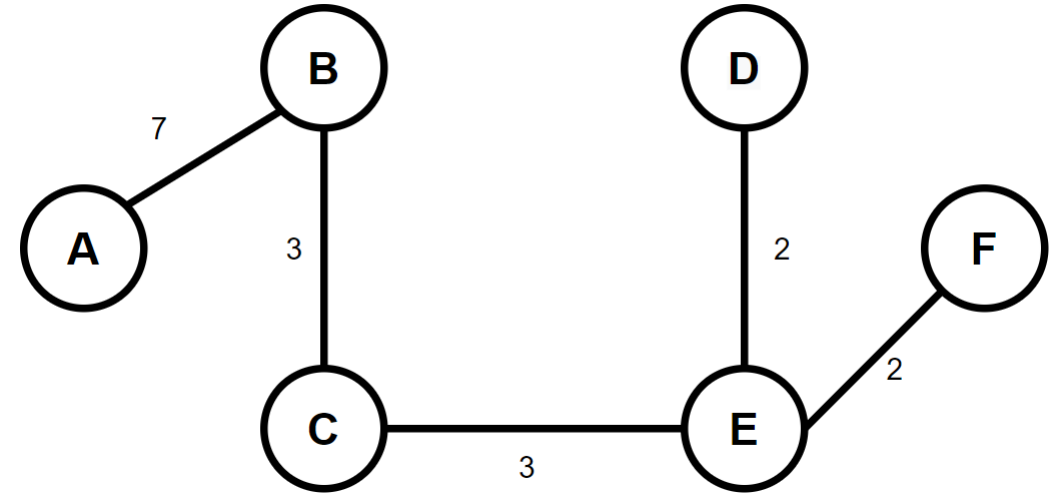
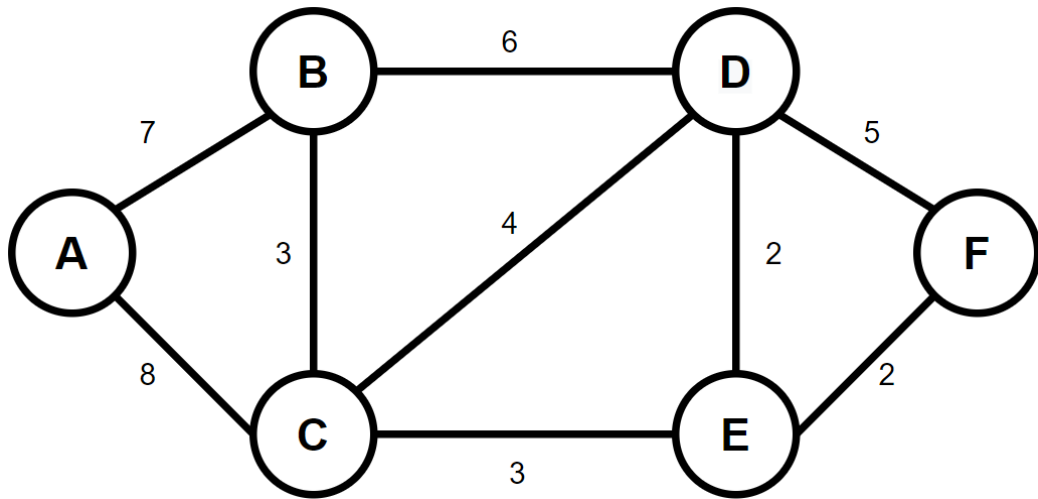
Or we could pick (2,3,8) and discard it because Vertex 2 and 3 are already in the MST. Same for Vertex 3 and Vertex 4 in (3,4,9).

MST – Kruskal's Algorithm

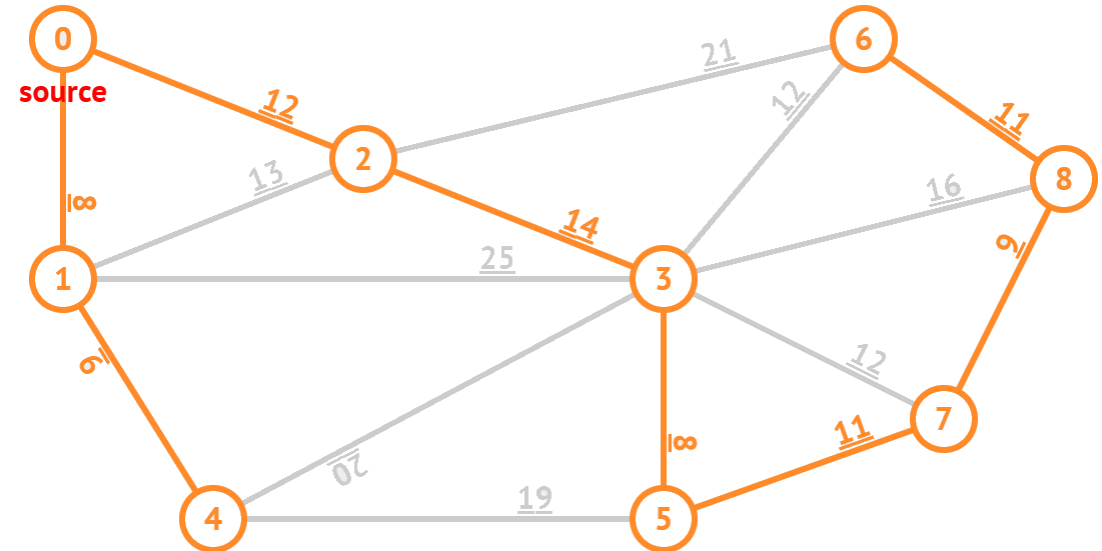
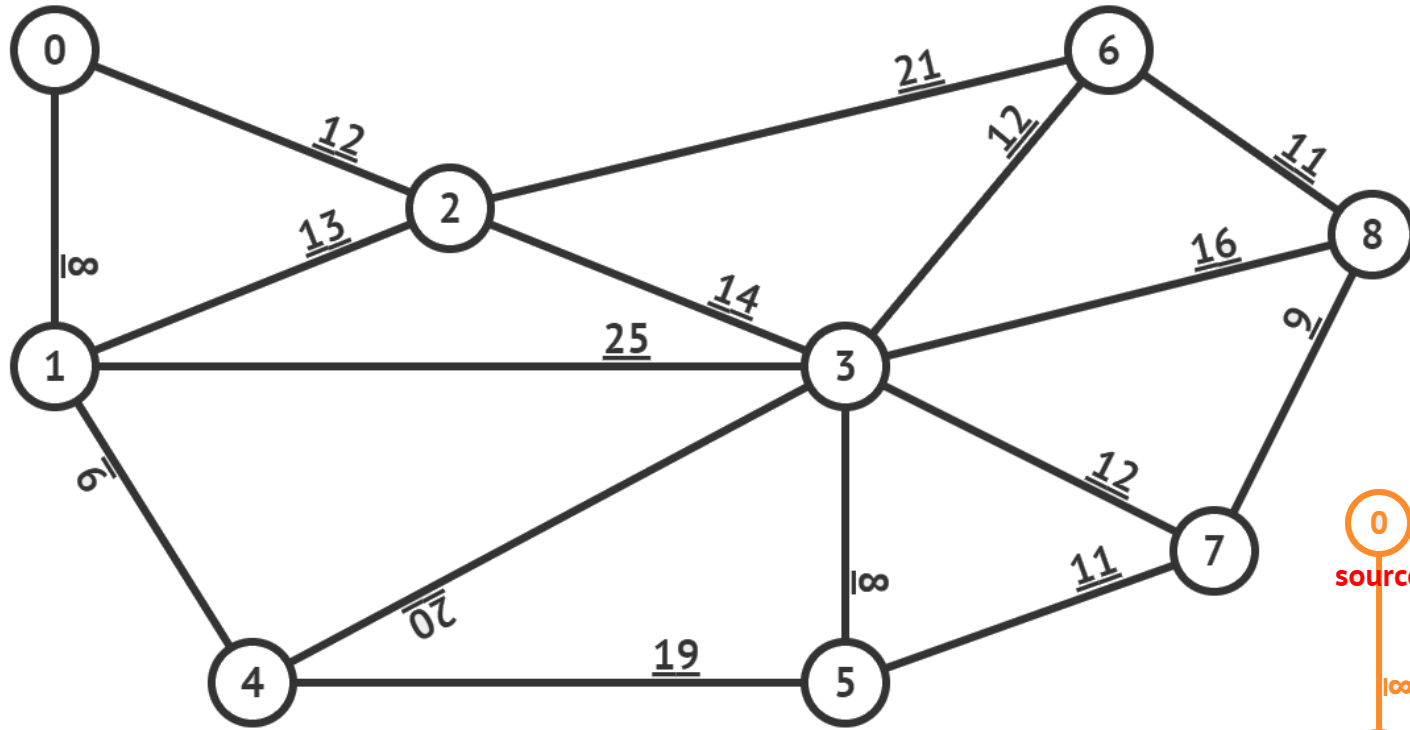


MST – Kruskal's Algorithm

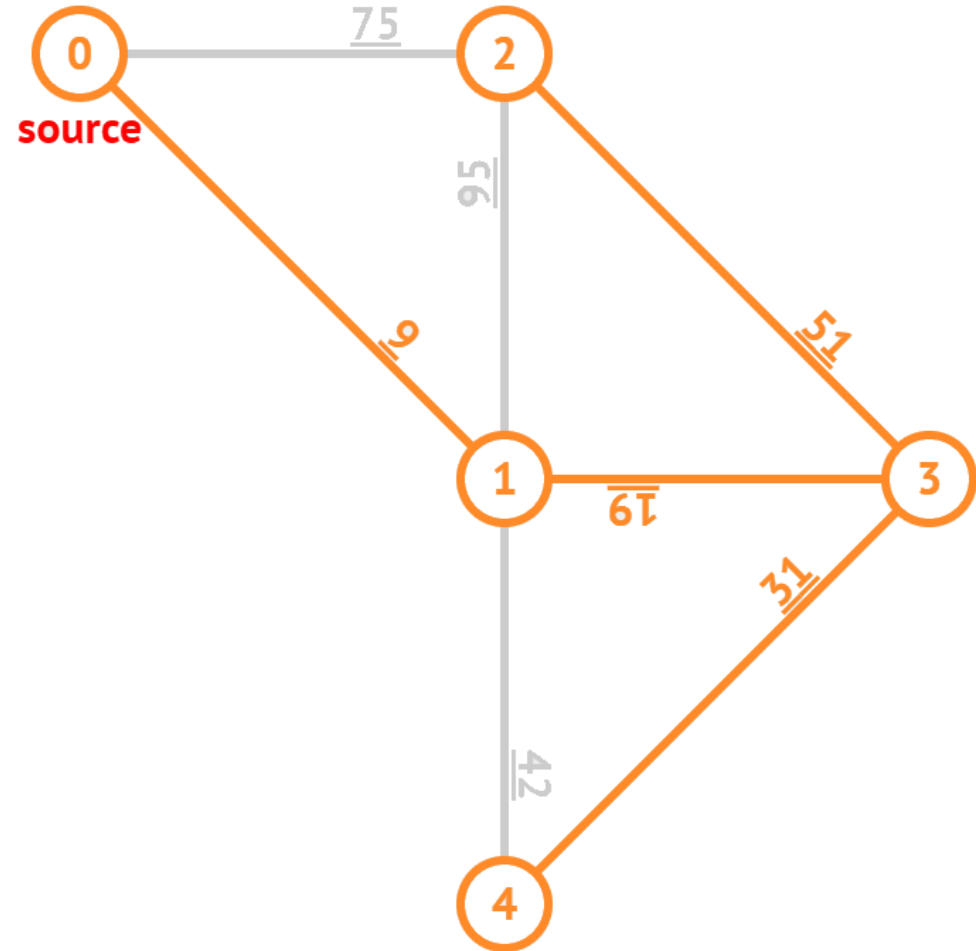
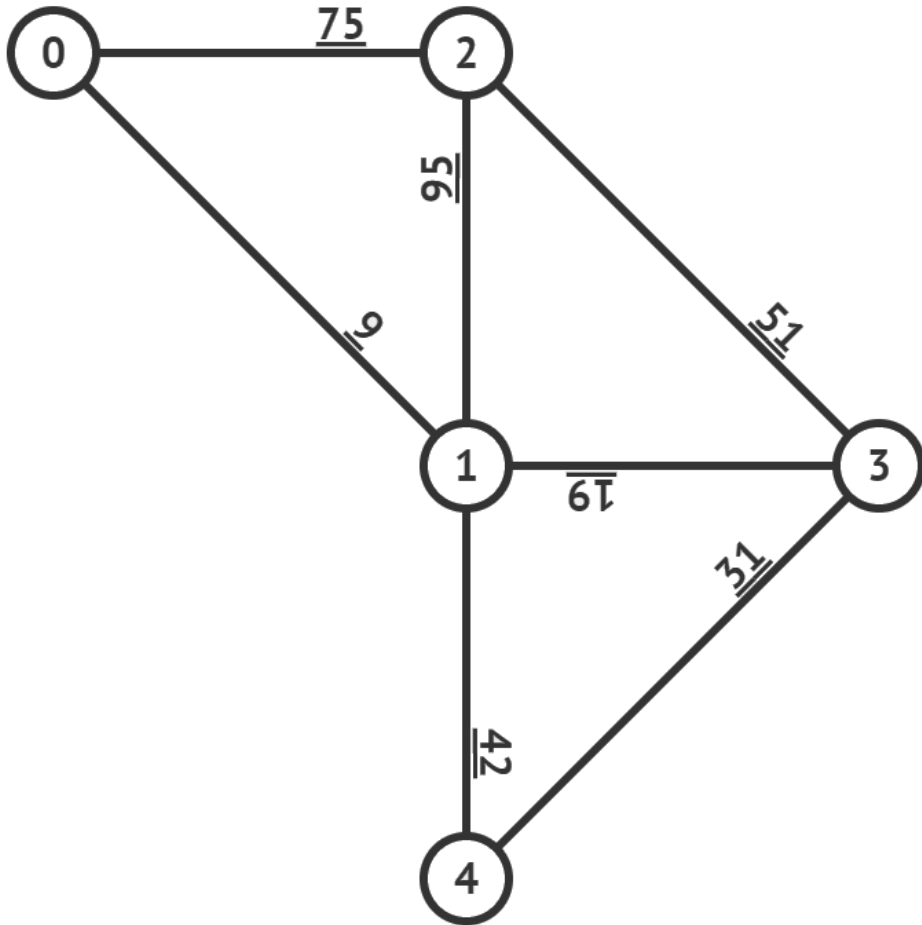
What is the Minimum Spanning Tree for this graph?



MST – Kruskal's Algorithm



MST – Kruskal's Algorithm



Kruskal's Algorithm

What is the run time of Kruskal's Algorithm?

$$O(E \log_2 E)$$

The most time consuming part of the algorithm is the sorting of the edges.

Prim vs Kruskal

You can't really ask which algorithm is better in a given case without considering specific details of problem you are trying to solve.

Use Prim's Algorithm when you have a graph with lots of edges.

Kruskal's Algorithm is general faster when you have a graph with fewer edges.

Prim vs Kruskal

Kruskal

builds a minimum spanning tree by adding one edge at a time. The next line is always the shortest (minimum weight) ONLY if it does NOT create a cycle.

Prims

builds a minimum spanning tree by adding one vertex at a time. The next vertex to be added is always the one nearest to a vertex already on the graph.

