CSE 3318

Week of 07/23/2024

Instructor : Donna French

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.

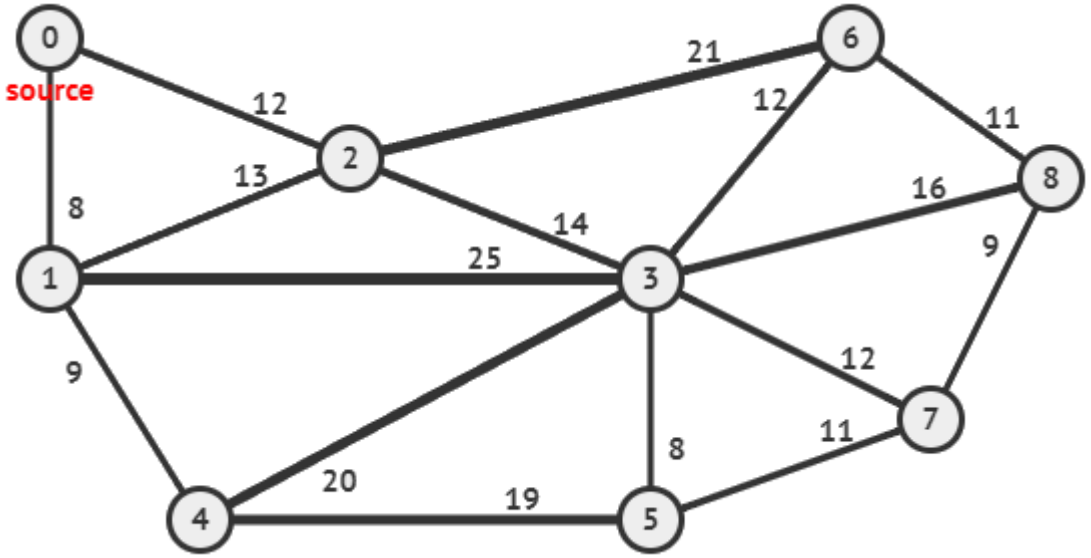
Use Prim’s Algorithm when you have a graph with lots of edges.

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

How many edges will our MST have?

How many vertices?
9

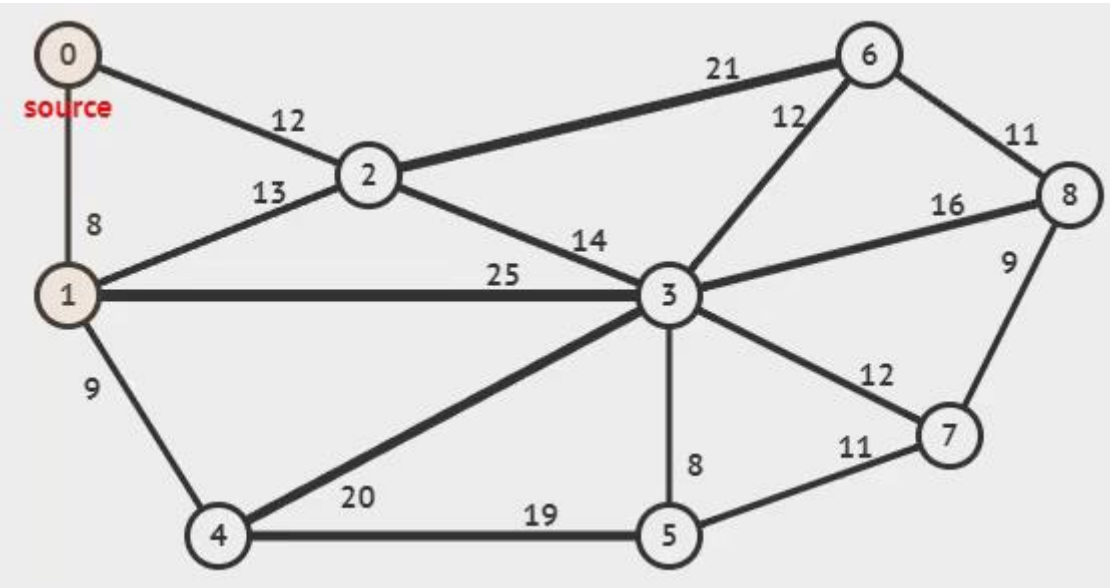
How many edges in the MST?
8



All edges

(0,1,8)
(3,5,8)
(4,1,9)
(7,8,9)
(5,7,11)
(6,8,11)
(0,2,12)
(3,6,12)
(3,7,12)
(1,2,13)
(2,3,14)
(3,8,16)
(4,5,19)
(3,4,20)
(2,6,21)
(1,3,25)

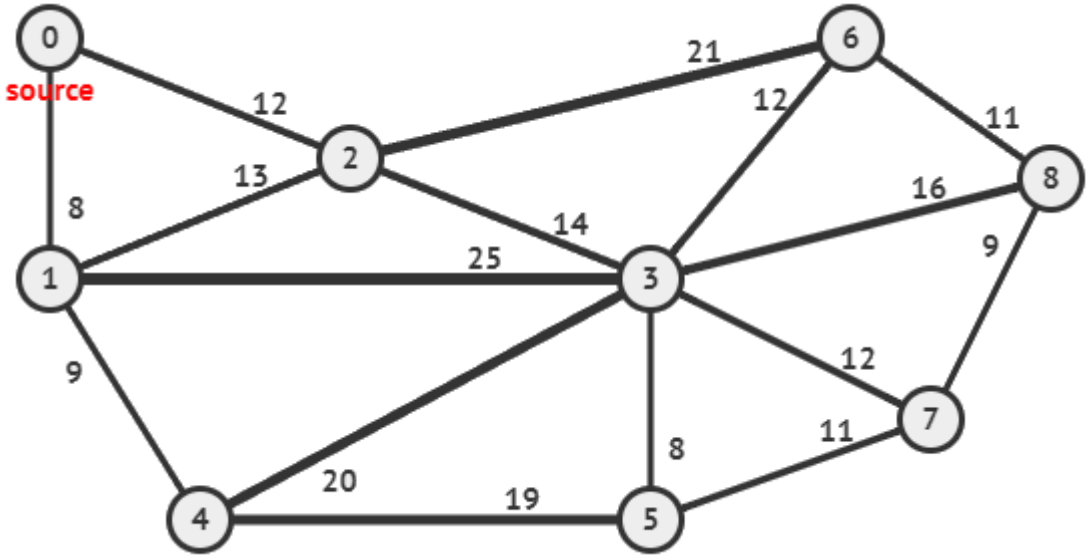
Kruskal will find all edges and sort them and then test each edge.



How many edges will our MST have?

How many vertices?
9

How many edges in the MST?
8



(0,1,8)
(0,2,12)

(1,2,13)
(1,3,25)
(1,4,9)

(4,5,19)
(4,3,20)

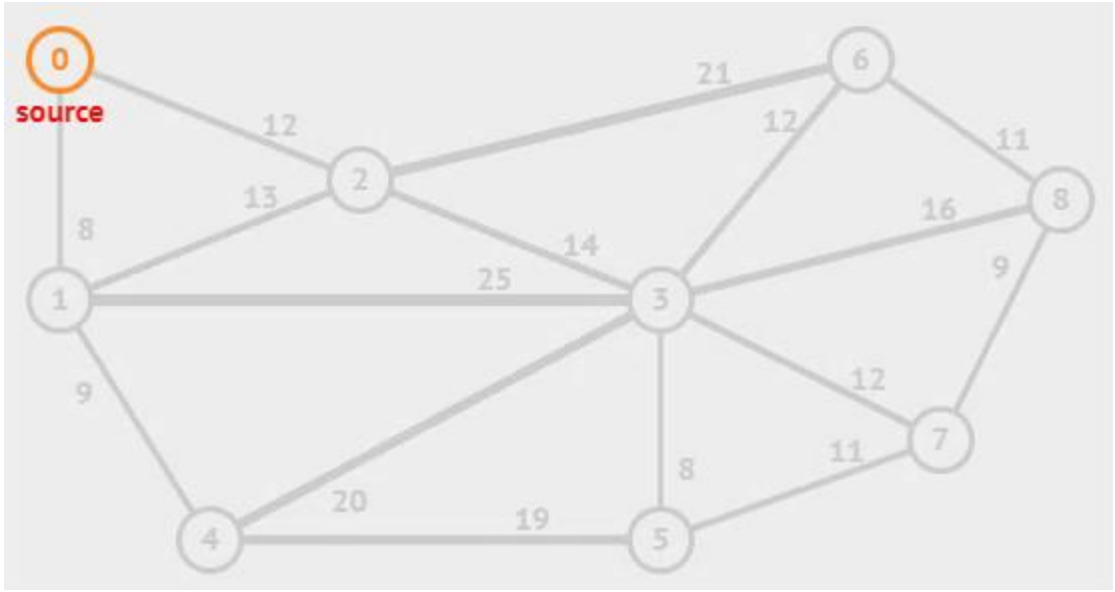
(2,3,14)
(2,6,21)

(3,5,8)
(3,6,12)
(3,7,12)
(3,8,16)

(5,7,11)

(7,8,9)
(8,6,11)

Prim's will start at a given vertex and connect the vertices until the MST is found



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

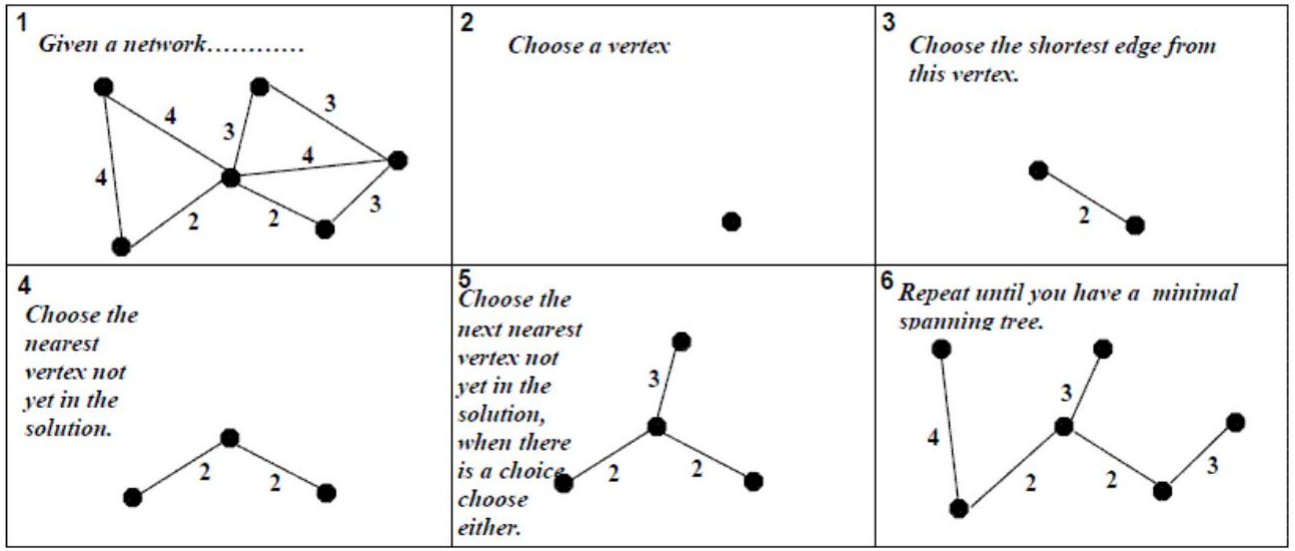
Prim always joins a "new" vertex to an "old" vertex, so that every stage is a tree.

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.

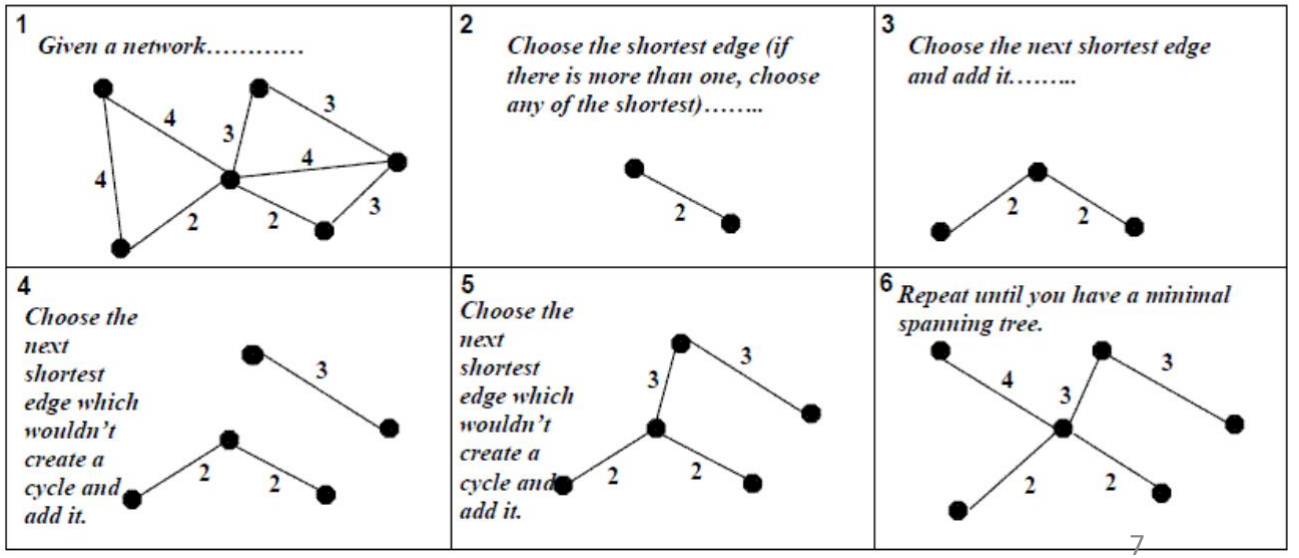
Kruskal has a larger complexity than Prim and could result in an incomplete/unconnected result if interrupted.

Prim vs Kruskal

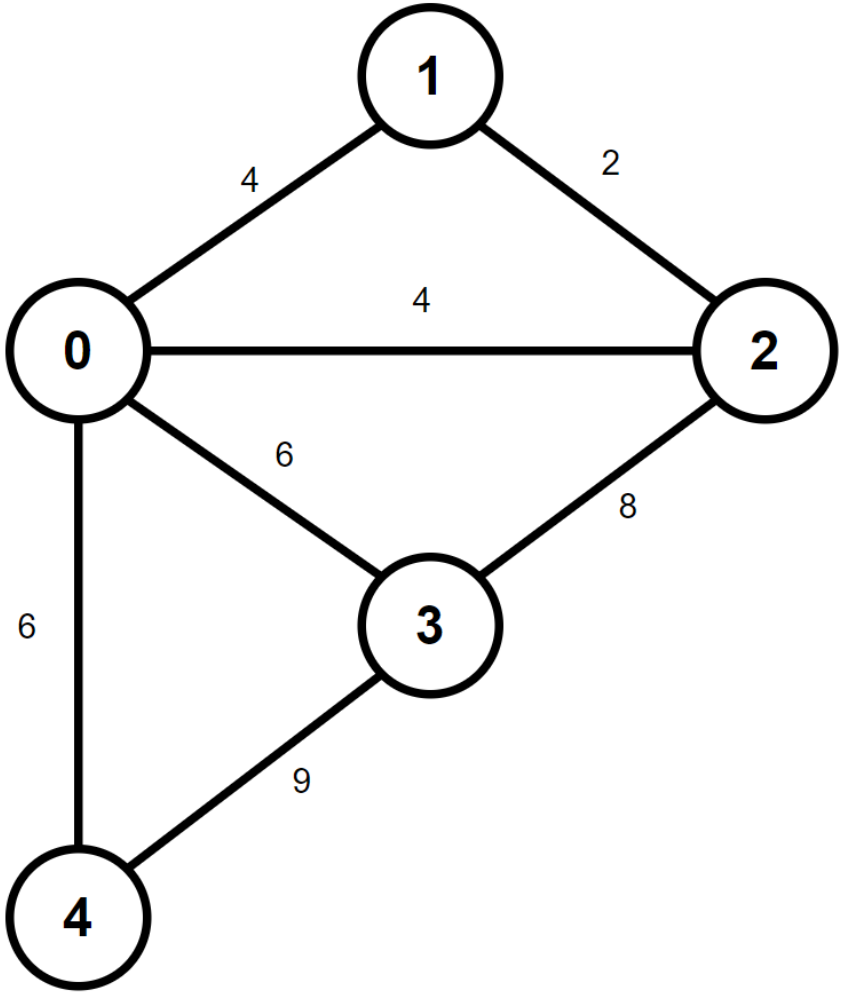
Prim's Algorithm



Kruskal's Algorithm



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)

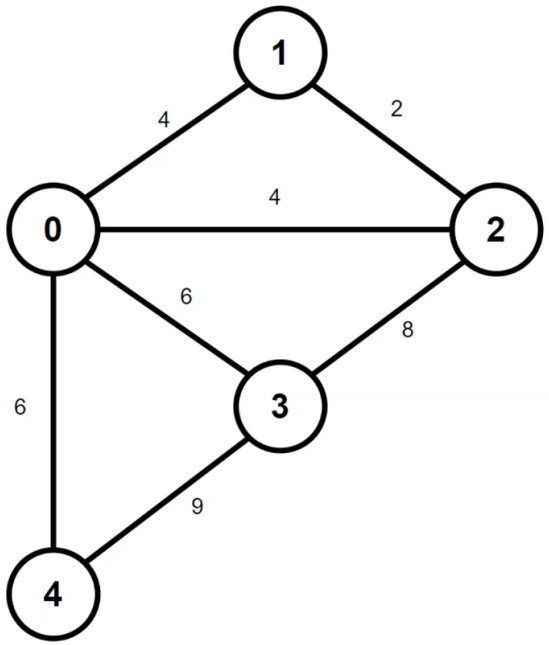
Add (0,1,4) to MST
Add (1,2,2) to MST
Remove (0,2,4) – causes cycle
Add (0,3,6) to MST
Add (0,4,6) to MST

