ADITYA ENGINEERING COLLEGE (A)

ADVANCED DATA STRUCTURES

By

Dr. A. Vanathi

Associate Professor

Dept. of Computer Science and Engineering

Aditya Engineering College(A)

Surampalem

Course Objectives:

COB 1:	To make the students learn External Sorting and Hashing Techniques.
COB 2:	To impart the knowledge on Priority Queues.
COB 3:	To provide knowledge on Efficient Binary Search trees and Multiway Search Trees.
COB 4:	To enable the students know the significance of Digital Search Trees.
COB 5:	To facilitate the students learn String Processing Algorithms.

Course Outcomes:

At the end of the Course, Student will be able to:

CO 1:	Demonstrate the External Sorting and Hashing.
CO 2:	Illustrate the concepts of Priority Queues.
CO 3:	Analyze the Efficient Binary Search trees and Multiway Search Trees.
CO 4:	Compare the Digital Search Structures.
CO 5:	Apply the String Matching Algorithms to real time applications.

SYLLABUS

Unit I:

External Sorting: Introduction, K-way Merge Sort, Buffer Handling for parallel Operation, Run Generation, Optimal Merging of Runs, Huffman Tree.

Hashing: Introduction to Static Hashing, Hash Tables, Hash Functions, Different Hash Functions, Collision Resolution Techniques, Dynamic Hashing.

Unit II:

Priority Queues (Heaps): Introduction, Binary Heaps-Model and Simple Implementation, Basic Heap Operations, Other Heap Operations, Applications of Priority Queues, Binomial Heaps (or Queues), Binomial Heap Structure and Implementation, Binomial Queue Operations.

Unit III:

Efficient Binary Search Trees: Self-balancing Binary Search Tree, AVL Trees, Rotations-LL, RR, LR and RL, Searching, Insertion, Deletion operations on AVL Trees, Red-Black Tree, Properties and Representation of Red-Black Trees, Insertion and deletion operations on Red-Black Trees, Applications of Red-Black Trees

Unit IV:

Multiway Search Trees: M-Way Search Trees Definition and Properties, B-Tree Definition and Properties, Searching, Insertion and Deletion operations on B-Trees, B+ Tree, Insertion and Deletion operations on B+ Trees.

Digital Search Structures: Introduction to Digital Search Tree, Operations on Digital Search Trees- Insertion, Searching, and Deletion.

Unit V:

Digital Search Structures: Binary Tries, Compressed Binary Trie, Patricia, Searching Patricia, inserting into Patricia, delete a node from Patricia, Multiway Tries- Definition, Searching a Trie, Compressed Tries, Compressed Tries with Digit Numbers-Searching, Insertion, Deletion.

String Processing: String Operations, Brute-Force Pattern Matching, The Boyer-Moore Algorithm, The Knuth-Morris-Pratt Algorithm, The Longest Common Subsequence Problem (LCS).

Text Books:

1. Advanced Data Structures, Reema Thareja, S. Rama Sree, Oxford University Press.
2. Data Structures and Algorithm Analysis in C, Mark Allen Weiss, Second Edition, Pearson.

Reference Books:

1. Fundamentals Of Data Structures In C, Horowitz, Sahni, Anderson-Freed, Second edition.
2. Data Structures and Algorithms, A. V. Aho, J. E. Hopcroft, and J. D. Ullman, Pearson.
3. Introduction to Algorithms, Thomas H. Cormen, Charles E. Leiserson and Ronald L. Rivest, Third Edition, The MIT Press.
4. Advanced Data Structures, Peter Brass, Cambridge University Press.

External Sorting

- The term sorting means arranging the elements of an array in either ascending or descending order.
- There are two types of sorting:
  - Internal sorting and
  - External sorting.
- Internal sorting is concerned with the ordering of elements present in a file stored in computers memory.
- Eg:Bubble Sort, Insertion Sort, Selection Sort, Heap Sort, Quick Sort, Merge Sort and Radix Sort.
- External sorting is applied when there is voluminous data to be sorted that cannot fit in the memory.
- Because of their large volumes, the files are stored in external storage devices like magnetic tapes, magnetic disks etc

- The files present on the disk are read into the internal memory in terms of blocks, one block at a time.
- The block of records stored in internal memory are sorted making use of any of the existing internal sorting techniques.
- Each of the sorted block of records is called as a ***run***.
- The file is viewed as a collection of runs.
- The runs are written on to the external storage devices as and when required.
- The most popular method for sorting files on external storage devices is External Merge Sort also called as K-way merge sort.

K- WAY MERGE SORT

- External merge sort is performed in two phases.
- The first phase involves the ***run generation*** and the second phase involves the ***merging of runs to form a larger run.***
- This run generation is repeated and merging is continued till a single run is generated with the sorted file as its outcome.
- If k runs are merged at a time, the external merge sort is called as a k-way Merge Sort.
- The k-Way merge sort where k=2 is a 2-way merge sort.
- In 2 – Way merge sort, 2 runs are merged at a time to generate a single run twice as long.
- The merging process is repeated until a single run is generated.
- Eg.: Consider 6000 records are to be sorted and memory can hold 500 records.

2-WAY MERGE SORT

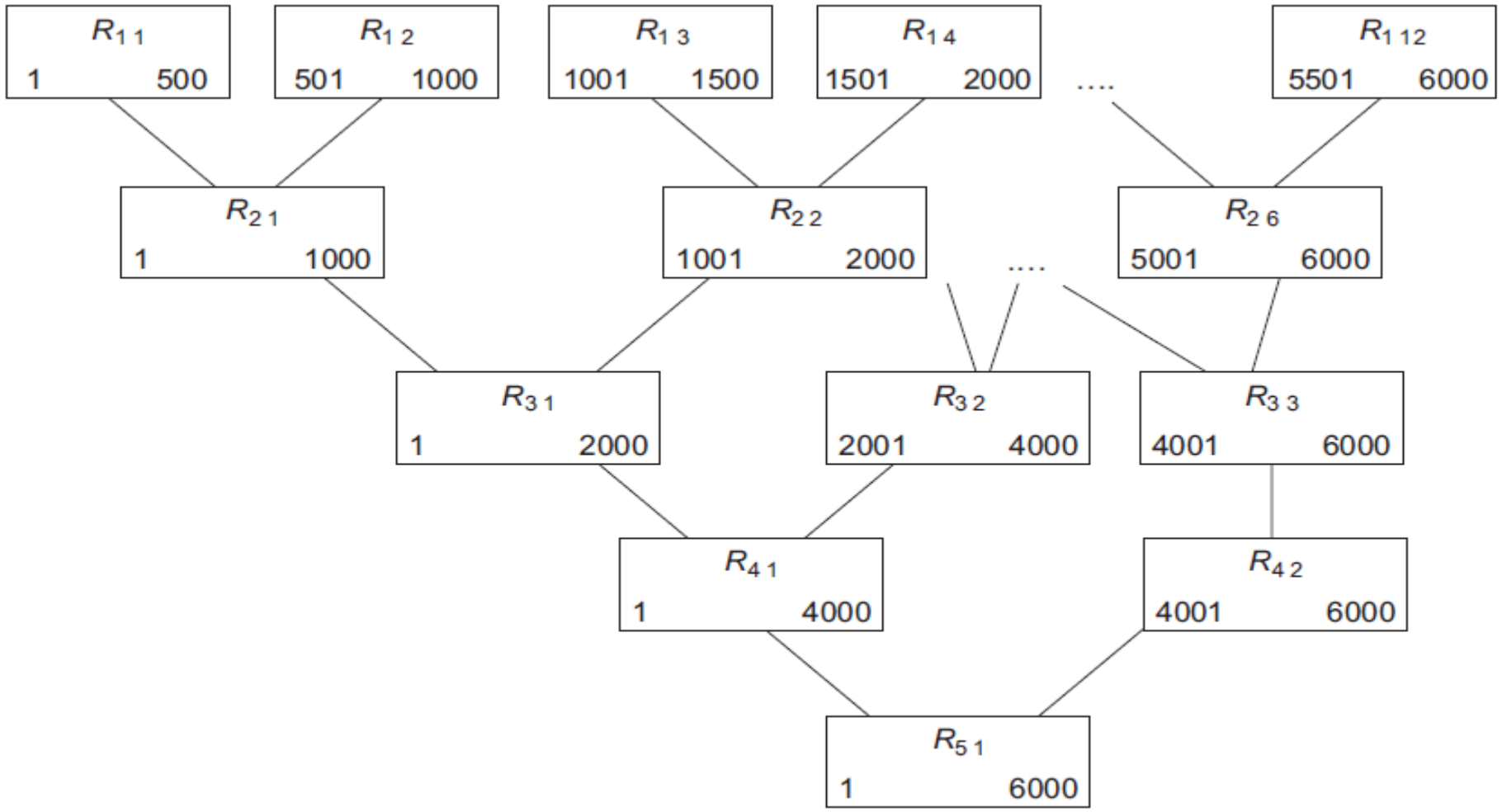
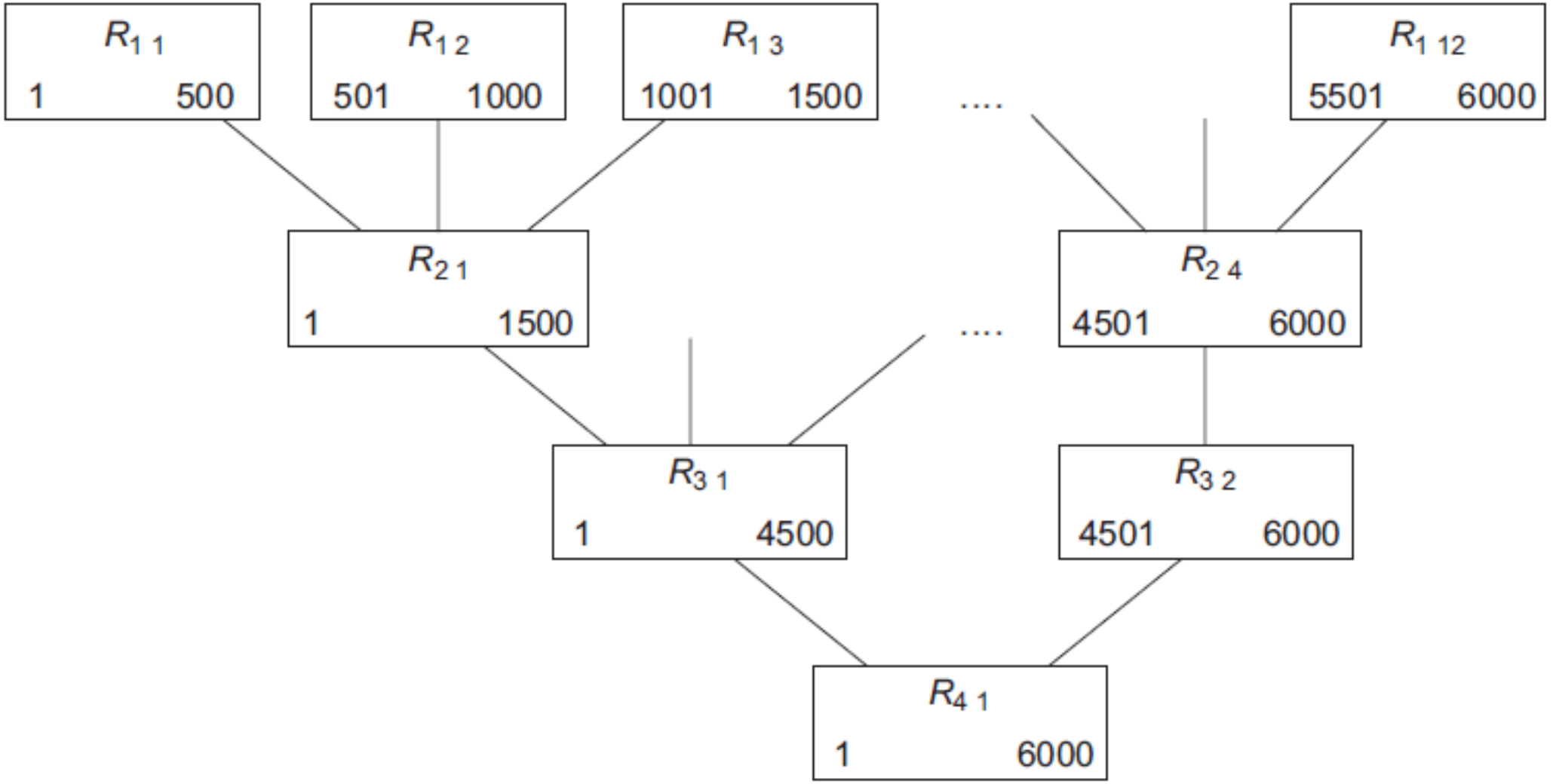
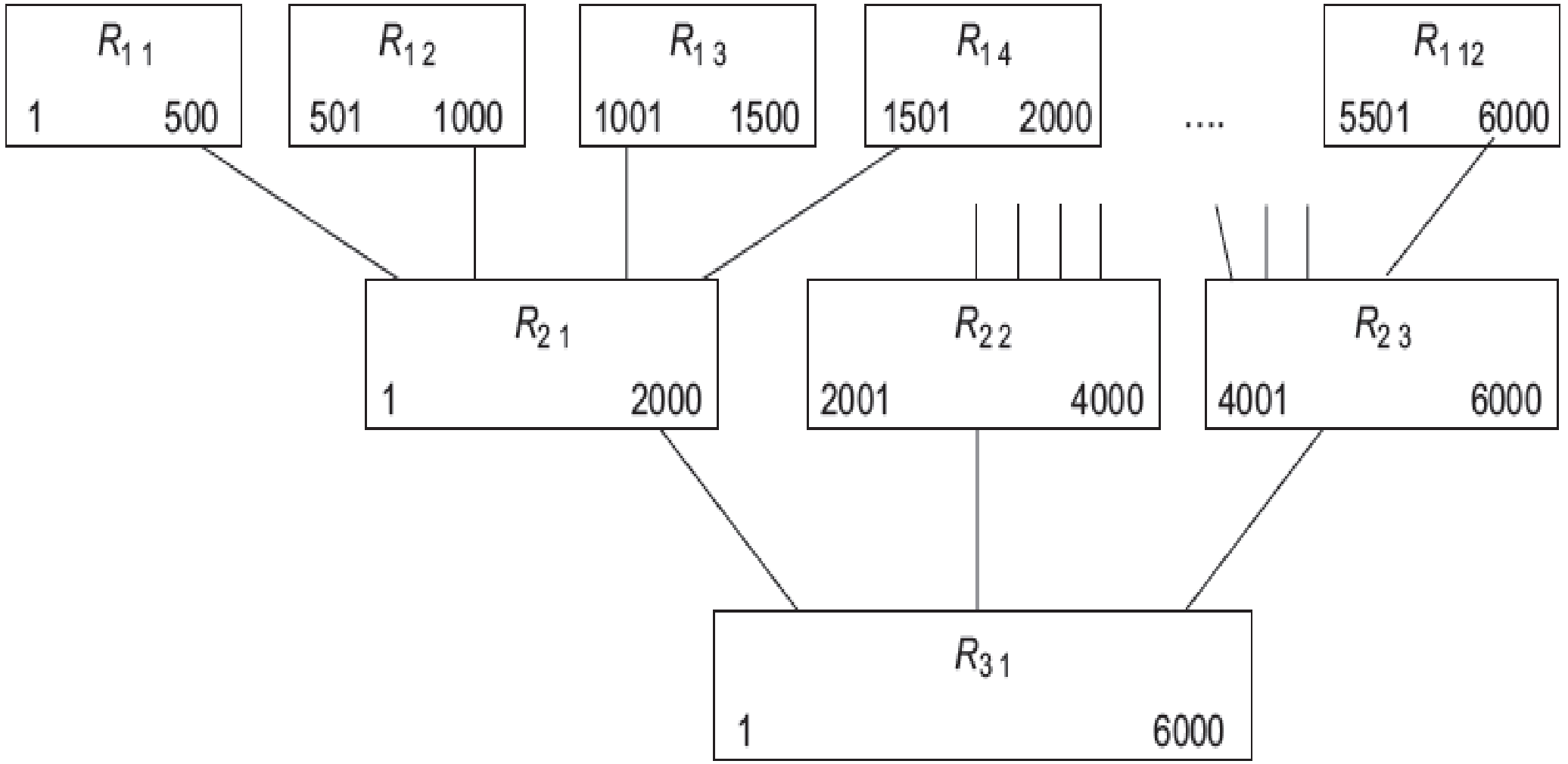