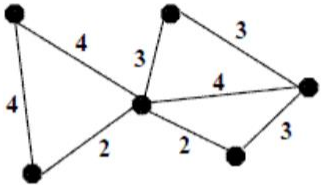

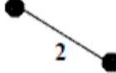
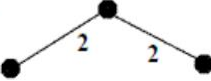
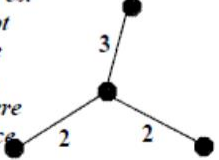
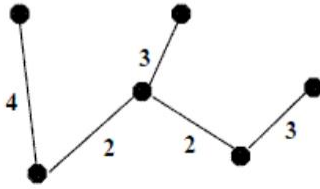
Prim vs Kruskal

Prim always joins a "new" vertex to an "old" vertex, so that every stage is a tree.

Kruskal allows both "new" to "new" and "old" to "old" to get connected, so it risks creating a circuit and must check for them every time.

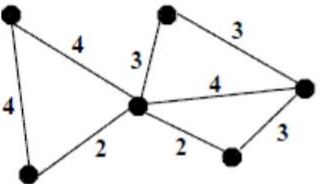

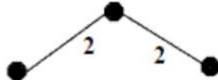
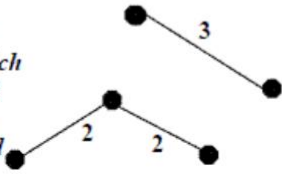
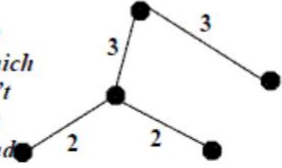
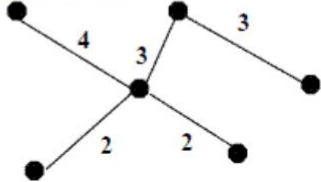
Kruskal has a larger complexity than Prim and could result in an incomplete/unconnected result if interrupted.

Prim's Algorithm

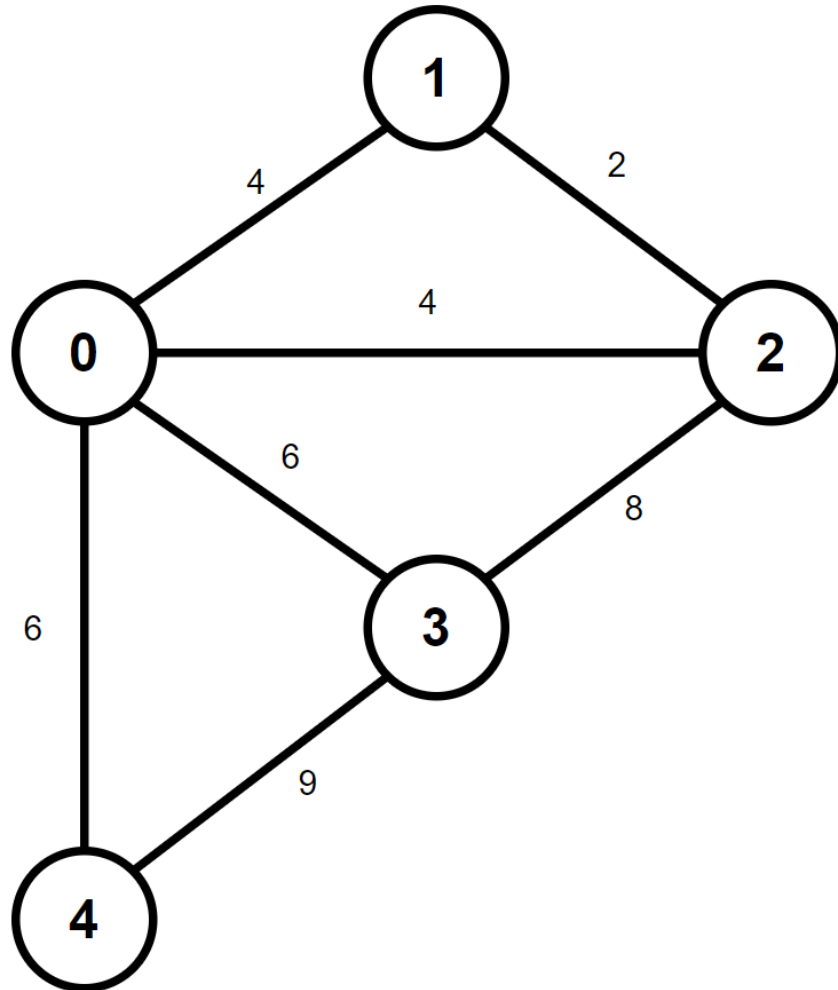
<p>1 Given a network.....</p> 	<p>2 Choose a vertex</p> 	<p>3 Choose the shortest edge from this vertex.</p> 
<p>4 Choose the nearest vertex not yet in the solution.</p> 	<p>5 Choose the next nearest vertex not yet in the solution, when there is a choice choose either.</p> 	<p>6 Repeat until you have a minimal spanning tree.</p> 

Prim vs Kruskal

Kruskal's Algorithm

<p>1 Given a network.....</p> 	<p>2 Choose the shortest edge (if there is more than one, choose any of the shortest).....</p> 	<p>3 Choose the next shortest edge and add it.....</p> 
<p>4 Choose the next shortest edge which wouldn't create a cycle and add it.</p> 	<p>5 Choose the next shortest edge which wouldn't create a cycle and add it.</p> 	<p>6 Repeat until you have a minimal spanning tree.</p> 

MST – Prim's Algorithm



(1,2,2)

(0,1,4)

(0,2,4)

(0,3,6)

(0,4,6)

(2,3,8)

(3,4,9)

Add (0,1,4) to MST

Add (1,2,2) to MST

Remove (0,2,4) – causes cycle

Add (0,3,6) to MST

Add (0,4,6) to MST

Having 5 vertices means we need 4 edges so when we reach 4 edges in our MST, we are done.

The MST has a distance of 18

```
student@Maverick:/media/sf_VM/CSE3318$ ./a.out PrimGraphA.txt
```

```
Enter starting vertex 0
```

```
(0,1,4)
```

```
(0,2,4)
```

```
(0,3,6)
```

```
(0,4,6)
```

```
-----  
Adding (0,1,4) to MST
```

```
(1,2,2)
```

```
(0,2,4)
```

```
(0,3,6)
```

```
(0,4,6)
```

```
-----  
Adding (1,2,2) to MST
```

```
(0,2,4)
```

```
(0,3,6)
```

```
(0,4,6)
```

```
(2,3,8)
```

```
-----  
Removing (0,2,4) - causes cycle
```

```
(0,3,6)
```

```
(0,4,6)
```

```
(2,3,8)
```

```
-----  
Adding (0,3,6) to MST
```

```
(0,4,6)
```

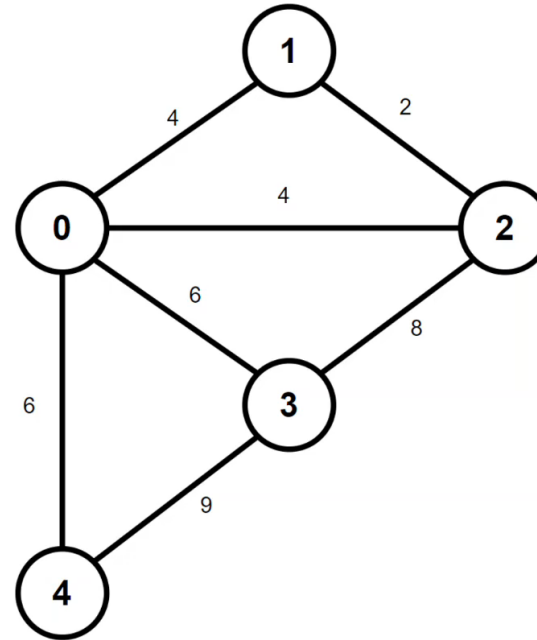
```
(2,3,8)
```

```
(3,4,9)
```

```
-----  
Adding (0,4,6) to MST
```

```
MST distance = 18
```

MST – Prim's Algorithm



-1, 4, 4, 6, 6
4, -1, 2, -1, -1
4, 2, -1, 8, -1
6, -1, 8, -1, 9
6, -1, -1, 9, -1

```
int main(int argc, char *argv[])
{
    int AdjacencyMatrix[MAX][MAX] = {};
    int StartingVertex = 0;
    int VertexCount = 0;

    VertexCount = ReadFile(argc, argv, AdjacencyMatrix);

    printf("Enter starting vertex ");
    scanf("%d", &StartingVertex);

    printf("\n\nMST distance = %d\n",
           Prims(AdjacencyMatrix, VertexCount, StartingVertex));

    return 0;
}
```

```
int row = 0, col = 0;

while (fgets(FileLine, sizeof(FileLine)-1, FH))
{
    token = strtok(FileLine, ",");

    while (token != NULL)
    {
        AdjacencyMatrix[row][col] = atoi(token);
        token = strtok(NULL, ",");
        col++;
    }
    row++;
    col = 0;
}
```

```
-1, 4, 4, 6, 6
4, -1, 2, -1, -1
4, 2, -1, 8, -1
6, -1, 8, -1, 9
6, -1, -1, 9, -1
```



```

int Prims(int AdjacencyMatrix[][MAX], int VertexCount, int StartingVertex)
{
    NODE *LinkedListHead = NULL, *TempPtr = NULL, *NextVertex = NULL;
    int Visited[MAX] = {};
    int MSTLength = 0;
    int EdgeCount = VertexCount - 1;
    int CurrentVertex = StartingVertex;

    Visited[StartingVertex] = 1;

    while(EdgeCount > 0)
    {
        // Create unvisited neighbor list
        // Find edge with the smallest distance that does not create a cycle
        // Add up distance, decrement edges, move current vertex
    }

    return(MSTLength);
}

```

```

typedef struct node
{
    int LE;
    int RE;
    int EdgeDistance;
    struct node *next_ptr;
}
NODE;

```

```
// Create unvisited neighbor list
for(int i = 0; i < VertexCount; i++)
```

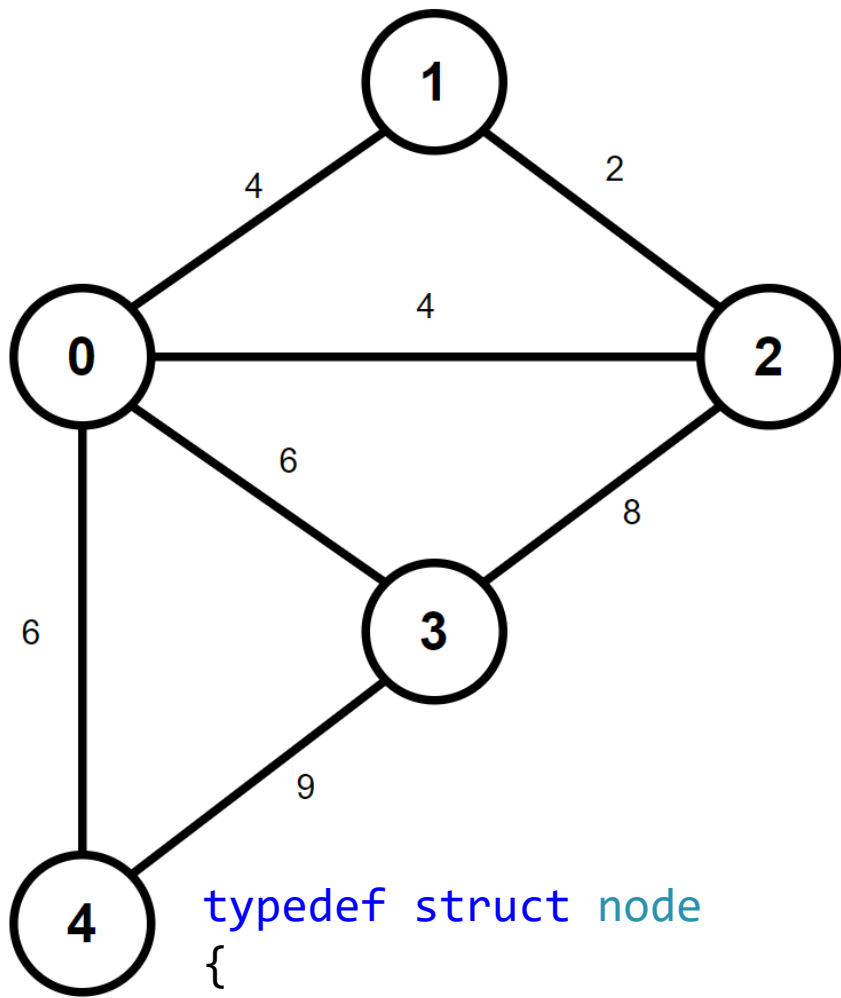
```
{
    if (Visited[i] == 0 &&
        AdjacencyMatrix[CurrentVertex][i] != -1)
    {
        InsertNode(CurrentVertex, i,
                    AdjacencyMatrix[CurrentVertex][i], &LinkedListHead);
    }
}
```

```
DisplayLinkedList(LinkedListHead);
```

-1	4	4	6	6
4	-1	2	-1	-1
4	2	-1	8	-1
6	-1	8	-1	9
6	-1	-1	9	-1

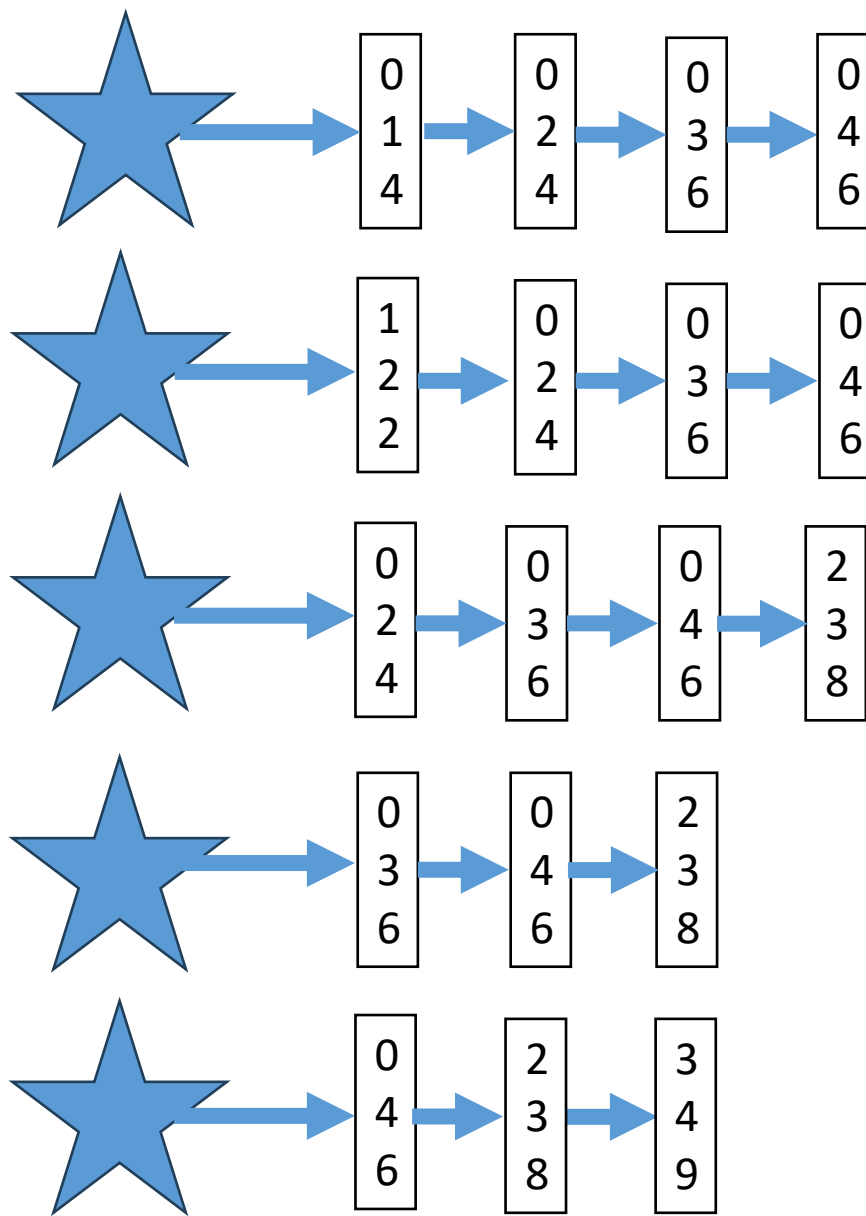
```
student@Maverick:/media/sf_VM/CS
Enter starting vertex 0
(0,1,4)
(0,2,4)
(0,3,6)
(0,4,6)
-----
Adding (0,1,4) to MST
(1,2,2)
(0,2,4)
(0,3,6)
(0,4,6)
-----
Adding (1,2,2) to MST
(0,2,4)
(0,3,6)
(0,4,6)
(2,3,8)
-----
Removing (0,2,4) - causes cycle
(0,3,6)
(0,4,6)
(2,3,8)
-----
Adding (0,3,6) to MST
(0,4,6)
(2,3,8)
(3,4,9)
-----
Adding (0,4,6) to MST

MST distance = 18
```



```

typedef struct node
{
    int LE;
    int RE;
    int EdgeDistance;
    struct node *next_ptr;
}
NODE;
  
