Lecture No.11

Data Structures & Algorithms

Bucket Sort Algorithm

Bucket or Bin Sort

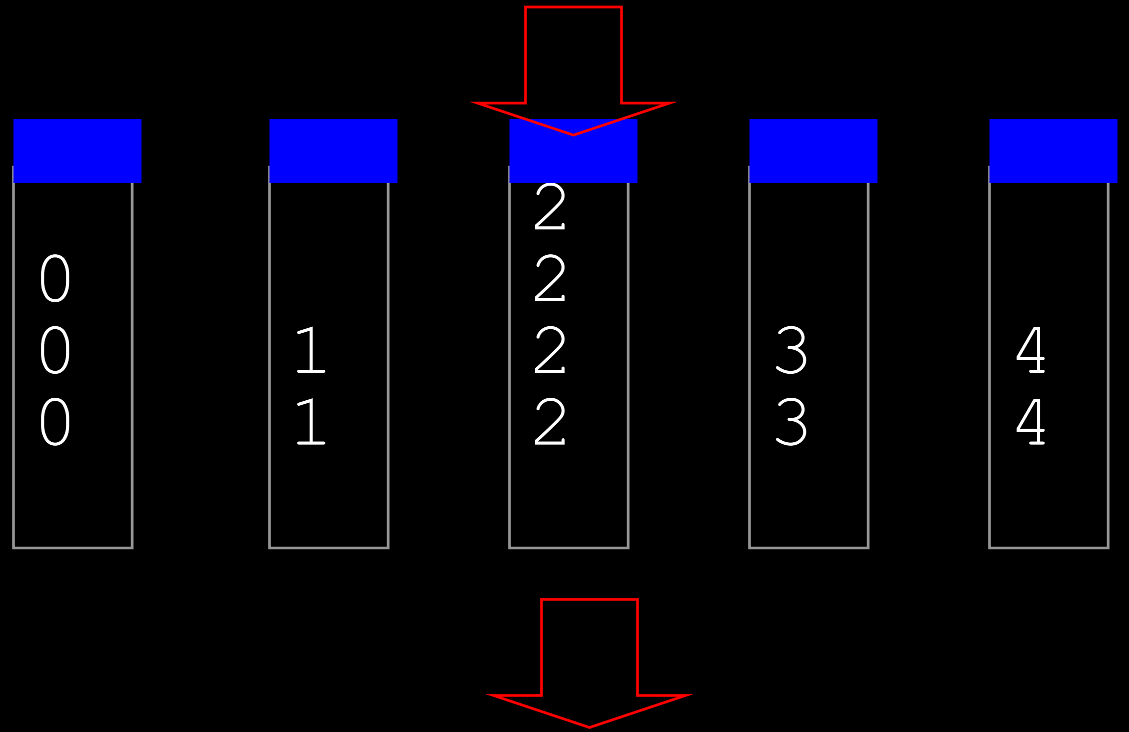
- **Bucket sort** is a **comparison sort algorithm** that works by distributing the elements of an array into a number of buckets and then each bucket is sorted individually using a separate sorting algorithm or by applying the bucket sort algorithm recursively.
- This algorithm is mainly useful when the input is uniformly distributed over a range

Bucket or Bin Sort

- Assume that the keys of the items that we wish to sort lie in a small fixed range and that there is only one item with each value of the key.
- Then we can sort with the following procedure:
- **1.** Set up an array of “bins” - one for each value of the key - in order,
- **2.** Examine each item and use the value of the key to place it in the appropriate bin.

Example

4	2	1	2	0	3	2	1	4	0	2	3	0



0	0	0	1	1	2	2	2	2	3	3	4	4

Bucket or Bin Sort

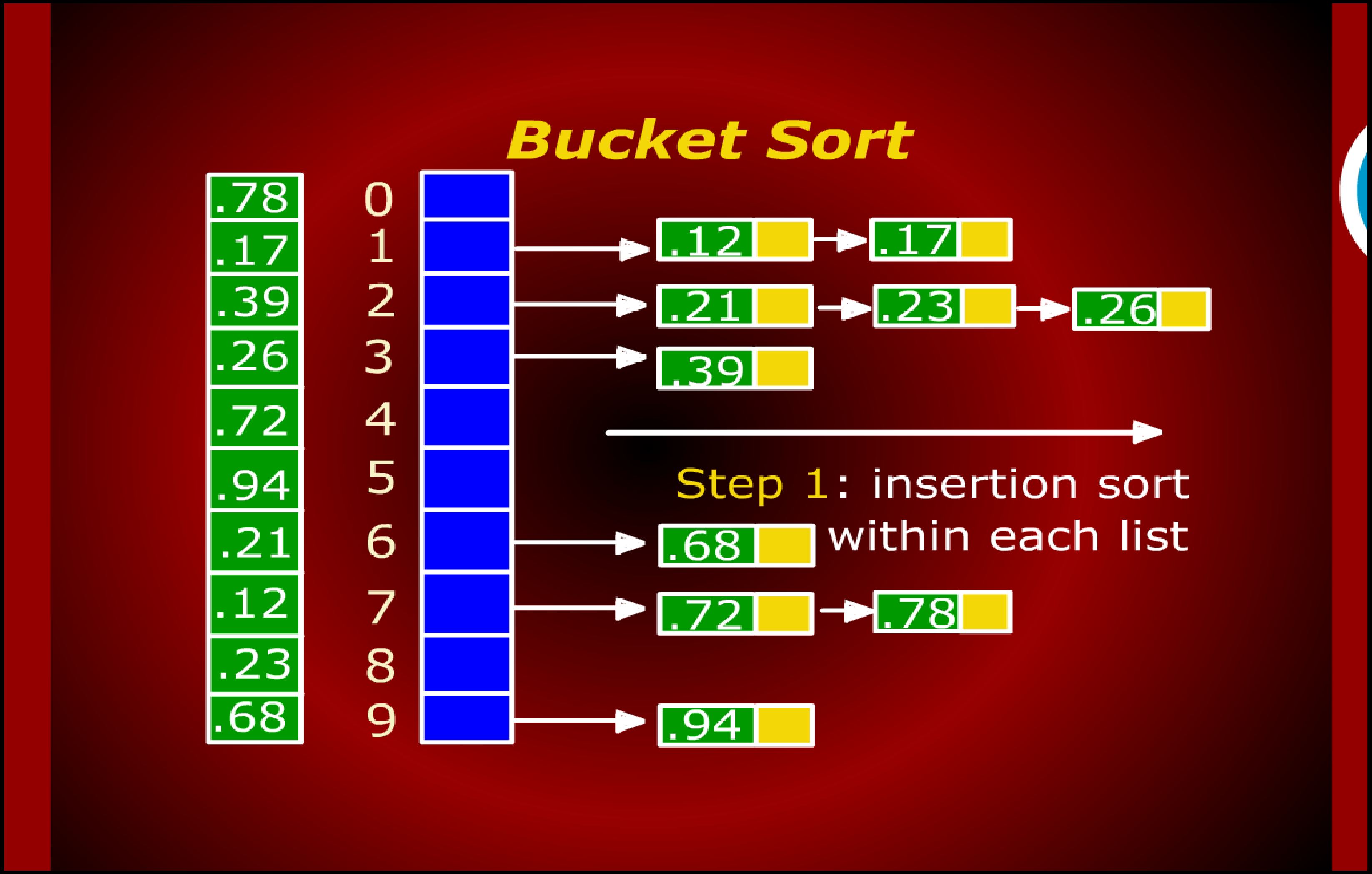
- **Algorithm**
- Create an empty array of size **n**(n empty buckets).
- Loop through the original array and put each array element in a “bucket”.
- Sort each of the non-empty buckets using insertion sort.
- Visit the buckets in order and put all elements back into the original array