Figure 1.1 2–way merge sort

3-WAY MERGE SORT



4-WAY MERGE SORT



Example: 2- way merge sort

- Consider 6000 records are available on a disk which are to be sorted. In the internal memory of the computer, only 500 records can be resided. The block size of the disk is 100 records. Sort the file using 2-way merge sort.
- **Solution:** The steps involved in sorting the file are as follows:

  Step 1: Read five blocks of data i.e. totally 500 records at a time from the file residing on the disk to internal memory. Sort these blocks using any internal sorting technique to generate 12 runs (runs = 6000 records / 500 records). These runs are written back on to the disk after sorting.

  Step 2: Merge 2 runs at a time to generate a new run in the next pass with size twice as long.

  Until all runs in the pass are processed

  Step 3: Repeat Step 2 until a single run is generated.

  Step 4: EXIT

Example: Perform 2-way merge sort

81 94 96 11 12 35 99 17 28 58 41 75 15

Unsorted Data on Disk

Assume k = 2 First step is to read 2 data items at a time into main memory, sort them and write them back to disk as runs of length 2. The initial blocks are

81 94 | 96 11 | 12 35 | 99 17

28 58 | 41 75 | 15

After internal sort, the runs are

81 94 | 11 96 | 12 35 | 17 99

28 58 | 41 75 | 15

Merge the runs of length 2 into runs of length 4. After merging & sorting, the runs are

11 81 94 96 | 12 17 35 99 | 28 41 58 75 | 15

Next step is to merge the runs of length 4 into runs of length 8.

11 12 17 35 81 94 96 99

15 28 41 58 75

Next step is to merge the runs of length 8 into runs of length 16.
The final sorted run is

11 12 15 17 28 35 41 58 75 81 94 96 99

k-Way_mergesort(File1, M, k)

File1 – unsorted file on disk

M –max records that can be stored & sorted in internal memory at a time

k – number of runs to be merged

Step 1: Run Generation

Repeat until all records in File1 are processed into runs

Read M records into main memory & sort internally.

Write this sorted sub-list (run) onto disk.

[END OF LOOP]

Step 2: Merging of Runs

Repeat until all runs are processed

Merge k runs into one sorted run with size as k times the input run size

Write this single run back onto disk

[END OF LOOP]

Step 3: If more than 1 run is generated in Step 2 then

Repeat Step 2

Else

The file on the external storage is sorted

Step 4: EXIT

Merging Runs - Implementation

- The k-Way merge sort includes merging of runs in every pass.
- The simplest merging technique is the k-way merge, where k runs are merged into one run.
- The merging process includes the following steps.
- Identify the first smallest record of each run and place it in the smallest set.
- The smallest of the records in the smallest set is the smallest record overall. This record is put in the output run and removed from the corresponding run.
- The next smallest record in the corresponding run is moved to the smallest set.
- This process is repeated until the initial runs are empty.

Example:

Consider 3 runs with 4 records each. Show the merging process in the 3-way merge sort

Run1 - 3 ,5 , 12, 15

Run2 - 2, 4, 10, 17

Run3 - 1, 6, 8, 18

Output Run – 1, 2, 3, 4, 5, 6, 8, 10, 12, 15, 17 18

Solution: The step by step process of merging includes

Consider the smallest record of each run and add it to the smallest set: {3, 2, 1}

- Step 1: The three records in the smallest set are {3, 2, 1}. Remove the smallest record 1, from the third run and put it in the output run: {1}. Move 6 to the smallest set.

- Step 2: The three records in smallest set are {3, 2, 6}. Remove 2 from the second run. output run: {1, 2}. Move 4 to the smallest set.

- Step 3: smallest set is {3, 4, 6}. Remove 3 from the first run.
  output run: {1, 2, 3}. Move 5 to the smallest set.

- Step 4: smallest set is {5, 4, 6}. Remove 4 from the second run.
  output run: {1, 2, 3, 4}. Move 10 to the smallest set.

- Step 5: smallest set is {5, 10, 6}. Remove 5 from the first run.
  output run: {1, 2, 3, 4, 5}. Move 12 to the smallest set.

- Step 6: smallest set is {12, 10, 6}. Remove 6 from the third run.
  output run: {1, 2, 3, 4, 5,6}. Move 8 to the smallest set.

- Step 7: smallest set is {12, 10, 8}. Remove 8 from the third run.

  output run: {1, 2, 3, 4, 5, 6, 8}. Move 18 to the smallest set.
- Step 8: smallest set is {12, 10, 18}. Remove 10 from the second run.

  output run: {1, 2, 3, 4, 5, 6, 8, 10}. Move 17 to the smallest set.
- Step 9: smallest set is {12, 17, 18}. Remove 12 from the first run.

  output run: {1, 2, 3, 4, 5, 6, 8, 10, 12}. Move 15 to the smallest set.
- Step 10: smallest set is {15, 17, 18}. Remove 15 from the first run.

  output run: {1, 2, 3, 4, 5, 6, 8, 10, 12, 15}.The first run is empty, the merge follows as a 2-way merge instead of a 3-way merge.
- Step 11: smallest set is {17, 18}. Remove 17 from the second run . The output run: {1, 2, 3, 4, 5, 6, 8, 10, 12, 15, 17}. Now, the second run is also empty, only the third run remains non-empty.
- Step 12: smallest set is {18}. Remove 18 and append to the output run.
- The result of merging the three initial runs is {1, 2, 3, 4, 5, 6, 8, 10, 12, 15, 17, 18}.

Buffer handling for parallel operations

- In k-way Merge sorting technique, we need atleast

  k-input buffers and 1-output buffer

The major steps involved in the k-way Merge sort are:

Read: Load the records from Runs into input buffers

Merge: Perform Merge on input buffers & store result in output buffer

Write: Store the records from the output buffer onto the Disk.

- Instead of performing these operations in serial, performing these operations in parallel improves the efficiency,

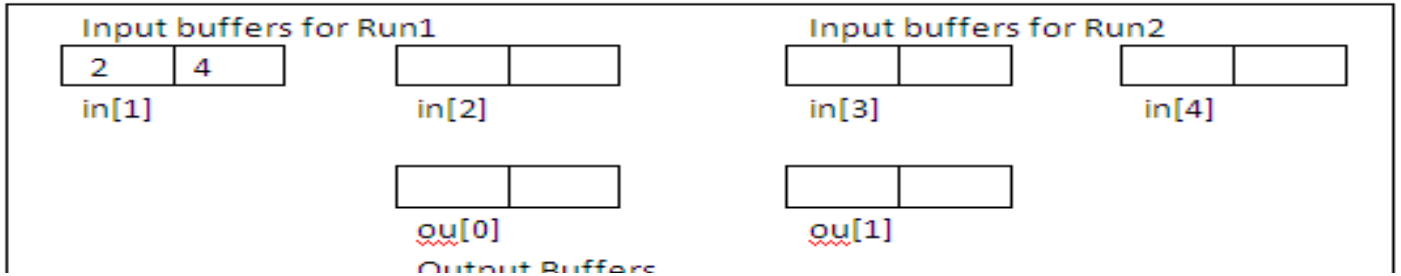
  i.e. we can perform Read, Merge and Write operations simultaneously.
- But k-input buffers and 1-output buffer strategy is not sufficient to handle the parallel operations.
- We require 2k-input buffers(read & merge) and 2-output buffers(merge & write) to handle the operations in parallel.

Buffer handling for parallel operations

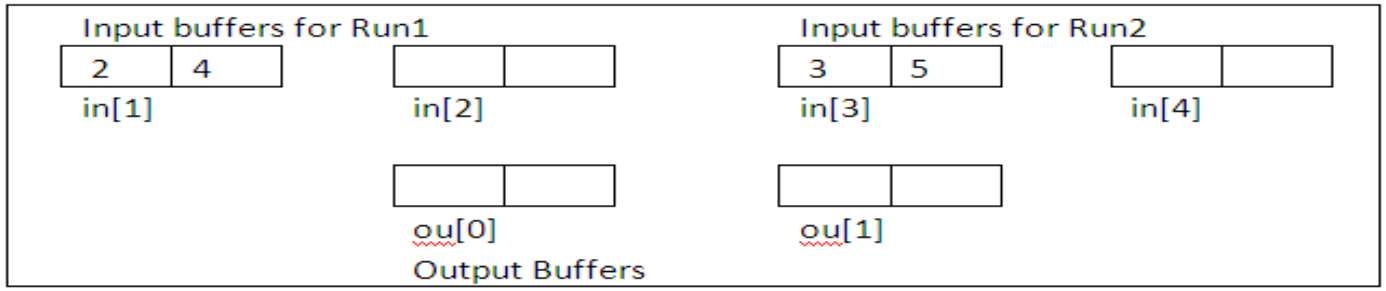
- Using buffer handling for parallel operations, implement 2-way merge for the runs shown below.