```



```

student@maverick:/media/sf_VM/CS
Enter starting vertex 0
(0,1,4)
(0,2,4)
(0,3,6)
(0,4,6)

-----
Adding (0,1,4) to MST
(1,2,2)
(0,2,4)
(0,3,6)
(0,4,6)

-----
Adding (1,2,2) to MST
(0,2,4)
(0,3,6)
(0,4,6)
(2,3,8)

-----
Removing (0,2,4) - causes cycle
(0,3,6)
(0,4,6)
(2,3,8)

-----
Adding (0,3,6) to MST
(0,4,6)
(2,3,8)
(3,4,9)

-----
Adding (0,4,6) to MST

MST distance = 18
  
```

```

// Find edge with the smallest distance that does not create
TempPtr = LinkedListHead;
while (TempPtr != NULL && NextVertex == NULL)
{
    if(Visited[TempPtr->LE] == 1 &&
        Visited[TempPtr->RE] == 0)
    {
        NextVertex = TempPtr;
    }
    else
    {
        printf("Removing (%d,%d,%d) - causes cycle\n",
            LinkedListHead->LE,LinkedListHead->RE,
            LinkedListHead->EdgeDistance);
        DeleteNode(&LinkedListHead);
        DisplayLinkedList(LinkedListHead);
    }
    TempPtr = TempPtr->next_ptr;
}

```

```

student@Maverick:/media/sf_VM/CS
Enter starting vertex 0
(0,1,4)
(0,2,4)
(0,3,6)
(0,4,6)
-----
Adding (0,1,4) to MST
(1,2,2)
(0,2,4)
(0,3,6)
(0,4,6)
-----
Adding (1,2,2) to MST
(0,2,4)
(0,3,6)
(0,4,6)
(2,3,8)
-----
Removing (0,2,4) - causes cycle
(0,3,6)
(0,4,6)
(2,3,8)
-----
Adding (0,3,6) to MST
(0,4,6)
(2,3,8)
(3,4,9)
-----
Adding (0,4,6) to MST

MST distance = 18

```

```
MSTLength += NextVertex->EdgeDistance;
```

```
CurrentVertex = NextVertex->RE;
```

```
Visited[CurrentVertex] = 1;
```

```
EdgeCount--;
```

```
printf("Adding (%d,%d,%d) to MST\n",  
        NextVertex->LE, NextVertex->RE,  
        NextVertex->EdgeDistance);
```

```
NextVertex = NULL;
```

```
DeleteNode(&LinkedListHead);
```

Prim's Algorithm

Implementing Prim using an array as the priority queue

$O(V^2)$

Implementing Prim using a binary heap as the priority queue

$O(E \log_2 V)$

Implementing Prim using a Fibonacci heap as the priority queue

$O(V \log_2 V + E)$

So what is the runtime of using a linked list?

```
while (EdgeCount > 0)  $V - 1$ 
```

$(V - 1)(V)$

```
{
```

```
    // Create unvisited neighbor list
```

```
    for(int i = 0; i < VertexCount; i++)
```

$V^2 - V$

```
    {
```

V

```
    }
```

```
    DisplayLinkedList (LinkedListHead);
```

$O(V^2)$

```
    // Find the edge with the smallest distance
```

```
    TempPtr = LinkedListHead;
```

```
    while (TempPtr != NULL && NextVertex == NULL)
```

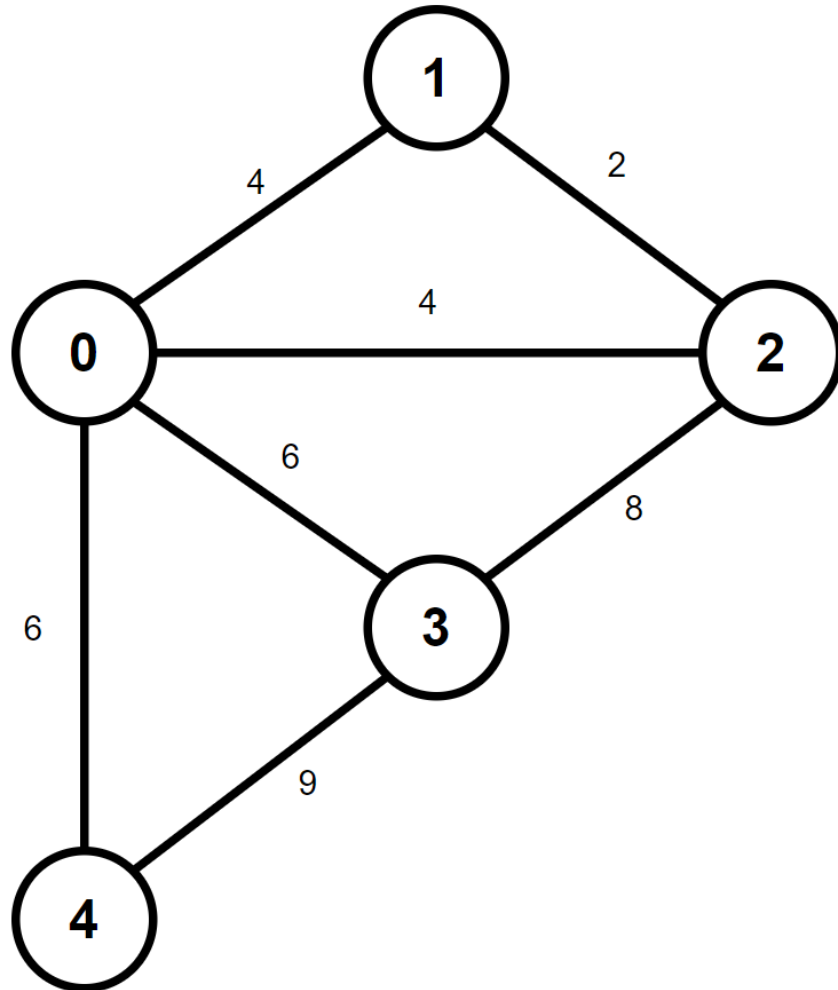
```
    {
```

```
    }
```

```
    //Housekeeping code and deleting the used node
```

```
}
```

MST – Kruskal's Algorithm



List all edges in order by weight and choose the minimum...

(1,2,2)

(0,1,4)

(0,2,4)

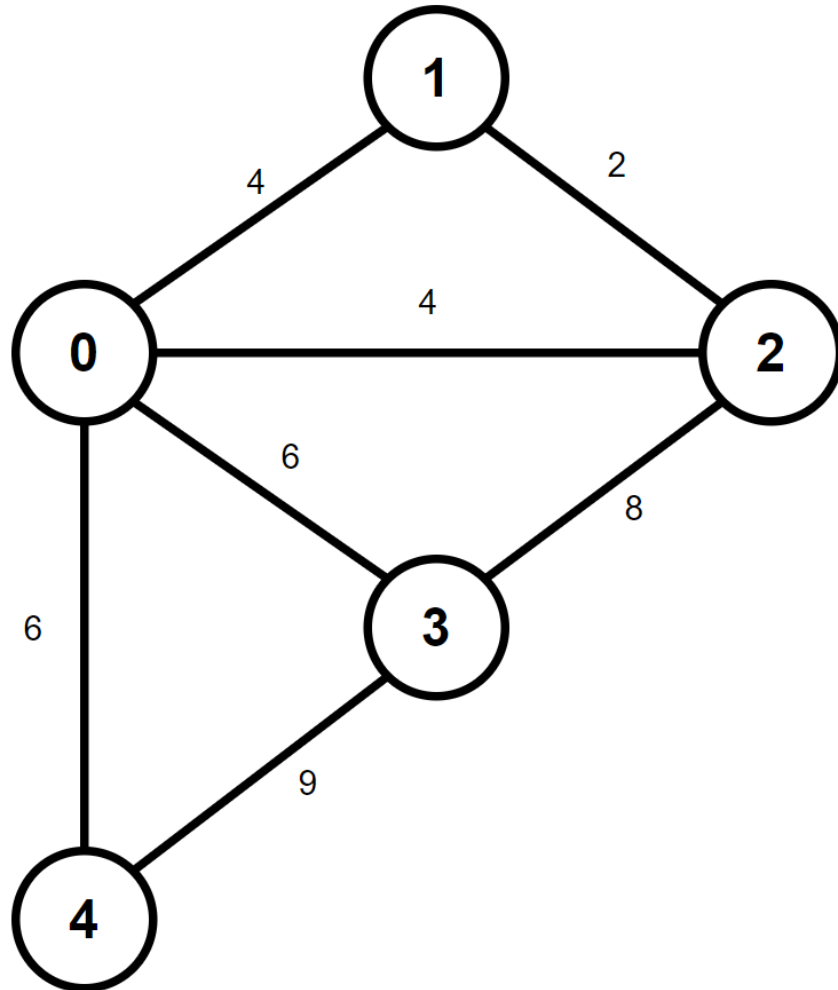
(0,3,6)

(0,4,6)

(2,3,8)

(3,4,9)

MST – Kruskal's Algorithm



List all edges in order by weight and choose the minimum...

(1,2,2)
(0,1,4)
(0,2,4)
(0,3,6)
(0,4,6)
(2,3,8)
(3,4,9)

(1,2,2)
(0,1,4)
(0,2,4)
(0,3,6)
(0,4,6)
(2,3,8)
(3,4,9)

```
(1,2,2)
(0,1,4)
(0,2,4)
(0,3,6)
(0,4,6)
(2,3,8)
(3,4,9)

-----
Adding (1,2,2) to MST
Adding (0,1,4) to MST
Removing (0,2,4) - causes cycle
(0,3,6)
(0,4,6)
(2,3,8)
(3,4,9)

-----
Adding (0,3,6) to MST
Adding (0,4,6) to MST

MST distance = 18
```

```
for (int i = 0; i < VertexCount; i++)
{
    // Create neighbor list
    for(int j = 0; j < VertexCount; j++)
    {
        if (Visited[j] == 0 && AdjacencyMatrix[i][j] != -1)
        {
            InsertNode(i,j, AdjacencyMatrix[i][j], &LinkedListHead);
        }
    }
    Visited[i] = 1;
}
```

```
memset(Visited, 0, sizeof(Visited));
```

```
DisplayLinkedList(LinkedListHead);
```

```
while (EdgeCount > 0)
{
    // Find the edge with the smallest distance

    MSTLength += NextVertex->EdgeDistance;
    EdgeCount--;
    printf("Adding (%d,%d,%d) to MST\n",
           NextVertex->LE, NextVertex->RE, NextVertex->EdgeDistance);
    NextVertex = NULL;
    DeleteNode(&LinkedListHead);
}
```

```
TempPtr = LinkedListHead;
while (TempPtr != NULL && NextVertex == NULL)
{
    if (Visited[TempPtr->LE] == 1 &&
        Visited[TempPtr->RE] == 1)
    {
        printf("Removing (%d,%d,%d) - causes cycle\n",
            LinkedListHead->LE,
            LinkedListHead->RE, LinkedListHead->EdgeDistance);
        DeleteNode(&LinkedListHead);
        DisplayLinkedList(LinkedListHead);
    }
    else
    {
        NextVertex = TempPtr;
        Visited[TempPtr->RE] = 1;
        Visited[TempPtr->LE] = 1;
    }
    TempPtr = TempPtr->next_ptr;
}
```

Dynamic Programming

Let's look at a familiar recursive problem

Calculate the n th Fibonacci value.

The recursive algorithm is

```
if  $n \leq 2$ , then set  $f = n$   
else  $f = \text{fib}(n-1) + \text{fib}(n-2)$   
return  $f$ 
```

n	Fibonacci
0	0
1	1
2	1
3	2
4	3
5	5
6	8
7	13
8	21
9	34
10	55
11	89

Dynamic Programming

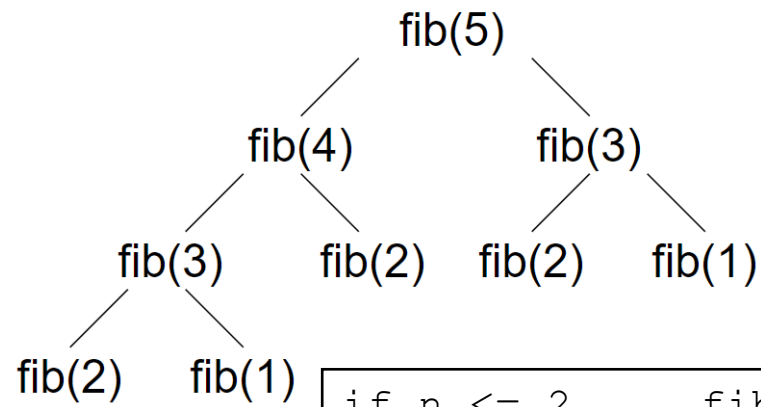
if $n \leq 2$,

then set $f = n$

else $f = \text{fib}(n-1) + \text{fib}(n-2)$

return f

$\text{fib}(5)$



```

if n <= 2      fib(5)
  return 1
else
  return fib(n-1) +
    fib(n-2)
  
```

```

if n <= 2      fib(4)
  return 1
else
  return fib(n-1) +
    fib(n-2)
  
```

```

if n <= 2      fib(3)
  return 1
else
  return fib(n-1) +
    fib(n-2)
  
```

```

if n <= 2      fib(3)
  return 1
else
  return fib(n-1) +
    fib(n-2)
  
```

```

if n <= 2      fib(2)
  return 1
else
  return fib(n-1) +
    fib(n-2)
  
```

```

if n <= 2      fib(2)
  return 1
else
  return fib(n-1) +
    fib(n-2)
  
```

```

if n <= 2      fib(1)
  return 1
else
  return fib(n-1) +
    fib(n-2)
  