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
```

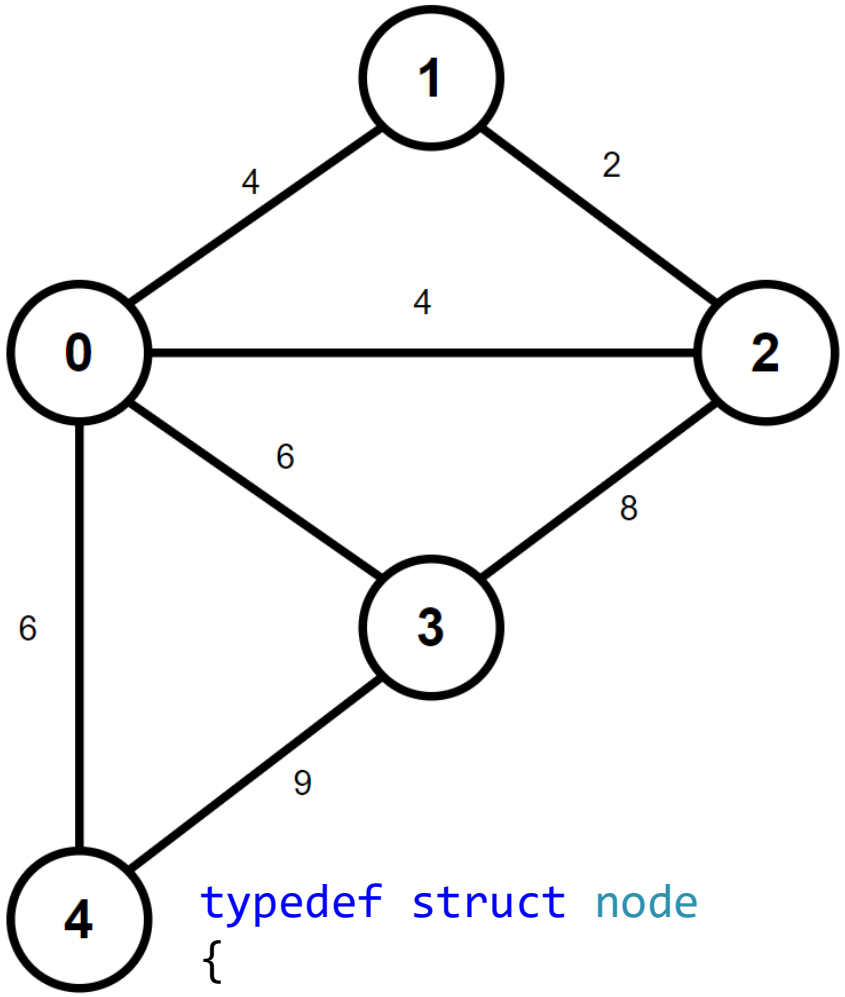
```
int Prims(int AdjacencyMatrix[][MAX], int VertexCount, int StartingVertex)
{
   NODE *LinkedListHead = NULL, *TempPtr = NULL, *NextVertex = NULL;
   int Visited[MAX] = {};
   int MSTLength = 0;
   int EdgeCount = VertexCount - 1;
   int CurrentVertex = StartingVertex;

   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);
```

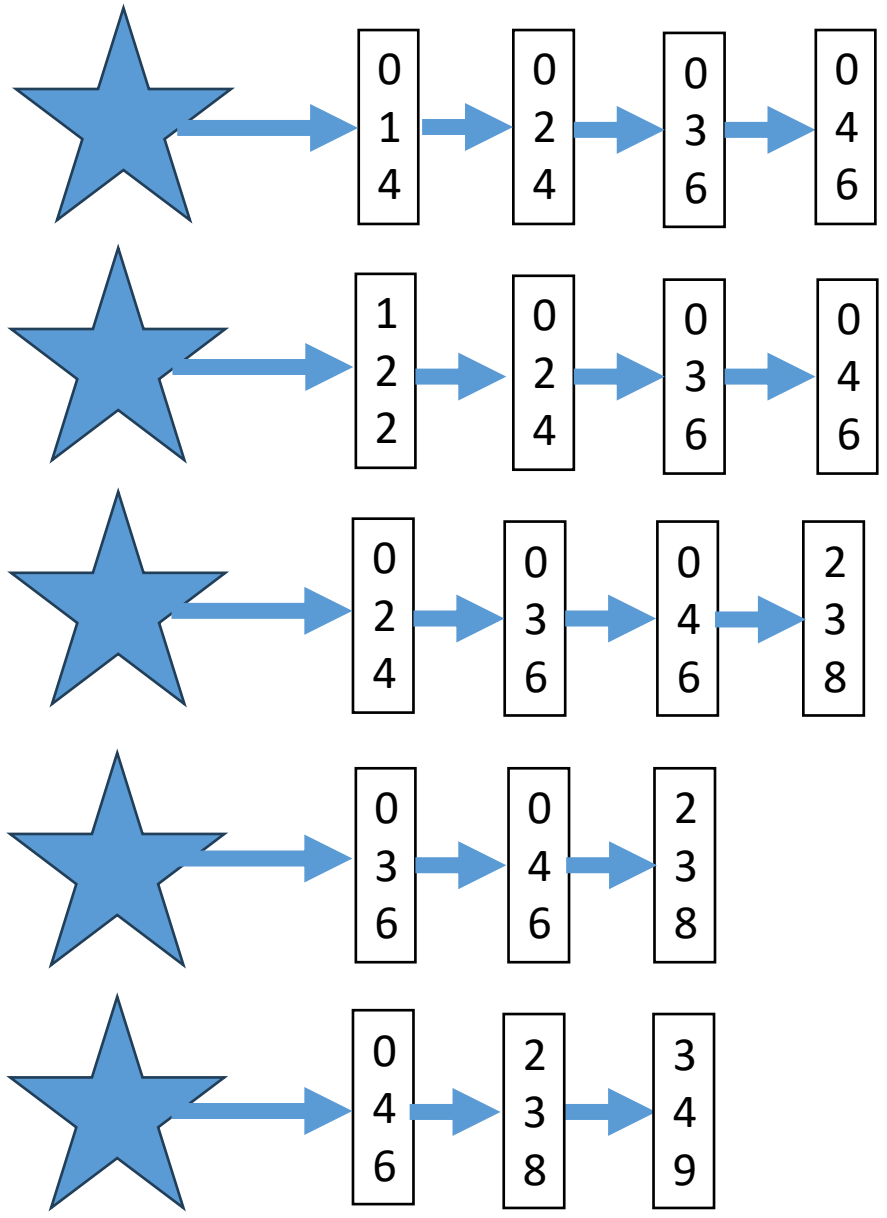
```
typedef struct node
{
    int LE;
    int RE;
    int EdgeDistance;
    struct node *next_ptr;
}
NODE;
```

```
// 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], &Lin
   }
}

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

      DisplayLinkedList(LinkedListHead);

      // Find the edge with the smallest distance
      TempPtr = LinkedListHead;
      while (TempPtr != NULL && NextVertex == NULL)
      {
      }

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

V - 1

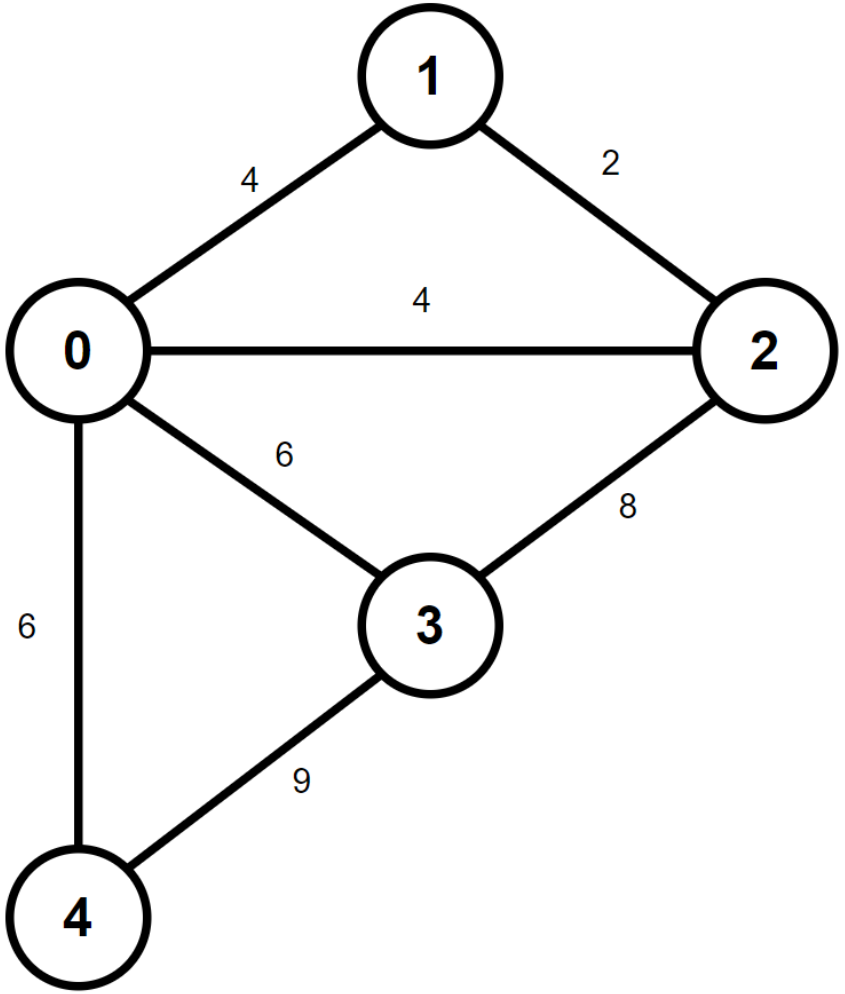
V

(V – 1)(V)

$V^2 - V$

$O(V^2)$

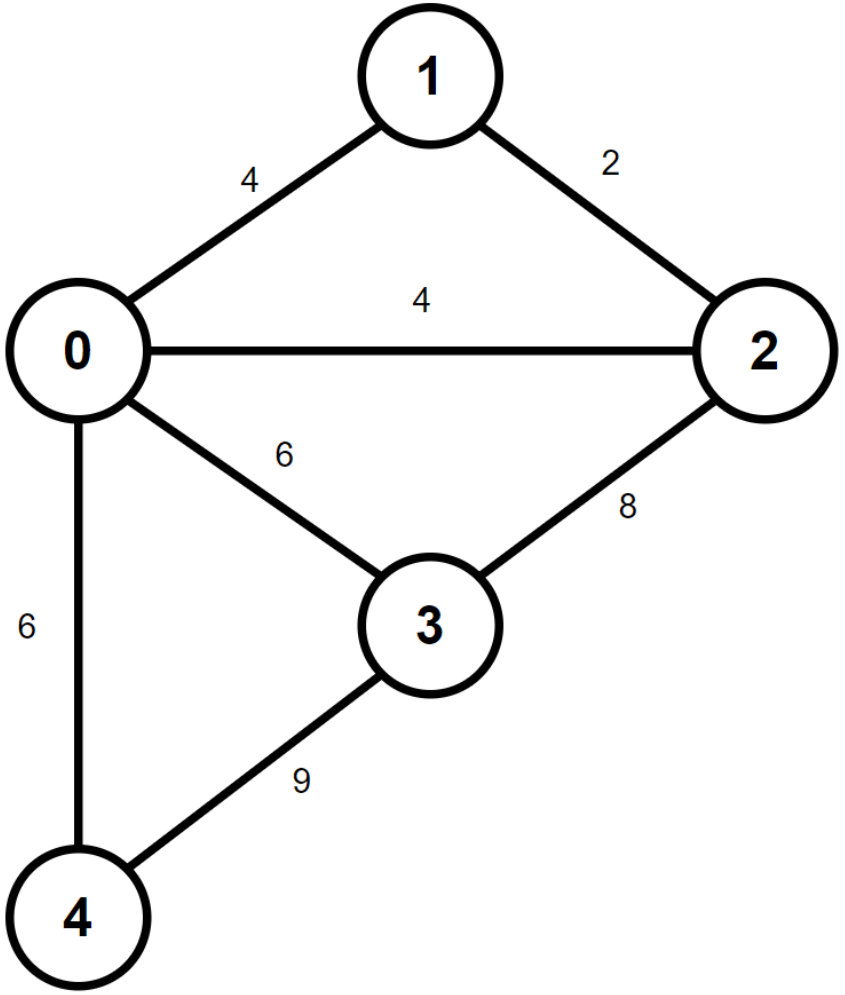
MST – Kruskal's Algorithm



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)
--------------------
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;
}
```

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.

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 $O(V\log_2 V+E)$

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.

Kruskal's performance is mainly determined by the sorting step which dominates the overall complexity.

Space complexity is $O(V + E)$

Dynamic Programming

Let's look at a familiar recursive problem

Calculate the *n*th Fibonacci value.

The recursive algorithm is

```
if n <= 1, then set f = n
else f = fib(n-1) + 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

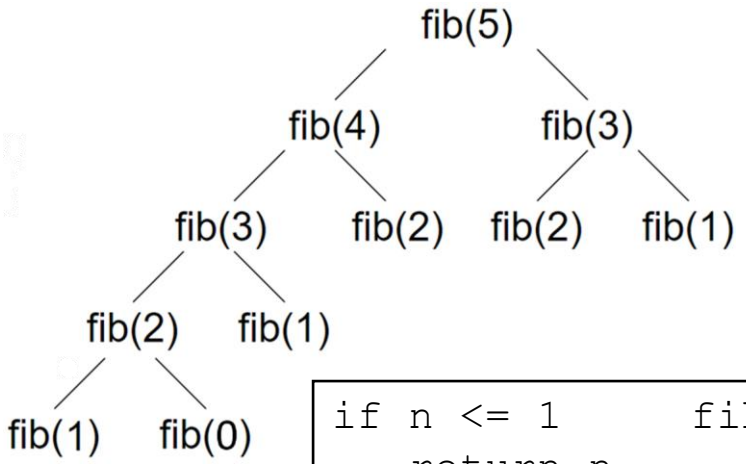
Dynamic Programming

if n <= 1,

then set f = n

else f = fib(n-1) + fib(n-2)

return f




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

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

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

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

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

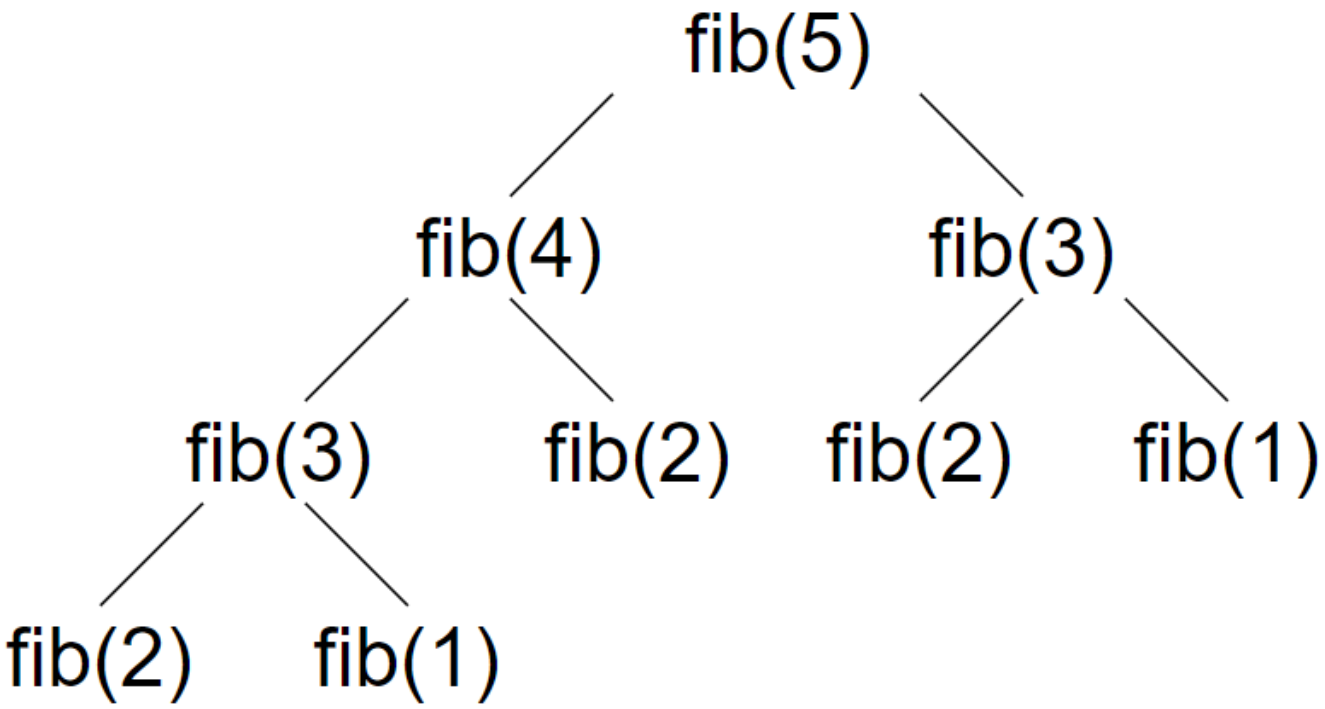
```
if n <= 1      fib(2)
   return n
else
   return fib(n-1) +
          fib(n-2)
```

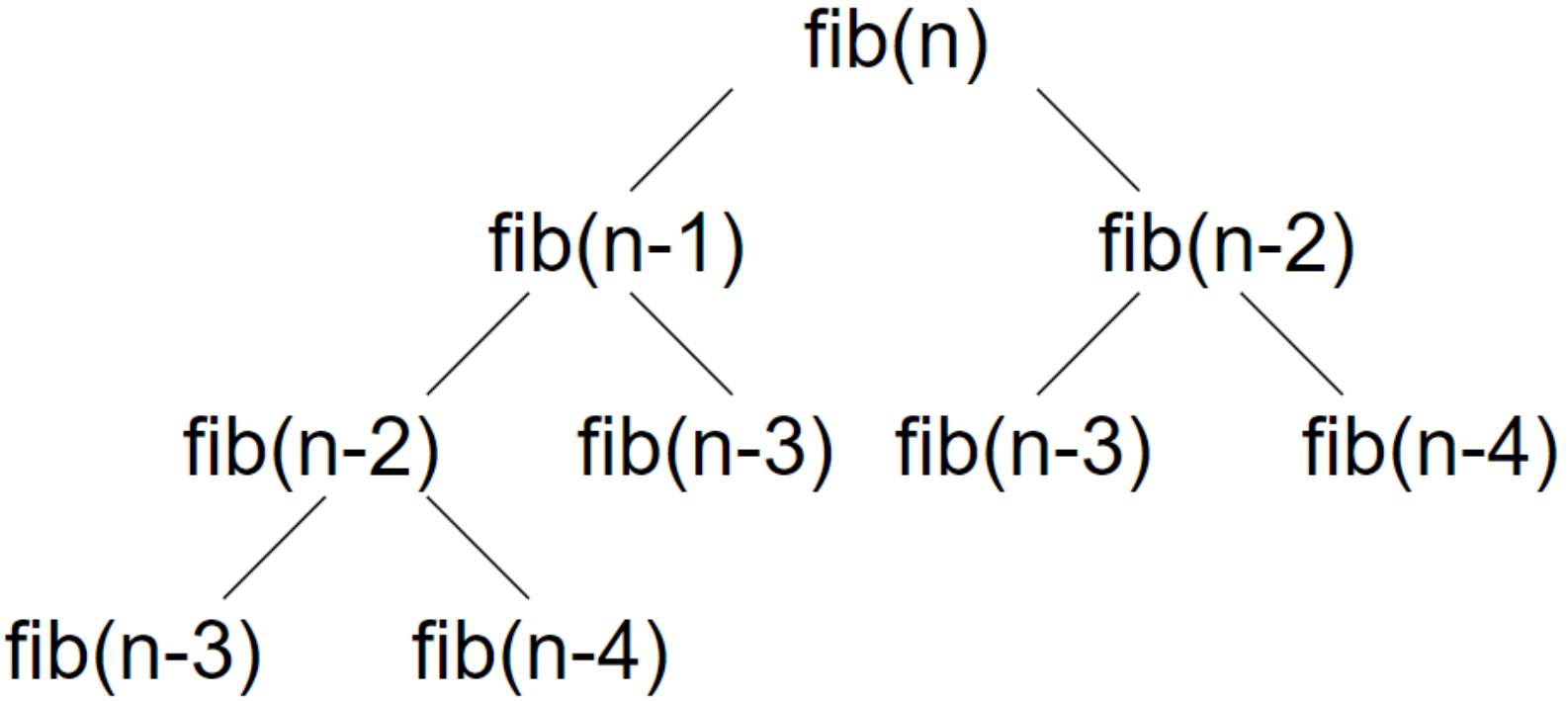
```
if n <= 1      fib(1)
   return n
else
   return fib(n-1) +
          fib(n-2)
```