- Let block size= 2 and each buffer can hold 2 records.
- As 2-way merge is to be used, the input buffers used are 4 and the output buffers used are 2.
- Let in[1],in[2] represent the input buffers for loading the records from Run1 and in[3],in[4] represent the input buffers for loading the records from Run2.
- Let ou[0] and ou[1] represent the output buffers.
- Initially all the input and output buffers are empty.
- The steps involved are

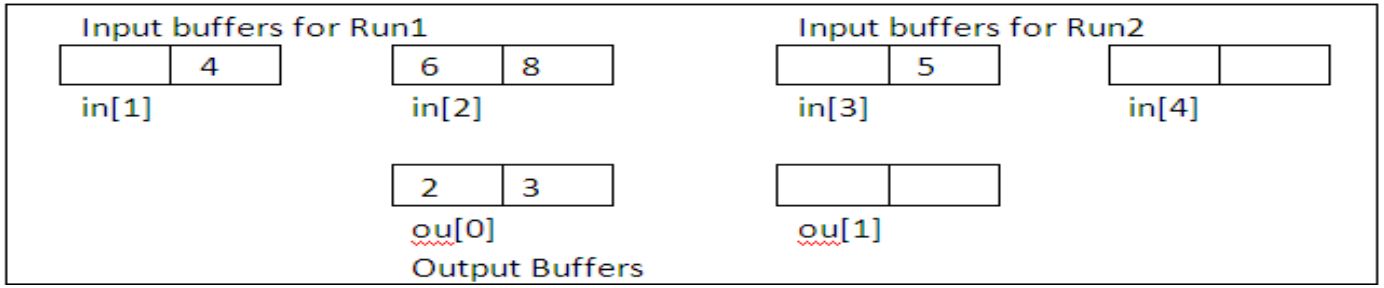
step1



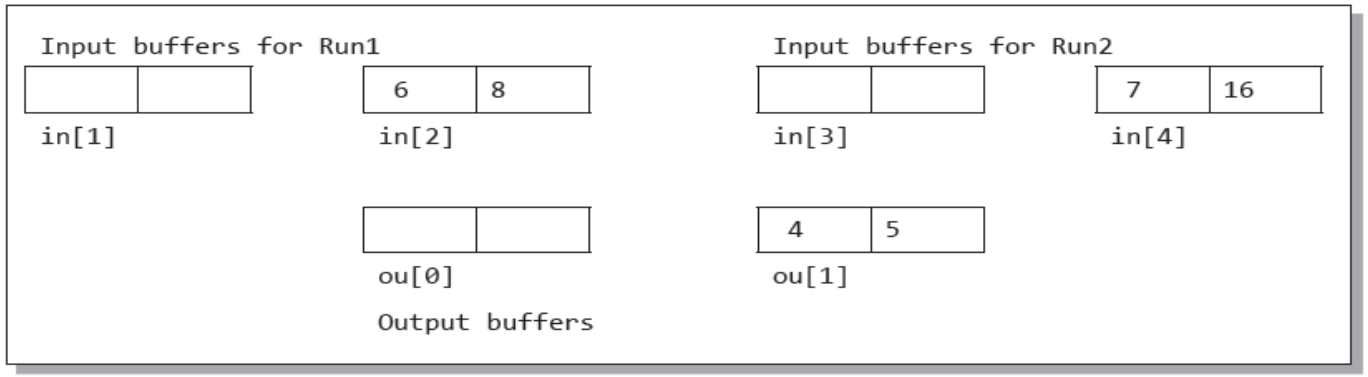
step2



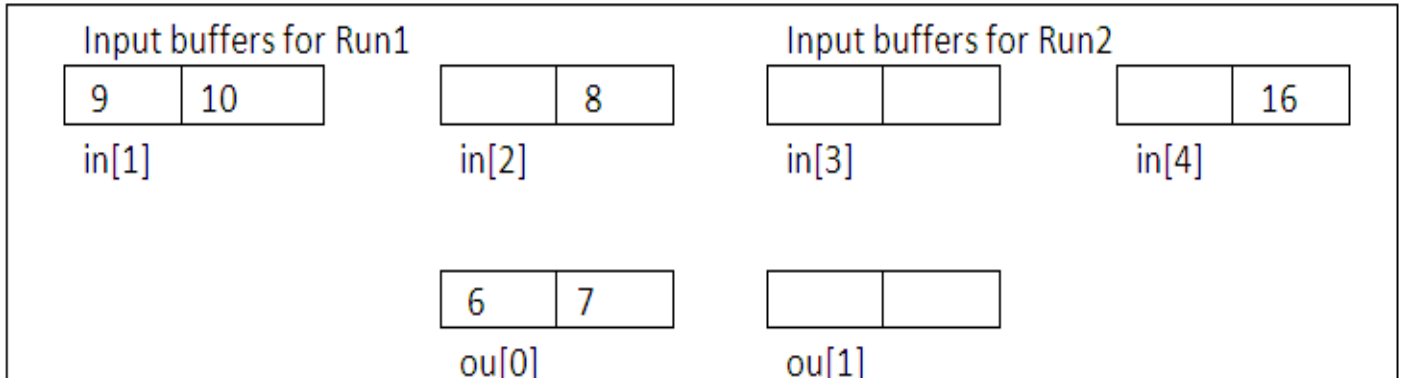
step3



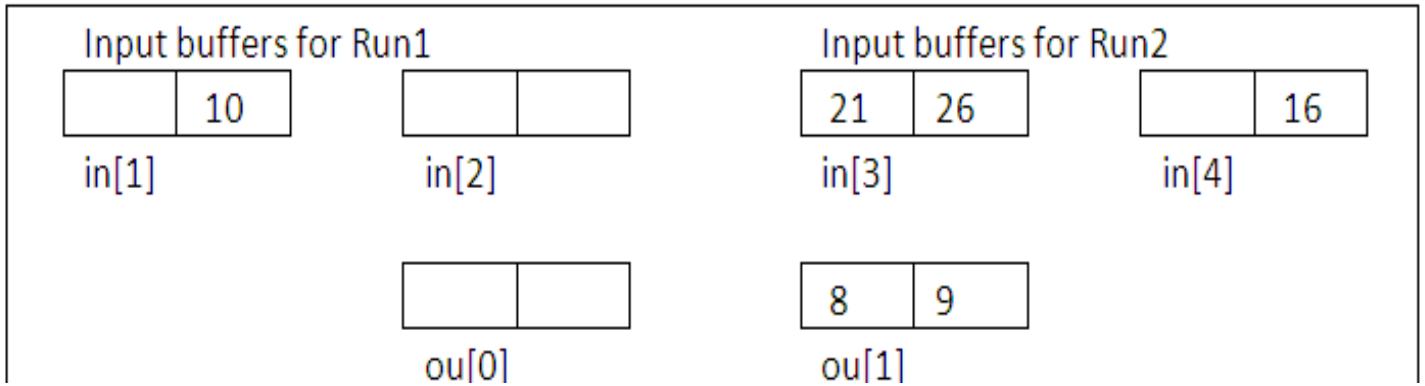
step4



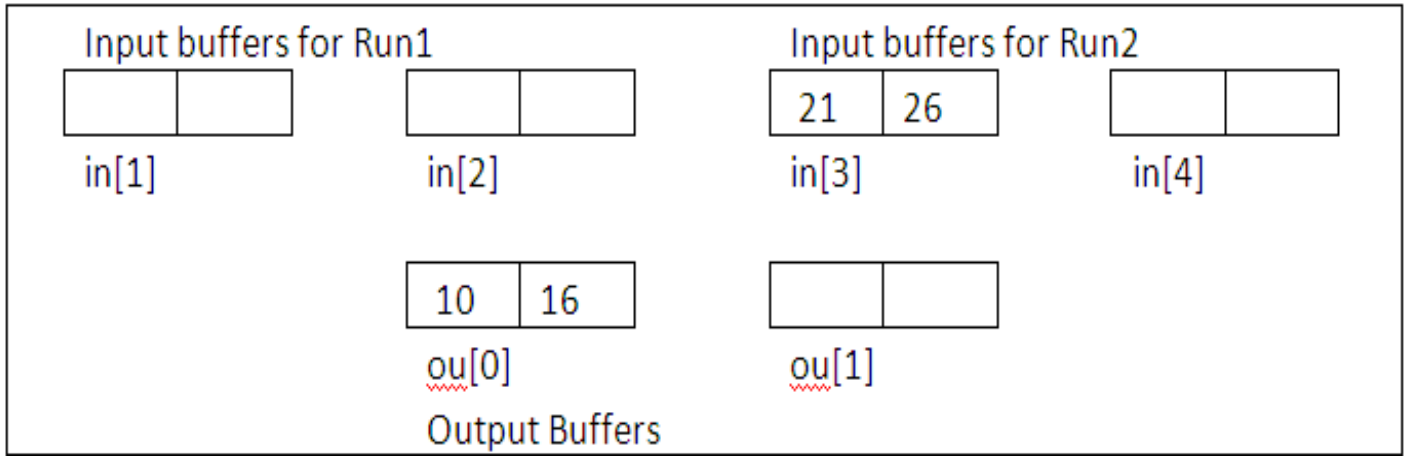
step5



step6



step7



- After Step 7, merging will be delayed until another record is loaded from Run1 into in[1].
- Processing has to be held up due to lack of input records from the run. Simply assigning 2 buffers per run does not solve the problem. Therefore, input buffers have to be assigned to the runs cleverly to avoid the input delay,
- An individual buffer may be assigned to any run depending upon need. This type of buffer that is assigned to a run when needed is called a floating buffer and the k–way merge algorithm using floating buffers is called *buffering algorithm*.
- In the buffer assignment strategy of floating buffers, at any time there will be at least one input buffer for each run. The remaining buffers will be filled on a priority basis.

The following assumptions are made for buffering algorithm

1. For performing “read” and “write” onto a disk simultaneously, we take two disk drives.
2. While read or write operation is being processed, the CPU cannot make reference to the same block of memory.
3. Input and output buffers are of the same size.
4. End of each run has a sentinel record with a very large key, say +∞ and all other records have key value less than that of sentinel record.
5. The time to merge into an output buffer equals the time to read a block.
6. In case of equal keys, the run with smallest index is chosen for the next read operation.
7. Input buffers are queued in k queues, one queue for each run. Empty buffers are placed on a linked stack.

k-way merge using floating buffers

Let k be the no. of runs, s be the buffer size

Step 1: Set up k linked queues, insert the first block of each run in an input buffer attached to the corresponding queue. The remaining k input buffers are pushed into a linked stack of empty input buffers. Let the output buffer be represented as ou.

Step 2: Let LastKey[i] be the key value recently placed into the input buffer from run i where $1 \le i \le k$. Let N = i where LastKey[i] is minimum. N is the run from which the next block is to be loaded. If LastKey[N] $\neq +\infty$, then insert the next block from run N into free input buffer.

Step 3: Select s minimum keys from all the input buffers, merge them into the output buffer ou and initiate the writing of ou onto disk. Merging continues until the output buffer gets full. If an input buffer becomes empty before the output buffer gets full, the next buffer on the same queue is considered and the empty buffer is returned to the stack of empty buffers. However, if an input buffer becomes empty and the output buffer gets $+\infty$ merged into it, the empty buffer is left on the queue. If the output buffer gets $+\infty$ merged into it, the contents of ou are copied onto disk, the merge terminates and the control goes to Step 6.

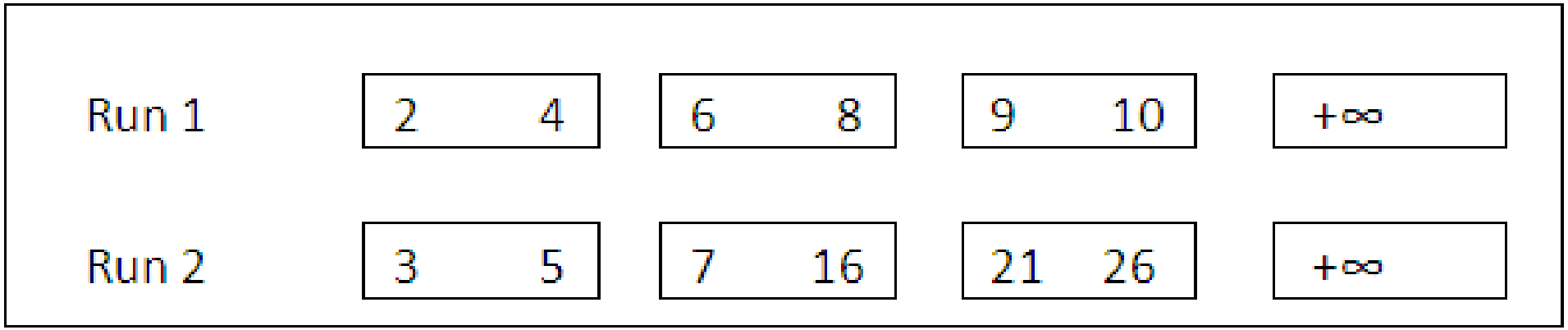
Step 4: Wait for any ongoing disk input/output to complete.

Step 5: Repeat Steps 2 to 4 until there is no input block to be read.