```

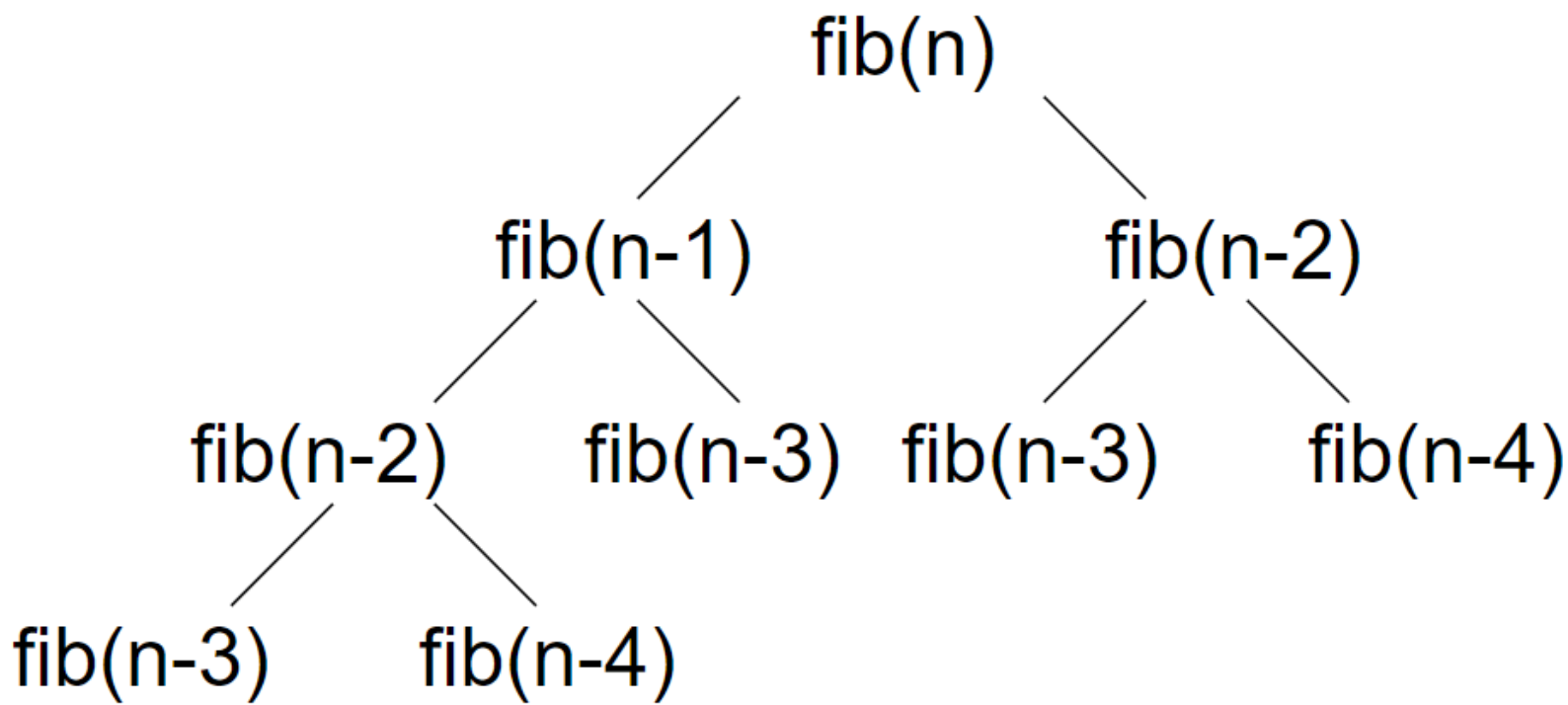
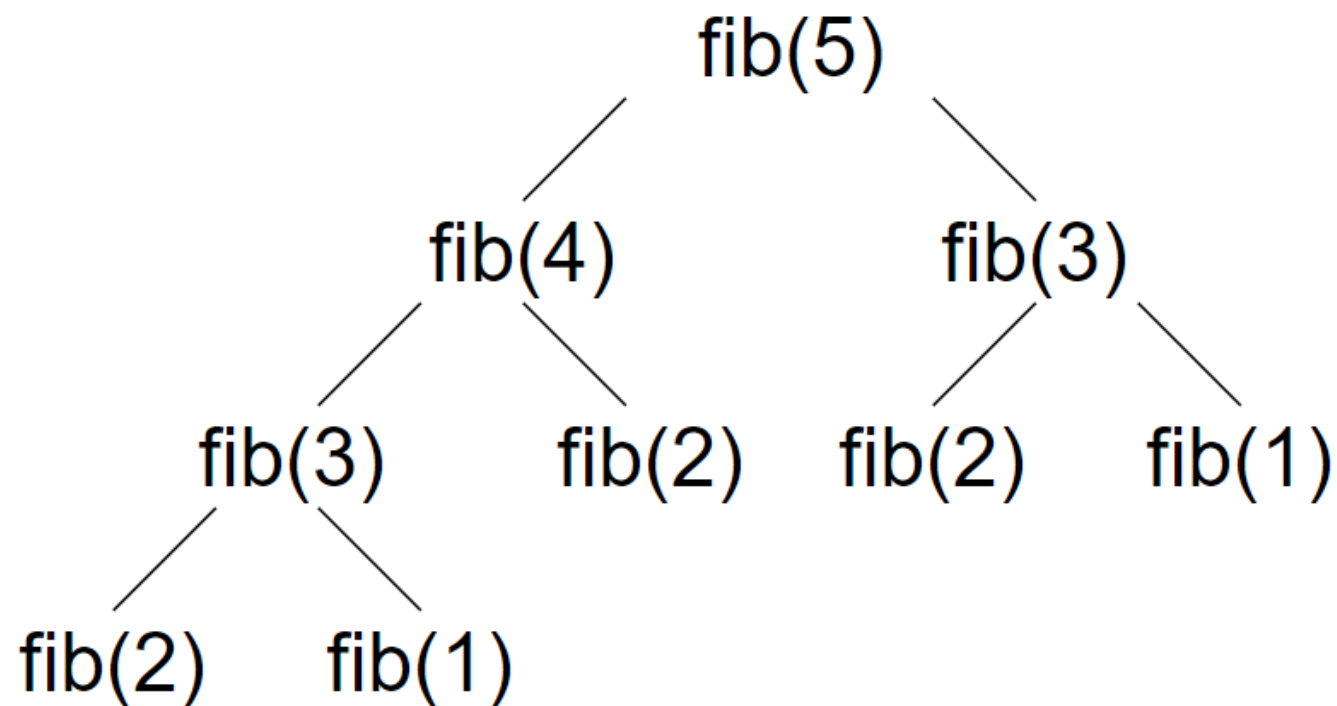
```

if n <= 2      fib(2)
  return 1
else
  return fib(n-1) +
    fib(n-2)
  
```

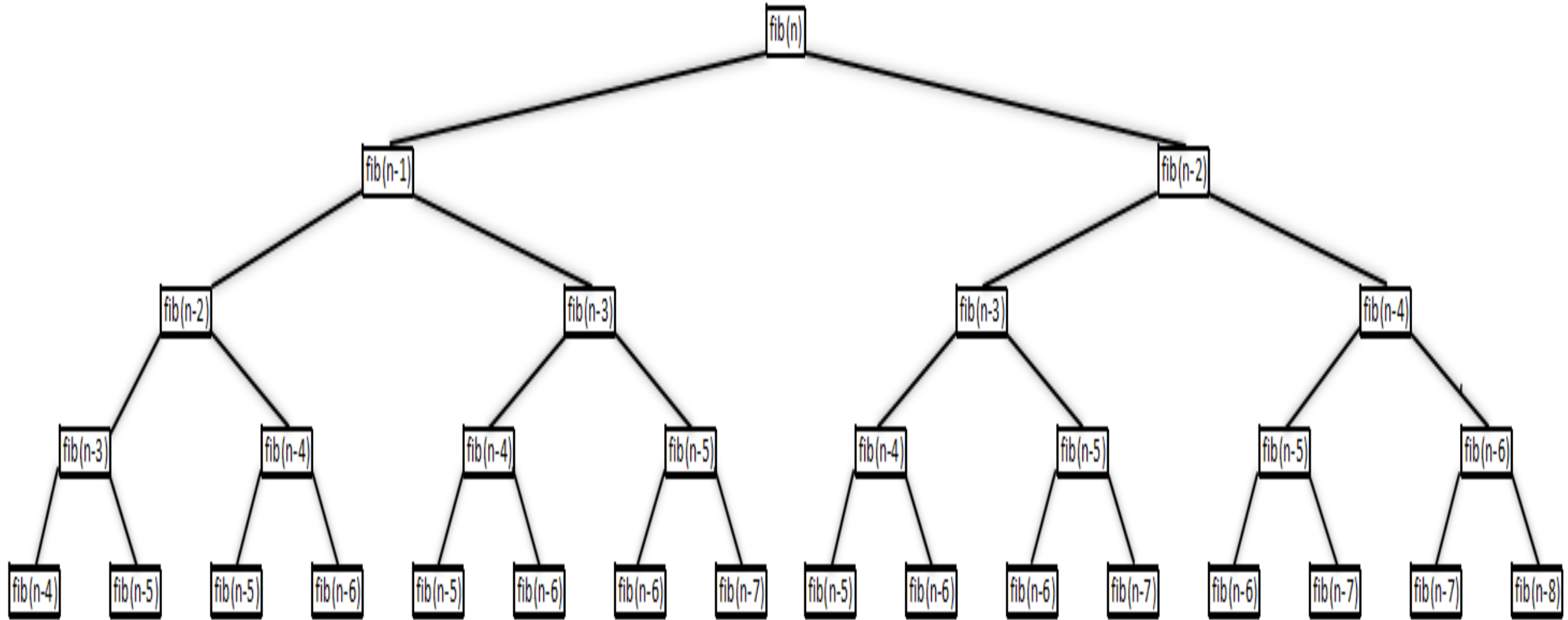
```

if n <= 2      fib(1)
  return 1
else
  return fib(n-1) +
    fib(n-2)
  
```

Dynamic Programming



Dynamic Programming



Dynamic Programming

General, powerful algorithm design technique

Dynamic Programming is approximately "careful brute force"

Trying to get to polynomial time

Break a problem into subproblems, solve those subproblems and reuse those solutions.

Dynamic Programming

Dynamic Programming (DP) is an algorithmic technique for solving an optimization problem by breaking it down into simpler subproblems and utilizing the fact that the optimal solution to the overall problem depends upon the optimal solution to its subproblems.

Dynamic Programming

There are two patterns used in solving DP problems

1. Tabulation – Bottom up
2. Memoization – Top Down

Dynamic Programming

MEMOIZATION

MEMOIZATION?

MEMORIZATION?



Dynamic Programming

There are two patterns used in solving DP problems

1. Tabulation – Bottom up

I will study the theory of Dynamic Programming in CSE3318, then I will practice some problems using DP and hence I will master Dynamic Programming.

2. Memoization – Top Down

To master Dynamic Programming, I would have to practice DP problems and to practice problems, I would have to study some theory of Dynamic Programming from CSE3318.

Dynamic Programming

Tabulation Method – Bottom Up Dynamic Programming

As the name itself suggests starting from the bottom and cumulating answers to the top.

Let's apply this method to the Fibonacci problem.

What is the bottom/base case of Fibonacci?

Dynamic Programming

Tabulation Method – Bottom Up Dynamic Programming

What is the bottom/base case of Fibonacci?

Let's look at the classic definition of Fibonacci...

if $n \leq 2$,
then set $f = n$

else $f = \text{fib}(n-1) + \text{fib}(n-2)$

return f

So for $n = 0$ or $n = 1$ or $n = 2$,
 n is always returned

Dynamic Programming

Tabulation Method – Bottom Up Dynamic Programming

	n	Fibonacci
	0	0
	1	1
if (n == 0 n == 1 n == 2)	2	1
{	3	2
return n;	4	3
}	5	5
	6	8
	7	13
	8	21
	9	34
	10	55
	11	89

Dynamic Programming

Tabulation Method – Bottom Up Dynamic Programming

```
if (n == 0 || n == 1)
{
    return n;
}

for (i = 2; i <= n; i++)
{

```

n	Fibonacci
0	0
1	1
2	1
3	2
4	3
5	5
6	8
7	13
8	21
9	34
10	55
11	89

Dynamic Programming

Tabulation Method – Bottom Up Dynamic Programming

```
if (n == 0 || n == 1)
{
    return n;
}

for (i = 2; i <= n; i++)
{
}
```

n	Fib	
0	0	
1	1	
2	1	0 + 1
3	2	1 + 1
4	3	1 + 2
5	5	2 + 3
6	8	3 + 5

Dynamic Programming

Tabulation Method – Bottom Up Dynamic Programming

```
if (n == 0 || n == 1)
{
    return n;
}

int First = 0, Second = 1;
for (i = 2; i <=n; i++)
{
    Fib = First + Second;
    First = Second;
    Second = Fib;
}

return Second;
```

n	Fib	
0	0	
1	1	
2	1	0 + 1
3	2	1 + 1
4	3	1 + 2
5	5	2 + 3
6	8	3 + 5

Dynamic Programming

Tabulation Method – Bottom Up Dynamic Programming

```
if (n == 0 || n == 1)
{
    return n;
}
int First = 0, Second = 1;
for (i = 2; i <=n; i++)
{
    Fib = First + Second;
    First = Second;
    Second = Fib;
}
return Fib;
```

n = 5

	1 st	2 nd	Fib
0		1	1
1		2	2
2		3	3
3		5	5

n	Fib	
0	0	
1	1	
2	1	0 + 1
3	2	1 + 1
4	3	1 + 2
5	5	2 + 3
6	8	3 + 5

Dynamic Programming

Memoization

In computing, memoization or memoisation is an optimization technique used primarily to speed up computer programs by storing the results of expensive function calls and returning the stored result when the same inputs occur again.

Dynamic Programming

```
unsigned long long int fibonacci(unsigned int n)
{
    if (n == 0 || n == 1)
    {
        return n;
    }
    else
    {
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}
```

fib(5)

fib(3) = 2

fib(4) = 3

fib(5) = 5

Dynamic Programming

fib(5)

Dynamic Programming

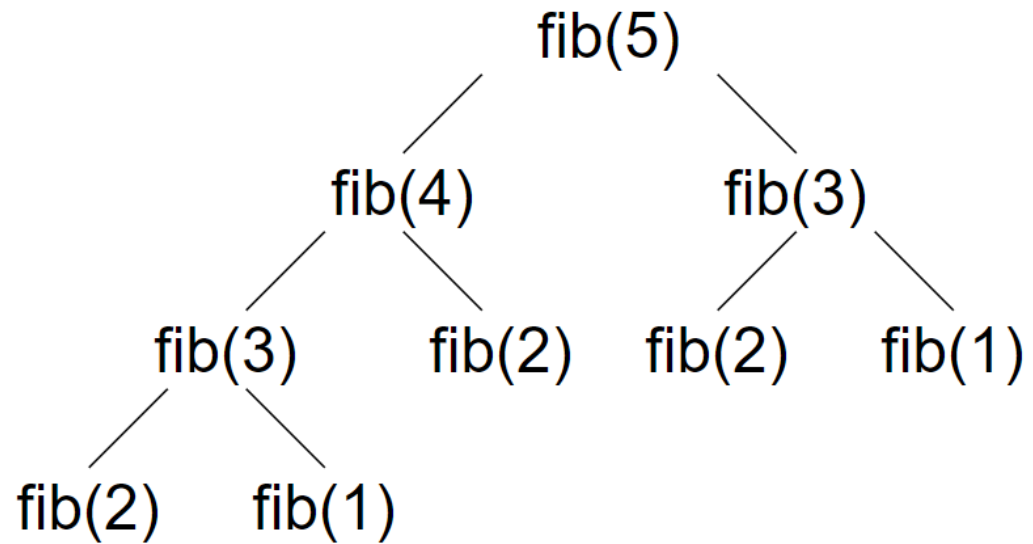
The memoized program for a problem is like the recursive version with a small modification that it looks into a lookup table before computing solutions.