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

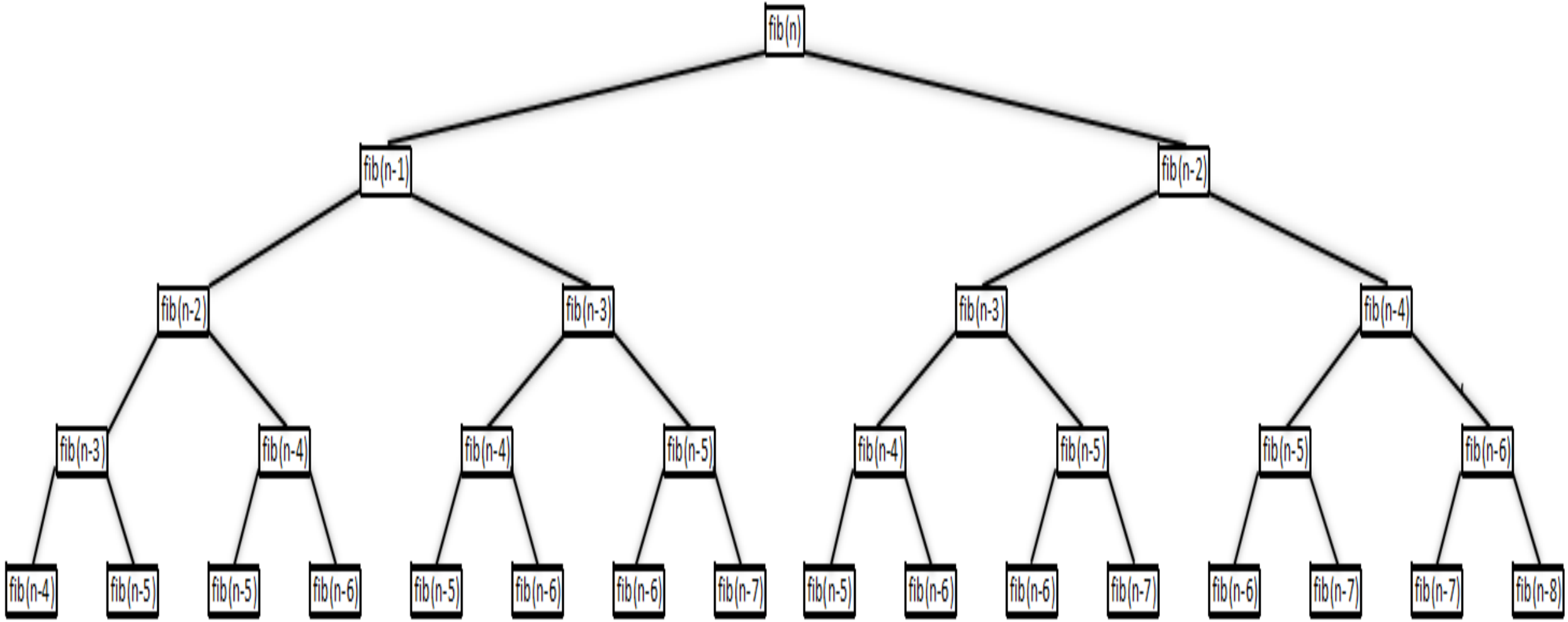
```
if n <= 1      fib(1)
   return n
else
   return fib(n-1) +
          fib(n-2)
```

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

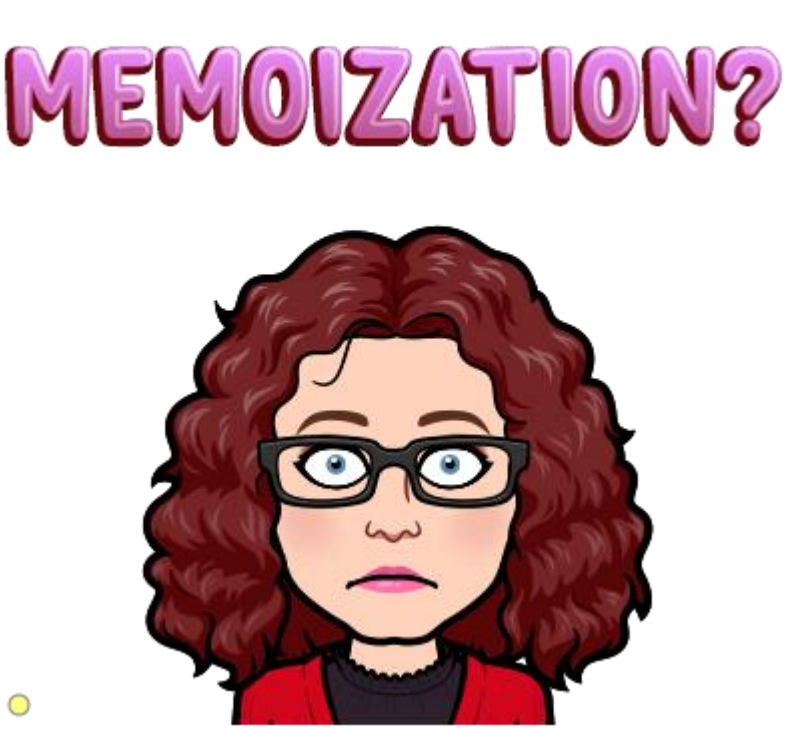
There are two patterns used in solving DP problems

1. Tabulation – Bottom up

2. Memoization – Top Down

Dynamic Programming

MEMOIZATION



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 <= 1,
then set f = n
else f = fib(n-1) + fib(n-2)
return f

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

Dynamic Programming

Tabulation Method – Bottom Up Dynamic Programming

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

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	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

1st	2nd	Fib
0	1	1
1	1	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

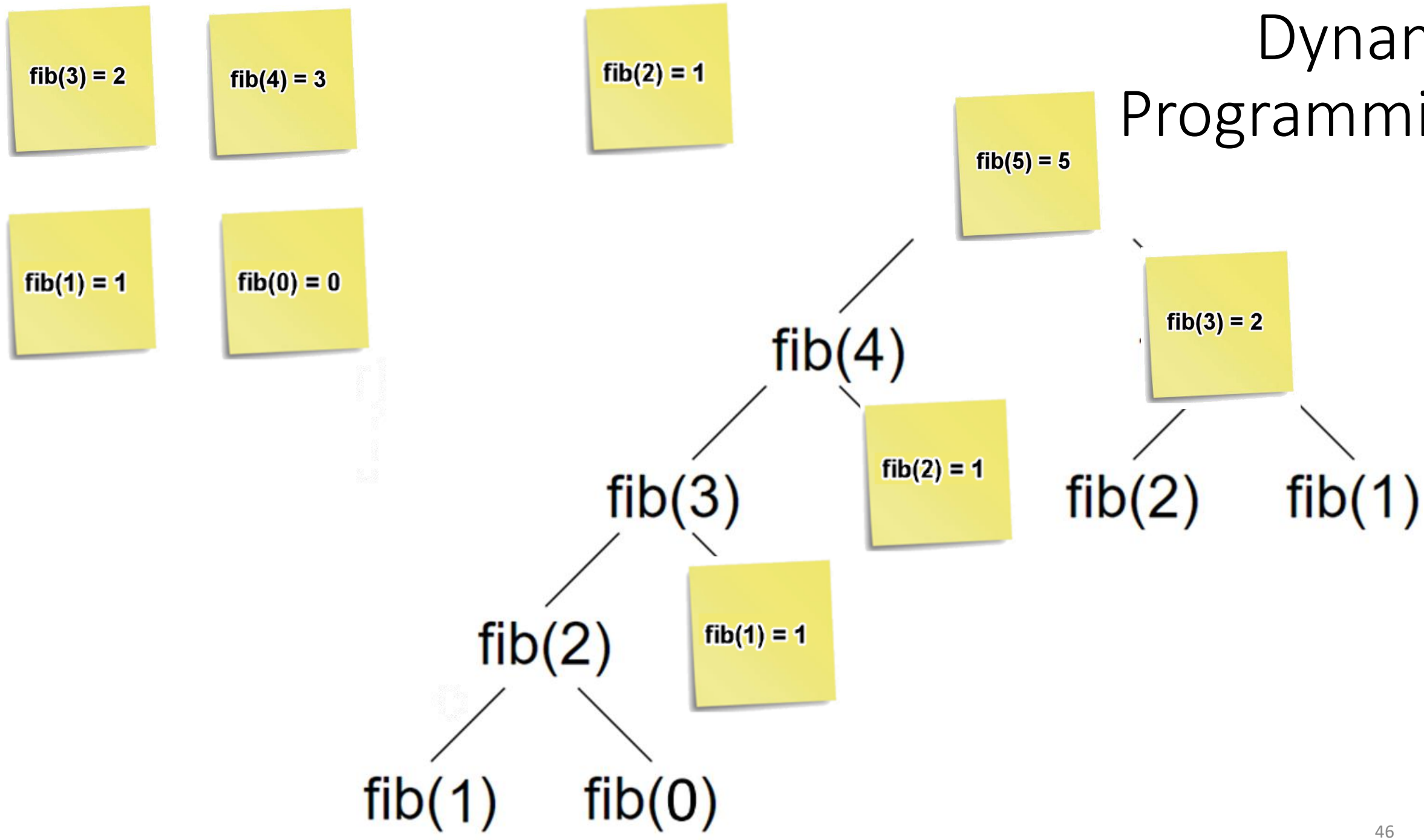
In computing, memoization or memoisation is an optimization technique used primarily to speed up computer programs by storing the results of expensive function calls and returning the stored result when the same inputs occur again.

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)

Dynamic Programming



Dynamic Programming

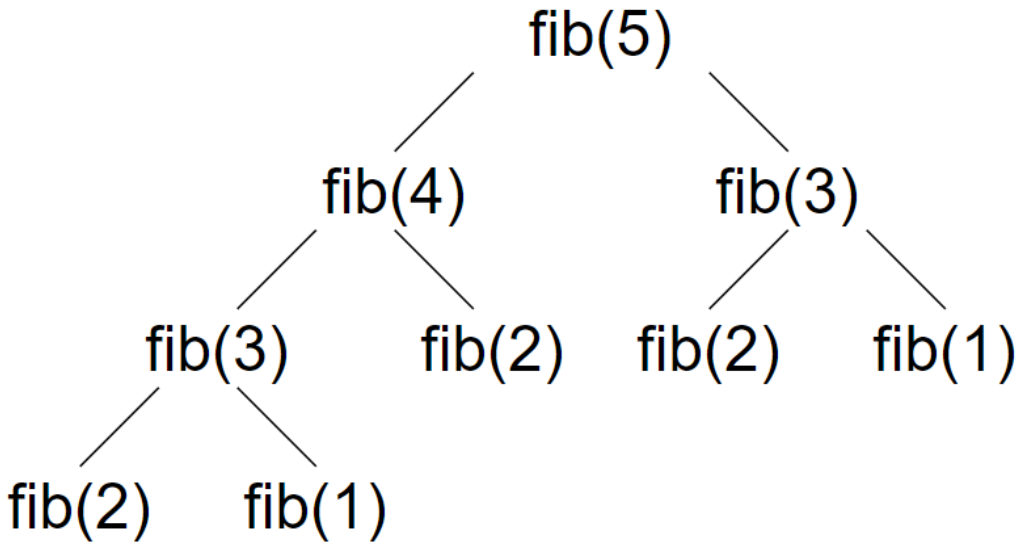
The memoized program for a problem is like the recursive version with a small modification that it looks into a lookup table before computing solutions.

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

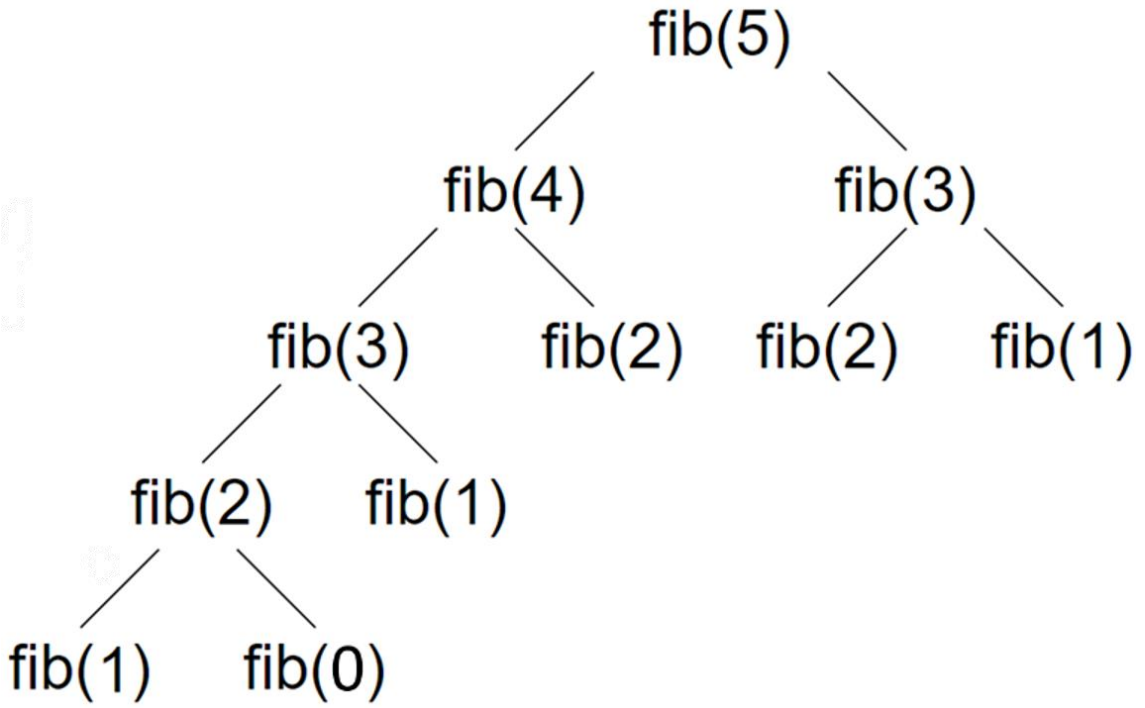
Whenever we need the solution to a subproblem, we first look into the lookup table.