Step 6: EXIT

2-way merge with floating buffers

Implement k-Way merge with floating buffers for the runs shown below. Assume that the last block of each run is loaded with the sentinel record with key +∞. The blocks in Run1 and Run2 are as follows:



2-way merge with floating buffers

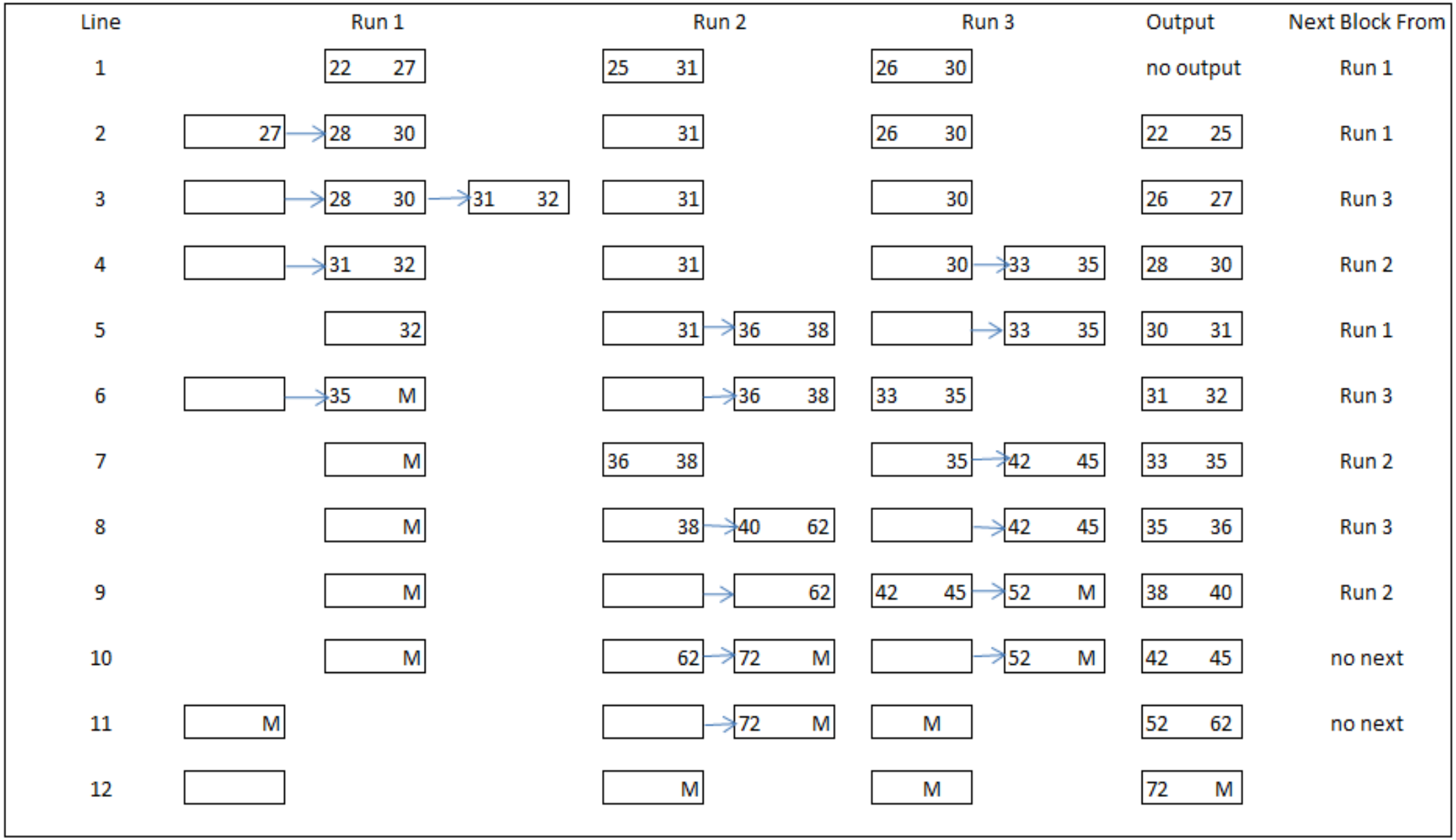
Line	Run 1	Run 2	Output	Next Block From
1	2 4	3 5	no output	Run 1
2	4 → 6 8	5	2 3	Run 2
3	→ 6 8	→ 7 16	4 5	Run 1
4	8 → 9 10	16	6 7	Run 1
5	→ 10 → M	16	8 9	Run 2
6	→ M	→ 21 26	10 16	Run 2
7	M	→ M	21 26	no next
8			M M	

3-way merge with floating buffers

Implement k-Way merge with floating buffers for the runs shown below. Assume that the last block of each run is loaded with the sentinel record with key +∞. The blocks in Run1, Run2 and Run3 are as follows:

Run 1	22	27	28	30	31	32	35	+∞
Run 2	25	31	36	38	40	62	72	+∞
Run 3	26	30	33	35	42	45	52	+∞

3-way merge with floating buffers



MERGING OF RUNS USING TOURNMENT TREES

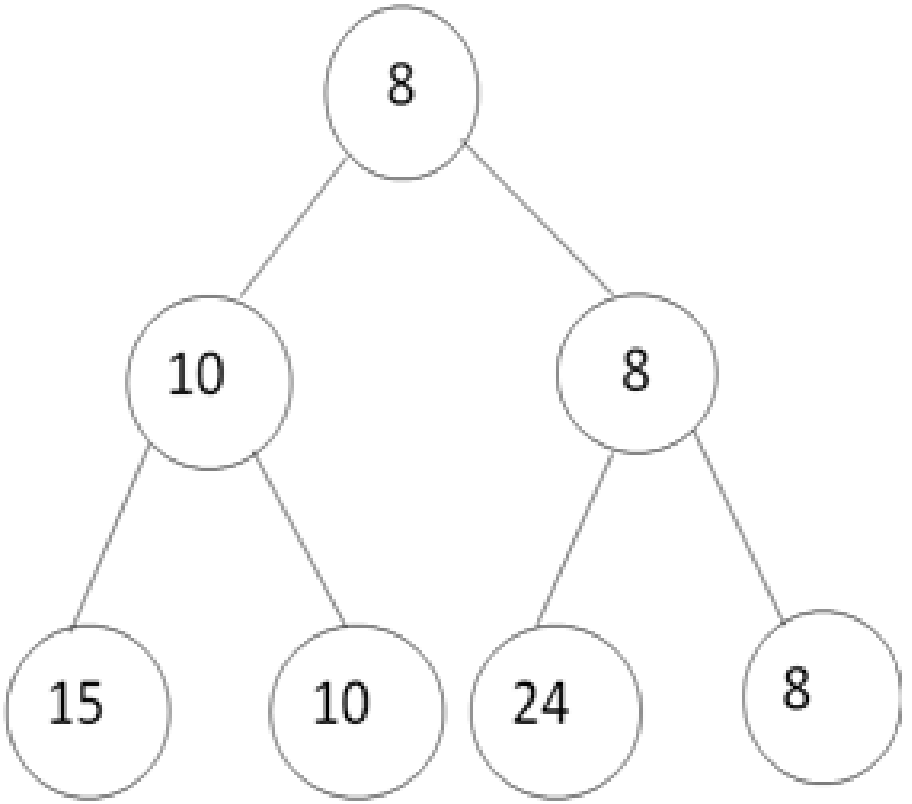
- Using internal sorting techniques, it is possible to generate runs whose size = internal memory.
- But, by using the loser's tree, it can be performed in a better way.
- A loser tree can be understood clearly by knowing firstly about Winner trees.
- A **winner tree** is a complete binary tree where each internal node represents the smaller of its two children.
- The root consists of the smallest value in the entire tree.
- As like a tournament tree, each internal node of the winner tree represents the winner(smallest) of a tournament and the root node is the overall winner(overall smallest) of the entire tournament.
- The keys or records are present in the external nodes or leaf nodes only.

- **Example -** Consider merging of 4 runs using Winner tree and Loser tree for the Runs shown below.

Run 1	Run2	Run 3	Run 4
15	10	24	8
25	18	56	12
30	20	68	15
46	93	76	28

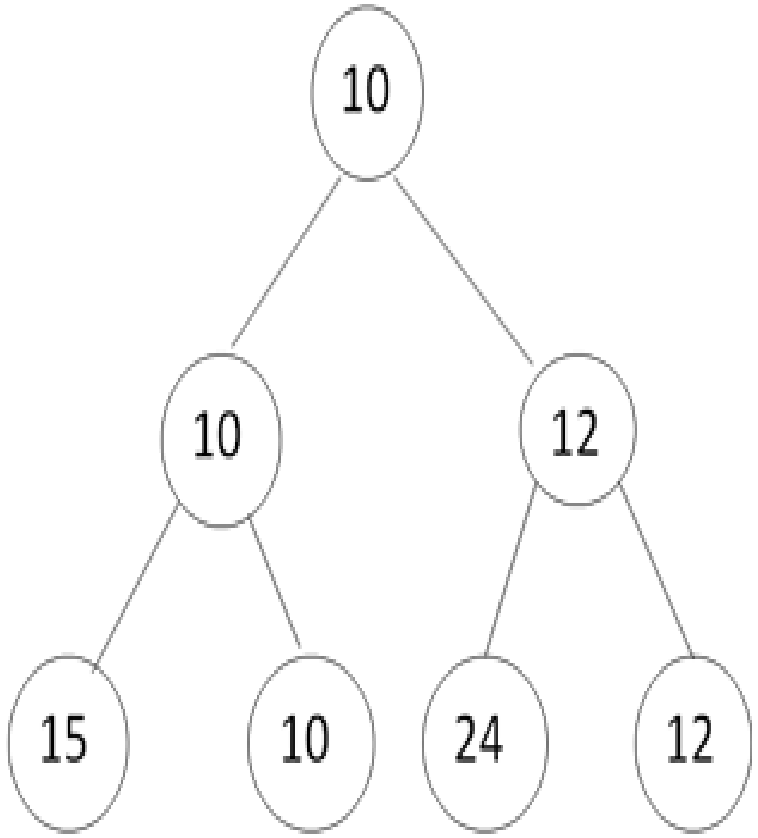
Winner tree

- The first key of each run is initially loaded into the external nodes of the winner tree.
- Comparison takes place at every level of the tree, starting from the leaf nodes towards the root.
- The internal nodes 10 and 8 represent the winner of the respective tournaments played between 15, 10 and 24, 8 respectively.
- The root node of the winner tree consists of the overall winner i.e., 8.
- This value pointed by root is stored onto the disk.



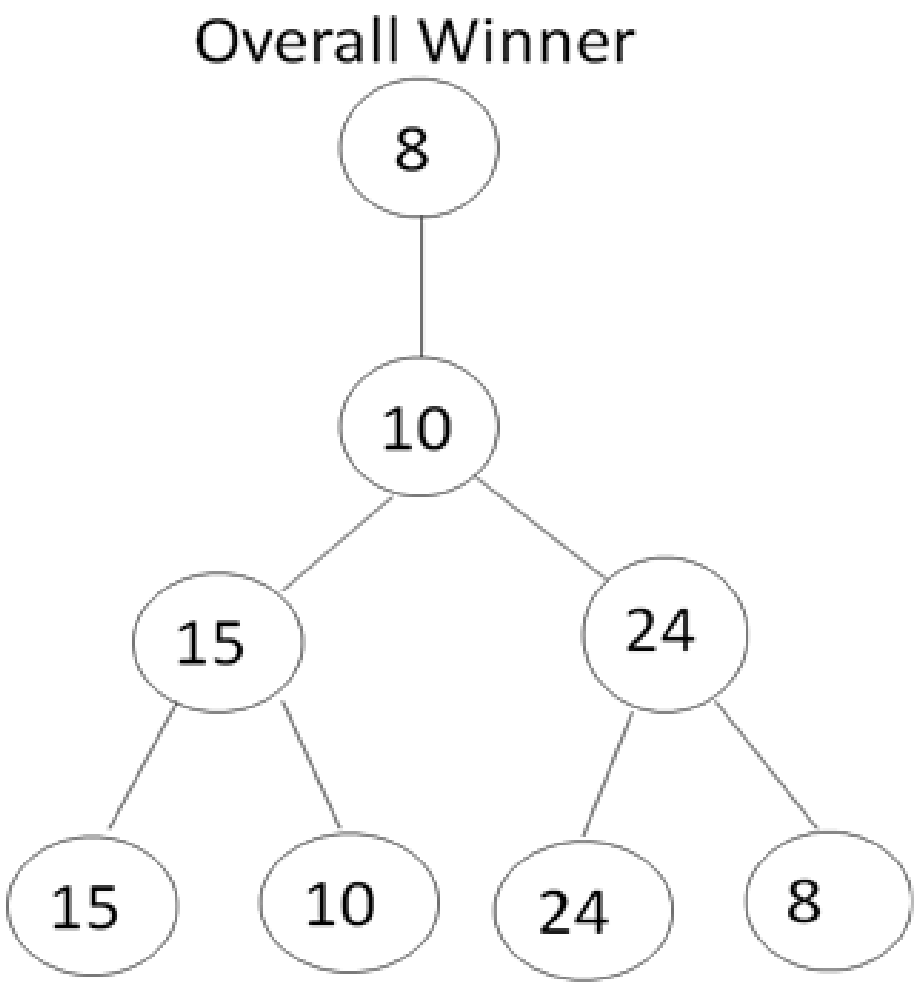
Winner Tree after first record is output

- When a record is output in the Winner tree, the next record to be placed in the external node is taken from the run from which the smallest record is output in the previous pass
- The smallest record at root i.e 8 is taken from Run 4. So, the next value that is input is to be taken from Run4.
- The next value in Run4 to be input is 12.
- The value 12 is compared against the other values which are basically the losers in the previous pass.
- The overall winner 10, is stored onto disk.
- The process repeats until all the elements from the runs are processed.



Loser tree

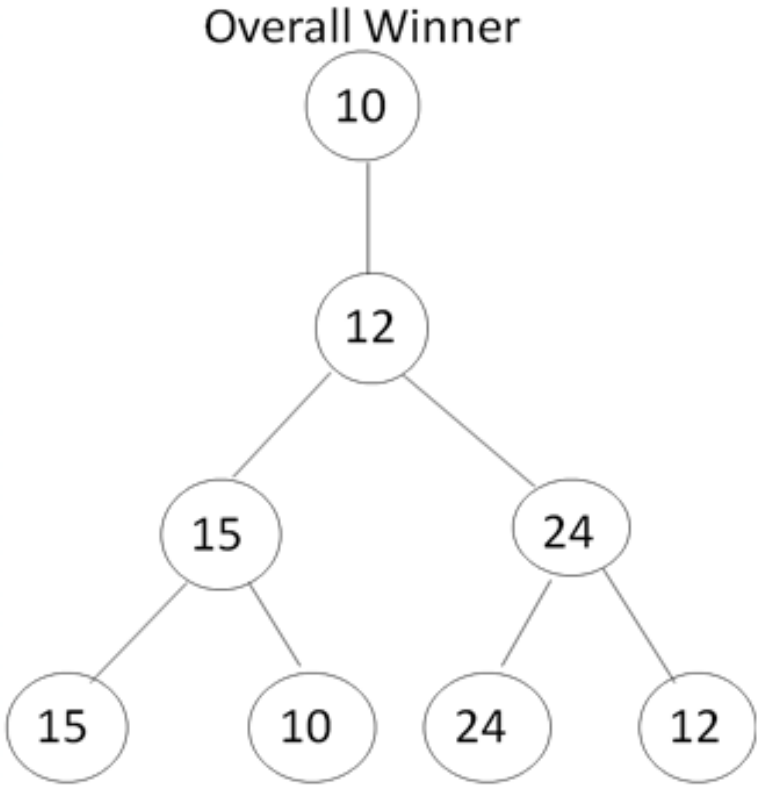
- Another efficient way of generating runs is by using Loser trees in which the number of comparisons are reduced.
- The winner tree can be restructured in such a way that instead of placing pointers to winners as internal nodes, pointers to losers can be placed as internal nodes. Such trees are called as Loser trees.
- The root node consists of the loser of the winner nodes in the previous level.
- A special node is added on top of the root node to represent the overall winner of the tournament.
- The leaf nodes of Loser tree consist of the first records of all the runs as in Winner tree.