We initialize a lookup array with all initial values as NULL.

Whenever we need the solution to a subproblem, we first look into the lookup table.

If the precomputed value is there, then we return that value; otherwise, we calculate the value and put the result in the lookup table so that it can be reused later.

```
unsigned long long int fibonacci(unsigned int n)
{
    if (n == 0 || n == 1)
    {
        return n;
    }
    else
    {
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}
```



```

unsigned long long fibonacci(unsigned long long n, unsigned long long Memo[])
{
    unsigned long long result = 0;

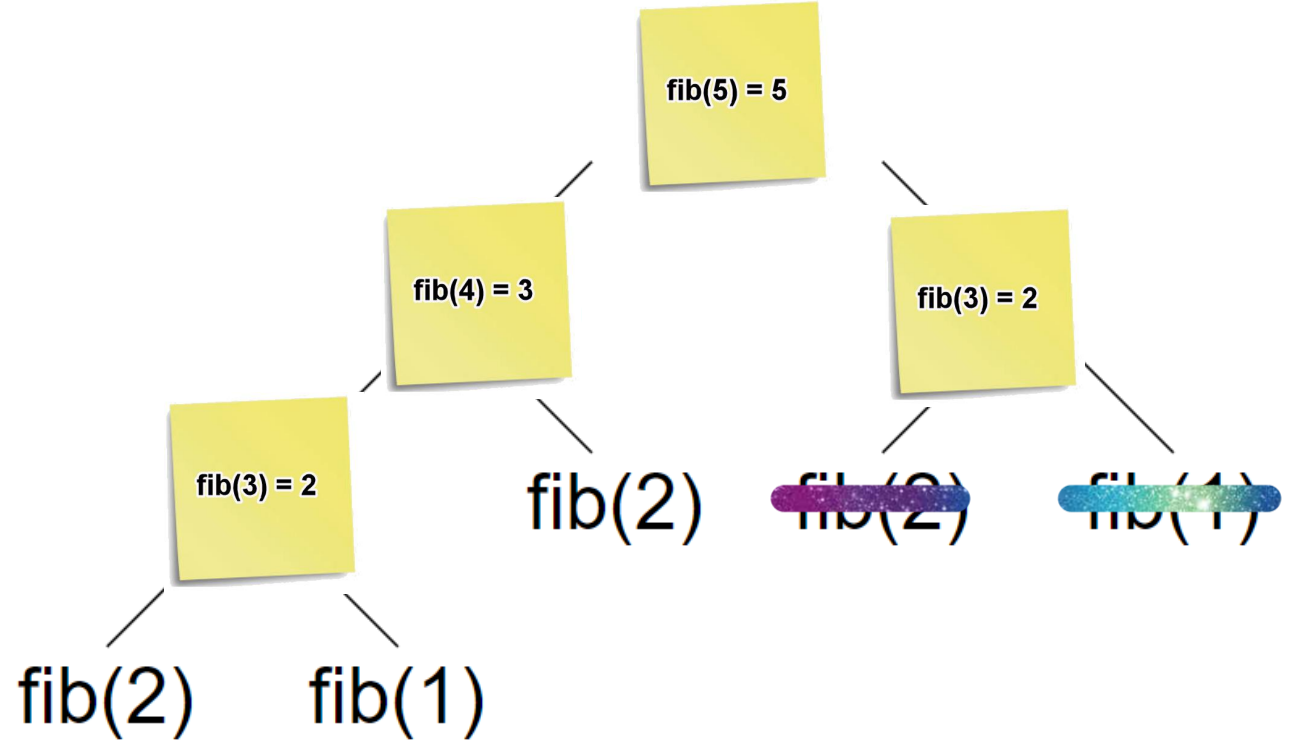
    if (n == 1 || n == 2)
    {
        result = 1;
    }
    else if (Memo[n] != '\0')
    {
        result = Memo[n];
    }
    else
    {
        result = fibonacci(n - 1, Memo) + fibonacci(n - 2, Memo);
        Memo[n] = result;
    }
    return result;
}

```

```

graph TD
    fib5["fib(5) = 5"] --> fib4["fib(4) = 3"]
    fib5 --> fib3_2["fib(3) = 2"]
    fib4 --> fib3_2_2["fib(3) = 2"]
    fib4 --> fib2_1["fib(2)"]
    fib3_2 --> fib2_2["fib(2)"]
    fib3_2 --> fib1_1["fib(1)"]
    fib2_1 --> fib2_3["fib(2)"]
    fib2_1 --> fib1_2["fib(1)"]
    fib2_2 --> fib2_4["fib(2)"]
    fib2_2 --> fib1_3["fib(1)"]

```



Dynamic Programming

There are two main properties of a problem that suggests it can be solved using Dynamic Programming.

1. Overlapping Subproblems
2. Optimal Substructure

Dynamic Programming

Overlapping Subproblems

Like Divide and Conquer, Dynamic Programming combines solutions to sub-problems.

Dynamic Programming is mainly used when solutions of same subproblems are needed again and again.

Dynamic Programming

In dynamic programming, computed solutions to subproblems are stored/memoized so that these don't have to be recomputed.

Dynamic Programming is not useful when there are no common (overlapping) subproblems because there is no point storing the solutions if they are not needed again.

For example, Binary Search doesn't have common subproblems.

Dynamic Programming

Optimal Substructure

A given problem has Optimal Substructure Property if the optimal solution of the given problem can be obtained by using optimal solutions of its subproblems.

Greedy Algorithms have the same property.



Dynamic Programming

Optimal Substructure

A given problem has Optimal Substructure Property if the optimal solution of the given problem can be obtained by using optimal solutions of its subproblems.

DP techniques exploit this property to split the problems into smaller subproblems and solve them instead.

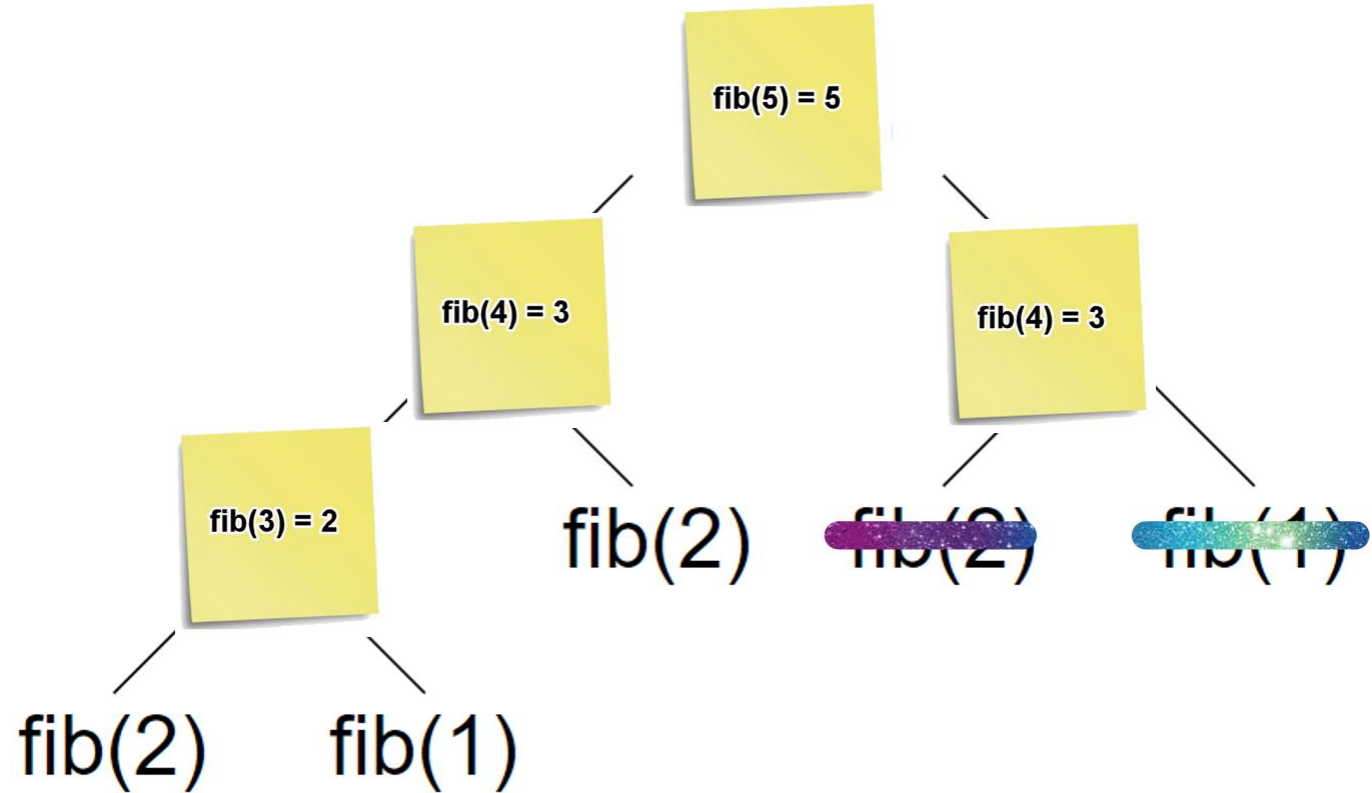
After the subproblems become sufficiently small, solving them becomes trivial.

Dynamic Programming

DP \approx recursion + memoization

Memoize (take a memo) and re-use solutions to subproblems that help solve the problem.

For any given value k , $\text{fib}(k)$ only recurses the first time it is called.



Dynamic Programming

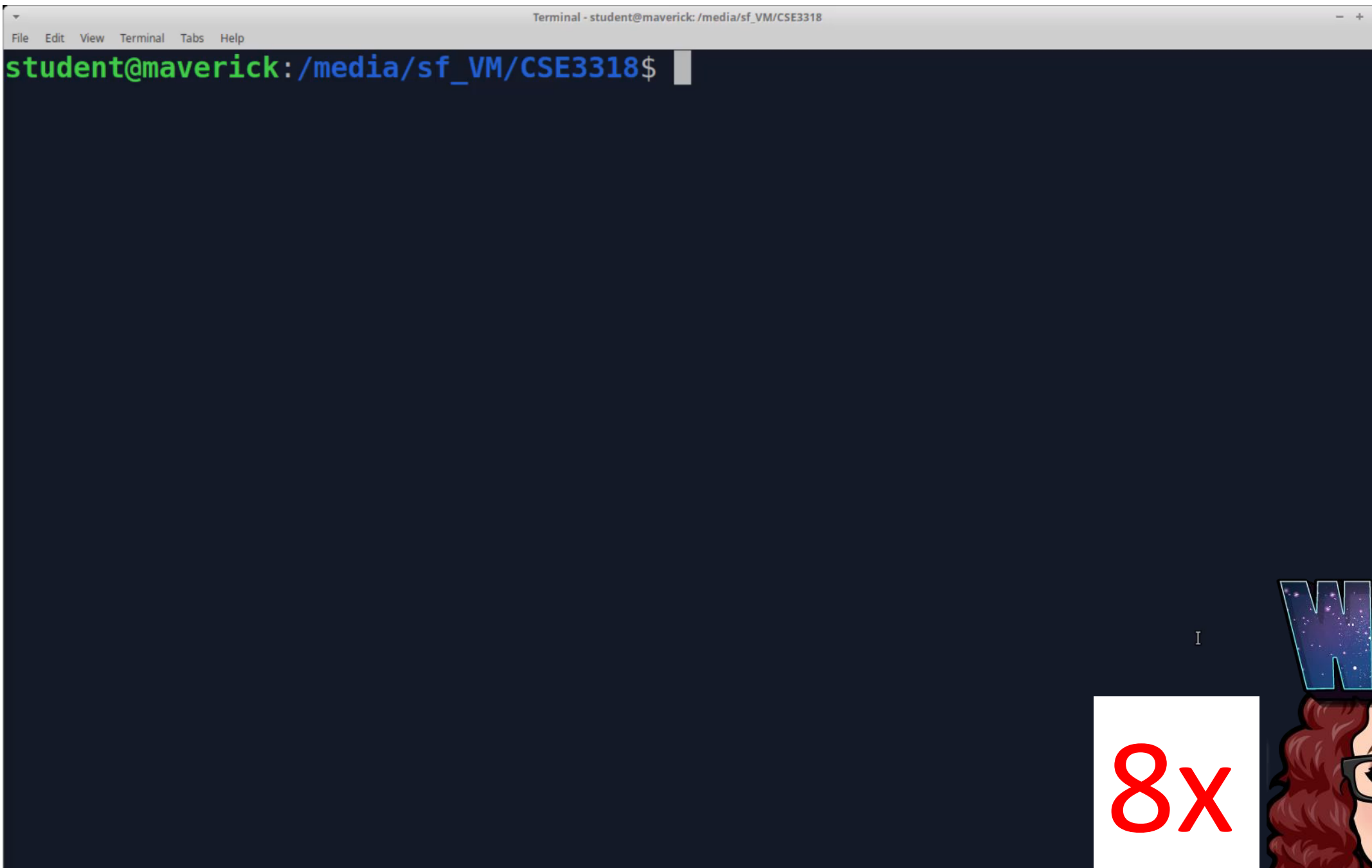
For any given value k , $\text{fib}(k)$ only recurses the first time it is called.

Memoized calls cost $\Theta(1)$

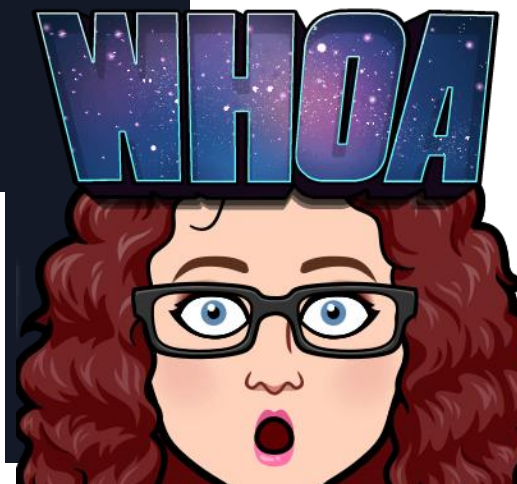
The number of non-memoized calls is n given that we are calculating $\text{fib}(n)$.