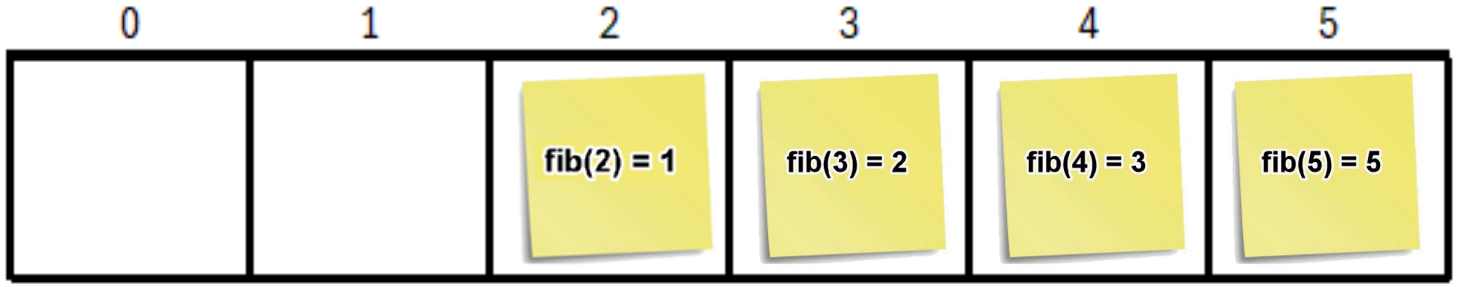
If the precomputed value is there, then we return that value; otherwise, we calculate the value and put the result in the lookup table so that it can be reused later.

```
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 Memo[100] = {};

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

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





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.

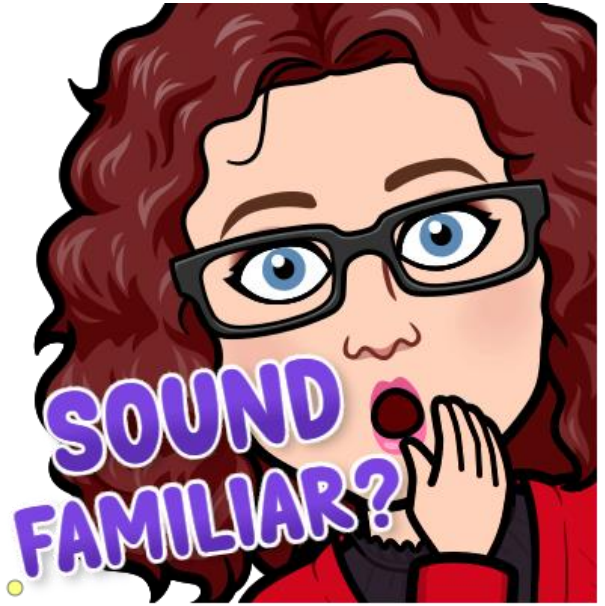
For example, Binary Search doesn't have common subproblems.

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.

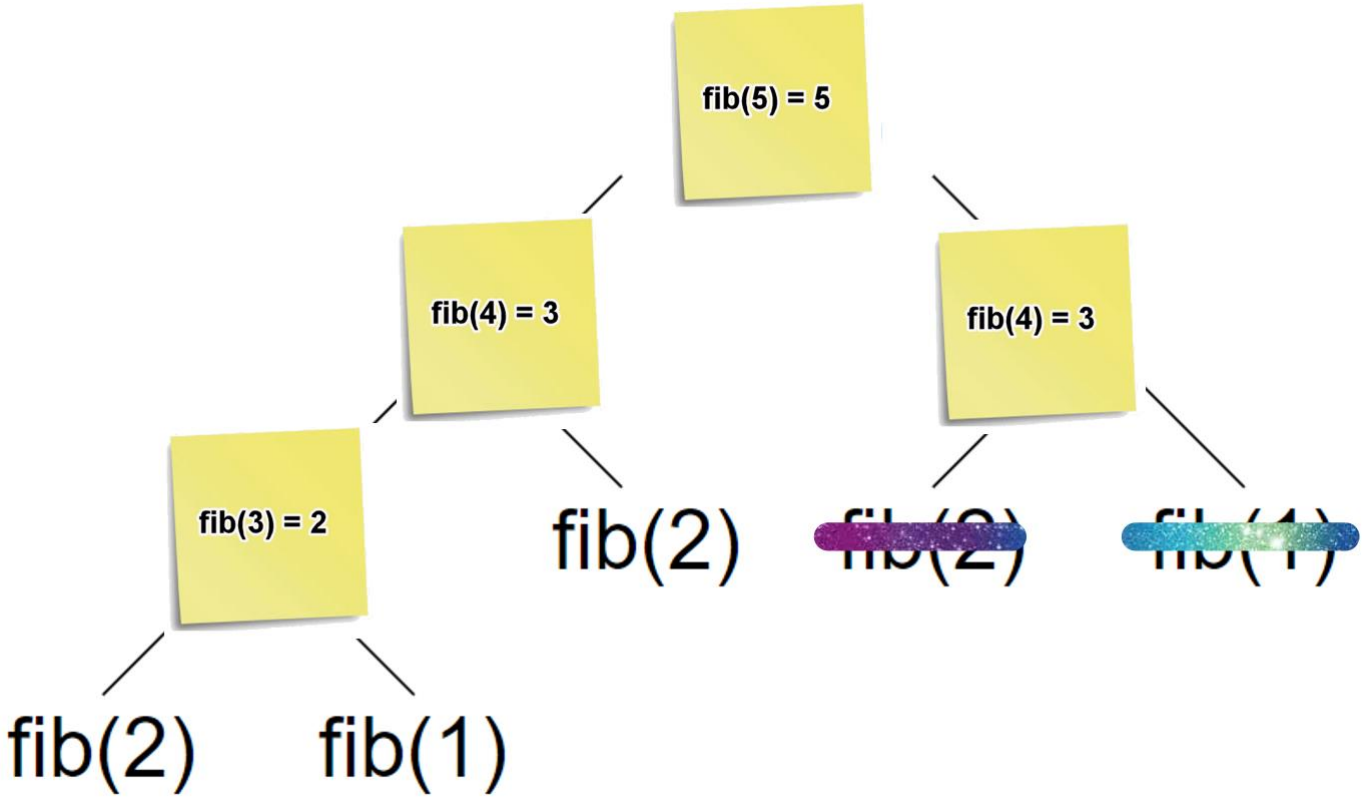
After the subproblems become sufficiently small, solving them becomes trivial.

Dynamic Programming

DP ≈ recursion + memoization

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

For any given value *k*, fib(*k*) only recurses the first time it is called.



Dynamic Programming

For any given value *k*, fib(*k*) only recurses the first time it is called.

Memoized calls cost Θ(1)

The number of non-memoized calls is *n* given that we are calculating fib(*n*).

The amount of nonrecursive work per call is Θ(1).

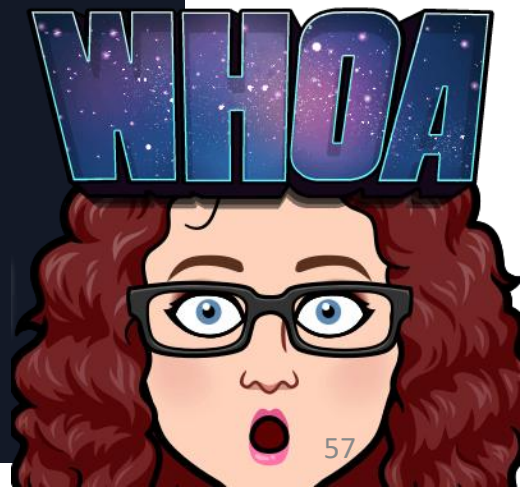
Total time of memoized version of Fibonacci is Θ(*n*)

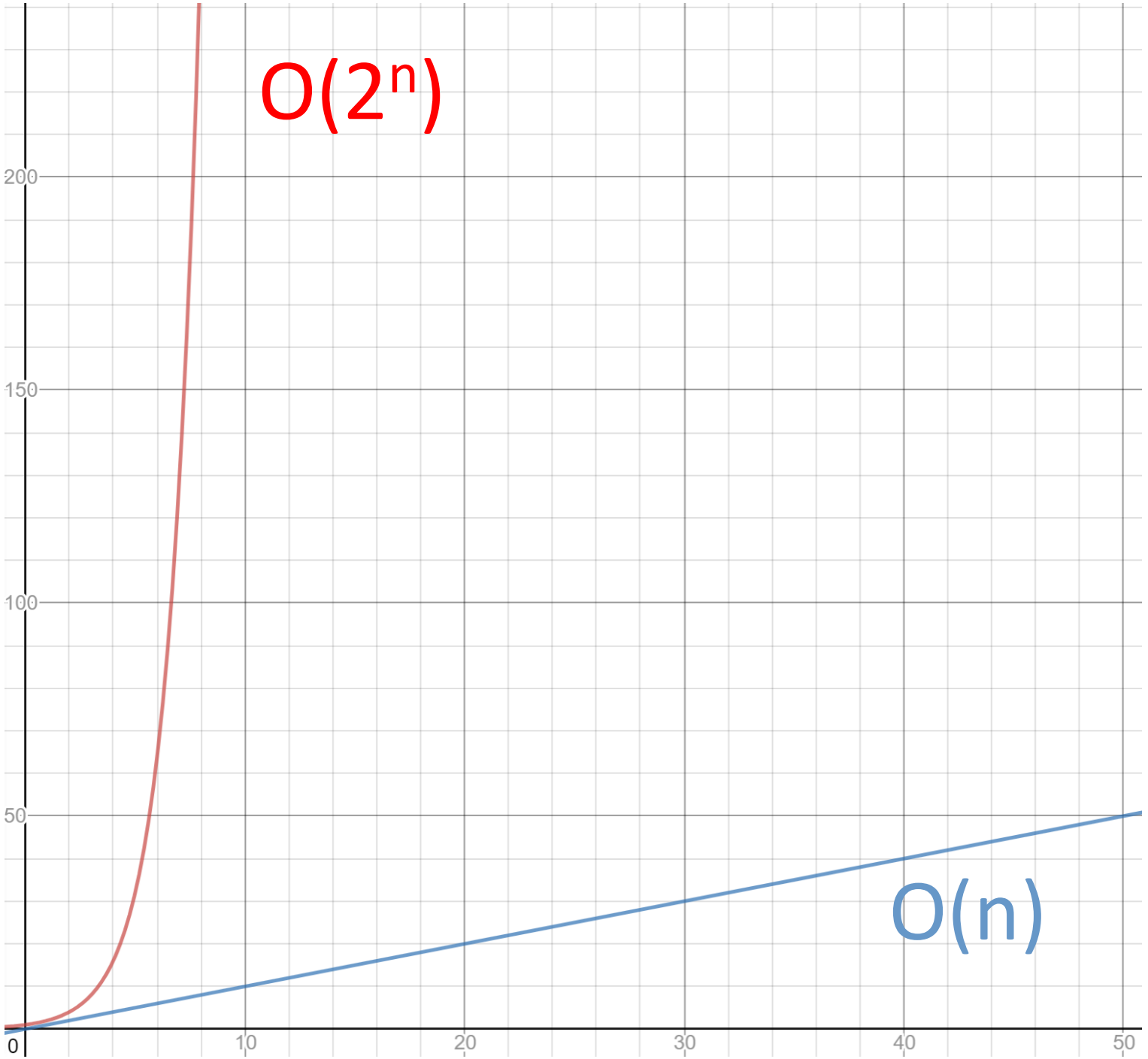
Terminal - student@maverick: /media/sf_VM/CSE3318

File Edit View Terminal Tabs Help

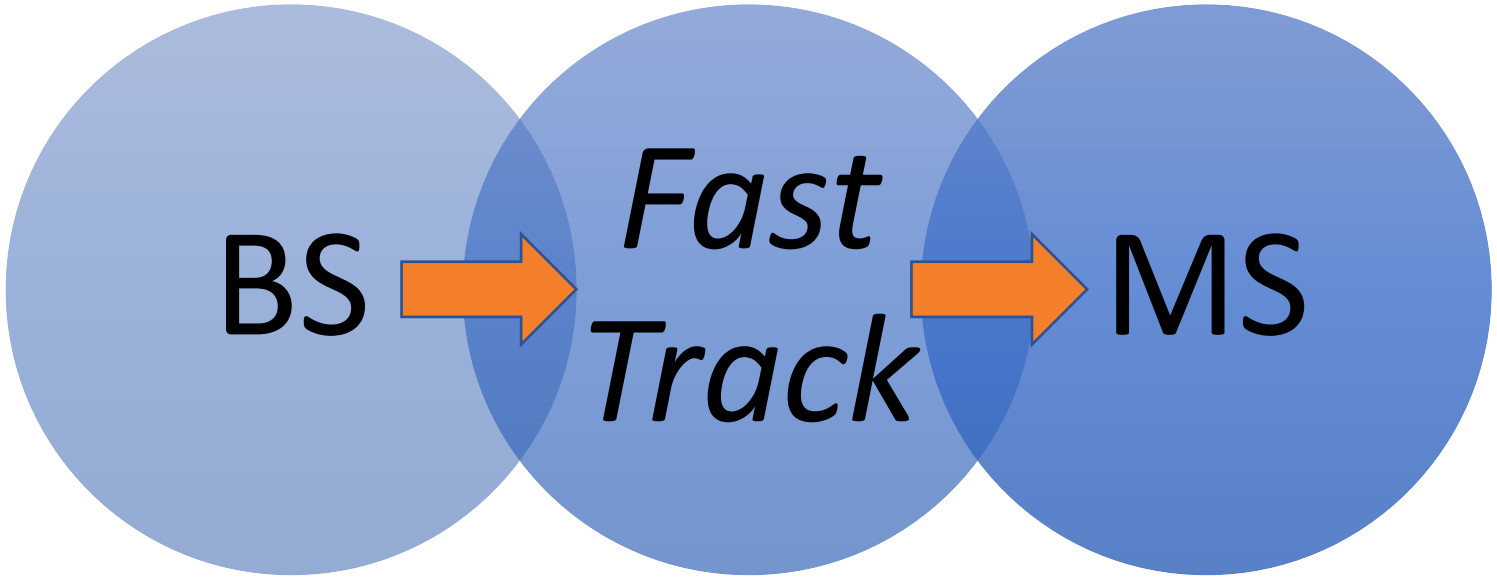
```
student@maverick:/media/sf_VM/CSE3318$
```

8x





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



UTA
Department of Computer Science and Engineering

Are you thinking about going to grad school?

Find out how to earn a Master of Science degree in 1 year with UTA's FastTrack program.

UNIVERSITY OF TEXAS ARLINGTON | DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Fast Track

M.S. in Computer Science

Master of Software Engineering

M.S. in Computer Engineering

60

UNIVERSITY OF TEXAS ARLINGTON | DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Fast Track

Outstanding undergraduate students in Computer Science, Software Engineering or Computer Engineering can begin satisfying requirements leading to a master's degree while completing their undergraduate studies.

61

M.S. Fast Track

Undergraduate students admitted to the Fast Track program may complete up to three graduate level courses that will count towards both their bachelors and master's degrees.

Successful Fast Track students will be able to join the master's program without completing the traditional application process.

Students who successfully complete the undergraduate Fast Track requirements do not pay an application fee or take the GRE.

Meet with an undergraduate advisor to discuss the program or email donna.french@uta.edu to see if you qualify.

62

M.S. Fast Track - Computer Science

Students who meet the following conditions may apply for the Fast Track Program

- Must be within 30 hours of completing the bachelor's degree
- Must have completed at least 30 hours of study at UTA
- Must have an overall GPA of at least 3.25
- Must have completed the following prerequisite foundation coursework with a minimum GPA of 3.3
  - o CSE 2312 – Computer Organization and Assembly Language Programming
  - o CSE 2315 – Discrete Structures
  - o CSE 3315 – Theoretical Concepts in Computer Science and Engineering
  - o CSE 3318 – Algorithms & Data Structures
  - o CSE 3320 – Operating Systems
  - o IE 3301 - Engineering Probability or MATH 3313 - Intro to Probability

63

M.S. Fast Track - Computer Science

Undergraduate Fast Track students must take a minimum of 2 and a maximum of 3 of the following graduate courses

Required

- o CSE 5311 – Design and Analysis of Algorithms

Choose 1 or 2 of the following

- o CSE 5301 – Data Analysis & Modeling Techniques
- o CSE 5306 – Distributed Systems
- o CSE 5317 – Design and Construction of Compilers
- o CSE 5331 – DBMS Models and Implementation Techniques
- o CSE 5360 – Artificial Intelligence I
- o CSE 5365 – Computer Graphics

64

M.S. Fast Track - Computer Engineering

Students who meet the following conditions may apply for the Fast Track Program

- Must be within 30 hours of completing the bachelor's degree
- Must have completed at least 30 hours of study at UTA
- Must have an overall GPA of at least 3.25
- Must have completed the following prerequisite foundation coursework with a minimum GPA of 3.3
  - o CSE 2312 – Computer Organization and Assembly Language Programming
  - o CSE 2315 – Discrete Structures
  - o CSE 3318 – Algorithms & Data Structures
  - o CSE 3320 – Operating Systems
  - o CSE 3442 – Embedded Systems I
  - o IE 3301 - Engineering Probability or MATH 3313 - Intro to Probability

65

M.S. Fast Track - Computer Engineering

Undergraduate Fast Track students must take a minimum of 2 and a maximum of 3 of the following graduate courses

Choose 2 or 3 of the following

- o CSE 5342 – Embedded Systems
- o CSE 5352 – IoT and Networking
- o CSE 5354 – Real-Time Operating Systems
- o CSE 5355 – Electromechanical Systems and Sensors
- o CSE 5356 – System on Chip (SoC) Design
- o CSE 5357 – Advanced Digital Logic Design
- o CSE 5358 – Microprocessor Systems

66

M.S. Fast Track - Software Engineering

Students who meet the following conditions may apply for the Fast Track Program

- Must be within 30 hours of completing the bachelor's degree
- Must have completed at least 30 hours of study at UTA
- Must have an overall GPA of at least 3.25
- Must have completed the following prerequisite foundation coursework with a minimum GPA of 3.3
  - o CSE 2312 – Computer Organization and Assembly Language Programming
  - o CSE 2315 – Discrete Structures
  - o CSE 3310 – Fundamental of Software Engineering
  - o CSE 3318 – Algorithms & Data Structures
  - o CSE 3320 – Operating Systems
  - o IE 3301 - Engineering Probability or MATH 3313 - Intro to Probability

67

M.S. Fast Track - Software Engineering

Undergraduate Fast Track students must take a minimum of 2 and a maximum of 3 of the following graduate courses

Required

- o CSE 5311 – Design and Analysis of Algorithms

Choose 1 or 2 of the following

- o CSE 5301 – Data Analysis & Modeling Techniques
- o CSE 5321 – Software Testing
- o CSE 5322 – Software Design Patterns
- o CSE 5325 – Software Engineering: Management, Maintenance, and Quality Assurance

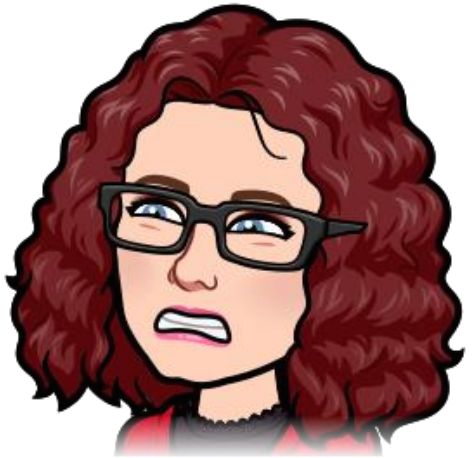
68

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?

$O(\log_2 n)$

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.

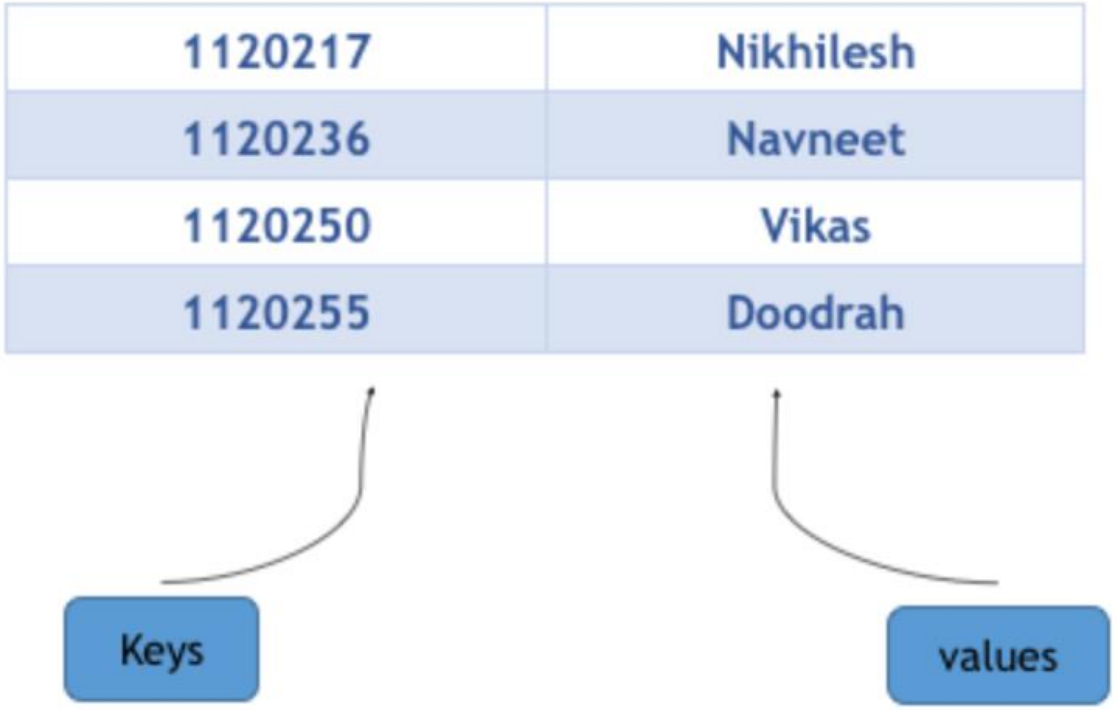
Hashing

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.

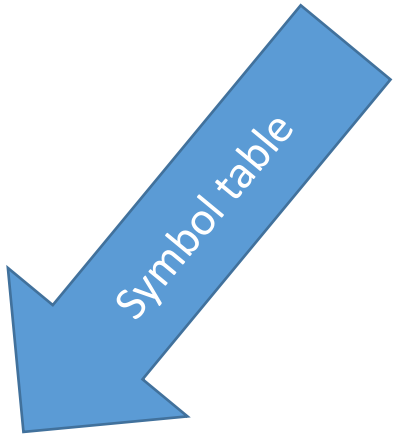


Hashing

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.

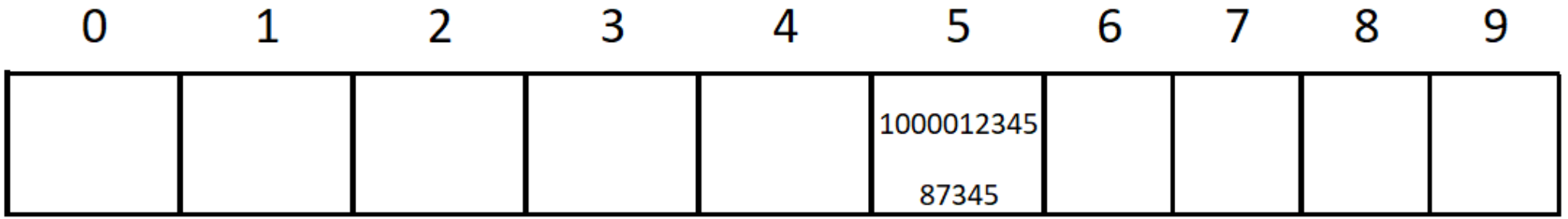
```
int GetHashCode(unsigned int student_id)
{
   return ((student_id) % MaxHashTableSize);
}
```

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.



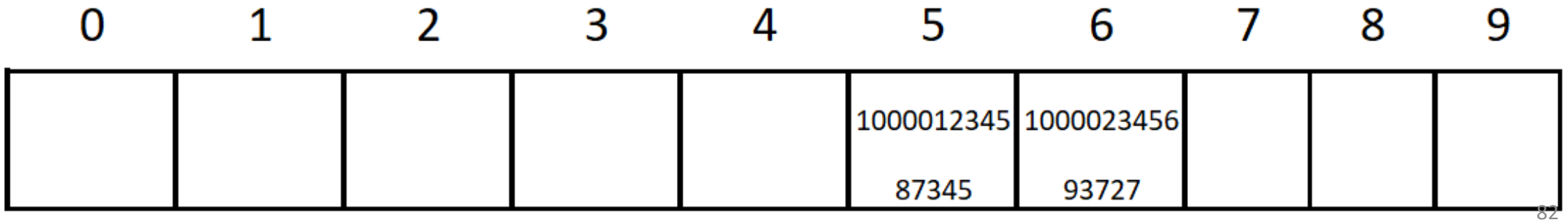
Hash Table

```
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.