LINEAR TIME SORTING

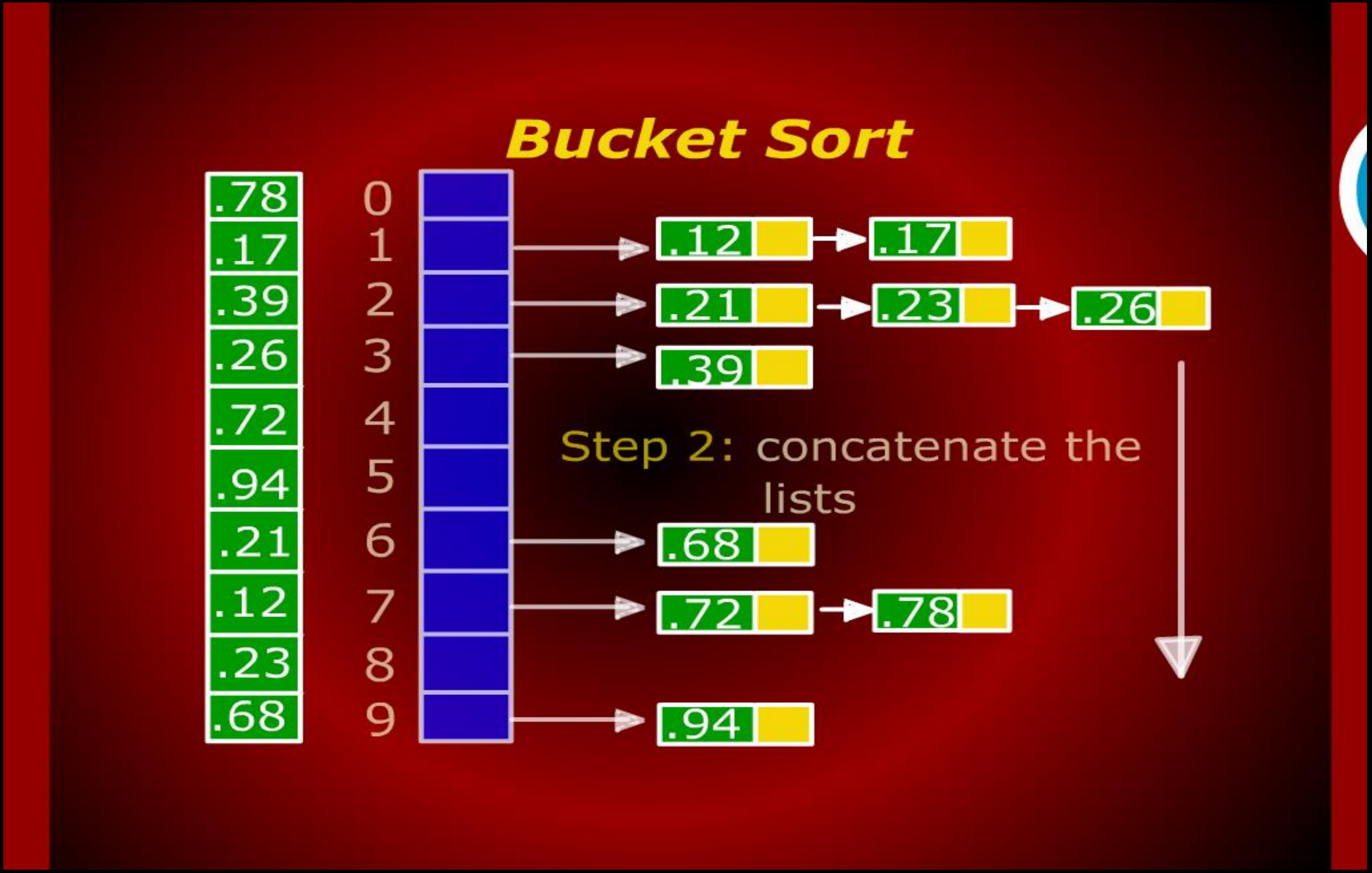
Bucket Sort

- Assumption: input elements are uniformly distributed over [0, 1]
- n inputs dropped into n equal-sized subintervals of [0, 1].

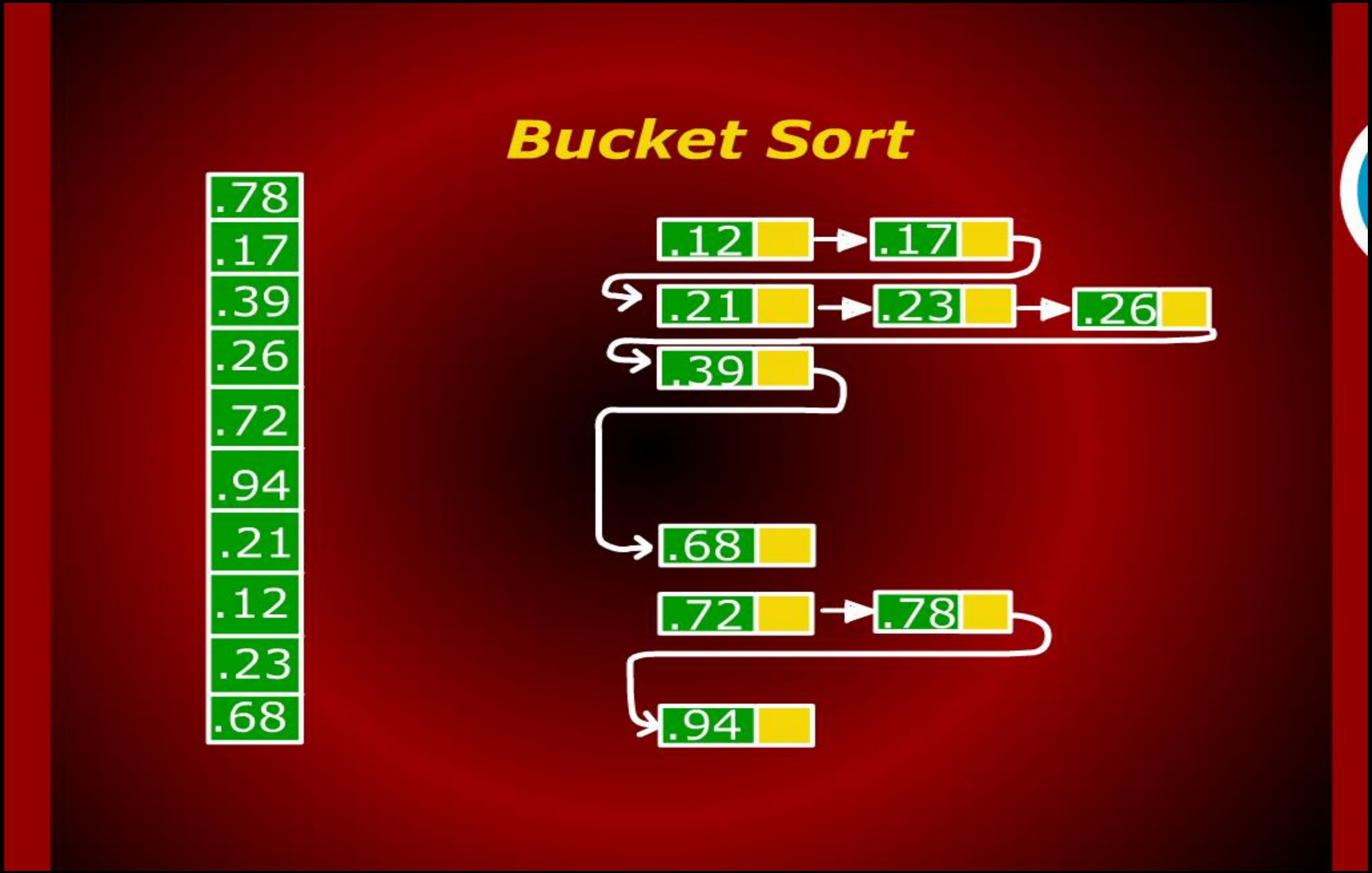
LINEAR TIME SORTING



LINEAR TIME SORTING



LINEAR TIME SORTING



LINEAR TIME SORTING

Radix Sort

- The main shortcoming of counting sort is that it is useful for small integers, i.e., 1..k where k is small.
- If k were a million or more, the size of the rank array would also be a million.
- Radix sort provides a nice work around this limitation by sorting numbers one digit at a time.

LINEAR TIME SORTING

Radix Sort

576		49[4]		9[5]4		[1]76		176
494		19[4]		5[7]6		[1]94		194
194		95[4]		1[7]6		[2]78		278
296	$\Rightarrow$	57[6]	$\Rightarrow$	2[7]8	$\Rightarrow$	[2]96	$\Rightarrow$	296
278		29[6]		4[9]4		[4]94		494
176		17[6]		1[9]4		[5]76		576
954		27[8]		2[9]6		[9]54		954

LINEAR TIME SORTING

Radix Sort

Here is the algorithm that sorts A[1..n] where each number is d digits long.

```
RADIX-SORT( array A, int n, int d)
      for i ← 1 to d
      do stably sort A w.r.t i^th lowest
            order digit
```

Radix Sort

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

Example: first pass

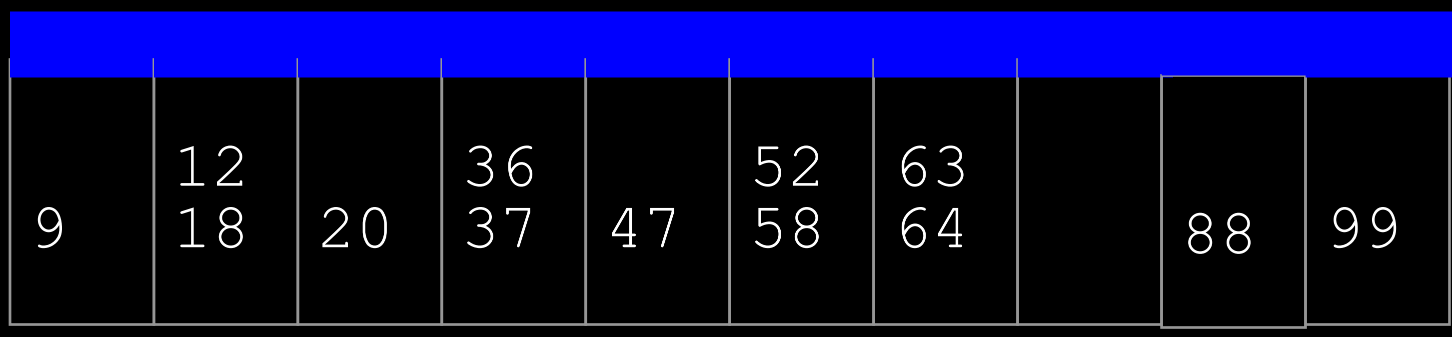
12	58	37	64	52	36	99	63	18	9	20	88	47

20		12 52	63	64		36	37 47	58 18 88	9 99

20	12	52	63	64	36	37	47	58	18	88	9	99

Example: second pass

20	12	52	63	64	36	37	47	58	18	88	9	99



9	12	18	20	36	37	47	52	58	63	64	88	99

Example: 1st and 2nd passes

12	58	37	64	52	36	99	63	18	9	20	88	47

sort by rightmost digit

20	12	52	63	64	36	37	47	58	18	88	9	99

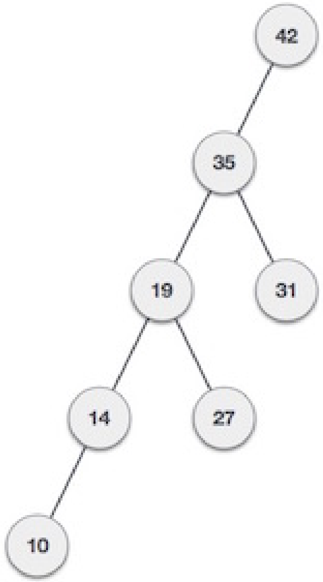
sort by leftmost digit

9	12	18	20	36	37	47	52	58	63	64	88	99

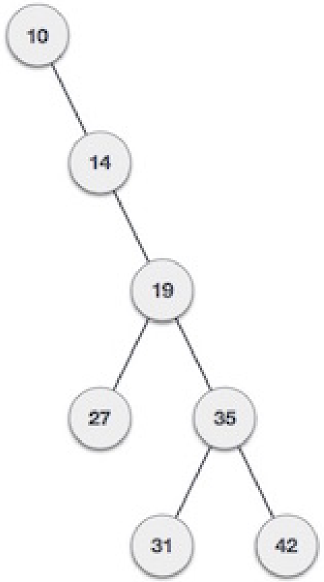
Data Structure and Algorithms - AVL Trees