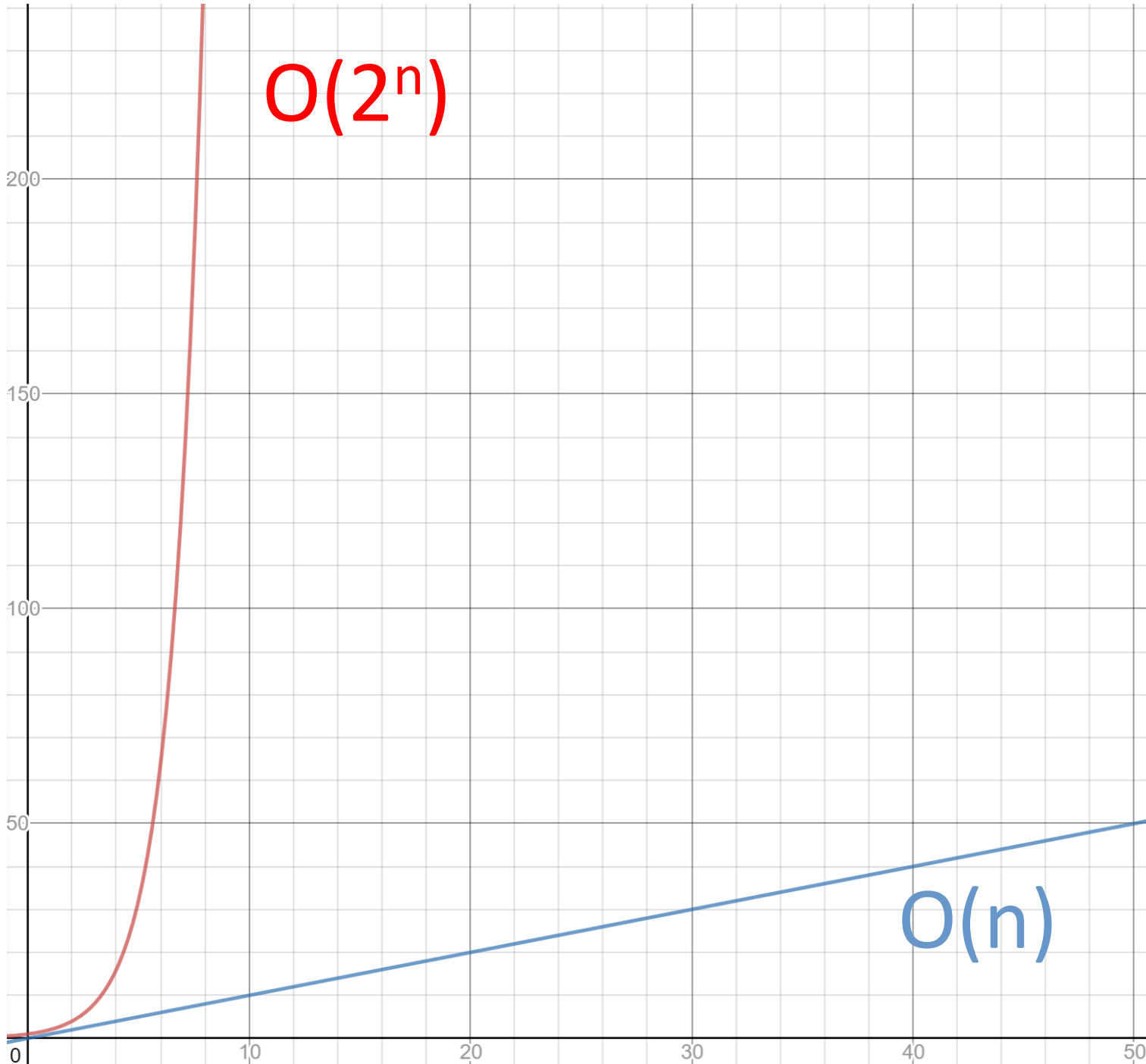
The amount of nonrecursive work per call is $\Theta(1)$.

Total time of memoized version of Fibonacci is $\Theta(n)$



8x





$$2^{50} = 1,125,899,906,842,624$$

Hashing

If you need the definition of a word in the dictionary, how do you find it?

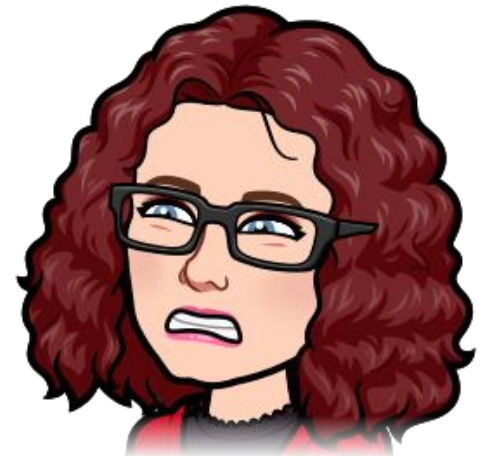
Do you start from page 1 and search until you find it?

$O(n)$

Do you divide the dictionary in half and decide if you need to look in the right half or left half and keep repeating that process until you find the word?

$O(\log_2 n)$

I HOPE NOT!



Hashing

Dictionary

Abstract Data Type - class of objects whose logical behavior is defined by a set of values and a set of operations

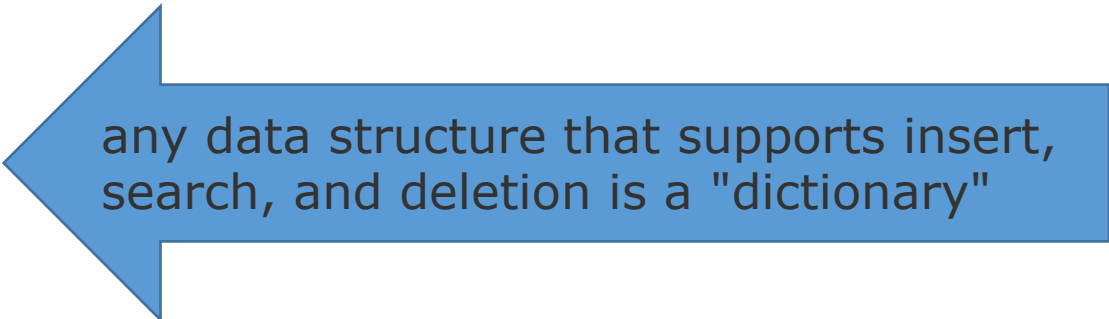
Maintains a set of items each with its own key

3 basic actions for a dictionary

Insert item

Delete item

Search for item



any data structure that supports insert, search, and deletion is a "dictionary"

Hashing

Dictionary

3 basic actions for a dictionary

- Insert item

 - Overwrite any existing key

- Delete item

- Search for item

 - Return the item with the given key or report that it does not exist

Hashing

For those that have studied C++, you might be wondering...

`std::map`

is that a dictionary?

Yes!

`std::map` is the C++ standard library implementation of a dictionary

Hashing

A **map** is a set where each element is a pair, called a key/value pair.

The key is used for sorting and indexing the data and must be unique.

The value is the actual data.

Duplicate keys are *not* allowed—a single value can be associated with each key.

This is called a one-to-one mapping.

Hashing

A map of students where **id number** is the key and **name** is the value can be represented graphically as

Notice that keys are arranged in ascending order.

Maps always arrange its keys in sorted order.

Here the keys are of string type; therefore, they are sorted lexicographically.

1120217	Nikhilesh
1120236	Navneet
1120250	Vikas
1120255	Doodrah

Keys

values

Hashing

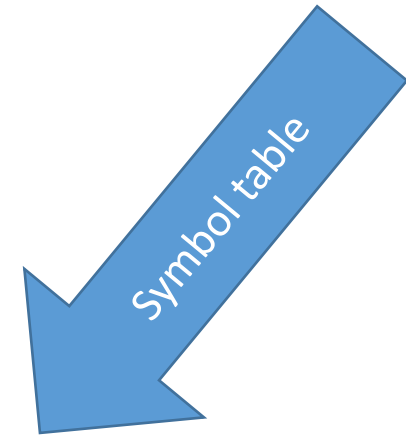
So what is a limitation of this implementation of a dictionary?



Hashing

Dictionaries have a lot of uses

- Databases use them for lookups
- Spell check
- Username and password verification
- Compilers and interpreters (translate variable name to memory address)
- Network router
- Cryptography



Hash Table

A **hash table** is a data structure which implements an associative array abstract data type which is a structure that can map keys to values.

In hashing, large keys are converted into small keys by using a **hash function**. The hash function employs an algorithm to compute an index into the array. These small keys are then stored in this array which is called the **hash table**.

The idea of hashing is to distribute entries uniformly across an array.

Hash Table

Let's say we need to store student id's and their corresponding pin numbers.

1000012345	87345
1000023456	93727
1000046536	87323
1000085848	34522
1000100349	82234
1002002020	73721

Hash Table

We could store that information in an array of structures.

0	1	2	3	4	5	6	7	8	9
1000012345 87345	1000023456 93727	1000046536 87323	1000085848 34522	1000100349 82234	1002002020 73721				

To find an item, we would perform a linear search – start at the beginning of the array and check each cell for a match.

Since this array has 10 cells, the performance would be based on the worst case which is the value we are looking for is in the last cell so $O(10)$ or $O(n)$ where n is the size of the array.

Hash Table

Instead of storing each student ID/pin in an array, let's use a **Hash Function** to determine which array element is used.

Given an input key, the hash function uses an algorithm to map the input key to an index in the array.

```
int GetHashCode(unsigned int student_id)
{
    return ((student_id) % MaxHashTableSize);
}
```


Hash Table

```
int GetHashCode(unsigned int student_id)
{
    return (student_id % MaxHashTableSize);
}
```

Input : 1000012345

Output : $1000012345 \% 10 = 5$

So we put 1000012345 and 87345 in array element 5.

0	1	2	3	4	5	6	7	8	9
					1000012345 87345				

Hash Table

```
int GetHashCode(unsigned int student_id)
{
    return (student_id % MaxHashTableSize);
}
```

Input : 1000023456

Output : $1000023456 \% 10 = 6$

So we put 1000023456 and 93727 in array element 6.

0	1	2	3	4	5	6	7	8	9
					1000012345 87345	1000023456 93727			

Hash Table

```
int GetHashCode(unsigned int student_id)
{
    return (student_id % MaxHashTableSize);
}
```

Input : 1000085848

Output : $1000085848 \% 10 = 8$

So we put 1000085848 and 34522 in array element 8.

0	1	2	3	4	5	6	7	8	9
					1000012345 87345	1000023456 93727		1000085848 34522	

Hash Table

```
int GetHashCode(unsigned int student_id)
{
    return (student_id % MaxHashTableSize);
}
```

Input : 1000100349

Output : $1000100349 \% 10 = 9$

So we put 1000100349 and 82234 in array element 9.

0	1	2	3	4	5	6	7	8	9
					1000012345 87345	1000023456 93727		1000085848 34522	1000100349 82234

Hash Table

```
int GetHashCode(unsigned int student_id)
{
    return (student_id % MaxHashTableSize);
}
```

Input : 1002002020

Output : $1002002020 \% 10 = 0$

So we put 1002002020 and 73721 in array element 0.

0	1	2	3	4	5	6	7	8	9
1002002020 73721					1000012345 87345	1000023456 93727		1000085848 34522	1000100349 82234

Hash Table

Now, when we want to search for a student ID, we give the student id to the hash function and it returns which array element to look in.

```
int GetHashCode(unsigned int student_id)
{
    return (student_id % MaxHashTableSize);
}
```

Input : 1000012345

Output : $1000012345 \% 10 = 5$

Array Element 5 has student ID 1000012345 and pin 87345.

0	1	2	3	4	5	6	7	8	9
1002002020 73721					1000012345 87345	1000023456 93727		1000085848 34522	1000100349 82234

Hash Table

Array with linear search – $O(n)$ where n is the size of the array.

0	1	2	3	4	5	6	7	8	9
1000012345 87345	1000023456 93727	1000046536 87323	1000085848 34522	1000100349 82234	1002002020 73721				

Hash Table

Hash function returns the exact array element containing information – $O(1)$

0	1	2	3	4	5	6	7	8	9
1002002020 73721					1000012345 87345	1000023456 93727		1000085848 34522	1000100349 82234

Hash Table – Hash Function

To achieve a good hashing mechanism, it is important to have a good hash function with the following basic requirements:

- Easy to compute - It should be easy to compute and must not become an algorithm in itself.
- Uniform distribution - It should provide a uniform distribution across the hash table and should not result in clustering.
- Minimize collisions: Collisions occur when pairs of elements are mapped to the same hash value. These should be avoided.

Hash Table - Collisions

```
int GetHashCode(unsigned int student_id)
{
    return (student_id % MaxHashTableSize);
}
```

Input : 1000046536

Output : $1000046536 \% 10 = 6$

So we would put 1000046536 and 87323 in array element 6 but another record is already there.

0	1	2	3	4	5	6	7	8	9
1002002020 73721					1000012345 87345	1000023456 93727		1000085848 34522	1000100349 82234



Hash Table – Hash Function

Irrespective of how good a hash function is, **collisions** are bound to occur.

Therefore, to maintain the performance of a hash table, it is important to manage collisions through various collision resolution techniques.

There are multiple techniques for dealing with collisions

- Open Addressing

- Chaining

Collision Resolution

Open Addressing

In open addressing, all item are stored in the hash table itself.

Each array cell contains either a record or NULL.

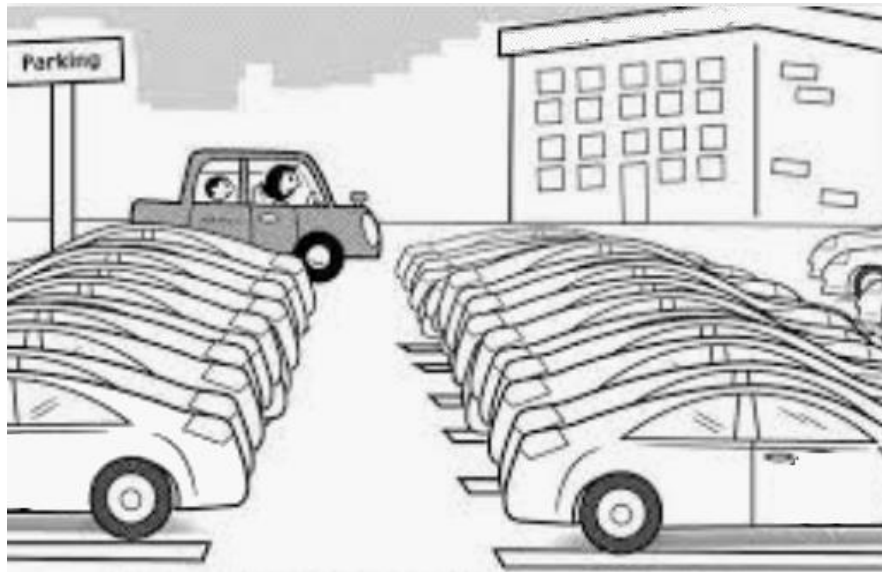
When inserting an item, if the array cell is already occupied, then open addressing takes the next cell.

Collision Resolution

Open Addressing

It keeps moving to the next cell until it finds an empty one.

If it reaches the end of the array, it starts over at the beginning and keeps looking





Collision Resolution

Open Addressing



This method either requires enough array space for every item to be stored or it needs to be able to reallocate the array in order to expand.

When searching for an element, the hash function takes the search to the array cell indicated by the hash, but then must perform a linear search to ensure that the correct item was found.

The array cells of the hash table are examined one by one until the desired item is found or the end of the array is found (meaning the item is not present).

Collision Resolution

Open Addressing



Linear Probing

In linear probing, we linearly probe for the next open cell.

The main problem with linear probing is clustering, many consecutive elements form groups and it starts taking time to find a free slot or to search an element.

Collision Resolution

Open Addressing

Quadratic Probing

Takes the original hash index and adds successive values of an arbitrary quadratic polynomial until an open cell is found.

Double Hashing

Perform a second hash to find a new cell

Collision Resolution

Open Addressing

Linear probing is easy to compute and has the best cache performance but suffers from clustering.

Double hashing has poor cache performance but no clustering. Double hashing requires more computation time as two hash functions need to be computed.

Quadratic probing lies between the two in terms of cache performance and clustering.