Loser tree

- In the first step, the overall winner is the output and therefore the value 8 is copied onto the disk.
- The next value is to be considered from Run 4 as the output value 8 is from Run 4.
- The value to be input is now 12.
- This is copied into the leaf nodes. But once the overall winner is output, the records with which the input needs to play these tournaments are readily available in the parent nodes.
- As a result, the sibling nodes are not accessed when the tournaments are being played.

Loser Tree after the first record is output



- In the Loser tree after the first record is outputted the value 12 in leaf node is compared with its parent 24 and not compared with its sibling 24.
- The root node consists of loser of the nodes 10 and 12 i.e 12.
- The overall winner consists of the winner of 10 and 12 (i.e. 10).
- The process repeats until all the elements from the runs are processed.

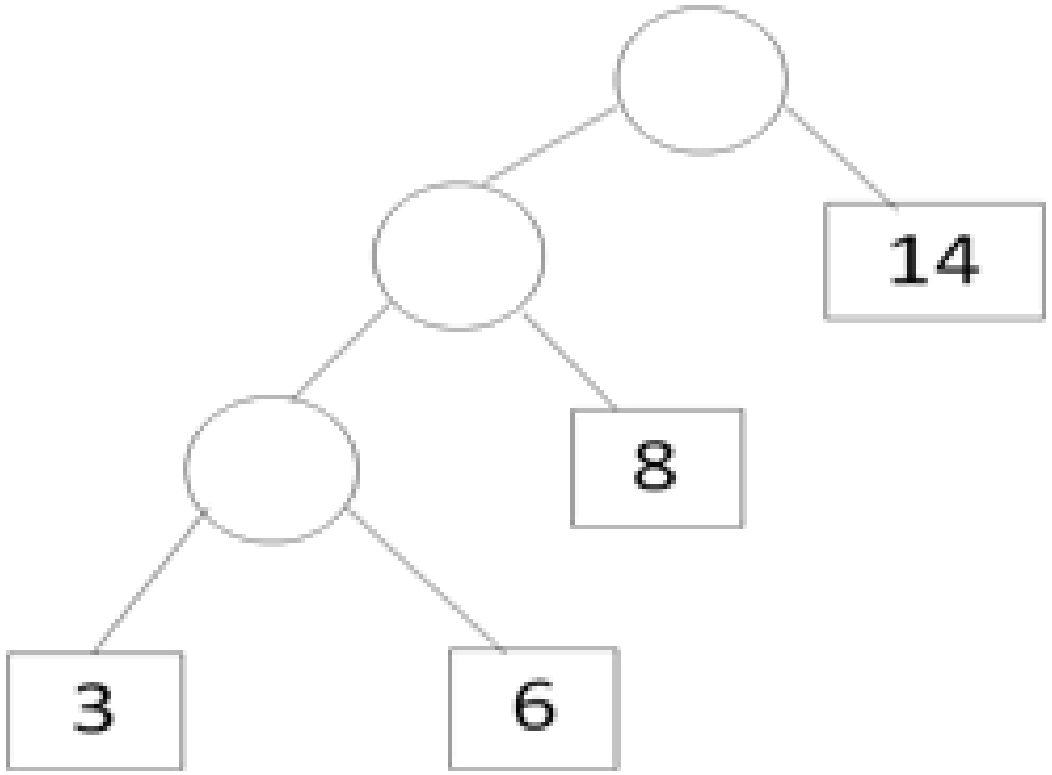
Optimal Merging of Runs (with different sizes)

- When runs of different sizes are to be merged, the process of merging is challenging and the main question is how the merging can be optimized.
- For merging the runs with different sizes, a merge tree is used.
- A merge tree is a tree constructed by merging the runs of different sizes.
- In a merge tree, the circular node is called as internal node and the square node is called as external node which is used to represent the initial runs.
- The possible ways of merging the runs of different sizes can be demonstrated clearly by considering an example of four runs of length 3,6,8 and 14 respectively.
- The runs are merged in two different ways using 2-way merge.

CASE 1: The merging is started by combining the runs of size 3 and 6 represented as external nodes. The output is a single run of size 9 represented as an internal node.

This run with size 9 is next merged with the run of size 8 (external node) which results in the run of size 17 represented as an internal node.

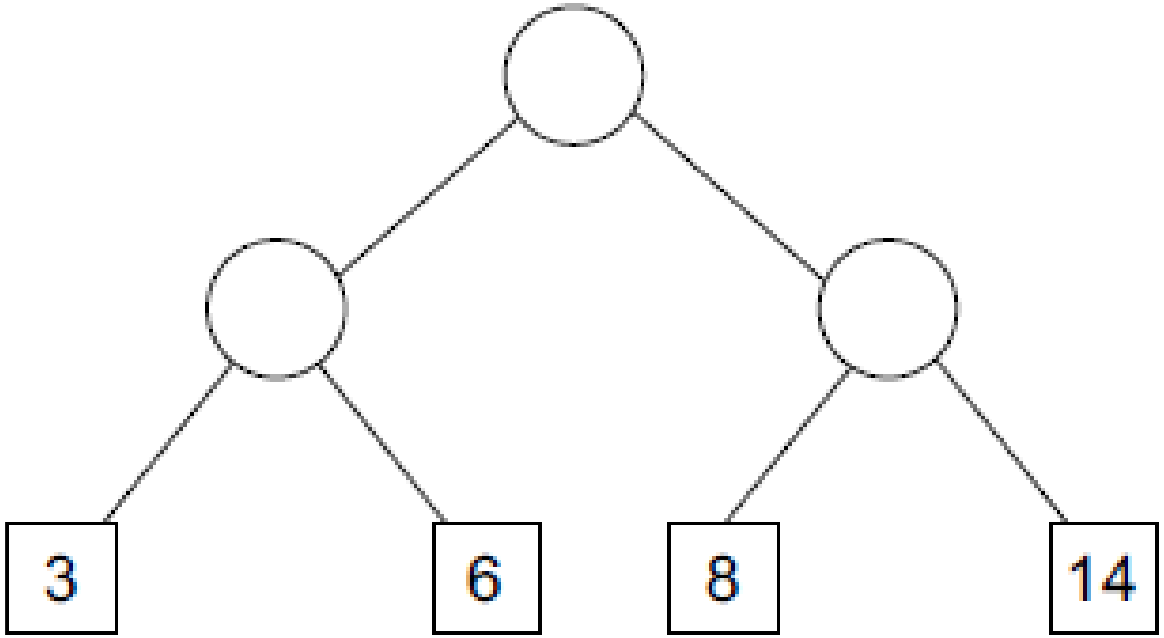
This run is finally merged with the run of size 14(external node) to result in a single run of size 31 represented as a root node.



CASE 2: The merging proceeds by combining the runs of size 3 & 6 represented as external nodes to result in a run of size 9 represented as an internal node.

The next step is to merge the next external nodes i.e runs of size 8 and 14 to result in a run of size 22 represented as an internal node.

Finally, the two internal node are merged i.e merge the run of size 9 with run of size 22 to obtain a single run of size 31 represented as a root node.



Optimal Merging of Runs

- Now the question is which among the two is optimal

  i.e which type of merge has the minimum merging time.

- This can be easily obtained by calculating the Weighted External Path Length.
- The Weighted External Path Length (WEPL) of a tree T is the total merge time calculated by adding the product of weight of an external node and the depth of the external node from the root node.
- The weight of the external node is the run size.
- The depth of the external node is the length of the path from the root node to the external node.
- The WEPL can be calculated using the equation

  WEPL(T) = ∑(weight of external node i)* (depth of node i from the root)

- WEPL(T) = S(weight of external node i) * (depth of node i from the root)
- For the CASE 1 Figure ,

  WEPL(T) = 3 * 3 + 6*3 + 8*2 + 14*1

  = 57
- For the CASE 2 Figure ,

  WEPL(T) = 3*2+6*2+8*2+14*2

  = 62

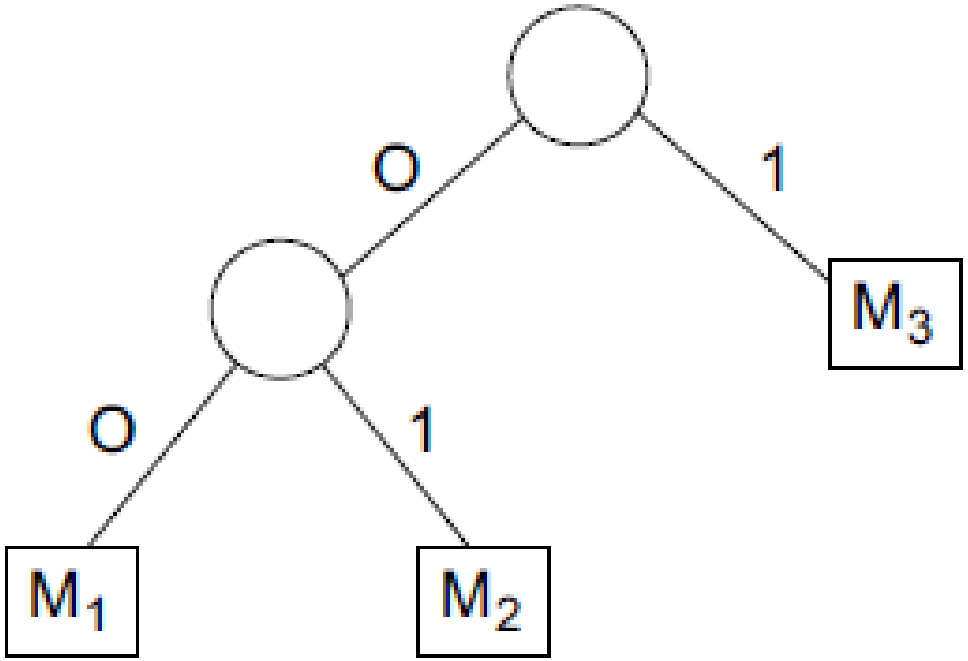
- We observe that CASE 1 Fig. has minimum WEPL. Therefore, the merging technique followed in CASE 1 Fig. is considered as *optimal merge*.

Huffman code

- Another way of finding a binary tree with minimum WEPL is by using Huffman Code.
- Consider the messages $M_1, M_2 \ldots\ldots M_{n+1}$ for which we have to derive the optimal set of codes.
- Each code is binary string used for transmission of messages.
- At the receiver end, the code is decoded using a decode tree, which is a binary tree, where the external nodes represent the messages.
- The binary bits in the code of a message determine the branching needed at each level to reach the correct external node.

- Let ‘0’ represent a left branch and ‘1’ represent a right branch of the decode tree.
- The codes 00, 01 and 1 represent the messages M_1, M_2 and M_3 respectively. These codes are called as Huffman codes.

- The size of message Mi is directly proportional to the number of bits in the code and the cost of decoding the code word is same as WEPL.
- The decoding time can be minimized by constructing a decode tree with minimum WEPL.
- Huffman constructed a Binary tree with minimum weighted external path length called the Huffman tree.
- The Huffman Algorithm is applied for constructing the Huffman tree.



Huffman tree

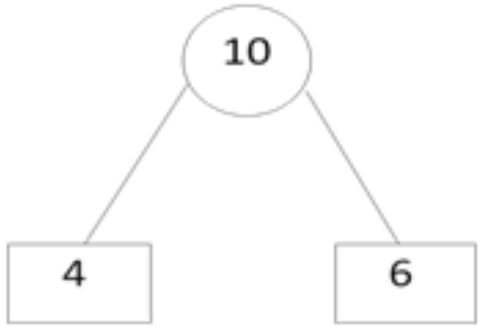
- The two main activities done using the Huffman Algorithm are constructing a Huffman tree and finding the Huffman codes by traversing the nodes from root to leaf nodes in the Huffman tree.
- In the Huffman tree, the leaf nodes represent the characters and the internal nodes represent the intermediary values.

- **Example :** Consider six characters q1 to q6 with the following values for each node.

  q1 = 4, q2 = 6, q3 = 8, q4 = 9, q5 = 15, q6 = 28. Construct a Huffman tree and find the code for each character.