AVL Trees

What if the input to binary search tree comes in a sorted (ascending or descending) manner? It will then look like this –



If input 'appears' non-increasing manner



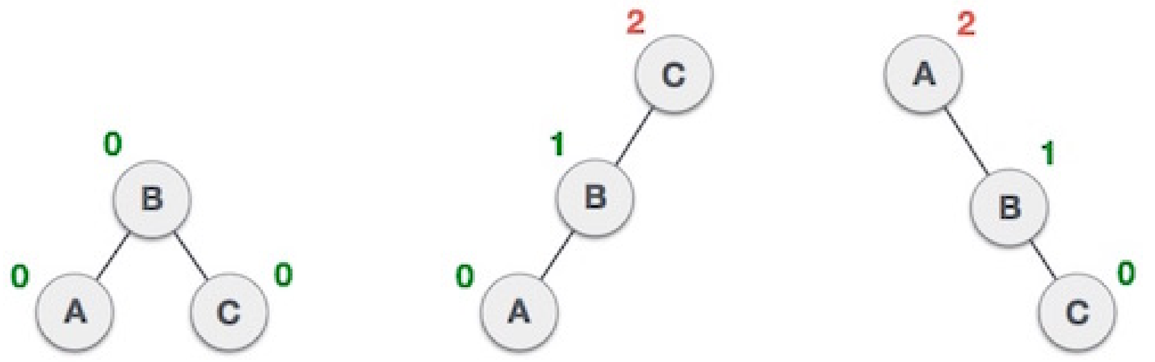
If input 'appears' in non-decreasing manner

It is observed that BST's worst-case performance is closest to linear search algorithms, that is O(n). In real-time data, we cannot predict data pattern and their frequencies. So, a need arises to balance out the existing BST.

Named after their inventor **Adelson**, **Velski** & **Landis**, **AVL trees** are height balancing binary search tree. AVL tree checks the height of the left and the right sub-trees and assures that the difference is not more than 1. This difference is called the **Balance Factor**.

AVL Trees

Here we see that the first tree is balanced and the next two trees are not balanced –



Balanced Not balanced Not balanced

In the second tree, the left subtree of **C** has height 2 and the right subtree has height 0, so the difference is 2. In the third tree, the right subtree of **A** has height 2 and the left is missing, so it is 0, and the difference is 2 again. AVL tree permits difference (balance factor) to be only 1.

```
BalanceFactor = height(left-sutree) − height(right-sutree)
```

If the difference in the height of left and right sub-trees is more than 1, the tree is balanced using some rotation techniques.

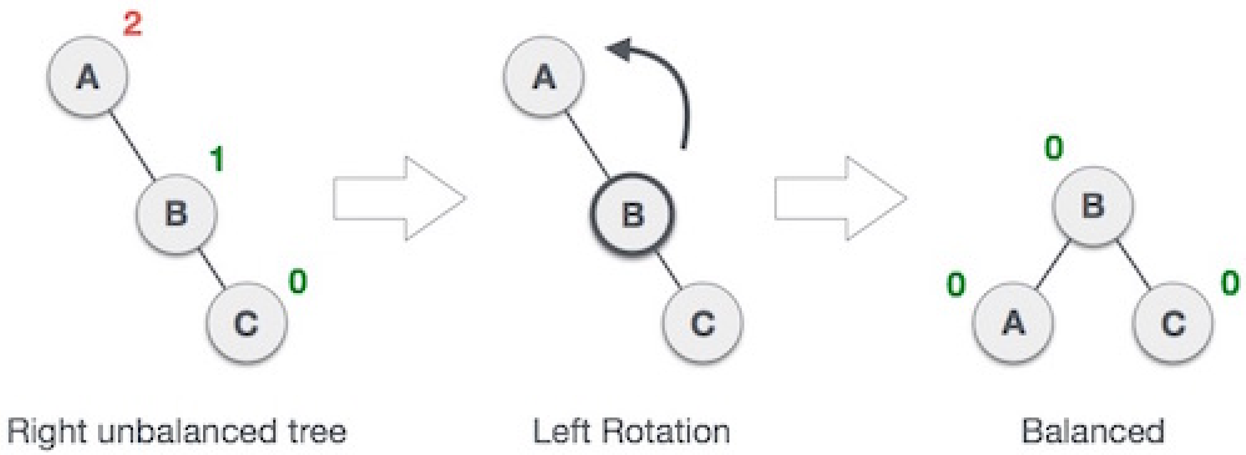
AVL Rotations

- To balance itself, an AVL tree may perform the following four kinds of rotations −
- Left rotation
- Right rotation
- Left-Right rotation
- Right-Left rotation
- The first two rotations are single rotations and the next two rotations are double rotations.
- To have an unbalanced tree, we at least need a tree of height 2.
- With this simple tree, let's understand them one by one.

AVL Rotations

Left Rotation

If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation −

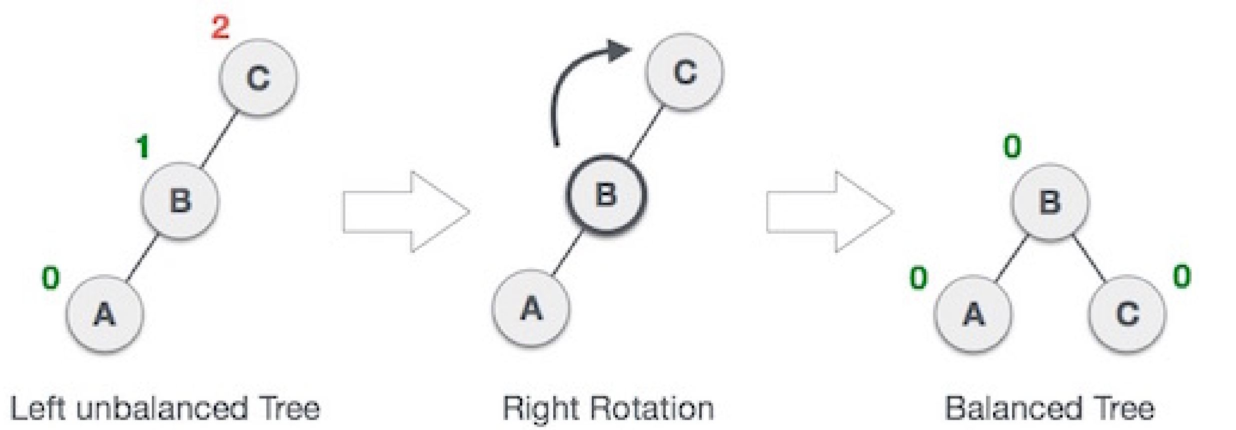


In our example, node **A** has become unbalanced as a node is inserted in the right subtree of A's right subtree. We perform the left rotation by making **A** the left-subtree of B.

AVL Rotations

Right Rotation

AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.

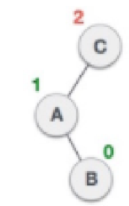
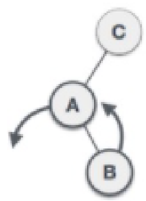
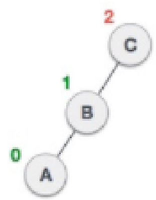
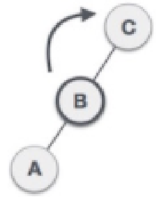
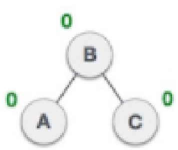


As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.

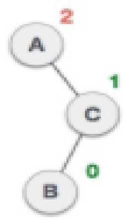
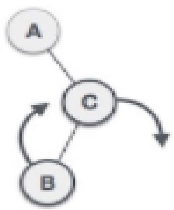
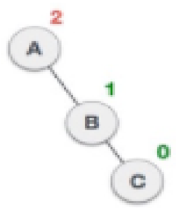
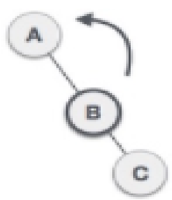
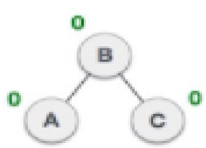
Left-Right Rotation

- Double rotations are slightly complex version of already explained versions of rotations.
- To understand them better, we should take note of each action performed while rotation.
- Let's first check how to perform Left-Right rotation.
- A left-right rotation is a combination of left rotation followed by right rotation.

AVL Rotations

State	Action
	A node has been inserted into the right subtree of the left subtree. This makes C an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.
	We first perform the left rotation on the left subtree of C. This makes A, the left subtree of B.
	Node C is still unbalanced, however now, it is because of the left-subtree of the left-subtree.
	We shall now right-rotate the tree, making B the new root node of this subtree. C now becomes the right subtree of its own left subtree.
	The tree is now balanced.