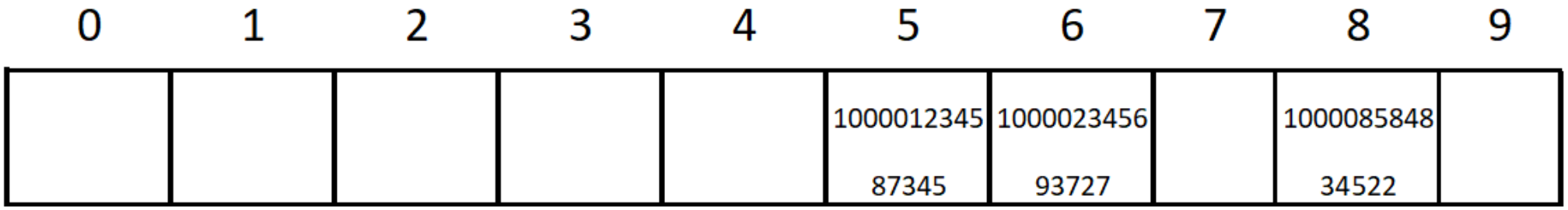


Hash Table

```
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.



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.

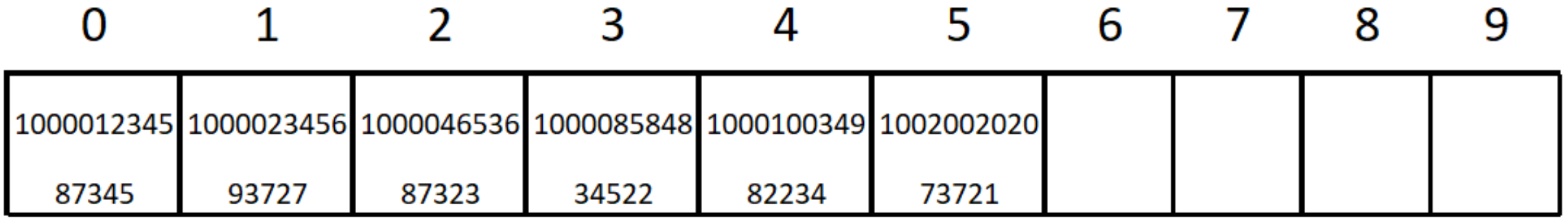
```
int GetHashCode(unsigned int student_id)
{
    return (student_id % MaxHashTableSize);
}
Input : 1000012345
Output : 1000012345 % 10 = 5
```

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

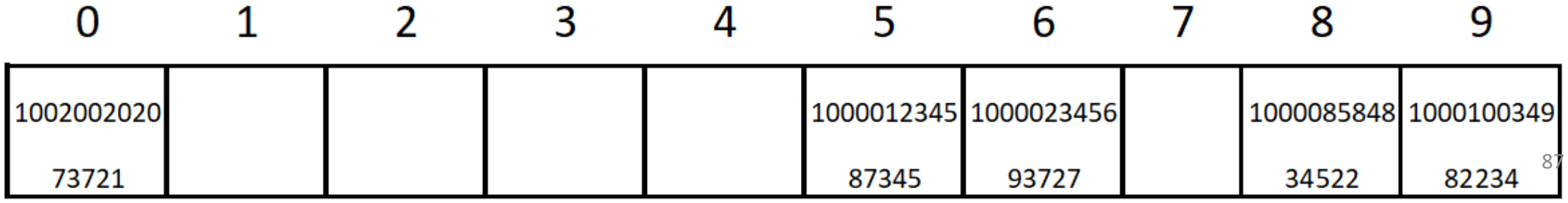
Hash Table

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



Hash Table

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



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.

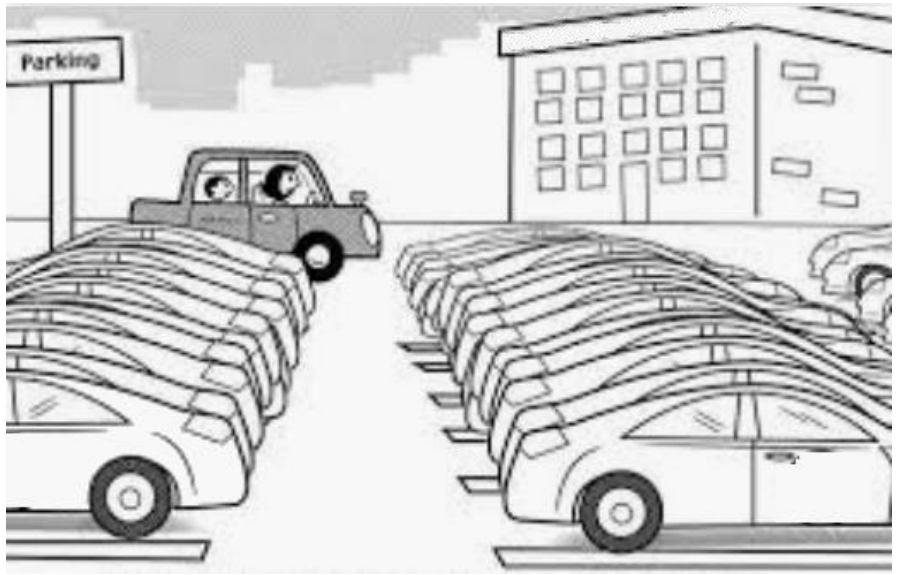
When inserting an item, if the array cell is already occupied, then open addressing takes the next cell.

Collision Resolution

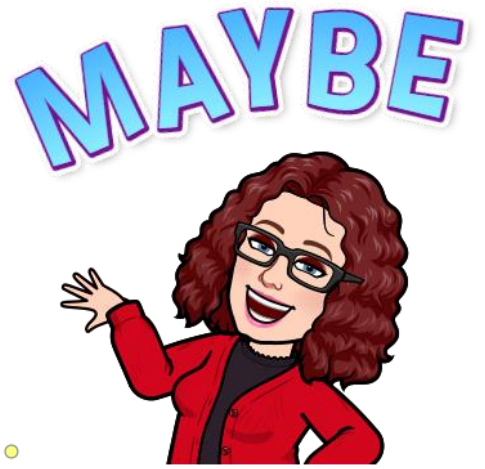
Open Addressing

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.

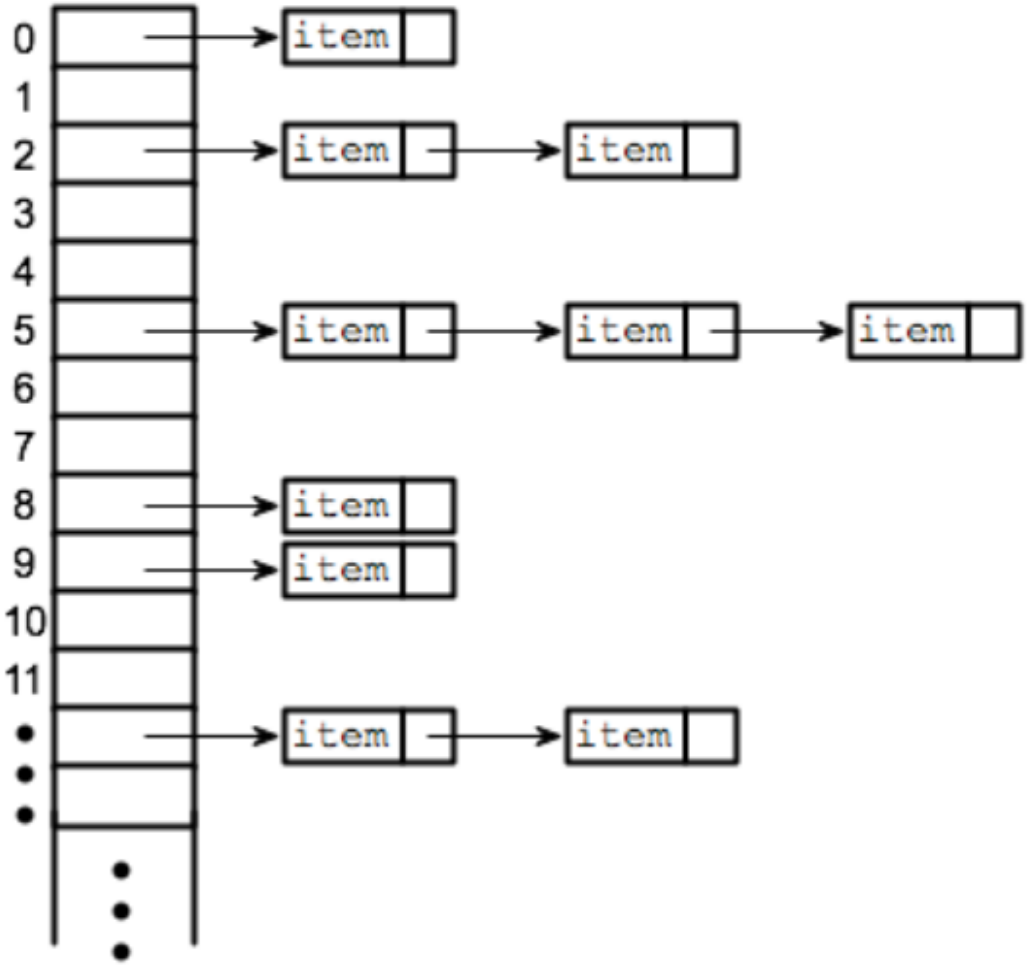
Collision Resolution

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.

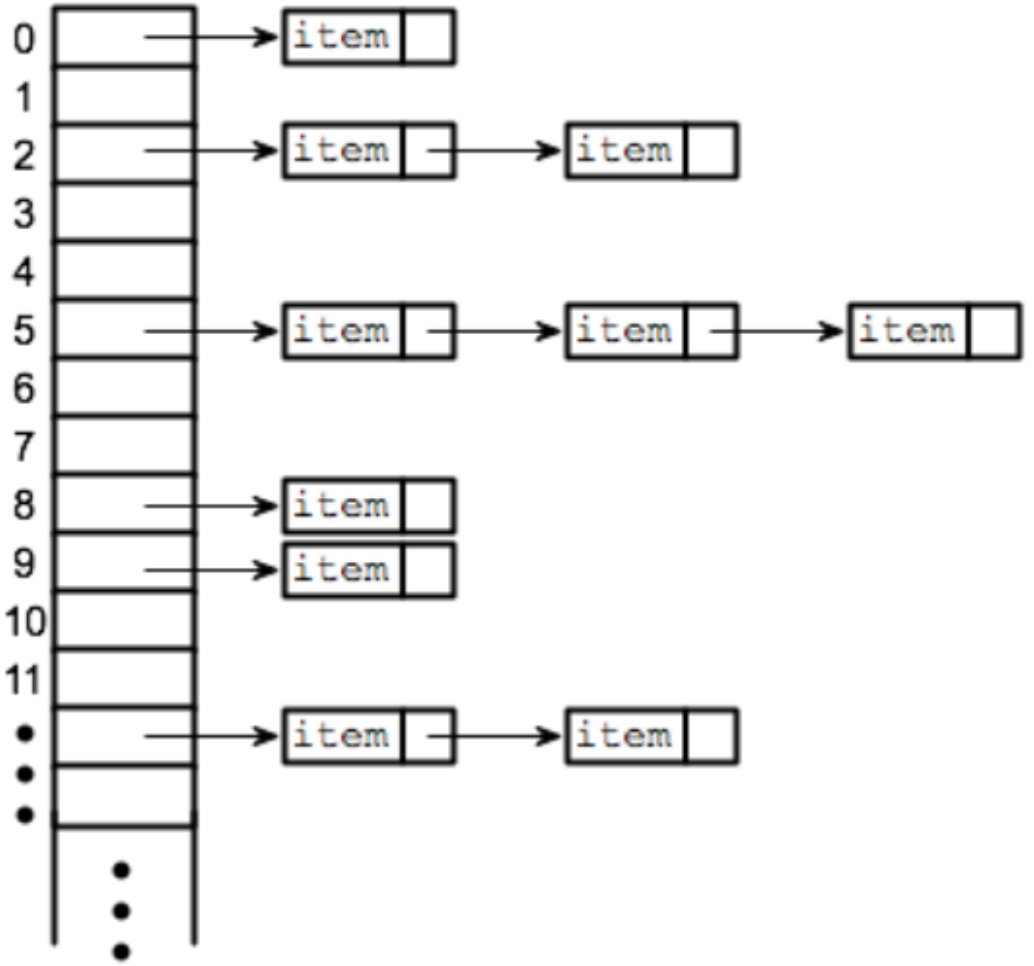


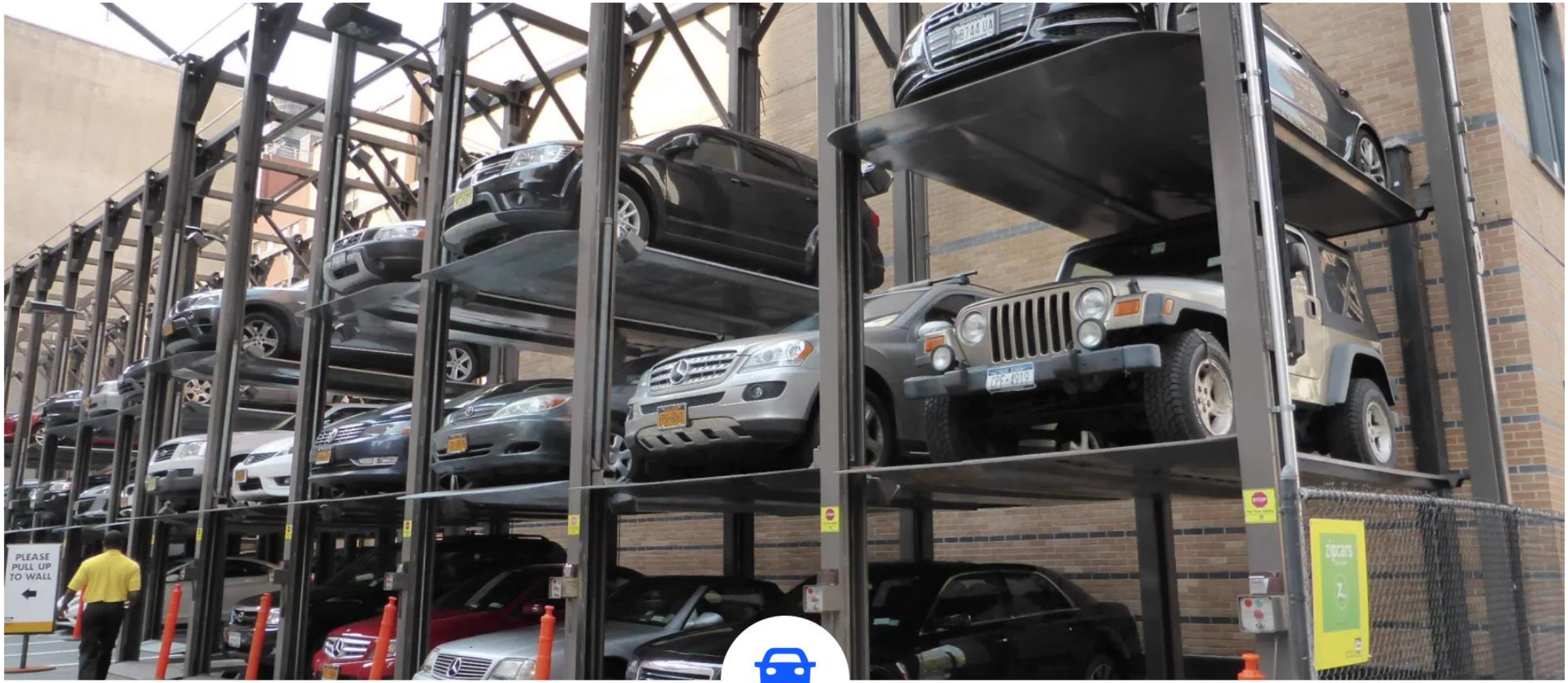
Collision Resolution

Separate chaining (open hashing)

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

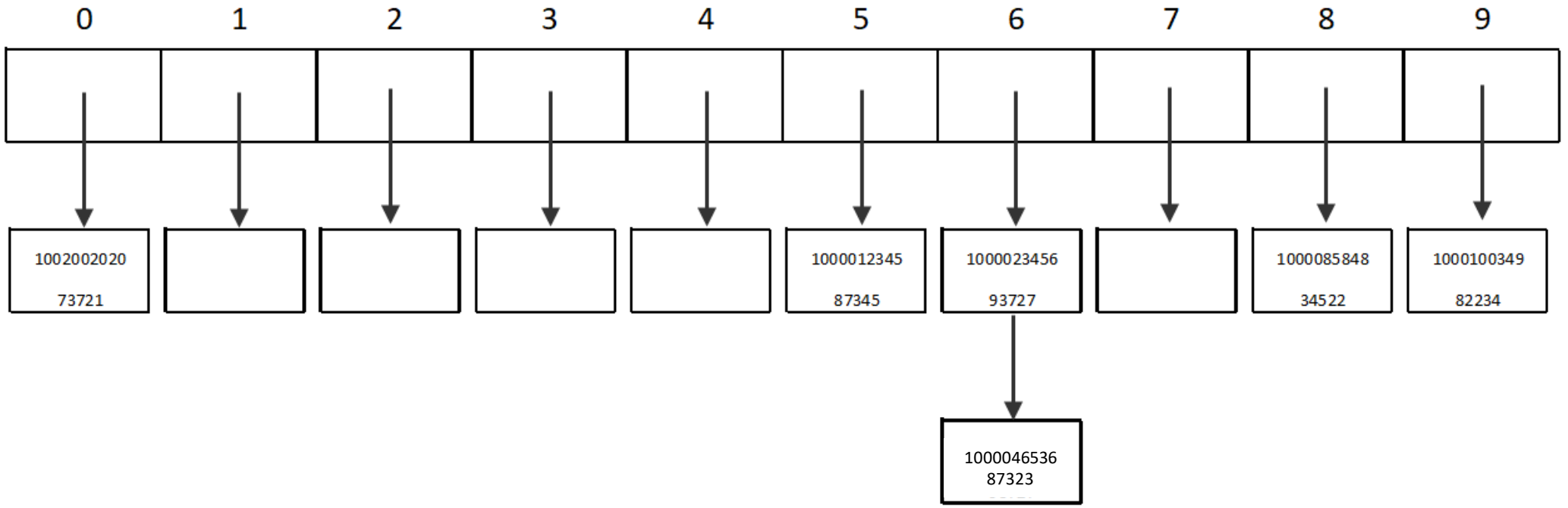
If there is any collision (i.e. two different elements have same hash value) then store both elements in the same linked list.




PLEASE
PULL UP
TO WALL

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.

If we are NOT using separate chaining for conflict resolution, then what happens if our array size is 10 and we need to store 11 items?

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.

Separate Chaining	Open Addressing
Chaining is simpler to implement.	Open Addressing requires more computation.
Does not run out of room.	Array may fill up
Less sensitive to the hash function and load factor.	Requires extra care to avoid clustering and the load factor.
Mostly used when it is unknown how many and how frequently keys may be inserted or deleted.	Used when the frequency and number of keys is known.
Cache performance is not good as keys are stored using linked list.	Provides better cache performance as everything is stored in the same table.
Waste of space - some parts of hash table are never used	A cell can be used even if an input doesn’t map to it.
Uses extra space for linked list pointers.	No links – everything in table

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?

What can we use for a hashing function?

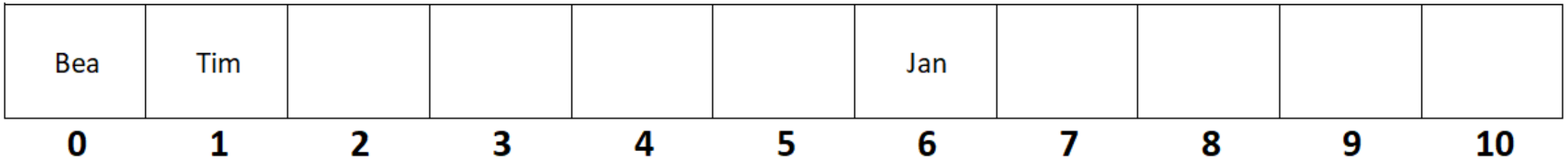
How about ASCII values?

Jan = J (74) + a (97) + n (110) = 281

Now MOD by size of array

281 % 11 = 6

Hashing



Tim = T(84) + i(105) + m(109) = 298

Now MOD by size of array

298 % 11 = 1

Bea = B(66) + e(101) + a(97) = 264

Now MOD by size of array

298 % 11 = 0

Hashing

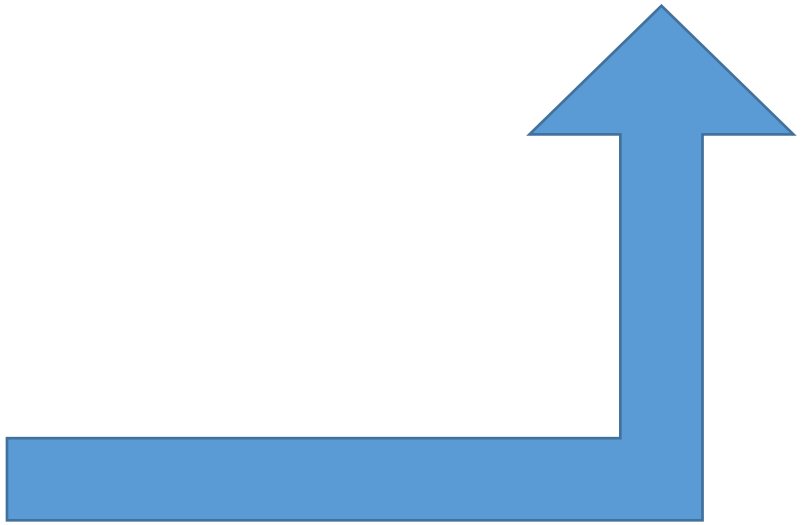