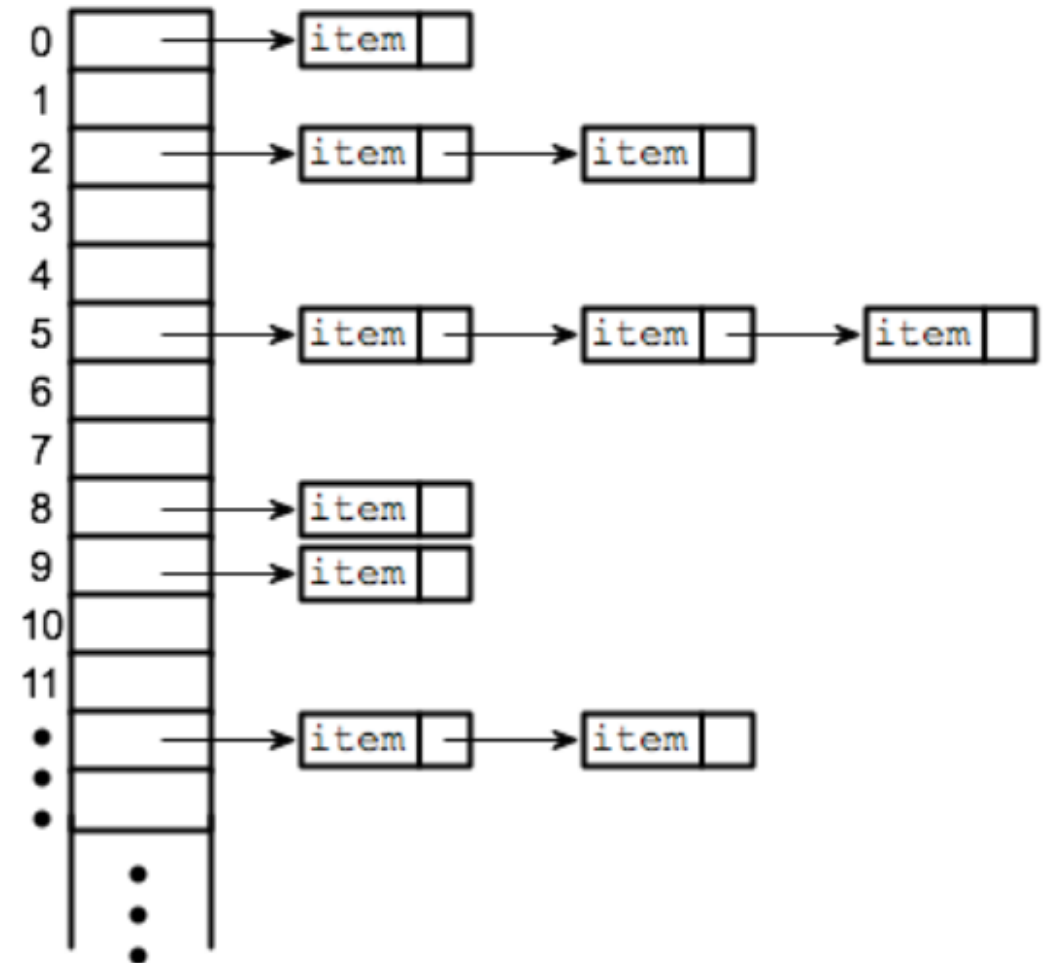
Collision Resolution

Separate chaining (open hashing)

Separate chaining is one of the most commonly used collision resolution techniques.

It is usually implemented using linked lists.

In separate chaining, each element of the hash table is a linked list.

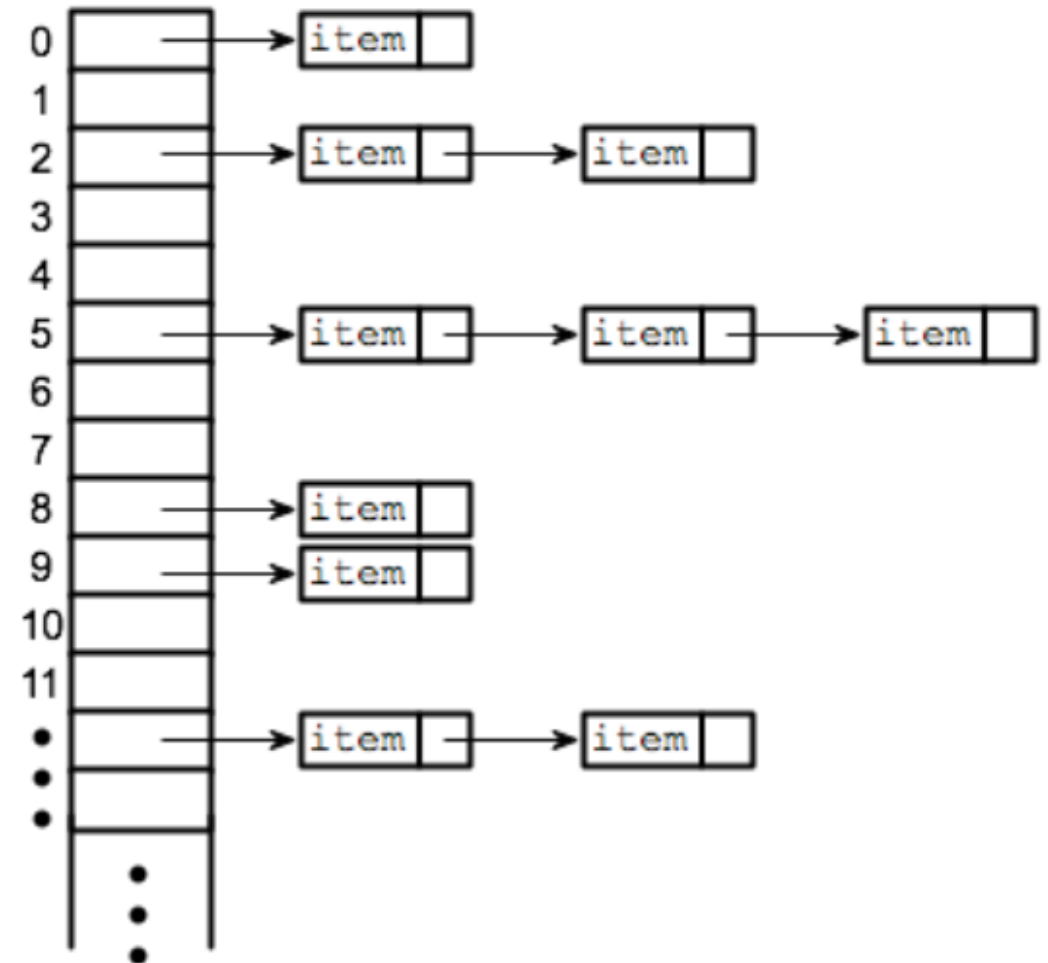


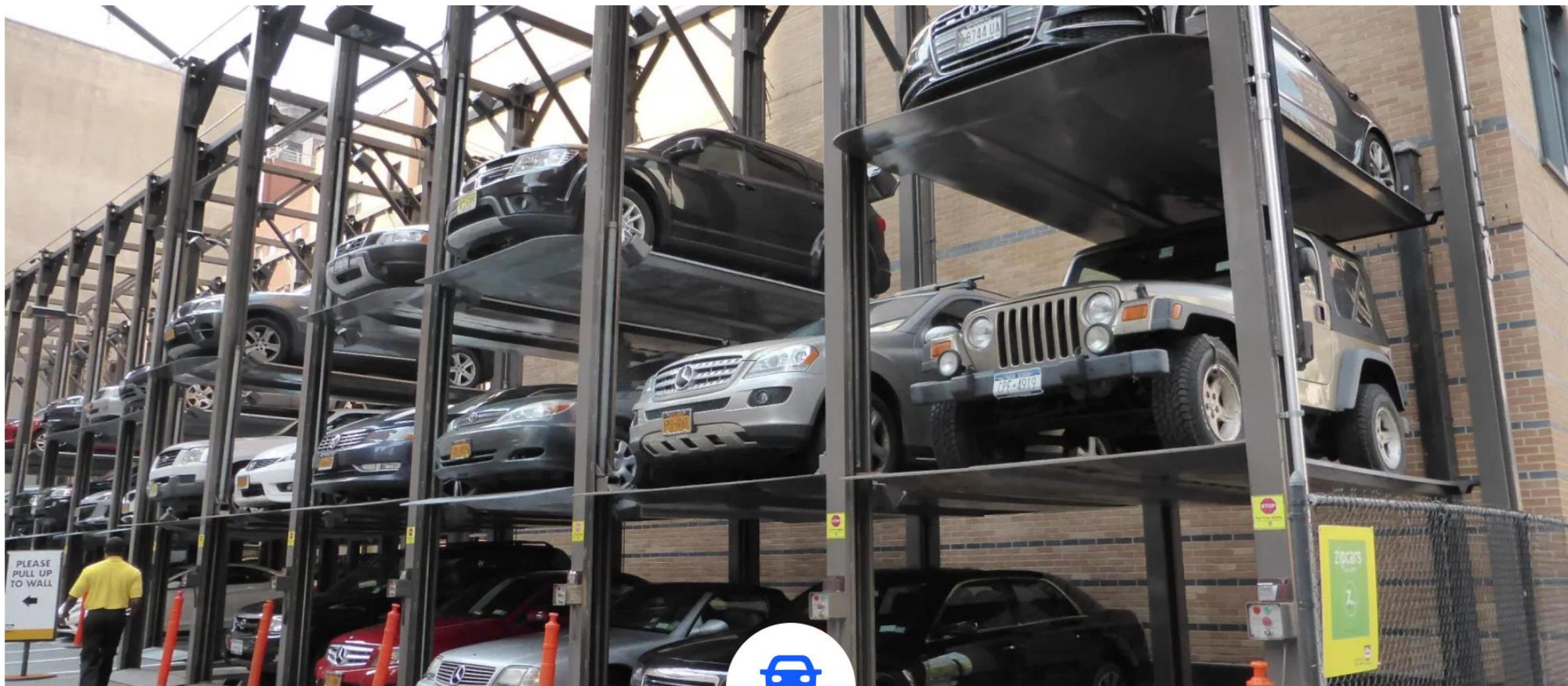
Collision Resolution

Separate chaining (open hashing)

To store an element in the hash table you must insert it into a specific linked list.

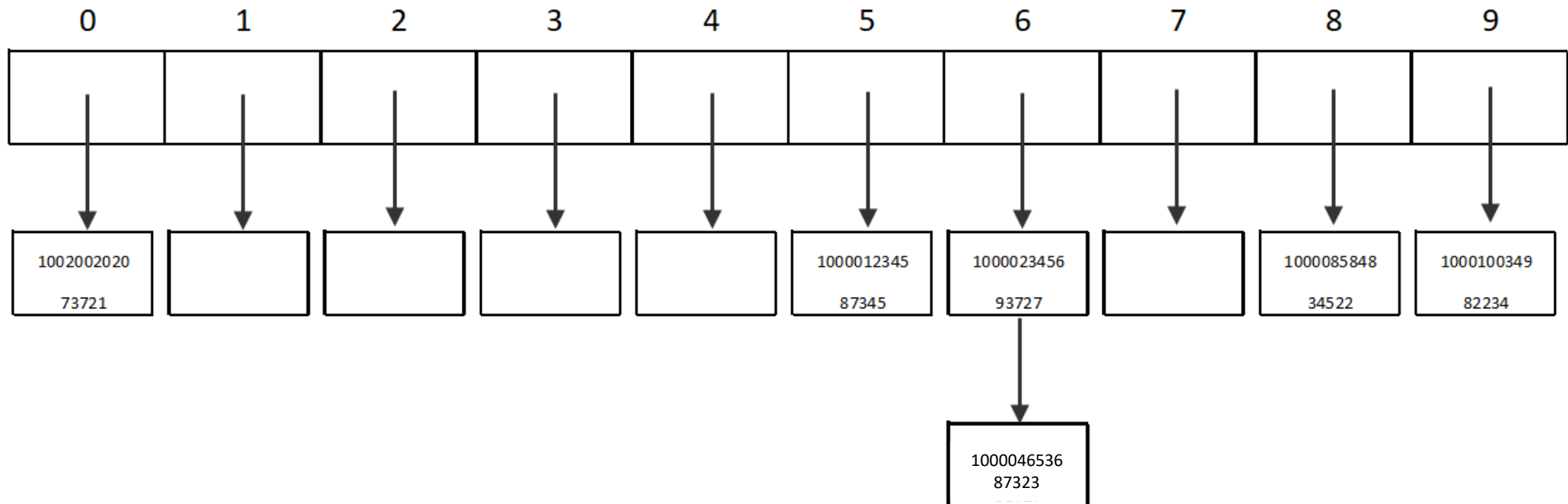
If there is any collision (i.e. two different elements have same hash value) then store both elements in the same linked list.





Collision Resolution

Separate chaining turns each array element into its own linked list. Any collisions are handled by adding the new element on to the end of the linked list for that index.



Collision Resolution

For separate chaining, the worst-case scenario is when all the entries are inserted into the same linked list. The lookup procedure may have to scan all its entries, so the worst-case cost is proportional to the number (n) of entries in the table so $O(n)$ which is the same as linear search.

A good hash function can prevent this worst-case scenario and make hash table look up always faster than linear search.

Hash tables, on average, are more efficient than search trees or any other table lookup structure. You can access an element in **$O(1)$** time. This is why hash tables are heavily used in many situations.

Hashing

The capacity of the hash table is the size of the array.

The load factor is the number of keys stored in the hash table divided by the capacity.

The size should be chosen so that the load factor is less than 1.

For instance, if we want to implement a German-English dictionary with 50,000 German words, we need a hash table that is larger than 50,000.

Hashing

Since the number of keys in the hash table is less than the capacity of the hash table, assuming that the keys are evenly distributed across indices, there will be few collisions, and most of the linked lists will be of length 1.

A few will be of length 2; a very few will be of length 3, and so on.

The probability that there is any linked list that is very much longer than the load factor is very small.

Hashing

Jan	Tim	Mia	Sam	Leo	Ted	Bea	Lou	Ada	Max	Zoe
0	1	2	3	4	5	6	7	8	9	10

If we had this array of names and we wanted to look up name, how would we find it?

We would have to start at the first element and search.

What is the time complexity of a linear search?

$O(n)$

What if we sorted and used a binary search?

$O(\log_2 n)$

Hashing

Jan	Tim	Mia	Sam	Leo	Ted	Bea	Lou	Ada	Max	Zoe
0	1	2	3	4	5	6	7	8	9	10

How about we use hashing?

$$\text{Jan} = J (74) + a (97) + n (110) = 281$$

What can we use for a hashing function?

Now MOD by size of array

$$281 \% 11 = 6$$

How about ASCII values?

Hashing

Bea	Tim					Jan				
0	1	2	3	4	5	6	7	8	9	10

$$\text{Tim} = T(84) + i(105) + m(109) = 298 \quad \text{Bea} = B(66) + e(101) + a(97) = 264$$

Now MOD by size of array

$$298 \% 11 = 1$$

Now MOD by size of array

$$264 \% 11 = 0$$

Hashing

If we apply the hash function to the rest of the names, our hash table will look like...

Bea	Tim	Leo	Sam	Mia	Zoe	Jan	Lou	Max	Ada	Ted
0	1	2	3	4	5	6	7	8	9	10

Now, how do we look up a name?