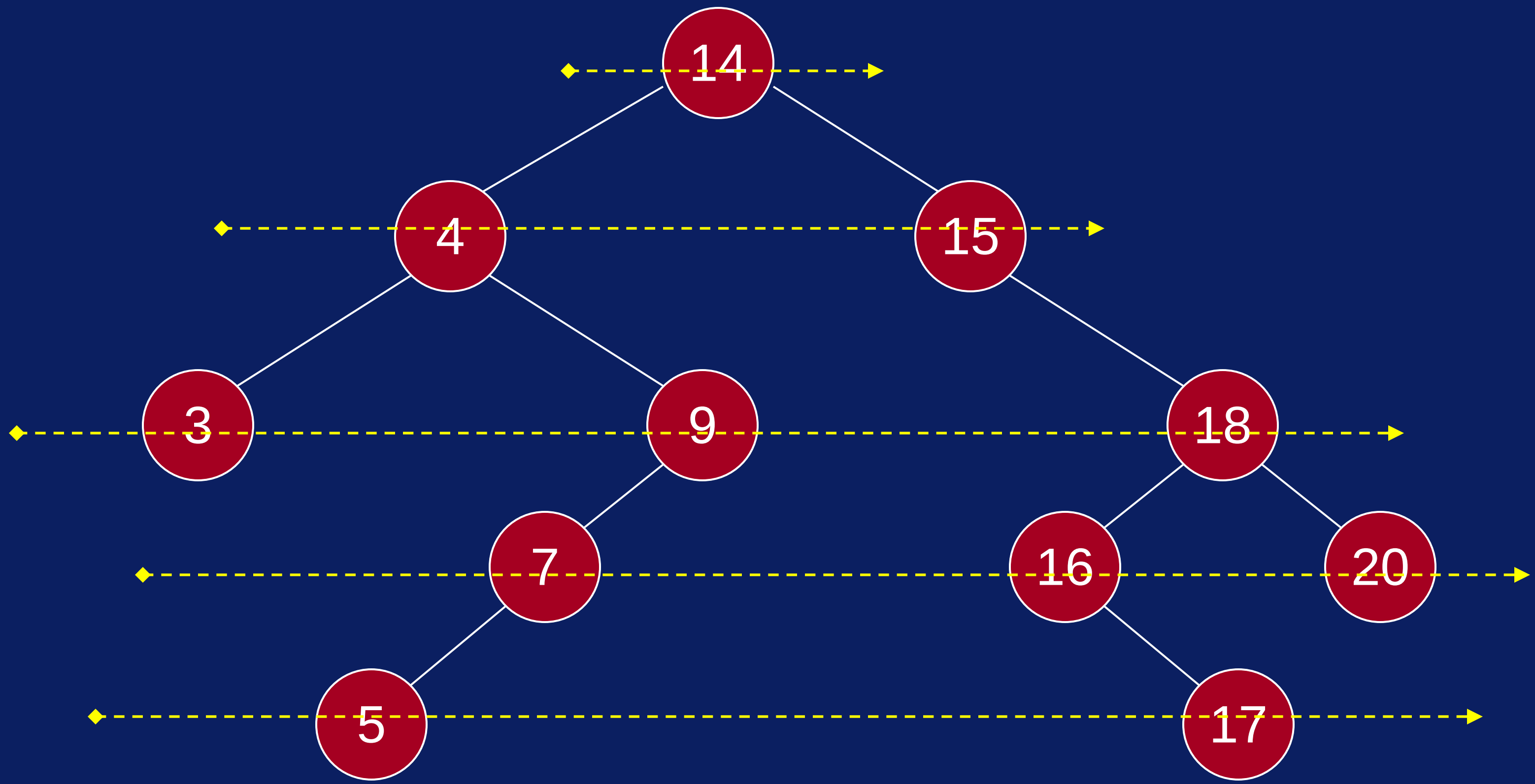
Right-Left Rotation

State	Action
	A node has been inserted into the left subtree of the right subtree. This makes A, an unbalanced node with balance factor 2.
	First, we perform the right rotation along C node, making C the right subtree of its own left subtree B. Now, B becomes the right subtree of A.
	Node A is still unbalanced because of the right subtree of its right subtree and requires a left rotation.
	A left rotation is performed by making B the new root node of the subtree. A becomes the left subtree of its right subtree B.
	The tree is now balanced.

Level-order Traversal

- There is yet another way of traversing a binary tree that is not related to recursive traversal procedures discussed previously.
- In level-order traversal, we visit the nodes at each level before proceeding to the next level.
- At each level, we visit the nodes in a left-to-right order.

Level-order Traversal

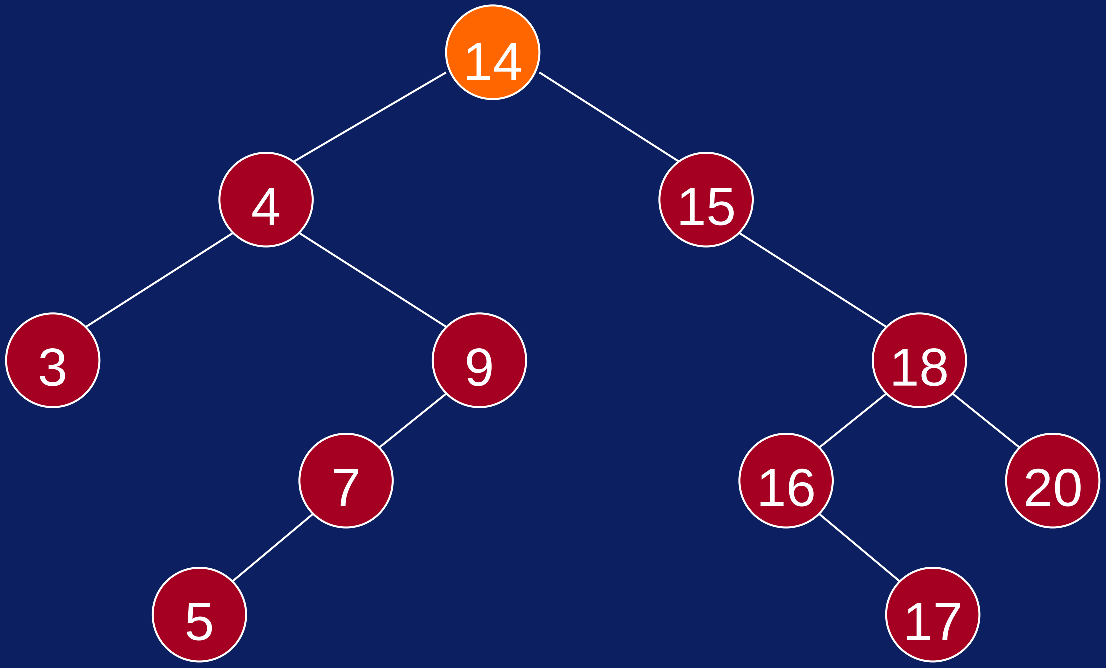


Level-order: 14 4 15 3 9 18 7 16 20 5 17

Level-order Traversal

- How do we do level-order traversal?
- Surprisingly, if we use a queue instead of a stack, we can visit the nodes in level-order.

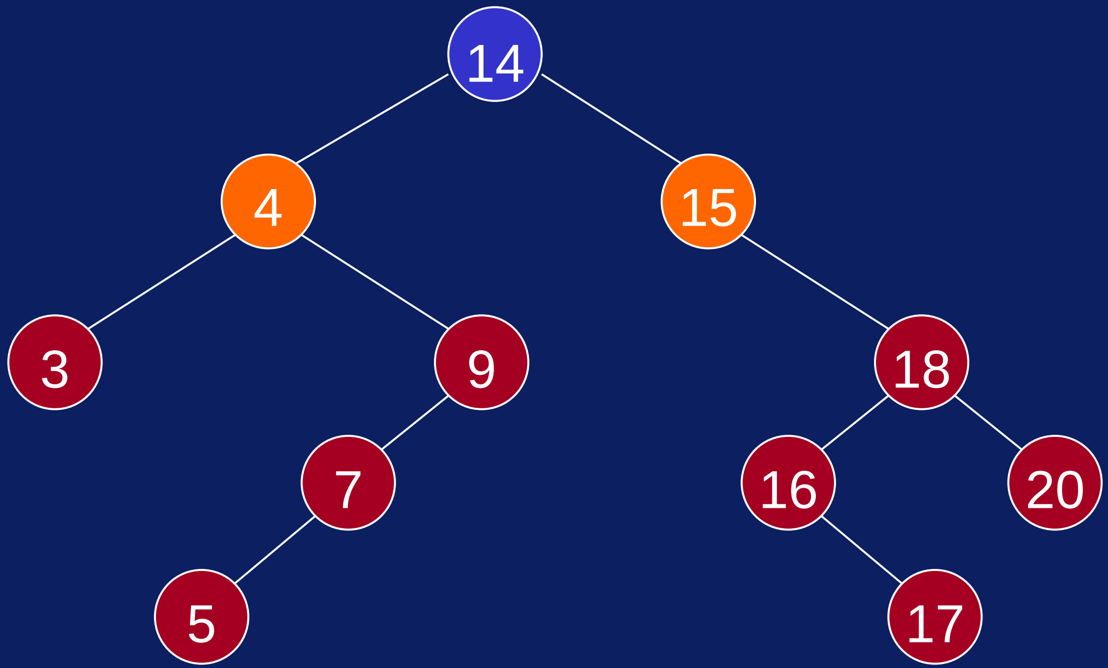
Level-order Traversal



Queue: 14

Output:

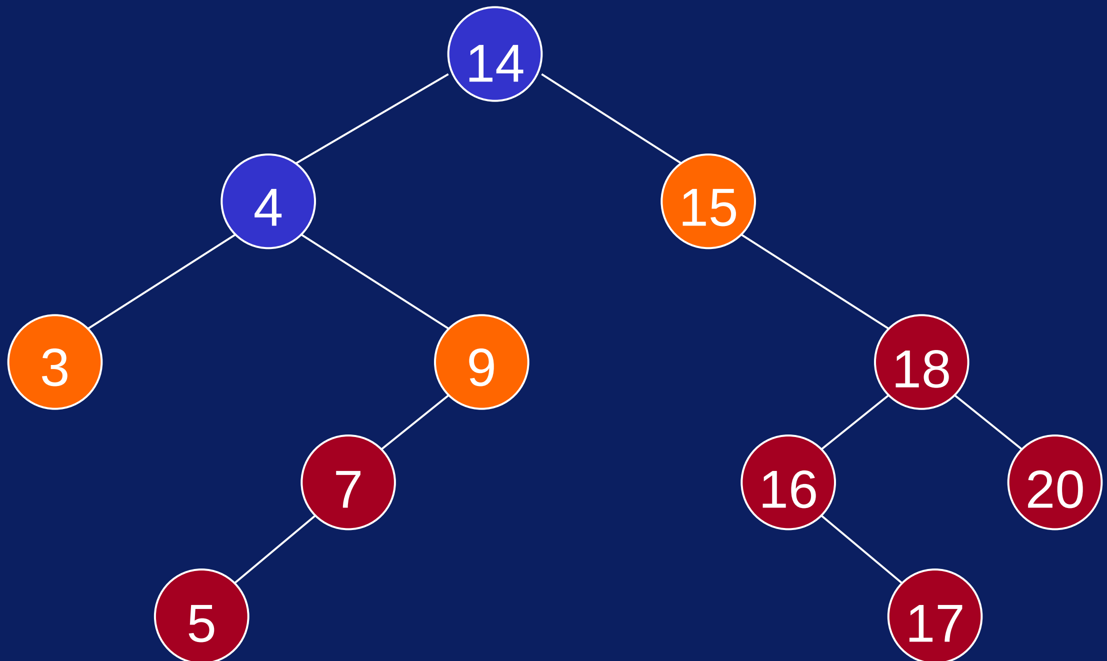
Level-order Traversal



Queue: 4 15

Output: 14

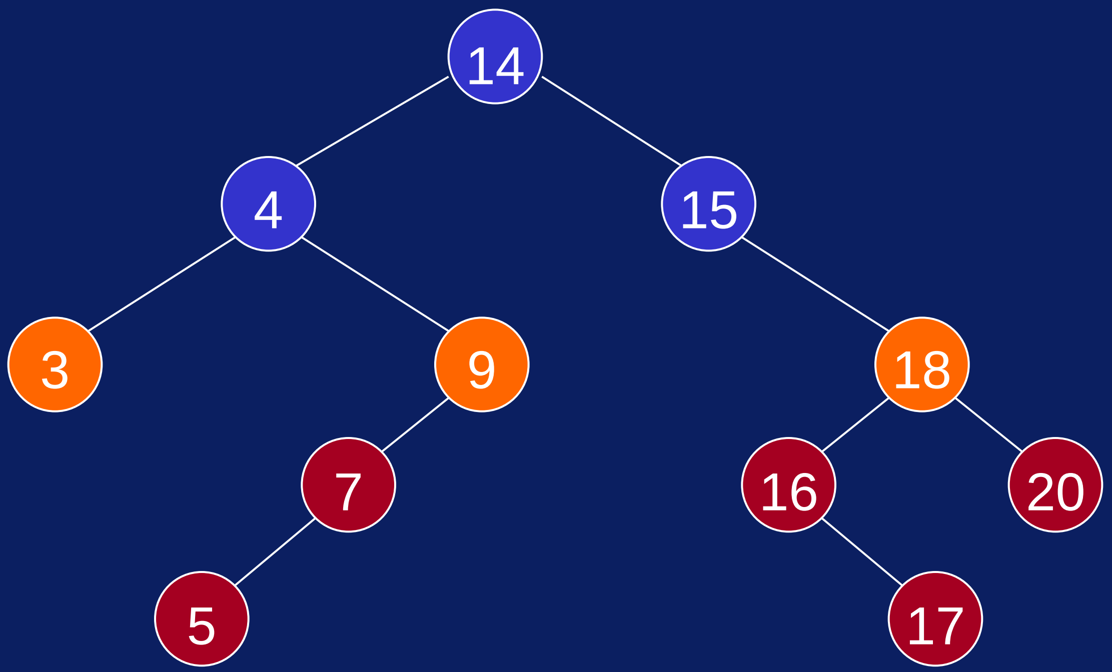
Level-order Traversal



Queue: 15 3 9

Output: 14 4

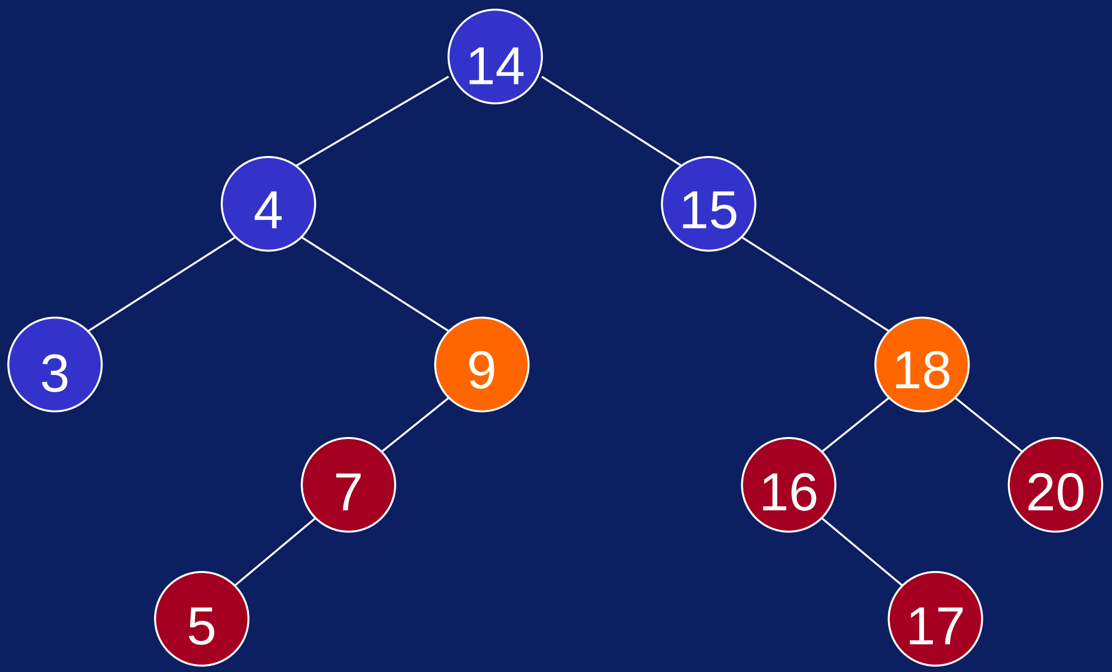
Level-order Traversal



Queue: 3 9 18

Output: 14 4 15

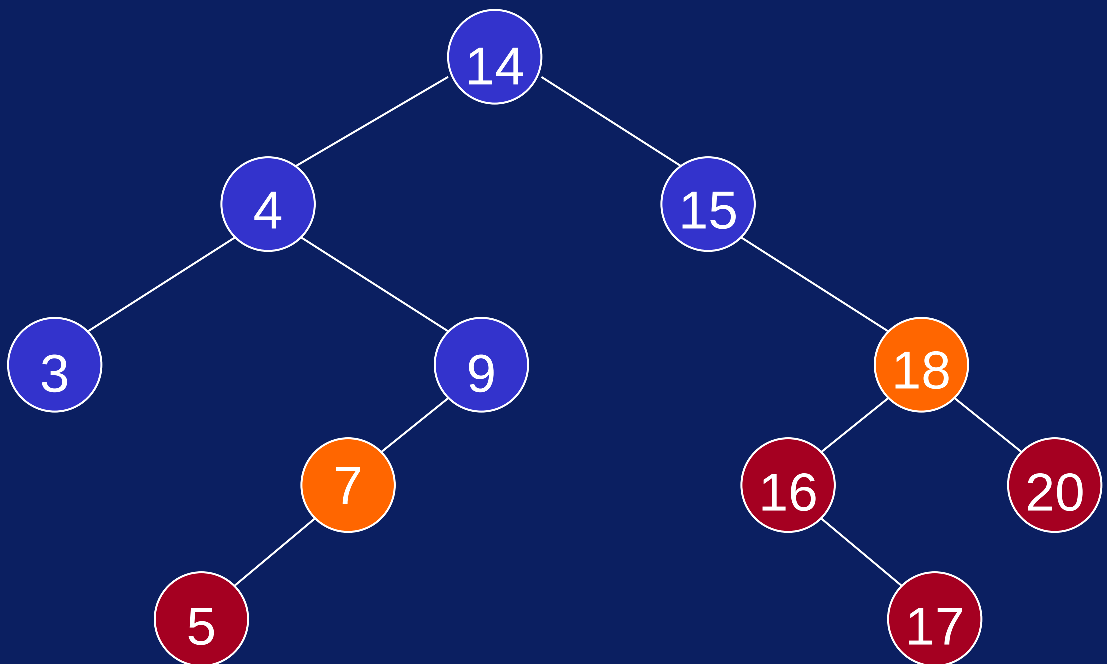
Level-order Traversal



Queue: 9 18

Output: 14 4 15 3

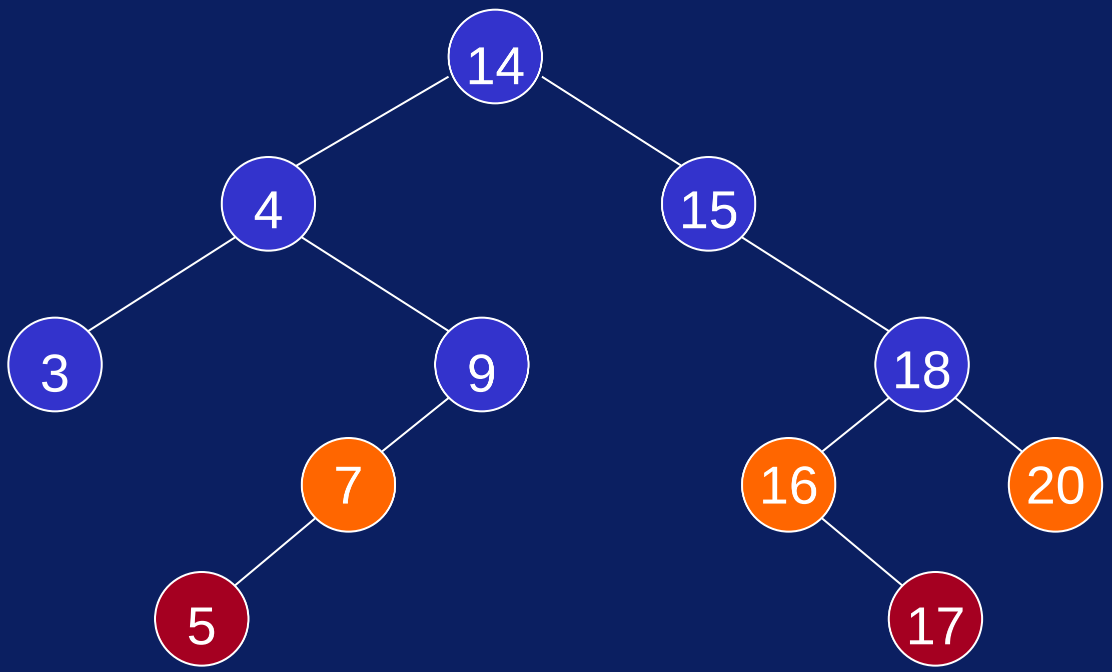
Level-order Traversal



Queue: 18 7

Output: 14 4 15 3   9

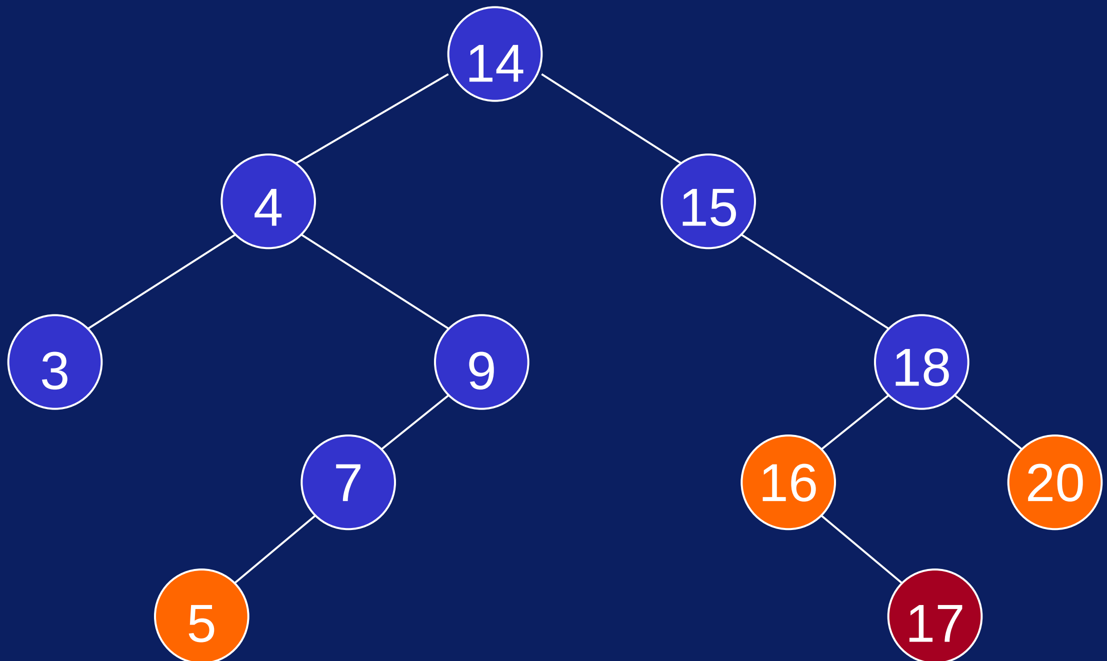
Level-order Traversal



Queue: 7 16 20

Output: 14 4 15 3 9 18

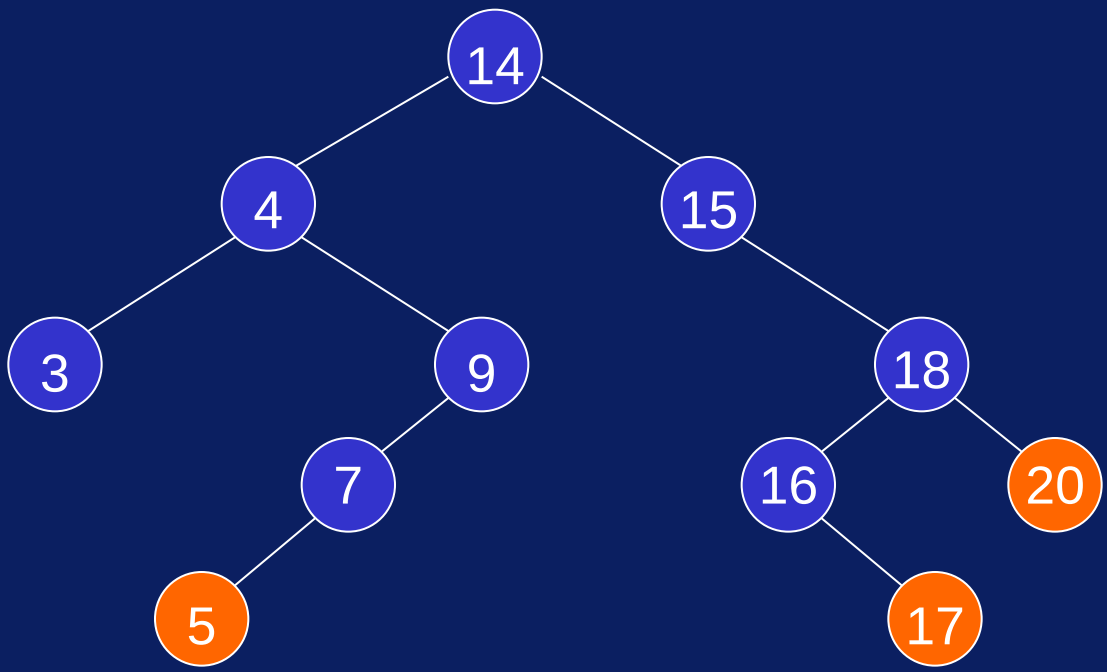
Level-order Traversal



Queue: 16 20 5

Output: 14 4 15 3 9 18 7

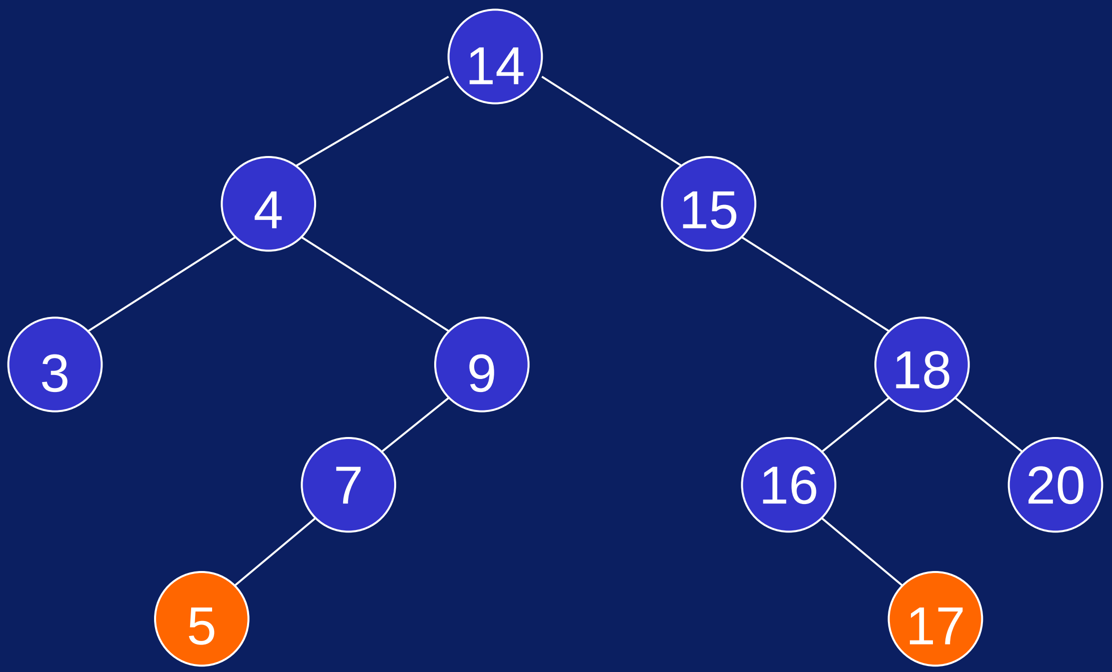
Level-order Traversal



Queue: 20 5 17

Output: 14 4 15 3   9 18   7 16

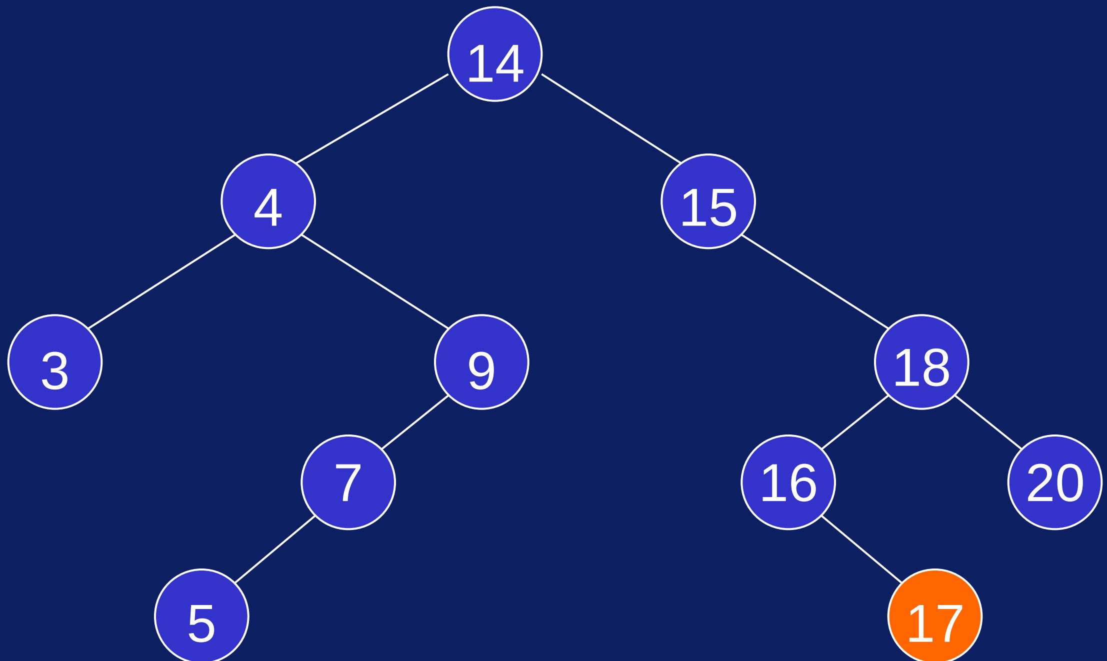
Level-order Traversal



Queue: 5 17

Output: 14 4 15 3   9 18   7 16 20

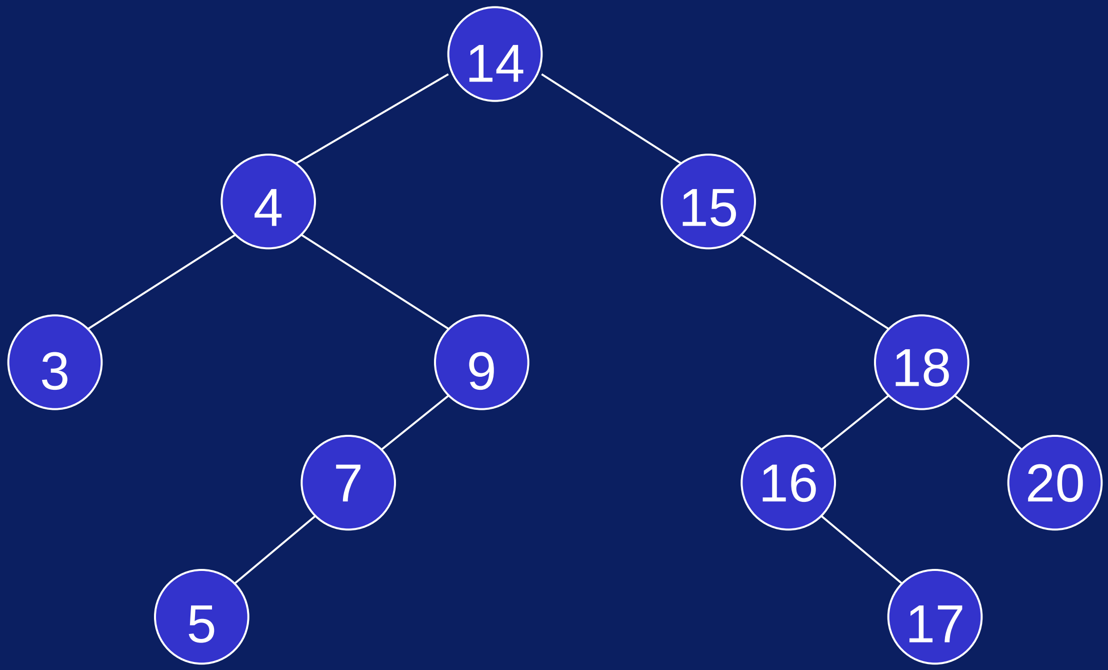
Level-order Traversal



Queue: 17

Output: 14 4 15 3 9 18 7 16 20 5

Level-order Traversal



Queue:

Output: 14 4 15 3 9 18 7 16 20 5 17

Hashing

Hashing

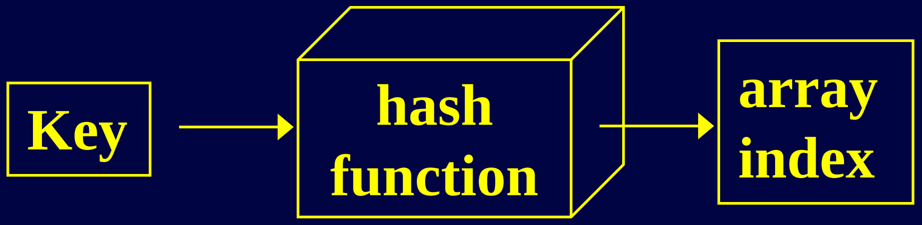
- Data Structure which is designed to use a special function called the Hash function which is used to map a given value with a particular key for faster access of elements.
- The efficiency of mapping depends of the efficiency of the hash function used.

Hashing

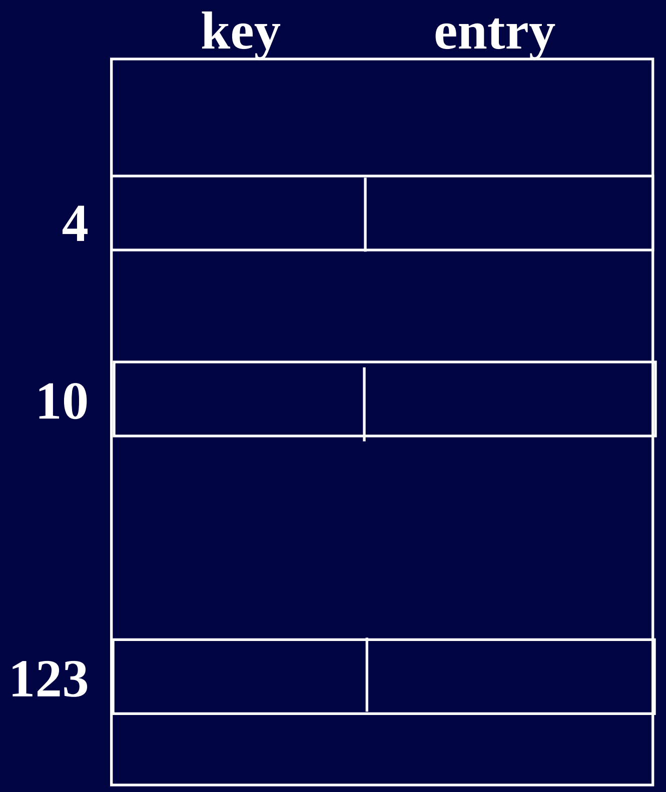
- Hash Table is a data structure which stores data in an associative manner.