If we apply the hash function to the rest of the names, our hash table will look like…

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

Ada = A(65) + d(100) + a(97) = 262 % 11 = 9

Time complexity?

Θ (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.

215 673 2387 folds to become index 872.

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**

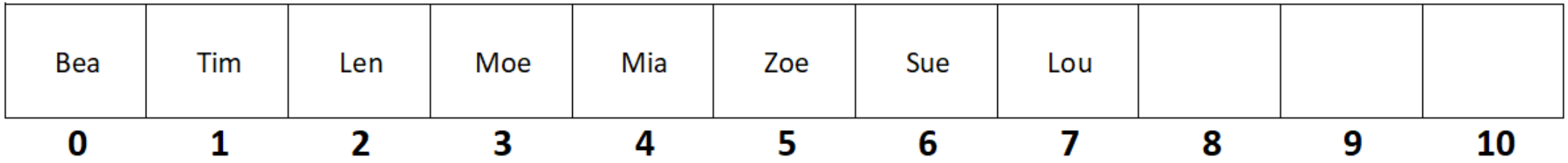
What happens when we try to add Len?

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.



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.

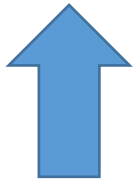
Open Addressing with linear probing would add Max at cell 9.

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….

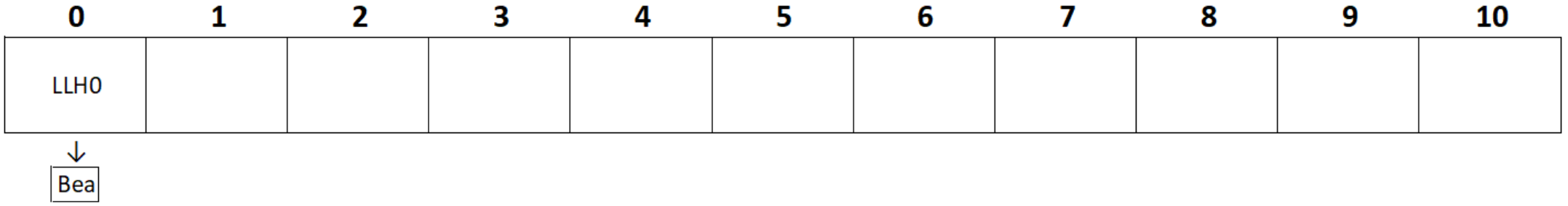
Bea	Tim			Mia	Zoe					
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**

Hashing with Separate Chaining

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

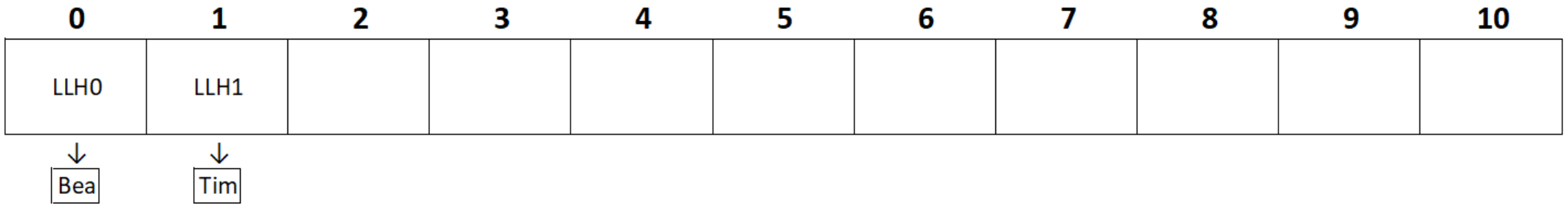
The linked list head is stored in the array – the array is now a `char` pointer array rather than just a `char` array.

Bea hashes to cell 0.

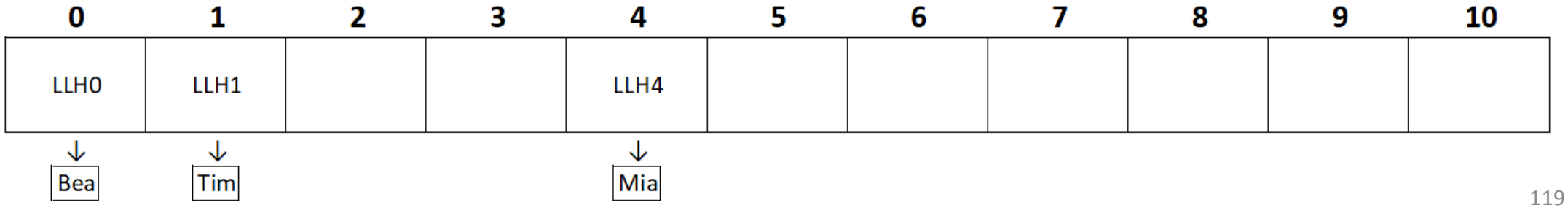


Hashing with Separate Chaining

Tim hashes to cell 1.

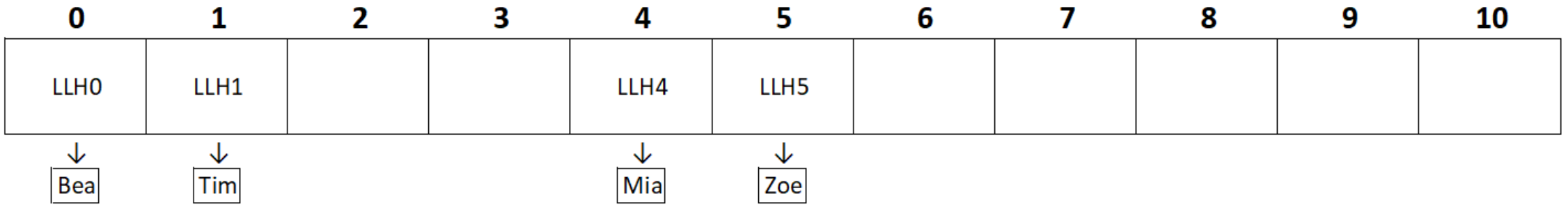


Mia hashes to cell 4.

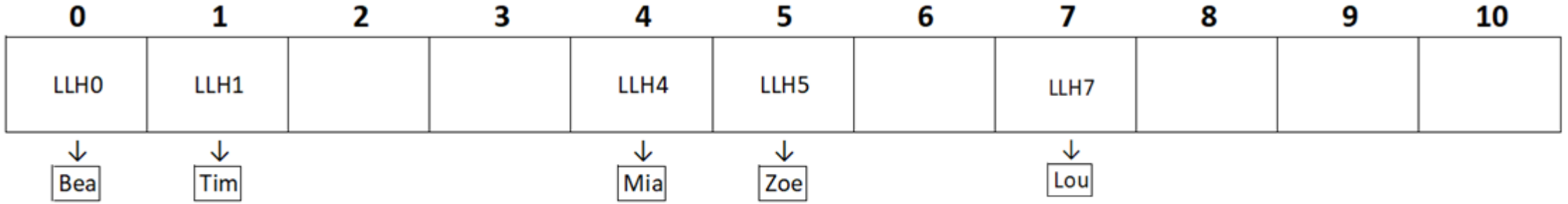


Hashing with Separate Chaining

Zoe hashes to cell 5.

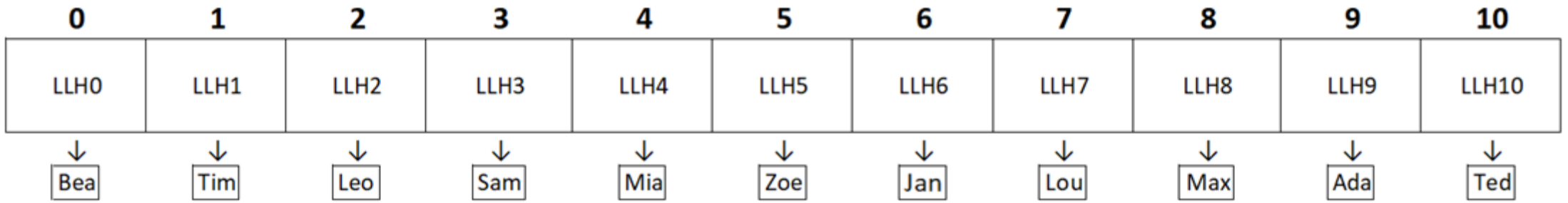


Lou hashes to cell 7.

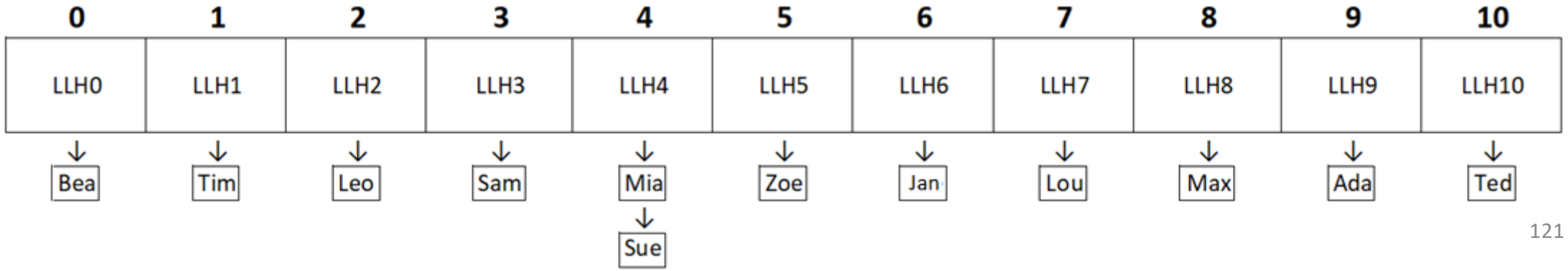


Hashing with Separate Chaining

We can add the rest...

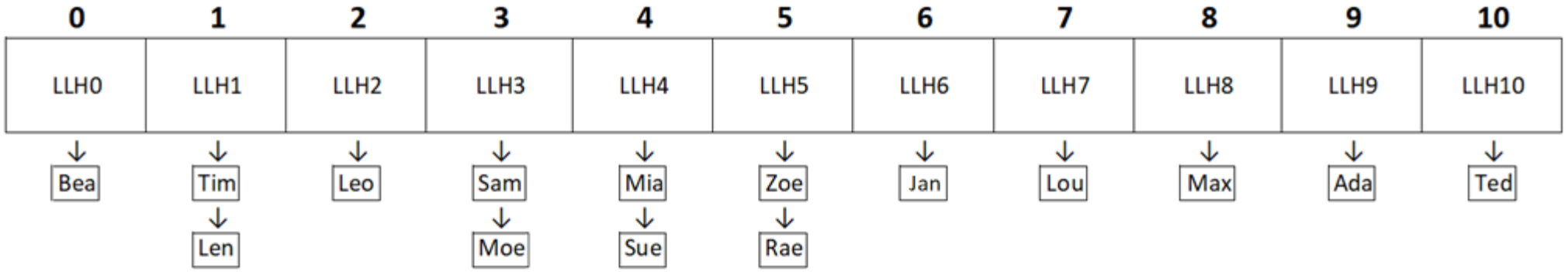


What happens when we try to add Sue which hashes to 4?



Hashing with Separate Chaining

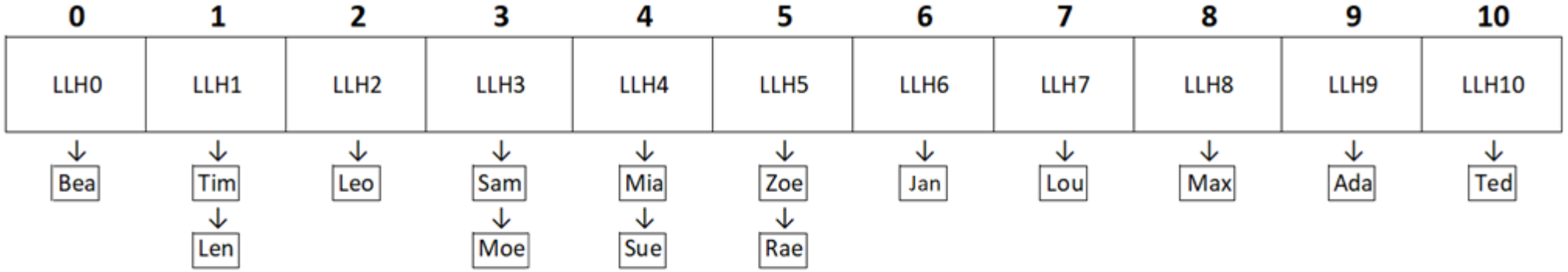
What happens when we add Len(1), Moe(3), Lou(7)



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

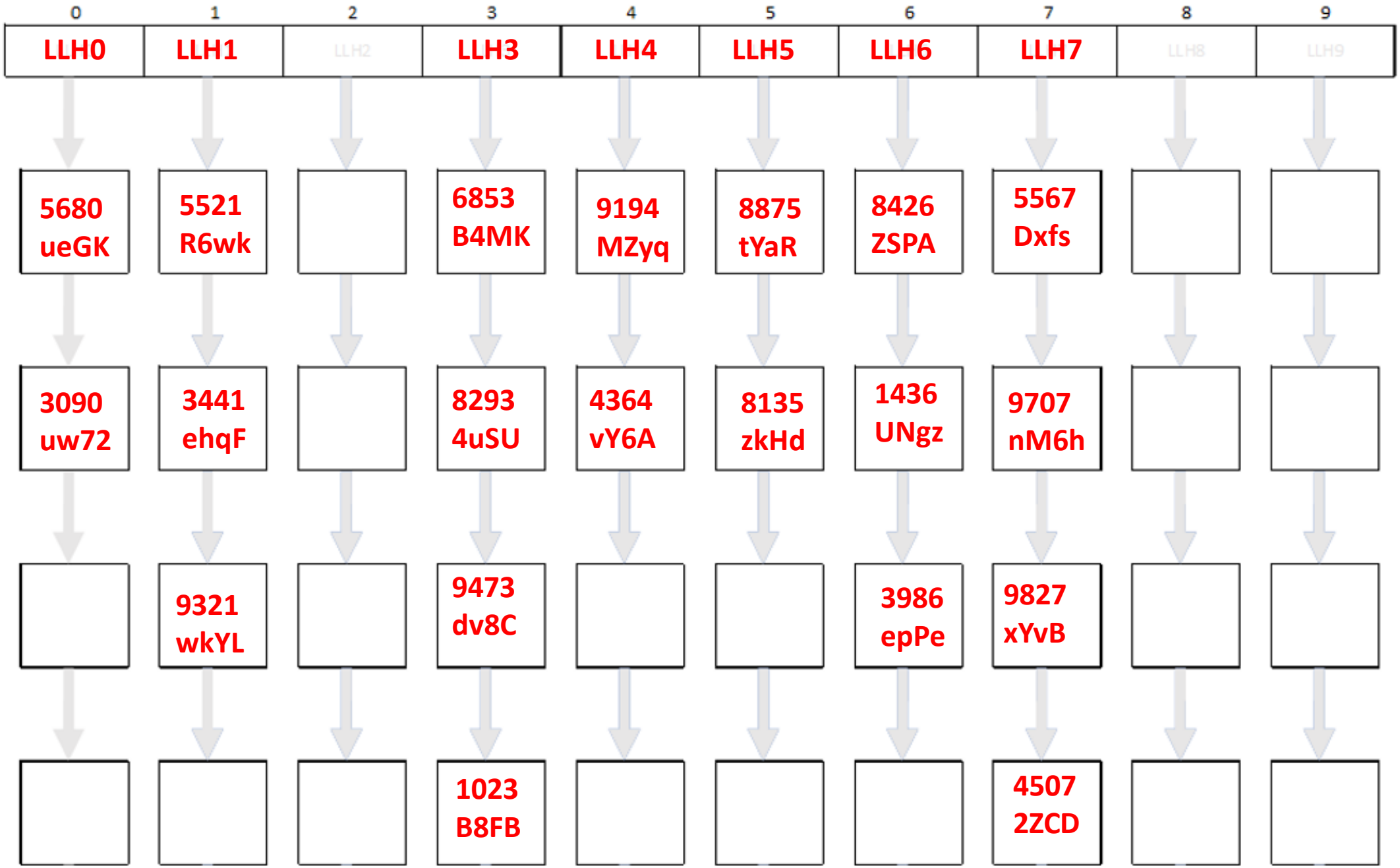
- *Associative arrays*: Hash tables are commonly used to implement many types of in-memory tables. They are used to implement associative arrays (arrays whose indices are arbitrary strings or other complicated objects).
- *Database indexing*: Hash tables may also be used as disk-based data structures 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

Part A – Complete the hash version using the same values/Part A data

0	1	2	3	4	5	6	7	8	9
8000 wyGK	7521 wsaR	0762 gL6h	5343 FuMK	3624 Gdgz	9035 728C	9466 BTyq	8697 uEPe	6708 ycfs	2699 xFuA

Part B - Create the hash table using the values from the Part B data. Use linear probing for resolving conflicts.

0	1	2	3	4	5	6	7	8	9
8000 Uezx	8697 epM4	0762 nMsx	5413 R63L	5343 B4P3	8755 4WF2	9466 BtMZ	3367 eh7d	6708 Dx4e	2699 ZSPX


0
1
2
3
4
5
6
7
8
9
LLH0
LLH1
LLH2
LLH3
LLH4
LLH5
LLH6
LLH7
LLH8
LLH9
5680
ueGK
5521
R6wk
6853
B4MK
9194
MZyq
8875
tYaR
8426
ZSPA
5567
Dxfs
3090
uw72
3441
ehqF
8293
4uSU
4364
vY6A
8135
zkHd
1436
UNgz
9707
nM6h
9321
wkYL
9473
dv8C
3986
epPe
9827
xYvB
1023
B8FB
4507
2ZCD