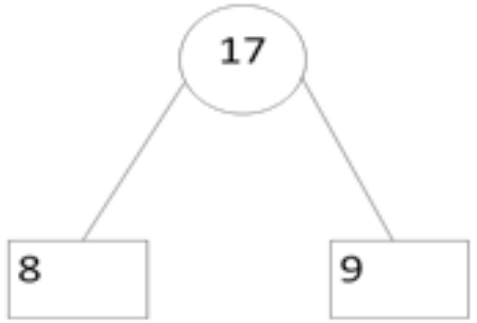
Step 1: create leaf nodes

- Step 2:



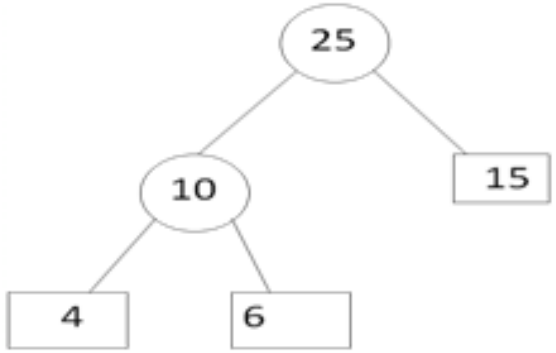
Select 2 min nodes from the given character list. q1 and q2 values are minimum. Construct a Huffman tree with q1 as left child and q2 as right child. Calculate the internal node value i.e., 10. Replace q1 and q2 in the list with this value 10. The list now contains 10 (internal node value), q3 = 8, q4 = 9, q5 = 15, and q6 = 28.

Step 3:



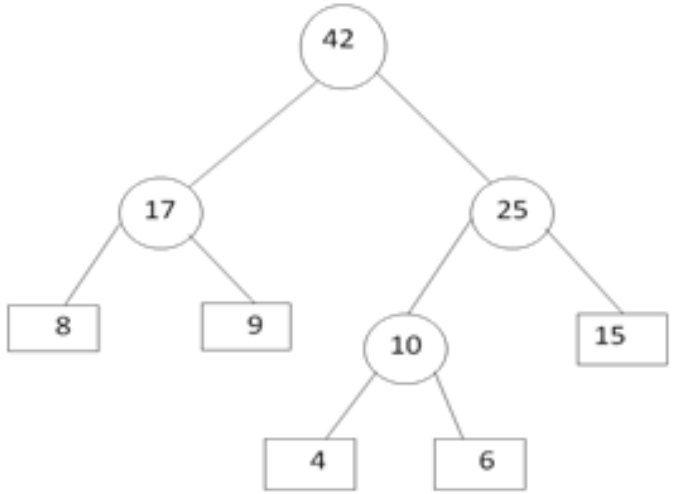
Consider the next 2 min nodes from the list. q3 and q4 are minimum. Construct a tree with q3 as left child and q4 as right child. Create an internal node with value 17 . Replace q3 and q4 in the list with this value 17. The list now contains 10, 17, q5 = 15, and q6 = 28.

- Step 4:



Consider the next 2 min nodes from the list. The values 10 and q5 are minimum. Construct the tree with the internal node 10 as left child and q5 as right child. Create an internal node with value 25. Replace 10 and q5 in the list with this value 25. The list now contains 17, 25, and q6 = 28.

Step 5:



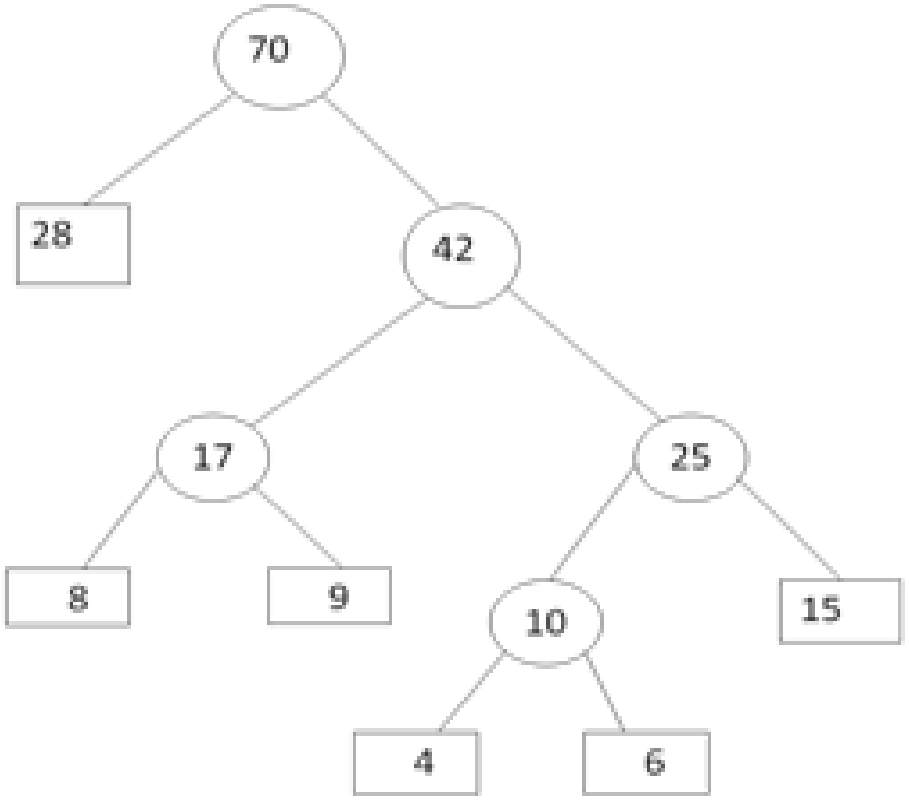
Consider the next 2 min nodes from the list. The values 17 and 25 are minimum. Construct the tree with the internal node 17 as left child and 25 as right child. Create an internal node with value as 42 . Replace 17 and 25 in the list with this value 42. The list now contains, 42 and q6 = 28.

Huffman tree

- Step 6:

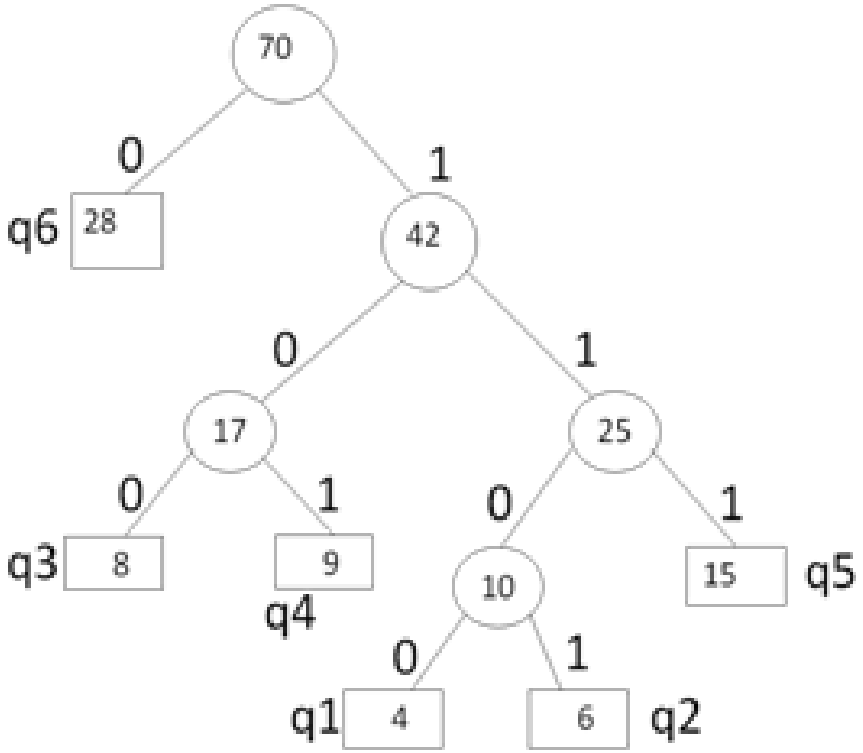
Construct the tree with the internal node 42 as right child and 28 as left child. Create an internal node with value as the sum of the values of left and right child i.e. 70

Consider the next two minimum nodes from the list. The list consists of only one value. The process stops and the final binary tree constructed is the Huffman tree



Huffman tree with codes

- Considering ‘0’ represents the left child and ‘1’ represents the right child of the Huffman tree , the Huffman Code for each of the character is known by traversing the Huffman tree from the root node to the character leaf node. The Huffman codes are q1= 1100, q2 =1101, q3=100, q4= 101 , q5=111, q6= 0.



Algorithm for constructing Huffman tree

- Step 1: Use the given characters in the list and create a leaf node for each character.
- Step 2: Select two nodes with minimum values in the given list of characters.
- Step 3: Construct a binary tree by creating a new internal node with value as the sum of two minimum values from the list. The minimum value of the list is considered as a left child and the next minimum value is considered as a right child. Replace the two nodes selected in the list with the internal node value.
- Step 4: Repeat Steps2 & 3 until the list has only one value.
- Step 5: EXIT

Thank You

ADITYA ENGINEERING COLLEGE (A)

ADVANCED DATA STRUCTURES

By

Dr. A. Vanathi

Associate Professor & HOD

Dept. of Computer Science and Engineering

Aditya Engineering College(A)

Surampalem

Basic Data Structures -Data Collections

- Linear structures
  - Array: Fixed-size
  - Linked-list: Variable-size
  - Stack: Add to top and remove from top
  - Queue: Add to back and remove from front

Non Linear structures

- Tree: A branching structure with no loops
- Graph: A more general branching structure, with less stringent connection conditions than for a tree

Operations performed

- ***Insertion*** *-Adding a new element.*
- ***Traversal*** *-Processing each element in the list*
- ***Searching*** *-Finding location of an element*
- ***Deletion*** *- Removing an element*
- ***Sorting*** *-Arranging elements in some type of order*
- ***Merging*** *-Combining two lists into a single list*

Array Operations and time Complexity

- Insert 3 at first location no. of shifts ?
- Delete 2 no. of shifts ?
- Time Complexity : O(n)

- Linear search O(1) : best O(n) : worst
- Binary Search O(log n) all cases and sorting

Advanced Data Structures

- Hash Tables
- Self- balancing Binary Search Trees,
- AVL Trees
  Red Black Trees
- Multi-way Search Trees:
- B-Trees
- B+ Trees.
-

Comparison of Time complexities

	Insert	Search	Delete
Unsorted list	O(1)	O(n)	O(n)
Unsorted array	O(n)	O(n)	O(n)
Sorted array	O(n)	O(log n)	O(n)
Trees	O(log n)	O(log n)	O(log n)
Array special case known keys $\in \{1, \dots, K\}$	O(1)	O(1)	O(1)

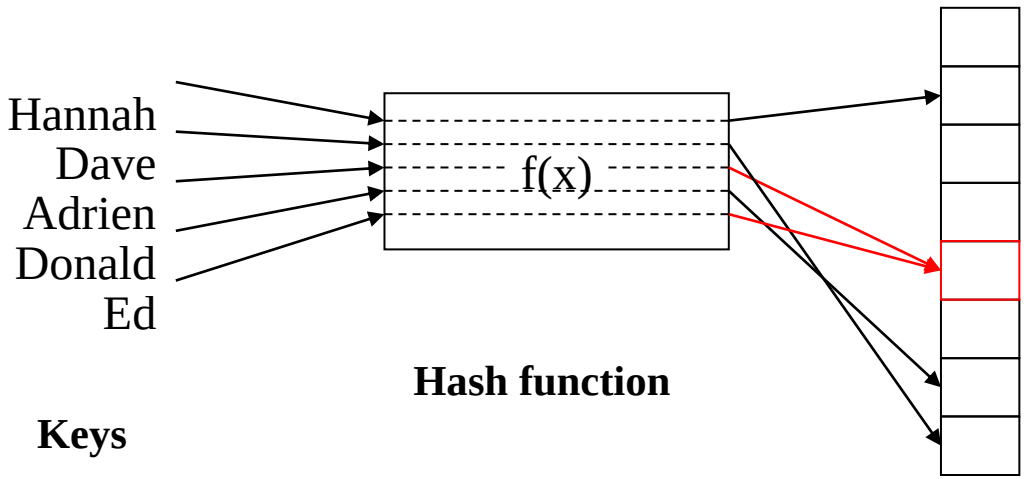
Why not just use an array with **direct addressing** (where each array cell corresponds to a key)?

Direct-addressing guarantees O(1) worst-case time for Insert/Delete/Search.

Hashing

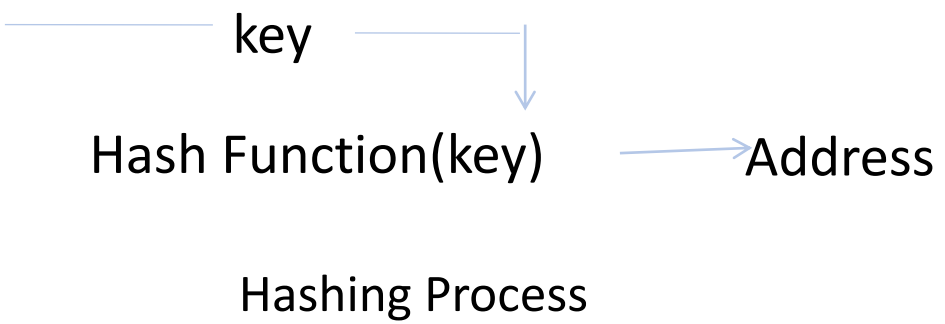
Hashing is a process of finding an address where the data is to be stored as well as located using a key with the help of the algorithmic function

Hashing is a method of directly computing the address of the record with the help of a key by using a suitable mathematical function called the hash function



Hash Table Dictionary Data Structure

Hashing



Hashing Process

Home Address	Data (key)
0	Kumar-36
1	Surya-27
2	Sundar-32
3	
4	Narayana-31
.	
.	
.	
8	
9	

The resulting address is used as the basis for storing and retrieving records and this address is called as *home address of the record*

For array to store a record in a hash table, hash function is applied to the key of the record being stored, returning an index within the range of the hash table

The item is then stored in the table of that index position

Hash function

A function that maps a key into the range [0 to Max − 1], the result of which is used as an index (or address) to hash table for storing and retrieving record

The address generated by hashing function is called as home address

All home addresses address to particular area of memory and that area is called as prime area

- It is easy to compute
- It satisfies uniform hashing

Hash Function

- Hash Function, h is simply a mathematical formula which when applied to the key, produces an integer which can be used as an index for the key in the hash table. The main aim of a hash function is that elements should be relatively randomly and uniformly distributed.