- In a hash table, data is stored in an array format, where each data value has its own unique index value.
- Access of data becomes very fast if we know the index of the desired data.
- Thus, insertion and search operations are very fast irrespective of the size of the data. Hash Table uses an array as a storage medium and uses hash technique to generate an index where an element is to be inserted or is to be located from.

Implementation : Hashing

- An *array* in which TableNodes are ***<u>not</u>*** stored consecutively
- Their place of storage is calculated using the key and a *hash function*

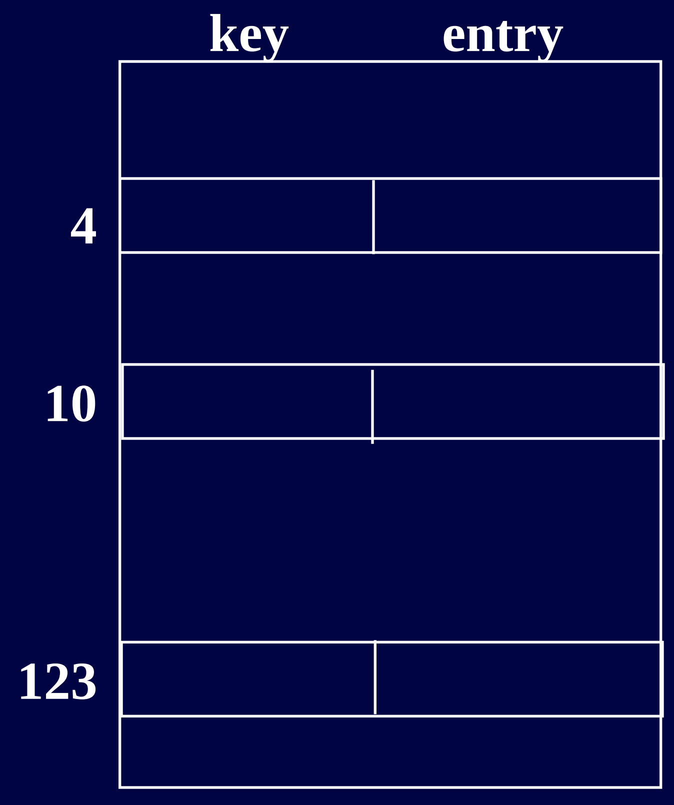


- Keys and entries are scattered throughout the array.



Hashing

- **<u>insert</u>**: calculate place of storage, insert TableNode; (1)
- **<u>find</u>**: calculate place of storage, retrieve entry; (1)
- **<u>remove</u>**: calculate place of storage, set it to null; (1)



All are constant time (1) !

Hashing

- We use an array of some fixed size *T* to hold the data. *T* is typically prime.
- Each key is mapped into some number in the range *0* to *T-1* using a *hash function*, which ideally should be efficient to compute.

Example: fruits

- Suppose our hash function gave us the following values:

    hashCode("apple") = 5
    hashCode("watermelon") = 3
    hashCode("grapes") = 8
    hashCode("cantaloupe") = 7
    hashCode("kiwi") = 0
    hashCode("strawberry") = 9
    hashCode("mango") = 6
    hashCode("banana") = 2

0	kiwi
1	
2	banana
3	watermelon
4	
5	apple
6	mango
7	cantaloupe
8	grapes
9	strawberry

Example

- Store data in a table array:

```
table[5] = "apple"
table[3] = "watermelon"