```
Pichu|1.00|4.4|B|Tiny Mouse|Static|1|172
Pikachu|1.04|13.2|B|Mouse|Static|2|025
Raichu|2.07|66.1|B|Mouse|Static|3|026
Charmander|2.00|18.7|B|Lizard|Blaze|1|004
Charmeleon|3.7|41.9|B|Flame|Blaze|2||005
Charizard|5.7|199.5|B|Flame|Blaze|3|006
Squirtle|1.08|19.8|B|Tiny Turtle|Torrent|1|007
Wartortle|3.03|49.6|B|Turtle|Torrent|2|008
Blastoise|5.03|188.5|B|Shellfish|Torrent|3|009
Igglybuff|1.00|2.2|B|Balloon|Cute Charm, Competitive|1|174
Jigglypuff|1.08|12.1|B|Balloon|Cute Charm,Competitive|2|039
Wigglytuff|3.03|26.5|B|Balloon|Cute Charm,Competitive|3|040
Ponyta|3.03|66.1|B|Fire Horse|Run Away, Flash Fire|1|077
Rapidash|5.07|209.4|B|Fire Horse|Run Away, Flash Fire|2|078
Psyduck|2.07|43.2|B|Duck|Damp, Cloud Nine|1|054
Golduck|5.07|168.9|B|Duck|Damp. Cloud Nine|2|055
Vaporeon|3.03|63.9|B|Bubble Jet|Water Absorb|2|134
Eevee|1.00|14.3|B|Evolution|Run Away, Adaptability|1|133
Jolteon|2.07|54.0|B|Lightning|Volt Absorb|3|135
Meowth|1.04|9.2|B|Scratch Cat|Pickup, Technician|1|052
Persian|3.03|70.5|B|Classy Cat|Technician, Limber|2|053
Bulbasaur|2.04|15.2|B|Seed|Overgrow|1|001
Ivysaur|3.03|28.7|B|Seed|Overgrow|2|002
Venusaur|6.07|220.5|B|Seed|Overgrow|3|003
Smoochum|1.04|13.2|F|Kiss|Oblivious, Forewarn|1|238
Jynx|4.07|89.5|F|Human Shape|Oblivious, Forewarn|2|124
Growlithe|2.04|41.9|B|Puppy|Intimidate, Flash Fire|1|058
Arcanine|6.03|341.7|B|Legendary|Intimidate, Flash Fire|2|059
Finneon|1.04|15.4|B|Wing Fish|Swift Swim, Storm Drain|1|456
Lumineon|3.11|52.9|B|Neon|Swift Swim, Storm Drain|2|457
Ditto|1.00|8.8|U|Transform|Limber|0|132
Nidoran|1.04|15.4|F|Poison Pin|Poison Point, Rivalry|1|029
Nidorina|2.07|44.1|F|Poison Pen|Poison Point, Rivalry|2|030
```

```c
typedef struct Pokemon
{
    char *name;
    float height;
    float weight;
    char gender;
    char *category;
    char *abilities;
    int evolution;
    int national_pokedex_number;
    struct Pokemon *next_ptr;
}
POKEMON;
```

Coding Assignment 6

CSE 3318

Coding Assignment 6 is over creating a hash table and using it to do the 3 basic dictionary actions - insert, delete and search.

Your hash table will be created using an integer array with separate chaining to resolve conflicts.

Step 1

For this assignment, you need to choose a subject for your dictionary. For my version of the assignment, I chose Pokémon – you may not also use Pokémon – you need to pick your own subject. Pick a subject that allows you to gather enough information to fulfill the minimum requirements of the assignment.

Structure

You will need to create a `struct` to hold each entry of the dictionary. My Pokémon example is

```
typedef struct Pokemon
{
    char *name;
    float height;
    float weight;
    char gender;
    char *category;
    char *abilities;
    int evolution;
    int national_pokedex_number;
    struct Pokemon *next_ptr;
}
POKEMON;
```

In this `struct`, the `char` pointers are used instead of `char` arrays since it is not known how much space those entries will need. Memory to hold those fields will be dynamically allocated using `malloc()` and the pointer returned by `malloc()` will be stored in the struct.

For your assignment, you will need a `struct` with a minimum of 5 fields – the parts shown in **bold** are required.

```
typedef struct YourStructName
{
   char *chararrayfield;
   float floatfield;
   char charfield;
   int intfield;
   struct YourStructName *next_ptr;
}
YOURSTRUCTNAME;
```

Your `struct` must be `typedef`ed and you will be expect to use the `typedef` throughout the program. You may add more fields if you want your dictionary to hold more information. Your member names must adequately describe the data being stored in the variable. Put your typedef in a text file and name it `Code6_xxxxxxxxxx_struct.txt` where `xxxxxxxxxx` is your student id.

Input File

Your input file must contain a minimum of 30 pipe delimited records that represent your dictionary entries. Here's the first 10 lines of my Pokédex file.

```
Pichu|1.00|4.4|B|Tiny Mouse|Static|1|172
Pikachu|1.04|13.2|B|Mouse|Static|2|025
Raichu|2.07|66.1|B|Mouse|Static|3|026
Charmander|2.00|18.7|B|Lizard|Blaze|1|004
Charmeleon|3.7|41.9|B|Flame|Blaze|2||005
Charizard|5.7|199.5|B|Flame|Blaze|3|006
Squirtle|1.08|19.8|B|Tiny Turtle|Torrent|1|007
Wartortle|3.03|49.6|B|Turtle|Torrent|2|008
Blastoise|5.03|188.5|B|Shellfish|Torrent|3|009
Igglybuff|1.00|2.2|B|Balloon|Cute Charm, Competitive|1|174
```

The fields in each entry must be pipe delimited and must be listed in the same order as your `struct`. Save this as a text file named `Code6_xxxxxxxxxx_InputFile.txt` where `xxxxxxxxxx` is your student id.

Hashing Function

Pick one of the methods discussed in class for creating your hash function (numeric keys, alphanumeric keys, folding). Pick which field(s) from your `struct` will be used for the hash function. Remember that whatever method you choose should result in a value that you can MOD with the hash table define to get an index that will always be within your array.

Write a small C program that prompts for input and calls your hash function and prints the result of the hash. You are proving that your hash function will work. Here's an example – your version may be slightly different depending on your input. THIS CODE IS ONLY FOR SHOWING THAT YOUR HASH FUNCTION WORKS. It is ONLY an example – modify it as needed to show that your hash function will work.

```
#include <stdio.h>
#define HASHTABLESIZE 10
int MyHashFunction(pass in value being used in hash)
{
   perform hash method
   return value % HASHTABLESIZE;
}