- Hash function produces a unique set of integers within some suitable range. Such function produces no collisions. But practically speaking, there is no hash function that eliminates collision completely. A good hash function can only minimize the number of collisions by spreading the elements uniformly throughout the array.

Properties of Good Hash Function

- Low Cost – executing time must be very small
- Determinism – same hash value must be generated for a given input.
- Uniformity – must map the keys as evenly as possible.

DIFFERENT HASH FUNCTIONS

Assuming numeric keys are being used, the hash functions mostly used are

1) Division method
2) Multiplication method
3) Mid-square method
4) Folding method

Hash Function-Division Method

- Division method is the most simple method of hashing an integer *x*. The method divides *x* by *M* and then use the remainder thus obtained. In this case, the hash function can be given as

  $$h(x) = x \bmod M$$

- The division method is quite good for just about any value of *M* and since it requires only a single division operation, the method works very fast. However, extra care should be taken to select a suitable value for *M*.

Hash Function-Division Method

- For example, *M* is an even number, then *h*(*x*) is even if *x* is even; and *h*(*x*) is odd if *x* is odd. If all possible keys are equi-probable, then this is not a problem. But if even keys are more likely than odd keys, then the division method will not spread hashed values uniformly.

- Generally, it is best to choose *M* to be a prime number because making *M* a prime increases the likelihood that the keys are mapped with a uniformity in the output range of values.

Hash Function-Division Method

- A potential drawback of the division method is that using this method, consecutive keys map to consecutive hash values. While on one hand this is good as it ensures that consecutive keys do not collide, but on the other hand it also means that consecutive array locations will be occupied. This may lead to degradation in performance.

- Example: Calculate hash values of keys 1234 and 5642.

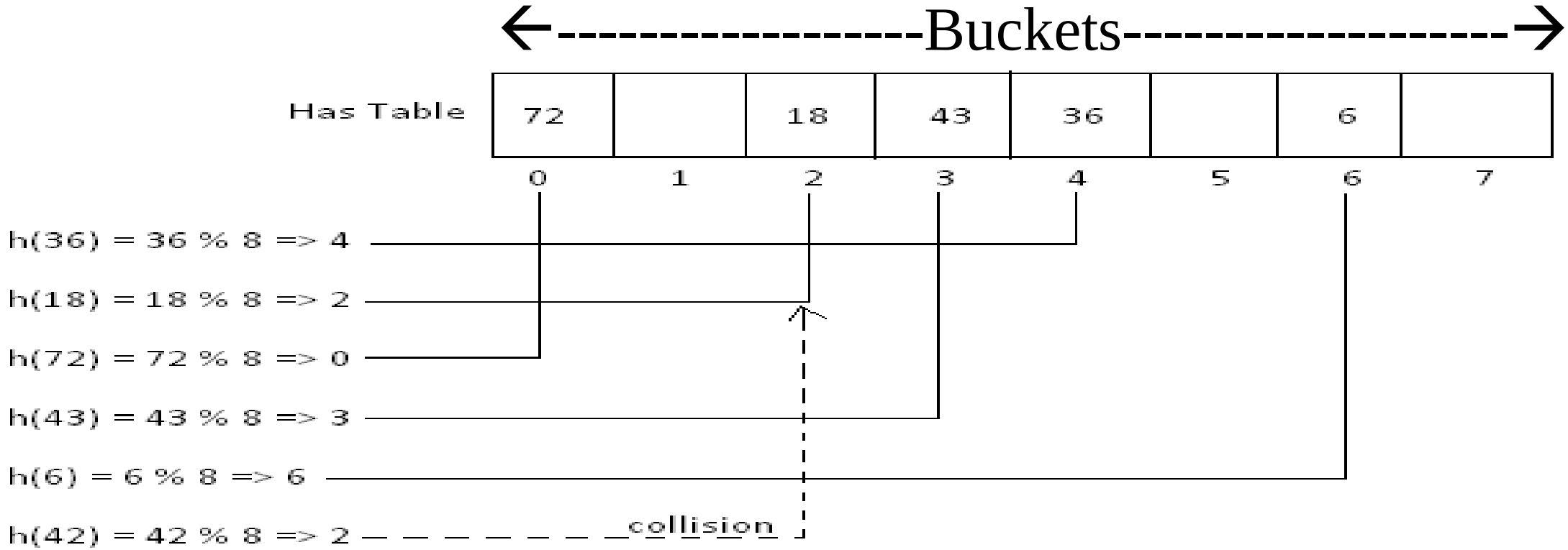
  Setting m = 97, hash values can be calculated as

  h(1234) = 1234 % 97 = 70

  h(5642) = 5642 % 97 = 16

Hash Function-Division Method

Consider a Hash table with 8 slots. i.e. array size 8.
Hash Function (HF) Hashcode = key % table size
The key values given are 36, 18, 72, 43, 6, 42



Hash Function-Multiplication Method

The steps involved in the multiplication method can be given as below:

Step 1: Choose a constant *A* such that *0 < A < 1.*

Step 2: Multiply the key *k* by *A*

Step 3: Extract the fractional part of *kA*

Step 4: Multiply the result of Step 3 by *m* and take the floor.

Hence, the hash function can be given as,

$$h(k) = \lfloor m (k A \bmod 1) \rfloor$$

where, *kA* mod 1 gives the fractional part of *kA* and m is the total number of indices in the hash table

The greatest advantage of the multiplication method is that it works practically with any value of *A*. Although the algorithm works better with some values than the others but the optimal choice depends on the characteristics of the data being hashed. Knuth has suggested that the best choice of A is

» (sqrt5 - 1) /2 = 0.6180339887

Hash Function-Multiplication Method

Example: Given a hash table of size 1000, map the key 12345 to an appropriate location in the hash table

We will use A = 0.618033, m = 1000 and k = 12345

h(12345) = $\lfloor 1000 (12345 \times 0.618033 \bmod 1) \rfloor$

= $\lfloor 1000 (7629.617385 \bmod 1) \rfloor$

= $\lfloor 1000 (0.617385) \rfloor$

= 617.385

= 617

Hash Function-Mid Square Method

Mid square method is a good hash function which works in two steps.

Step 1: Square the value of the key. That is, find k^2

Step 2: Extract the middle *r* bits of the result obtained in Step 1.

The algorithm works well because most or all bits of the key value contribute to the result. This is because all the digits in the original key value contribute to produce the middle two digits of the squared value. Therefore, the result is not dominated by the distribution of the bottom digit or the top digit of the original key value.

In the mid square method, the same *r* bits must be chosen from all the keys.

Therefore, the hash function can be given as,

h (k) = s

where, *s* is obtained by selecting r bits from k^2

Hash Function-Mid Square Method

Example: Calculate the hash value for keys 1234 and 5642 using the mid square method. The hash table has 100 memory locations.

Note the hash table has 100 memory locations whose indices vary from 0-99. this means, only two digits are needed to map the key to a location in the hash table, so *r = 2.*

When k = 1234, k2 = 1522756, h (k) = 27

When k = 5642, k2 = 31832164, h (k) = 21

Observe that 3rd and 4th digits starting from the right are chosen.

Hash Function-Folding Method

- The identifier x is partitioned into several parts, all but the last being of the same length.
- All partitions are added together to obtain the hash address for x.

  Shift folding

  Different partitions are added together to get h(x).

  Folding at the boundaries

  Identifier is folded at the partition boundaries, and digits falling into the same position are added together to obtain h(x). This is similar to reversing every other partition and then adding.

Hash Function-Folding Method

The shift folding method works in two steps.

Step 1: Divide the key value into a number of parts. That is divide k into parts, *k1, k2, …, kn,* where each part has the same number of digits except the last part which may have lesser digits than the other parts.

Step 2: Add the individual parts. That is obtain the sum of *k1 + k2 + .. + kn*. Hash value is produced by ignoring the last carry, if any.

Note that the number of digits in each part of the key will vary depending upon the size of the hash table. For example, if the hash table has a size of 1000. Then it means there are 1000 locations in the hash table. To address these 1000 locations, we will need at least three digits, therefore, each part of the key must have three digits except the last part which may have lesser digits.

Hash Function-Folding Method

Example: Given a hash table of 100 locations, calculate the hash value using shift folding method for keys- 5678, 321 and 34567.

Here, since there are 100 memory locations to address, we will break the key into parts where each part (except the last) will contain two digits.

Therefore,

Key	5678	321	34567
Parts	56 and 78	32 and 1	34, 56 and 7
Sum	134	33	97
Hash Value	34 (ignore the last carry)	33	97

Collision

- If a hash function maps two different keys to same location, then it is called as collision. Obviously, two records can not be stored in the same location.
- Therefore, a method used to solve the problem of collision also called collision resolution technique is applied.

The two most popular method of resolving collision are:

1) Collision resolution by open addressing (Closed Hashing)

2) Collision resolution by chaining -Closed Addressing(Open Hashing)

Collision

Collision Resolution by Open Addressing

- Once a collision takes place, open addressing computes new positions using a probe sequence and the next record is stored in that position.
- In this technique of collision resolution, all the values are stored in the hash table. The hash table will contain two types of values- either sentinel value (for example, -1) or a data value.
- The presence of sentinel value indicates that the location contains no data value at present but can be used to hold a value.

*The process of examining memory locations in the hash table is called **probing**.*

Open addressing technique can be implemented using

1. linear probing
2. quadratic probing and
3. double hashing.

Linear Probing

- Closed hashing indicates that the hashing is limited(closed) to this hash table only and no buckets could be added or linked.

- The simplest approach to resolve a collision is linear probing. In this technique, if a value is already stored at location generated by *h(k),* then the following hash function is used to resolve the collision.

$$h(k, i) = [h'(k) + i] \bmod m$$

- where, *m* is the size of the hash table, $h'(k) = k \bmod m$ and *i* is the probe number and varies from *0* to *m-1*.

Linear Probing

- Example: Consider a hash table with size = 10. Using linear probing insert the keys 72, 27, 36, 24, 63, 81 and 92 into the table.
- ***Let h'(k) = k mod m, m = 10***
- Initially the hash table can be given as,

0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Step1: Key = 72

h(72, 0) = (72 mod 10 + 0) mod 10

= (2) mod 10

= 2

Since, T[2] is vacant, insert key 72 at this location

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	-1	-1	-1	-1

Linear Probing

Step2: Key = 27

h(27, 0) = (27 mod 10 + 0) mod 10

= (7) mod 10

= 7

Since, T[7] is vacant, insert key 27 at this location

Step3: Key = 36

h(36, 0) = (36 mod 10 + 0) mod 10

= (6) mod 10

= 6

Since, T[6] is vacant, insert key 36 at this location

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	36	27	-1	-1

Linear Probing

Step4: Key = 24

h(24, 0) = (24 mod 10 + 0) mod 10

= (4) mod 10

= 4

Since, T[4] is vacant, insert key 24 at this location

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	24	-1	36	27	-1	-1

Step5: Key = 63

h(63, 0) = (63 mod 10 + 0) mod 10

= (3) mod 10

= 3

Since, T[3] is vacant, insert key 63 at this location

0	1	2	3	4	5	6	7	8	9
-1	-1	72	63	24	-1	36	27	-1	-1

Linear Probing

Step6: Key = 81

h(81, 0) = (81 mod 10 + 0) mod 10

= (1) mod 10

= 1

Since, T[1] is vacant, insert key 81 at this location

0	1	2	3	4	5	6	7	8	9
-1	81	72	63	24	-1	36	27	-1	-1

Step7: Key = 92

h(92, 0) = (92 mod 10 + 0) mod 10

= (2) mod 10

= 2

Now, T[2] is occupied, so we cannot store the key 92 in T[2].

Therefore, try again for next location. Thus probe, i = 1, this time.

Key = 92

h(92, 1) = (92 mod 10 + 1) mod 10

= (2 + 1) mod 10

= 3

Linear Probing

Now, T[3] is occupied, so we cannot store the key 92 in T[3]. Therefore, try again for next location. Thus probe, i = 2, this time.

Key = 92

h(92, 2) = (92 mod 10 + 2) mod 10

= (2 + 2) mod 10

= 4

Now, T[4] is occupied, so we cannot store the key 92 in T[4]. Therefore, try again for next location. Thus probe, i = 3, this time.