table[8] = "grapes"
table[7] = "cantaloupe"
table[0] = "kiwi"
table[9] = "strawberry"
table[6] = "mango"
table[2] = "banana"
```

0	kiwi
1	
2	banana
3	watermelon
4	
5	apple
6	mango
7	cantaloupe
8	grapes
9	strawberry

Example

- Associative array:

    table["apple"]
    table["watermelon"]
    table["grapes"]
    table["cantaloupe"]
    table["kiwi"]
    table["strawberry"]
    table["mango"]
    table["banana"]

0	kiwi
1	
2	banana
3	watermelon
4	
5	apple
6	mango
7	cantaloupe
8	grapes
9	strawberry

Example Hash Functions

- If the keys are strings the hash function is some function of the characters in the strings.
- One possibility is to simply add the ASCII values of the characters:

$$h(str) = \left(\sum_{i=0}^{length-1} str[i] \right) \% TableSize$$

$$Example: h(ABC) = (65 + 66 + 67) \% TableSize$$

Finding the hash function

```
int hashCode( char* s )
{
    int i, sum;
    sum = 0;
    for(i=0; i < strlen(s); i++ )
      sum = sum + s[i]; // ascii value
    return sum % TABLESIZE;
}
```

Example Hash Functions

- Another possibility is to convert the string into some number in some arbitrary base *b* (*b* also might be a prime number):

$$h(str) = \left(\sum_{i=0}^{length-1} str[i] \times b^{i} \right) \% T$$

$$Example: h(ABC) = (65b^{0} + 66b^{1} + 67b^{2}) \% T$$

Example Hash Functions

- If the keys are integers then *key%T* is generally a good hash function, unless the data has some undesirable features.
- For example, if $T = 10$ and all keys end in zeros, then $key\%T = 0$ for all keys.
- In general, to avoid situations like this, *T* should be a prime number.