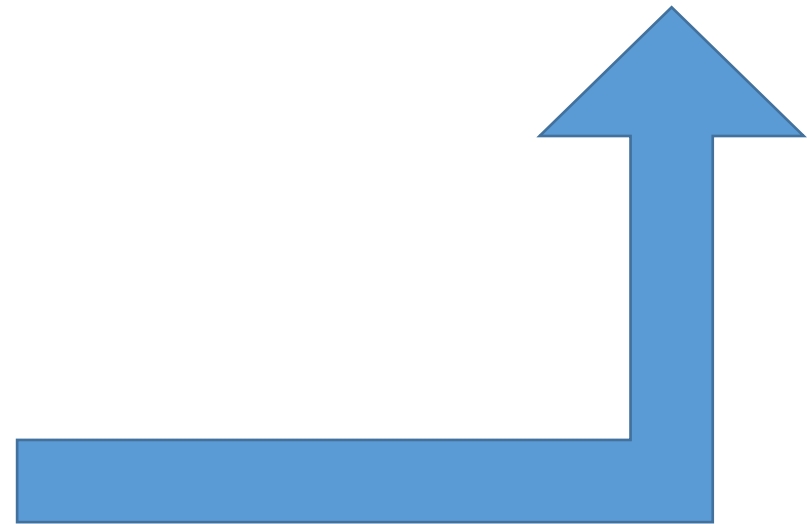
Hashing

Bea	Tim	Leo	Sam	Mia	Zoe	Jan	Lou	Max	Ada	Ted
0	1	2	3	4	5	6	7	8	9	10

Let's say we want to find Ada.

We apply the hash function

$$\text{Ada} = A(65) + d(100) + a(97) = 262 \% 11 = 9$$



Time complexity?

$\Theta(1)$

Hash Functions/Algorithms

Numeric keys

Divide the key by the number of cells in the array and take the remainder (use mod)

Alphanumeric keys

Divide the sum of the ASCII values by the number of cells in the array and take the remainder (use mod)

Folding Method

Rather than just adding the numbers of a phone number, for example, add the area code and then the prefix and then the exchange and use those 3 numbers as the array index.

214 772 2387 folds to become index 772.

Hashing with Open Addressing

Let's look at how Open Addressing handles collisions.

Bea	Tim			Mia	Zoe					
0	1	2	3	4	5	6	7	8	9	10

What happens when we try to add Sue?

$$S(83) + u(117) + e(101) = 301 \% 11 = 4$$

Mia is already in cell 4. So what does Open Addressing do with this collision?

Hashing with Open Addressing

Open Addressing with linear probing would add Sue at cell 6.

Bea	Tim			Mia	Zoe	Sue				
0	1	2	3	4	5	6	7	8	9	10

What happens when we try to add Len?

$$L(76) + e(101) + n(110) = 287 \% 11 = 1$$

Tim is already in cell 1. So what does Open Addressing do with this collision?

Hashing with Open Addressing

Open Addressing with linear probing would add Len at cell 2.

Bea	Tim	Len	Moe	Mia	Zoe	Sue	Lou			
0	1	2	3	4	5	6	7	8	9	10

Moe would hash to cell 3 so no issue.

Lou would hash to cell 7 so no issue.

Rae would hash to cell 5 but Zoe is already there.

What happens with Open Addressing?

Hashing with Open Addressing

Open Addressing with linear probing would add Rae at cell 8.

Bea	Tim	Len	Moe	Mia	Zoe	Sue	Lou	Rae	Max	Tod
0	1	2	3	4	5	6	7	8	9	10

Max would hash to cell 8 which is already occupied by Rae.


Open Addressing with linear probing would add Max at cell 9.

Tod would hash to cell 9 which is already occupied by Max.

Open Addressing with linear probing would add Tod at cell 10.

Hashing with Open Addressing

Bea	Tim	Len	Moe	Mia	Zoe	Sue	Lou	Rae	Max	Tod
0	1	2	3	4	5	6	7	8	9	10



So how do we look up Rae?

We hash it

$$R(82) + a(97) + e(101) = 280 \% 11 = 5$$

So we would look for Rae at cell 5.

Hashing with Separate Chaining

Let's do the same example but with Separate Chaining instead of Open Addressing.

Instead of storing the names in the actual array cells like with Open Addressing....

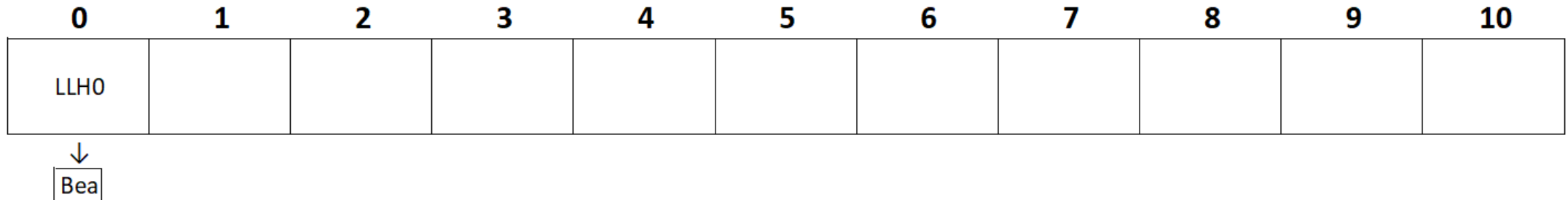
Bea	Tim			Mia	Zoe					
0	1	2	3	4	5	6	7	8	9	10

Hashing with Separate Chaining

When a key hashes to an array cell, a linked list is created.

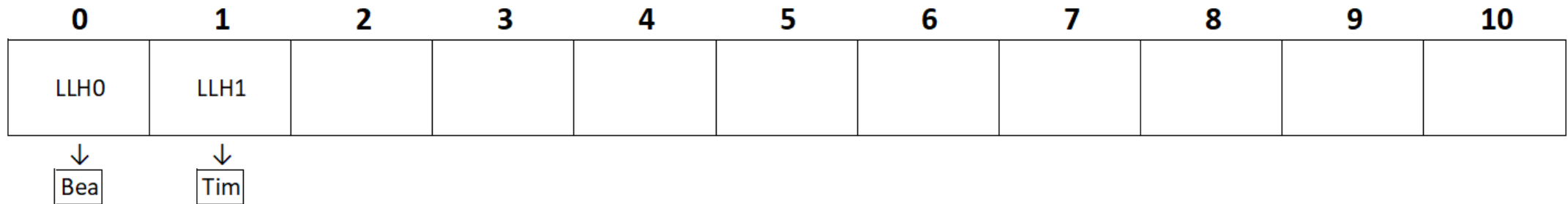
The linked list head is stored in the array – the array is now a `char` pointer array rather than just a `char` array.

Bea hashes to cell 0.

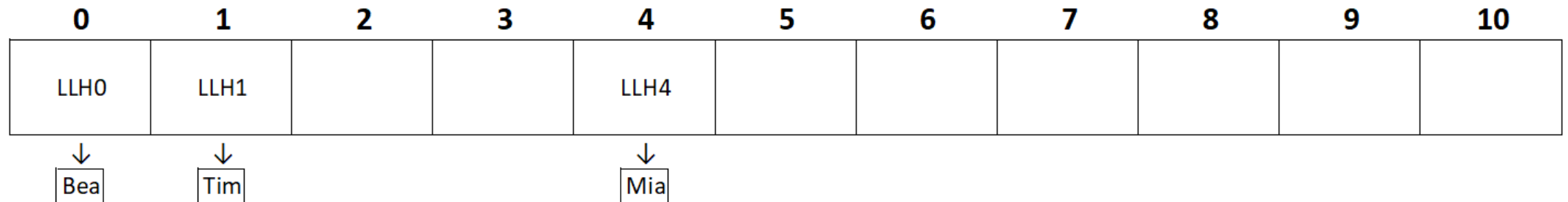


Hashing with Separate Chaining

Tim hashes to cell 1.

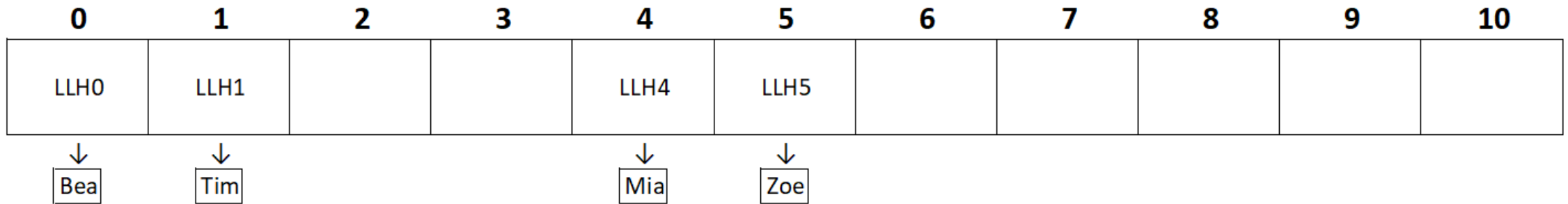


Mia hashes to cell 4.

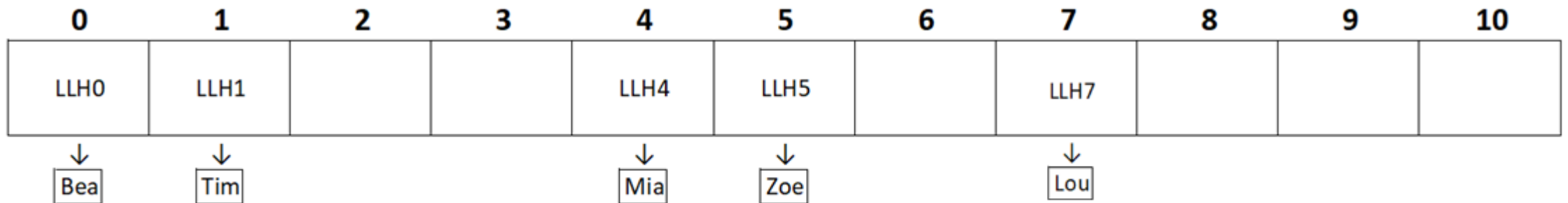


Hashing with Separate Chaining

Zoe hashes to cell 5.

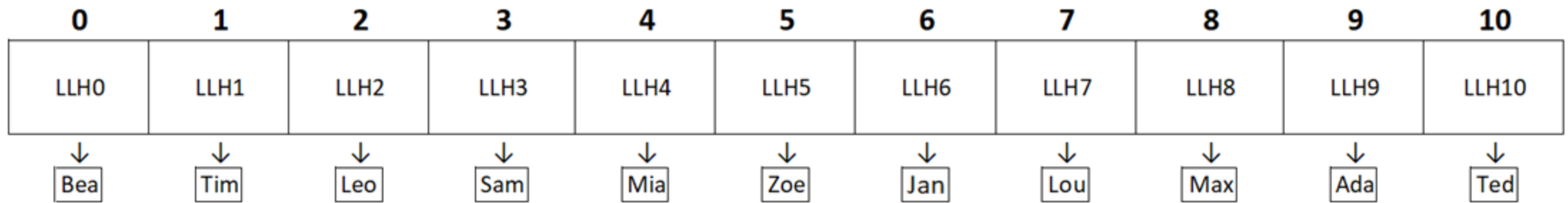


Lou hashes to cell 7.

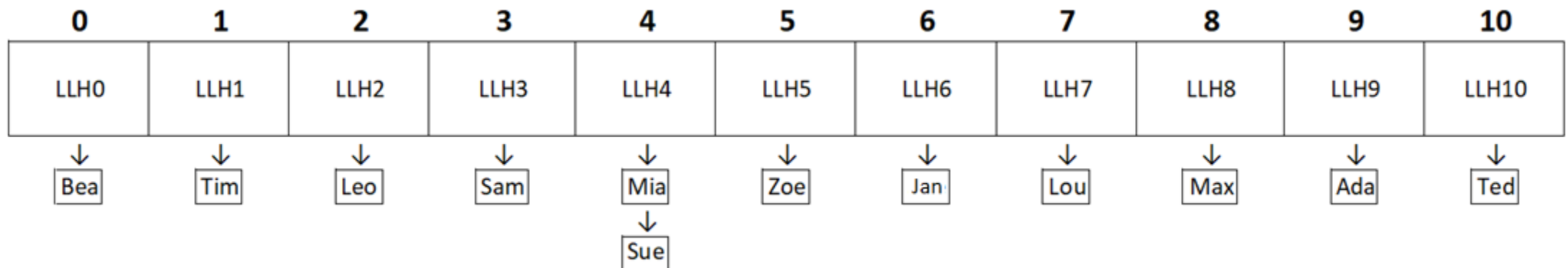


Hashing with Separate Chaining

We can add the rest...

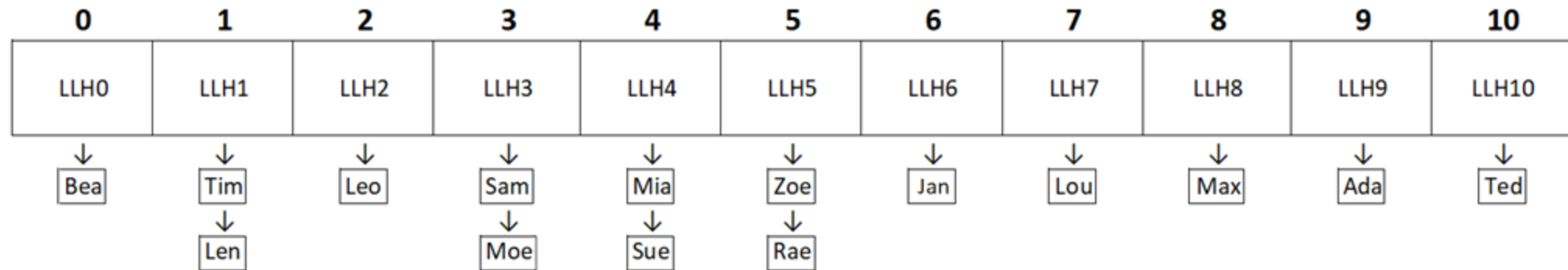


What happens when we try to add Sue which hashes to 4?



Hashing with Separate Chaining

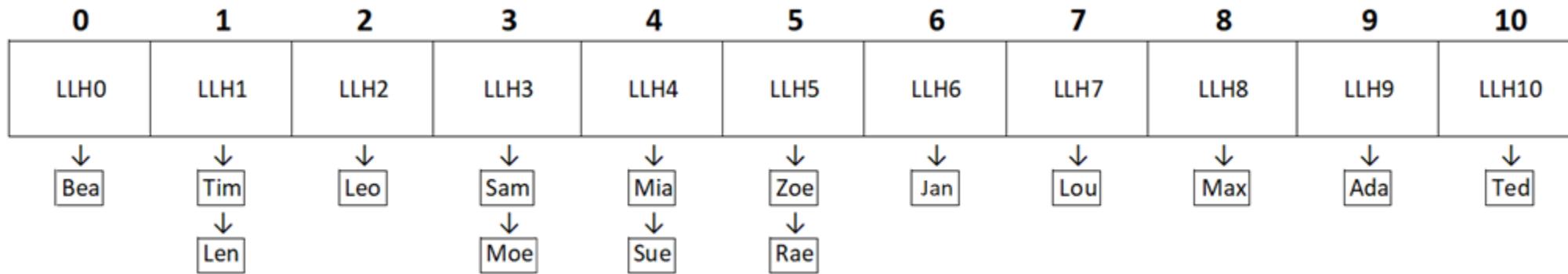
What happens when we add Len(1), Moe(3), Lou(7)



Any entries that hashes to match an existing entries gets added at the end of the linked list.

Hashing with Separate Chaining

When an item is hashed and needs to be added to the hash table,



we need to check if it is the first item to be added at that location.

If it is, then we create a linked list head, malloc a new node, put the data in the node and store the linked list head in the array.

If it is not the first, then we use the linked list head stored in the array cell and traverse to the end of the list and add a newly malloced node.

Hash Table – Uses

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