int main(void)
{
   char HashValue[20];
   printf("Enter value ");
   scanf("%s", HashValue);
   printf("The hash value for %s is %d\n”, HashValue, MyHashFunction(HashValue));
   return 0;
}
```

Compile your version and run and confirm that your function works. Save this code as `Code6_xxxxxxxxxx_HashFunction.c` where `xxxxxxxxxx` is your student id.

Step 2

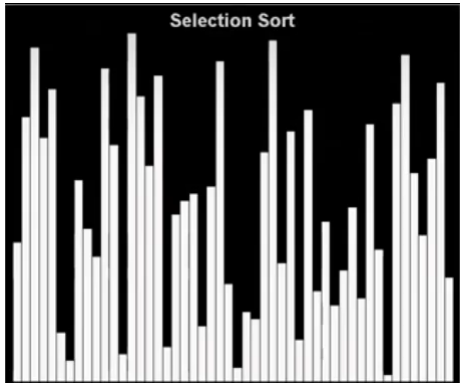
Submit your input file, `struct` and hashing function for approval. Submit these to me by email. If you want to zip the files, you can, but you can just send them as attachments. No pictures or screenshots will be accepted.

Three files must be submitted via email

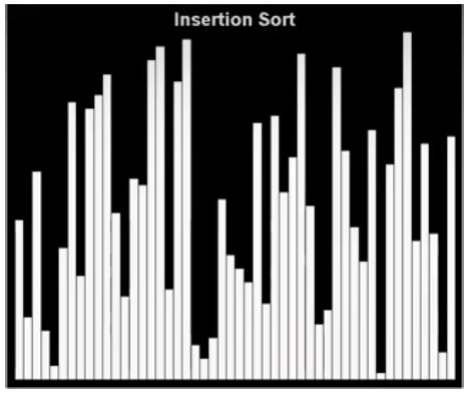
```
Code6_xxxxxxxxxx_struct.txt
Code6_xxxxxxxxxx_InputFile.txt
Code6_xxxxxxxxxx_HashFunction.c
```

Once you get approval, you will receive the rest of the instructions for the assignment. If you do not get approval, any code you turn in for Coding Assignment 6 will be assigned a grade of 0 regardless.

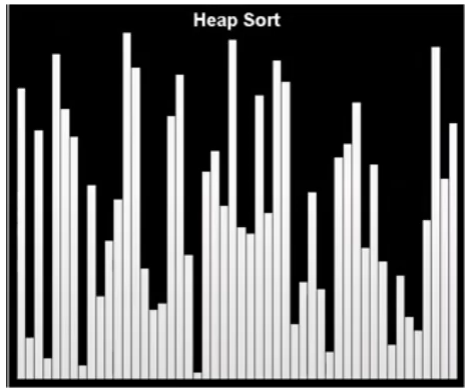
Sorts



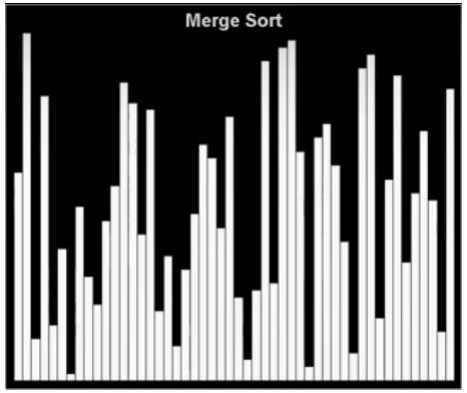
$O(n^2)$



$O(n^2)$

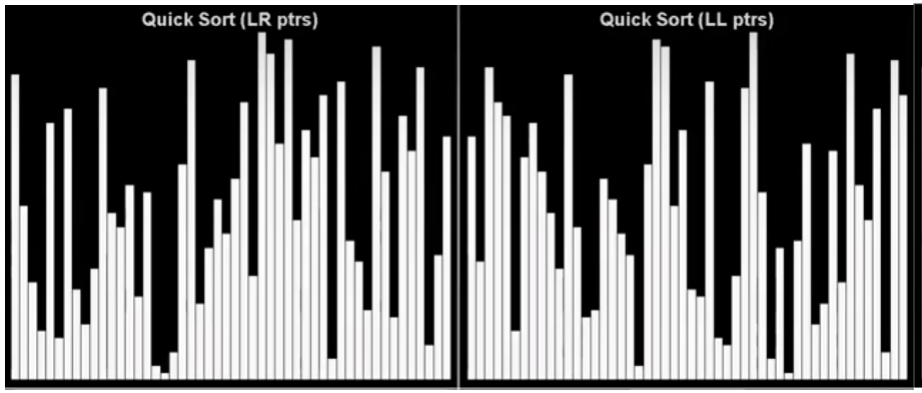


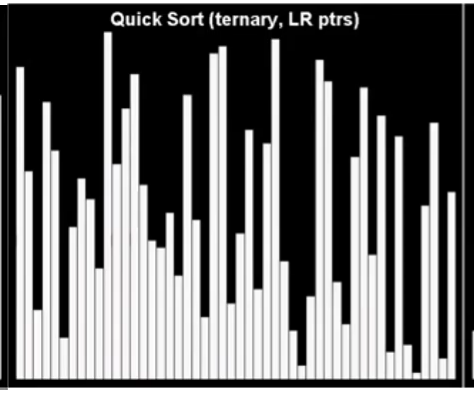
$O(n\log_2 n)$

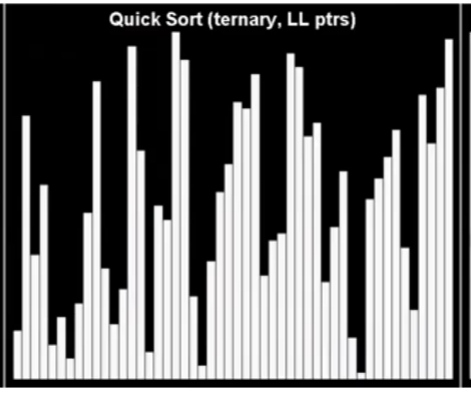


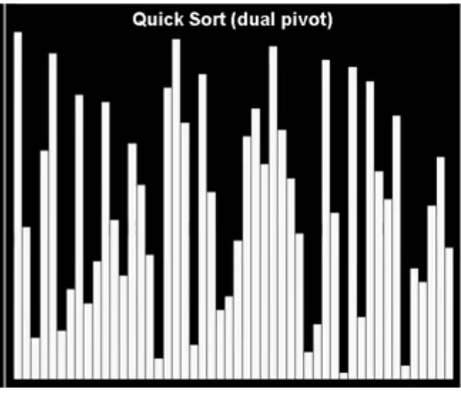
$O(n\log_2 n)$

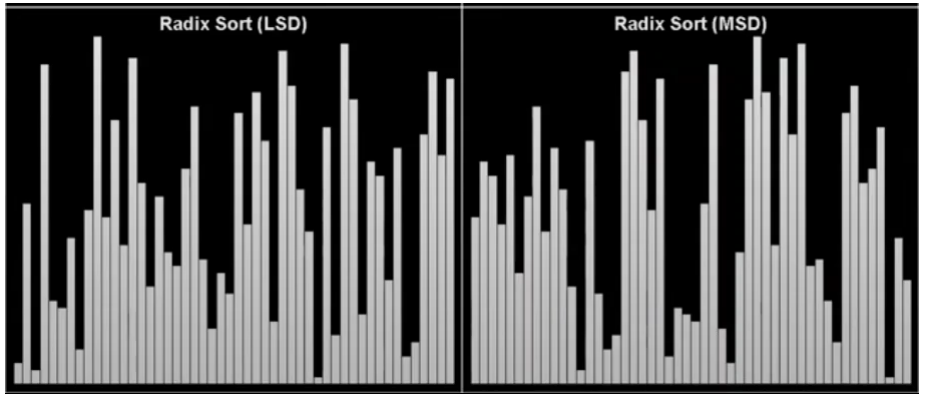
$O(n\log_2 n)$
$O(n^2)$

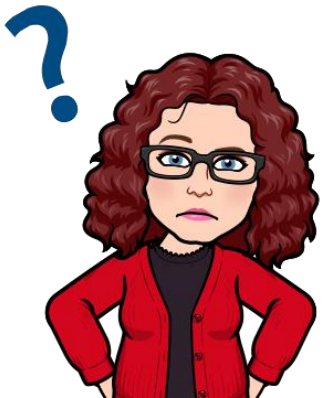












Radix Sort

Let's start with a list of numbers to sort

645, 86, 182, 35, 601, 8

Step 1

All numbers to be sorted need to be the same length.

Radix Sort

645, 86, 182, 35, 601, 8

How do we make 8 and 35 and 86 the same length as 648, 182 and 608?

645, 086, 182, 035, 601, 008

We add leading zeros/front pad our numbers with zeroes.

No difference in value between 001 and 1.

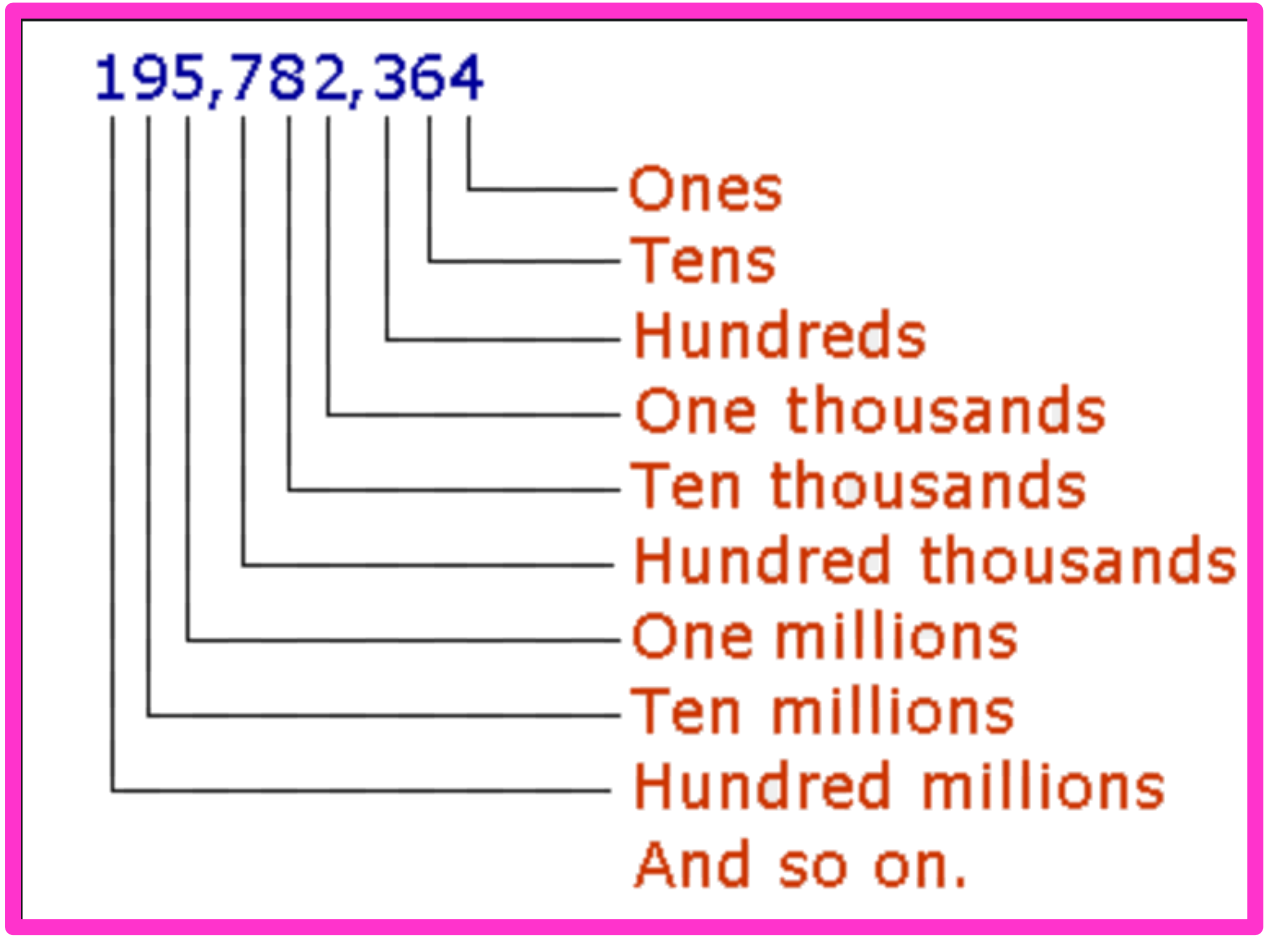
Big difference between 1 and 100 – add leading zeros to not change value.

Radix Sort

645, 086, 182, 035, 601, 008

Now we arrange our list by place value starting with ones.

601, 182, 645, 035, 086, 008

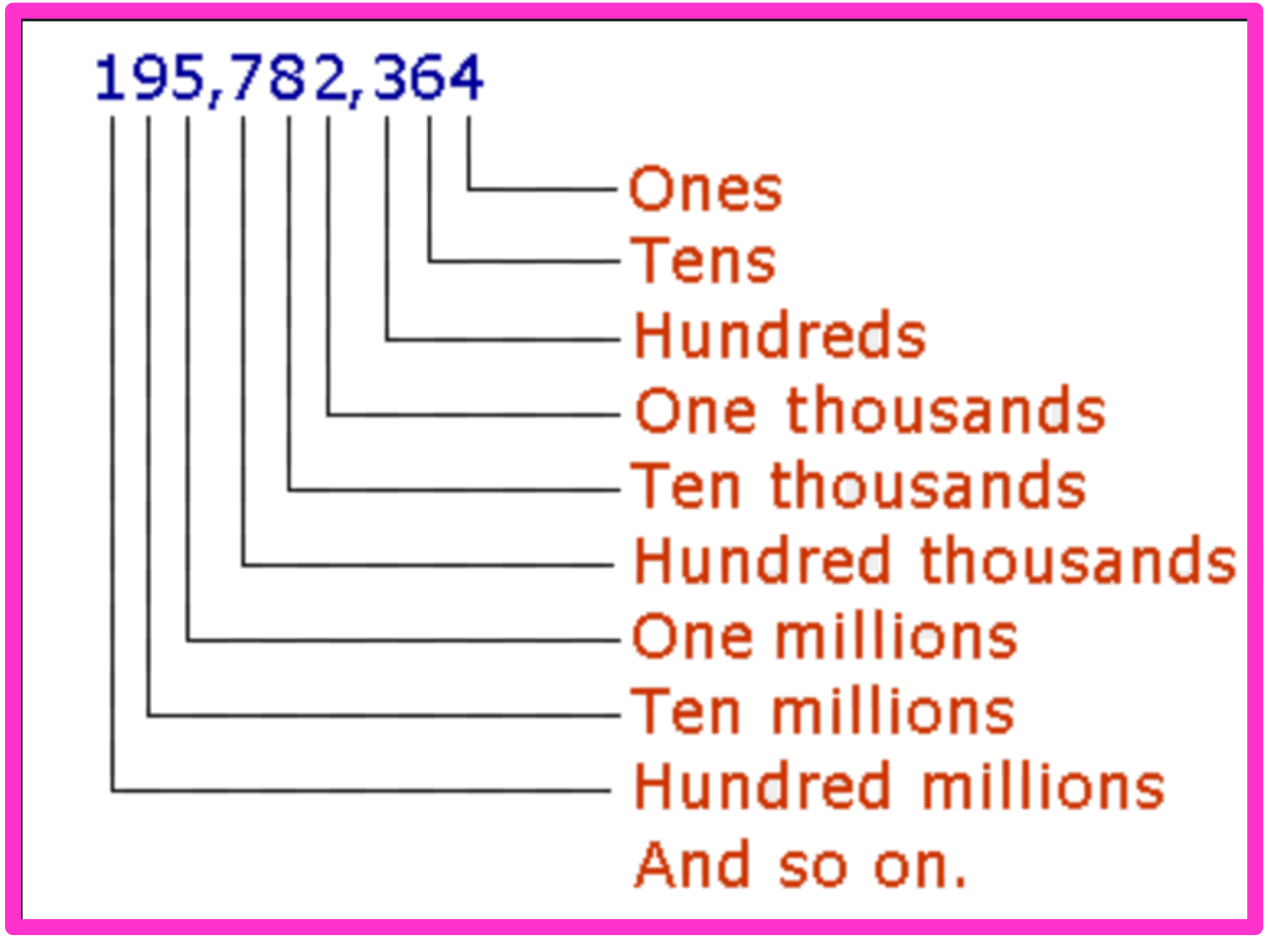


Radix Sort

601, 182, 645, 035, 086, 008

Now we arrange our list by place value tens.

601, 008, 035, 645, 182, 086

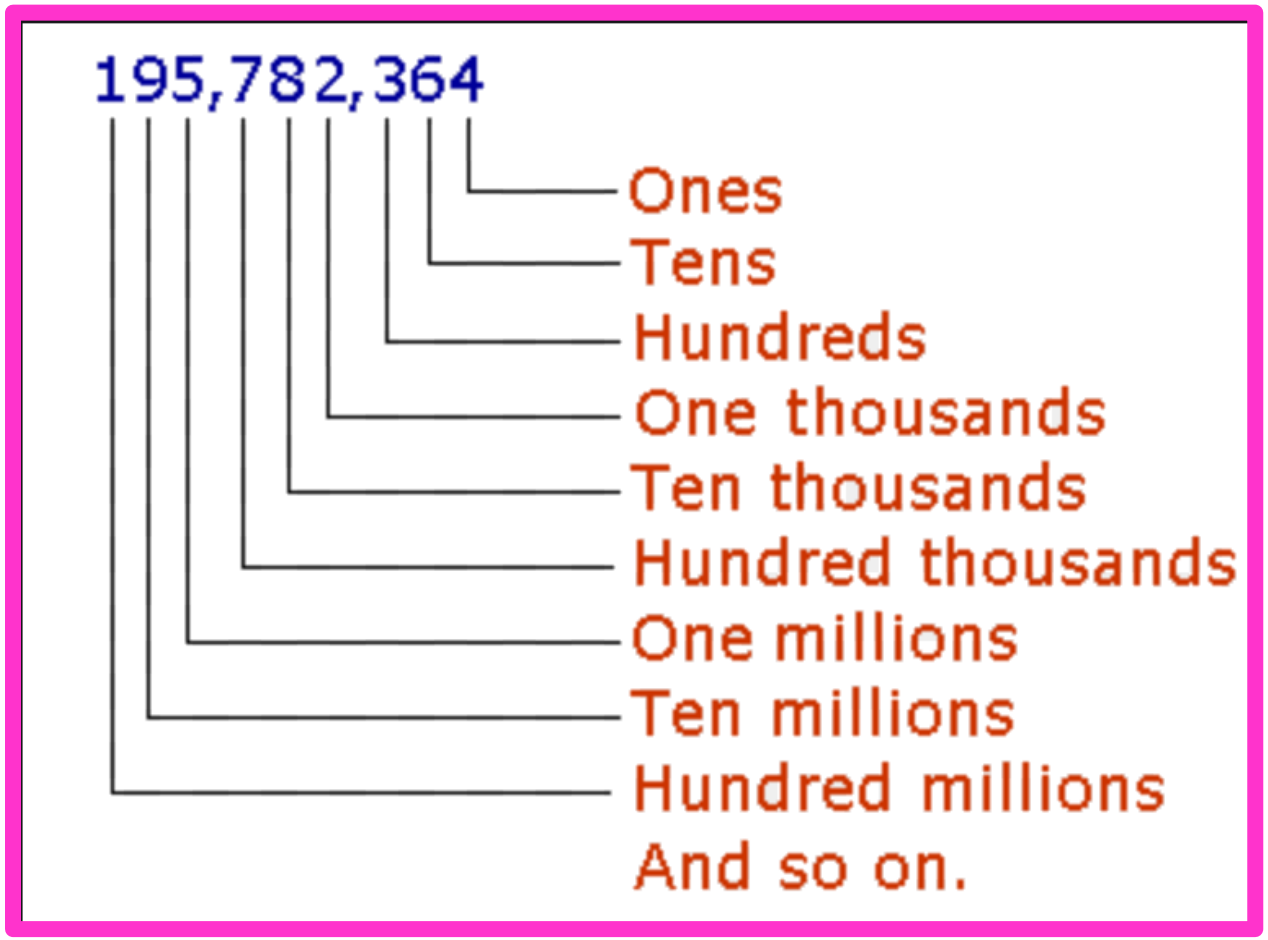


Radix Sort

601, 008, 035, 645, 182, 086

Now we arrange our list by place value hundreds.

008, 035, 086, 182, 601, 645



Radix Sort

645, 086, 182, 035, 601, 008

601, 182, 645, 035, 086, 008

601, 008, 035, 645, 182, 086

008, 035, 086, 182, 601, 645

Radix Sort

Radix Definition

The **base** of a system of numeration

So "Radix Sort" means a sort that uses the **base** of a system of numeration.

So can we use Radix Sort on numbers in bases other than 10?

Radix Sort

Let's start with a list of binary numbers

110, 11, 101, 1, 100, 111 , 10

Add leading zeros

110, 011, 101, 001, 100, 111 , 010

110, 011, 101, 001, 100, 111 , 010

Using ones, tens and hundreds place value is more applicable to the decimal system.

So let's talk about Most Significant Bit (MSB) and Least Significant Bit (LSB).

Radix Sort

MSB

the bit in a binary number which is of the greatest numerical value
the leftmost/first bit of a number

LSB

the bit in a binary number which is of the lowest numerical value.
the rightmost/last bit of a number

Radix Sort

110, 011, 101, 001, 100, 111 , 010

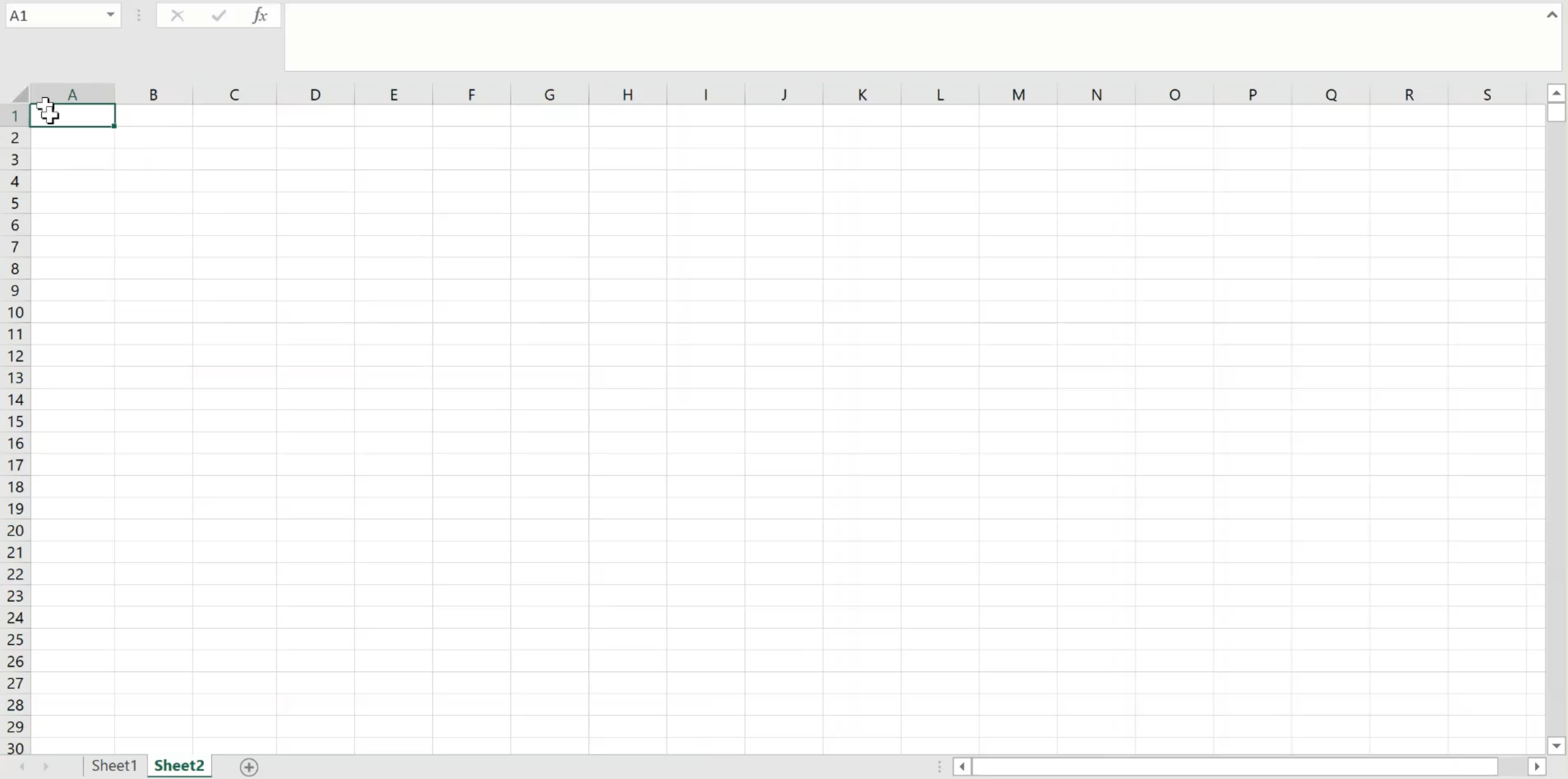
Let's arrange them starting with LSB and work towards the MSB.

110, 100, 010, 011, 101, 001, 111

100, 101, 001, 110, 010, 011, 111

001, 010, 011, 100, 101, 110, 111

Radix Sort


A1
fx
A B C D E F G H I J K L M N O P Q R S
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
Sheet1
Sheet2

Radix Sort Time Complexity

So what is the time complexity of Radix Sort?

Let's establish a few terms

d is the number of digits in the numbers in the list – use the number of digits in the biggest number in the list.

n is the number of numbers in the list

b is the base/radix we are using

Radix Sort Time Complexity

For every number in our list, *n*, we checked each digit.

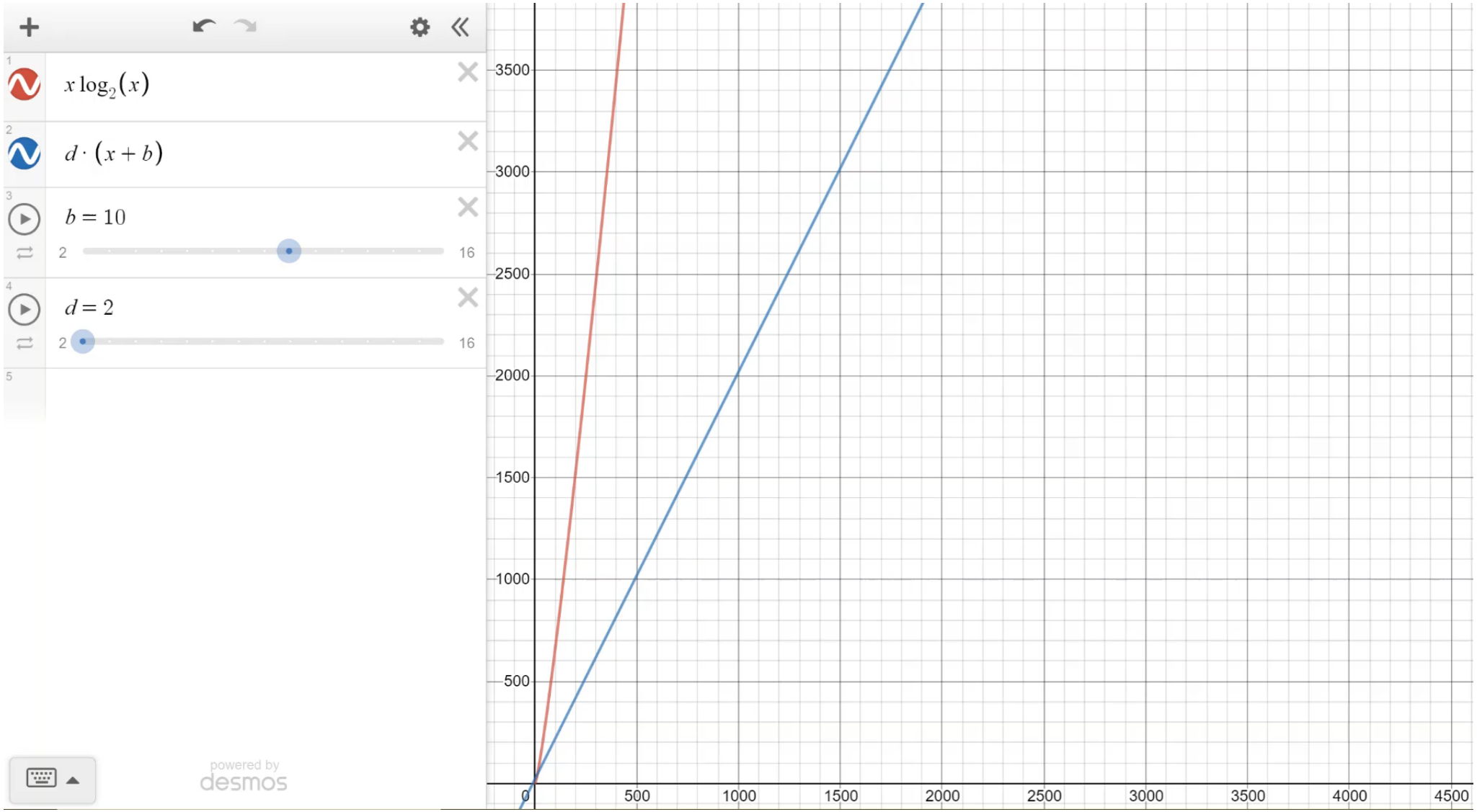
$d*n$

When we had a base 10 number, we had to check 10 digits -> 0-9.
When we had a base 2 number, we had to check 2 digits -> 0 and 1.

So the base influenced how much time we spent on each number.

$O(d * (n+b))$

Radix Sort Time Complexity



Radix Sort

Interesting Thoughts on Radix Sort

Radix Sort does not use any comparisons or swaps.

The time complexity of Radix Sort maintains its coefficient (number of digits) because of the significant impact of that value on the time complexity.

$O(nk)$

Even asymptotic notation does not wave away the impact of the number of digits.

Radix Sort

So if Radix Sorts has a better run time than most other sorts, why isn't it used more?

The biggest limitation of Radix Sort is that it is limited to data that only consists of digits or letters.

Radix Sort also does not actually move any elements around in the storage container (like an array) – it makes a new version of the storage container; therefore, has a higher space complexity than some other "slower" sorts. O(n+k).