Collision

Suppose our hash function gave us the following values:

- hash("apple") = 5
  hash("watermelon") = 3
  hash("grapes") = 8
  hash("cantaloupe") = 7
  hash("kiwi") = 0
  hash("strawberry") = 9
  hash("mango") = 6
  hash("banana") = 2

  hash("honeydew") = 6

- Now what?

0	kiwi
1	
2	banana
3	watermelon
4	
5	apple
6	mango
7	cantaloupe
8	grapes
9	strawberry

Collision

- When two values hash to the same array location, this is called a *collision*
- Collisions are normally treated as “first come, first served”—the first value that hashes to the location gets it
- We have to find something to do with the second and subsequent values that hash to this same location.

Solution for Handling collisions

- **Solution #1:** Search from there for an empty location
  - Can stop searching when we find the value *or* an empty location.
  - Search must be wrap-around at the end.

Solution for Handling collisions

- **Solution #2:** Use a second hash function
  - ...and a third, and a fourth, and a fifth, ...

Solution for Handling collisions

- **Solution #3:** Use the array location as the header of a linked list of values that hash to this location

Solution 1: Open Addressing

- This approach of handling collisions is called *open addressing*; it is also known as *closed hashing*.
- More formally, cells at $h_0(x)$, $h_1(x)$, $h_2(x)$, … are tried in succession where

$$h_i(x) = (hash(x) + f(i)) \ mod \ TableSize,$$

with $f(0) = 0$.

- The function, f, is the collision resolution strategy.

Linear Probing

- We use $f(i) = i$, i.e., f is a linear function of i. Thus

$$location(x) = (hash(x) + i) \ mod \ TableSize$$

- The collision resolution strategy is called *linear probing* because it scans the array sequentially (with wrap around) in search of an empty cell.

Linear Probing: insert

- Suppose we want to *add* seagull to this hash table
- Also suppose:
  - hashCode(“seagull”) = 143
  - table[143] is not empty
  - table[143] != seagull
  - table[144] is not empty
  - table[144] != seagull
  - table[145] is empty
- Therefore, put seagull at location 145

. . .	
141	
142	robin
143	sparrow
144	hawk
145	seagull
146	
147	bluejay
148	owl
. . .	

Linear Probing: insert

- Suppose you want to *add* hawk to this hash table
- Also suppose
  - hashCode("hawk") = 143
  - table[143] is not empty
  - table[143] != hawk
  - table[144] is not empty
  - table[144] == hawk
- hawk is already in the table, so do nothing.

. . .	
141	
142	robin
143	sparrow
144	hawk
145	seagull
146	
147	bluejay
148	owl
. . .	

Linear Probing: insert

- Suppose:
  - You want to add cardinal to this hash table
  - hashCode(“cardinal”) = 147
  - The last location is 148
  - 147 and 148 are occupied
- Solution:
  - Treat the table as circular; after 148 comes 0
  - Hence, cardinal goes in location 0 (or 1, or 2, or ...)

. . .	
141	
142	robin
143	sparrow
144	hawk
145	seagull
146	
147	bluejay
148	owl

Linear Probing: find

- Suppose we want to find hawk in this hash table
- We proceed as follows:
  - hashCode("hawk") = 143
  - table[143] is not empty
  - table[143] != hawk
  - table[144] is not empty
  - table[144] == hawk (found!)
- We use the same procedure for looking things up in the table as we do for inserting them

Linear Probing and Deletion

- If an item is placed in `array[hash(key)+4]`, then the item just before it is deleted
- How will probe determine that the "hole" does not indicate the item is not in the array?
- Have three states for each location
  - Occupied
  - Empty (never used)
  - Deleted (previously used)

Clustering

- One problem with linear probing technique is the tendency to form “clusters”.
- A cluster is a group of items not containing any open slots
- The bigger a cluster gets, the more likely it is that new values will hash into the cluster, and make it ever bigger.
- Clusters cause efficiency to degrade.

Quadratic Probing

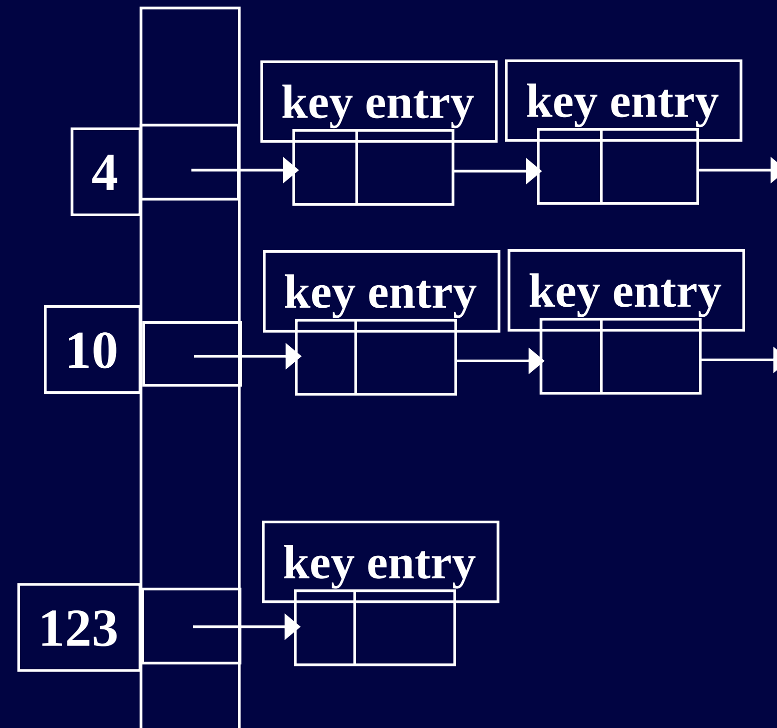
- Quadratic probing uses different formula:
  - Use $F(i) = i^2$ to resolve collisions
  - If hash function resolves to H and a search in cell H is inconclusive, try $H + 1^2$, $H + 2^2$, $H + 3^2$, …
- Probe

  array[hash(key)+1^2], then
  array[hash(key)+2^2], then
  array[hash(key)+3^2], and so on
  - Virtually eliminates primary clusters

Collision resolution: chaining

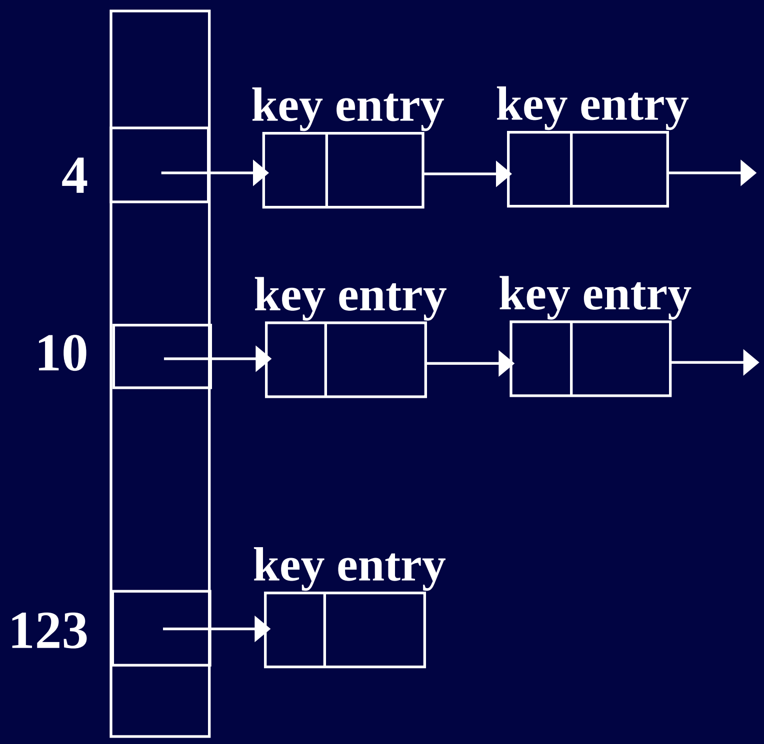
- Each table position is a linked list
- Add the keys and entries anywhere in the list (front easiest)

No need to change position!



Collision resolution: chaining

- Advantages over open addressing:
  - Simpler insertion and removal
  - Array size is not a limitation
- Disadvantage
  - Memory overhead is large if entries are small.



Applications of Hashing

- Compilers use hash tables to keep track of declared variables (symbol table).

- A hash table can be used for on-line spelling checkers — if misspelling detection (rather than correction) is important, an entire dictionary can be hashed and words checked in constant time.

Applications of Hashing

- Game playing programs use hash tables to store seen positions, thereby saving computation time if the position is encountered again.

- Hash functions can be used to quickly check for inequality — if two elements hash to different values they must be different.

When is hashing suitable?

- Hash tables are very good if there is a need for many searches in a reasonably stable table.
- Hash tables are not so good if there are many insertions and deletions, or if table traversals are needed — in this case, AVL trees are better.
- Also, hashing is very slow for any operations which require the entries to be sorted
  - e.g. Find the minimum key