Key = 92

h(92, 3) = (92 mod 10 + 3) mod 10

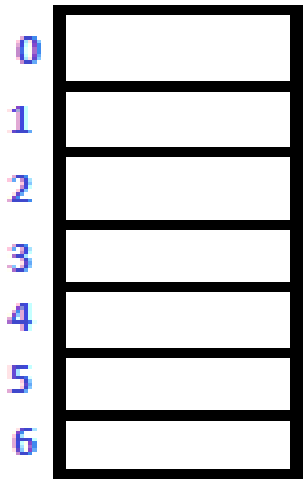
= (2 + 3) mod 10

= 5

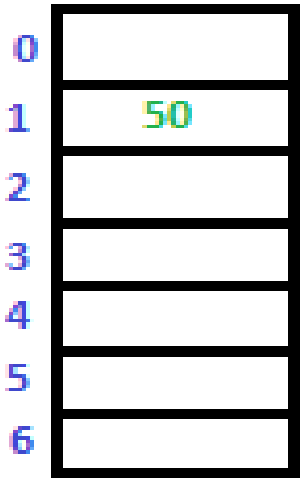
Since, T[5] is vacant, insert key 92 at this location

0	1	2	3	4	5	6	7	8	9
-1	-1	72	63	24	92	36	27	-1	-1

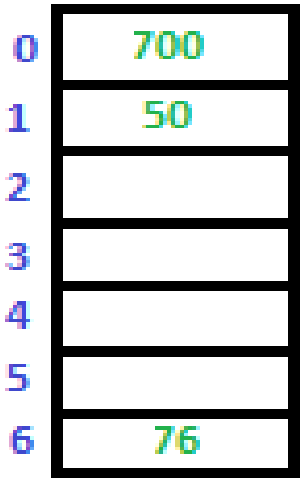
Linear Probing



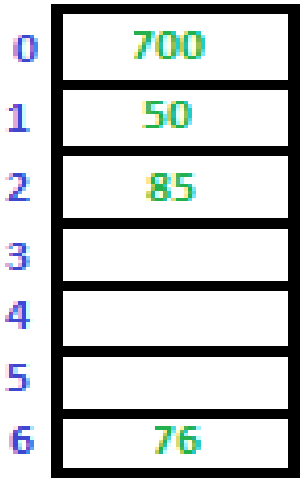
Initial Empty Table



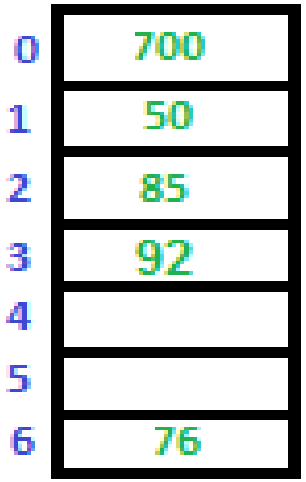
Insert 50



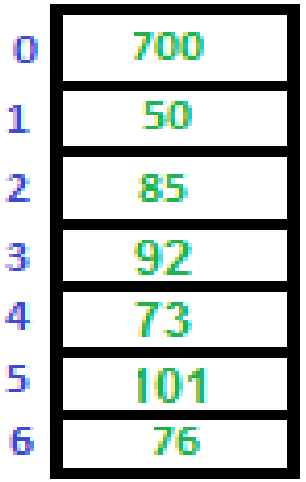
Insert 700 and 76



Insert 85: Collision Occurs, insert 85 at next free slot.



Insert 92, collision occurs as 50 is there at index 1. Insert at next free slot



Insert 73 and 101

Linear Probing

Searching a Value

- When searching a value in the hash table, the array index is re-computed and the key of the element stored at that location is checked with the value that has to be searched.

- If a match is found, then the search operation is successful. The search time in this case is given as O(1). Otherwise, if the key does not match, then the search function begins a sequential search of the array that continues until:

  – the value is found
  – the search function encounters a vacant location in the array, indicating that the value is not present

Linear Probing

Searching a Value

- The search function terminates because the table is full and the value is not present

- In worst case, the search operation may have to make (n-1) comparison, and the running time of the search algorithm may take time given as O(n). The worst case will be encountered when the table is full and after scanning all n-1 elements, the value is either present at the last location or not present in the table.

- Thus, we see that with increase in the number of collisions, the distance from the array index computed by the hash function and the actual location of the element increases, thereby increasing the search time.

Linear Probing

Pros and Cons

- Linear probing finds an empty location by doing a linear search in the array beginning from position *h(k).* Although, the algorithm provides good memory caching, through good locality of reference, but the drawback of this algorithm is that it results in clustering, and thus a higher risk that where there has been one collision there will be more. The performance of linear probing is sensitive to the distribution of input values.
- In linear probing as the hash table fills, clusters of consecutive cells are formed and the time required for a search increases with the size of the cluster. In addition to this, when a new value has to be inserted in to the table at a position which is already occupied, that value is inserted at the end of the cluster, which all the more increases the length of the cluster.

Linear Probing

Pros and Cons

- Generally, an insertion is made between two clusters that are separated by one vacant location. But with linear probing there are more chances that subsequent insertions will also end up in one of the clusters, thereby potentially increasing the cluster length by an amount much greater than one. More the number of collisions, higher the probes that are required to find a free location and lesser is the performance. This phenomenon is called ***primary clustering***. To avoid primary clustering, other techniques like quadratic probing and double hashing are used.

Quadratic Probing

- In this technique, if a value is already stored at location generated by h(k), then the following hash function is used to resolve the collision.

$$h(k, i) = [h'(k) + i^2] \bmod m$$

- where, *m* is the size of the hash table, *h'(k) = k mod m* and *i* is the probe number that varies from *0* to *m-1.*

- Quadratic probing eliminates the primary clustering phenomenon of linear probing because instead of doing a linear search, it does a quadratic search. For a given key *k,* first the location generated by *h'(k)* mod *m* is probed.

- If the location is free, the value is stored in it else, subsequent locations probed are offset by factors that depend in a quadratic manner on the probe number *i.* Quadratic probing performs better than linear probing.

Quadratic Probing

Example: Consider a hash table with size = 10. Using quadratic probing insert the keys 72, 27, 36, 24, 63, 81 and 101 into the table.

Let h'(k) = k mod m, m = 10 (Try to insert 92 after all are inserted)

Initially the hash table can be given as,

-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

We have,

h(k, i) = [h'(k) + i^2] mod m

Step1: Key = 72

h(72) = [72 mod 10 + 0 X 0] mod 10

= [72 mod 10] mod 10

= 2 mod 10

= 2

Since, T[2] is vacant, insert the key 72 in T[2]. The hash table now becomes,

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	-1	-1	-1	-1

Quadratic Probing

Step2: Key = 27

h(27) = [27 mod 10 + 0 X 0] mod 10

= [27 mod 10] mod 10

= 7 mod 10

= 7

Since, T[7] is vacant, insert the key 27 in T[7]. The hash table now becomes,

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	-1	27	-1	-1

Step3: Key = 36

h(36) = [36 mod 10 + 0 X 0] mod 10

= [36 mod 10] mod 10

= 6 mod 10

= 6

Since, T[6] is vacant, insert the key 36 in T[6]. The hash table now becomes,

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	36	27	-1	-1

Quadratic Probing

Step4: Key = 24

h(24) = [24 mod 10 + 0 X 0] mod 10

= [24 mod 10] mod 10

= 4 mod 10

= 4

Since, T[4] is vacant, insert the key 24 in T[4]. The hash table now becomes,

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	24	-1	36	27	-1	-1

Step5: Key = 63

h(63) = [63 mod 10 + 0X 0] mod 10

= [63 mod 10] mod 10

= 3 mod 10

= 3

Since, T[3] is vacant, insert the key 63 in T[3]. The hash table now becomes,

0	1	2	3	4	5	6	7	8	9
-1	-1	72	63	24	-1	36	27	-1	-1

Quadratic Probing

Step6: Key = 81

h(81) = [81 mod 10 + 0 X 0] mod 10

= [81 mod 10] mod 10

= 81 mod 10

= 1

Since, T[1] is vacant, insert the key 81 in T[1]. The hash table now becomes,

0	1	2	3	4	5	6	7	8	9
-1	81	72	63	24	-1	36	27	-1	-1

Quadratic Probing

Step7: Key = 101

h(101) = [101 mod 10 + 0 X 0] mod 10

= [101 mod 10 + 0] mod 10

= 1 mod 10

= 1

Since, T[1] is already occupied, the key 101 can not be stored in T[1]. Therefore, try again for next location. Thus probe, i = 1, this time.

Key = 101

h(101) = [101 mod 10 + 1X1] mod 10

= [101 mod 10 + 1] mod 10

= [1 + 1] mod 10

= 2 mod 10

= 2

Since, T[2] is already occupied, the key 101 can not be stored in T[2]. Therefore, try again for next location.

0	1	2	3	4	5	6	7	8	9
-1	81	72	63	24	-1	36	27	-1	-1

Quadratic Probing

Thus probe, i = 2, this time.

Key = 101

h(101) = [101 mod 10 + 2X2] mod 10

= [101 mod 10 + 4] mod 10

= [1 + 4] mod 10

= 5mod 10

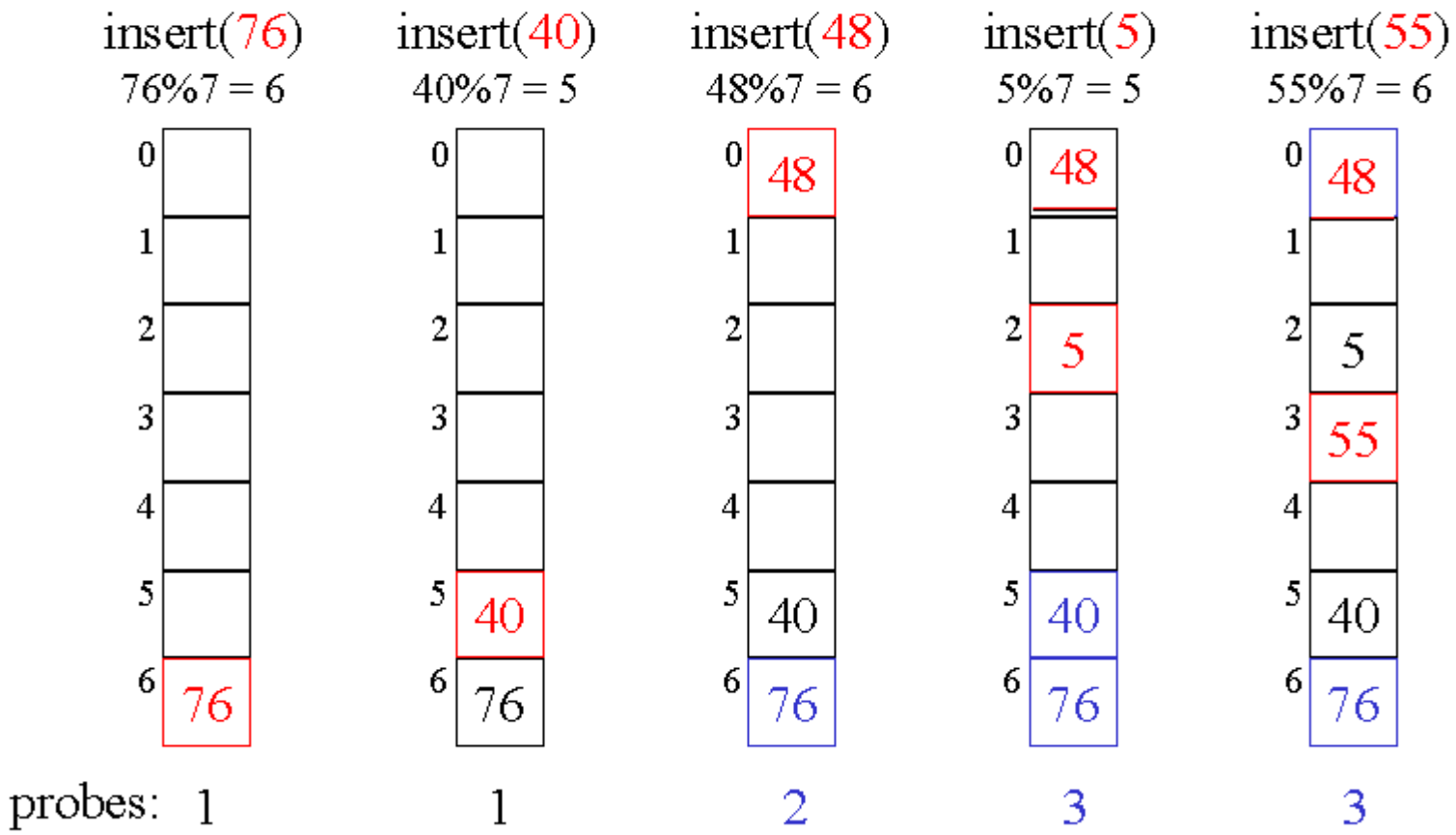
= 5

Since, T[5] is vacant, insert the key 101 in T[5]. The hash table now becomes,

0	1	2	3	4	5	6	7	8	9
-1	81	72	63	24	101	36	27	-1	-1

Quadratic Probing

Quadratic Probing Example ☺



Quadratic Probing

Searching a Value

While searching for a value using quadratic probing technique, the array index is re-computed and the key of the element stored at that location is checked with the value that has to be searched. If the desired key value matches the key value at that location, then the element is present in the hash table and the search is said to be successful. In this case the search time is given as *O(1).* However, if the value does not match then, the search function begins a search of the array that continues until:

- the value is found
- the search function encounters a vacant location in the array, indicating that the value is not present
- the search function terminates because the table is full and the value is not present

Quadratic Probing

Cons and Pros

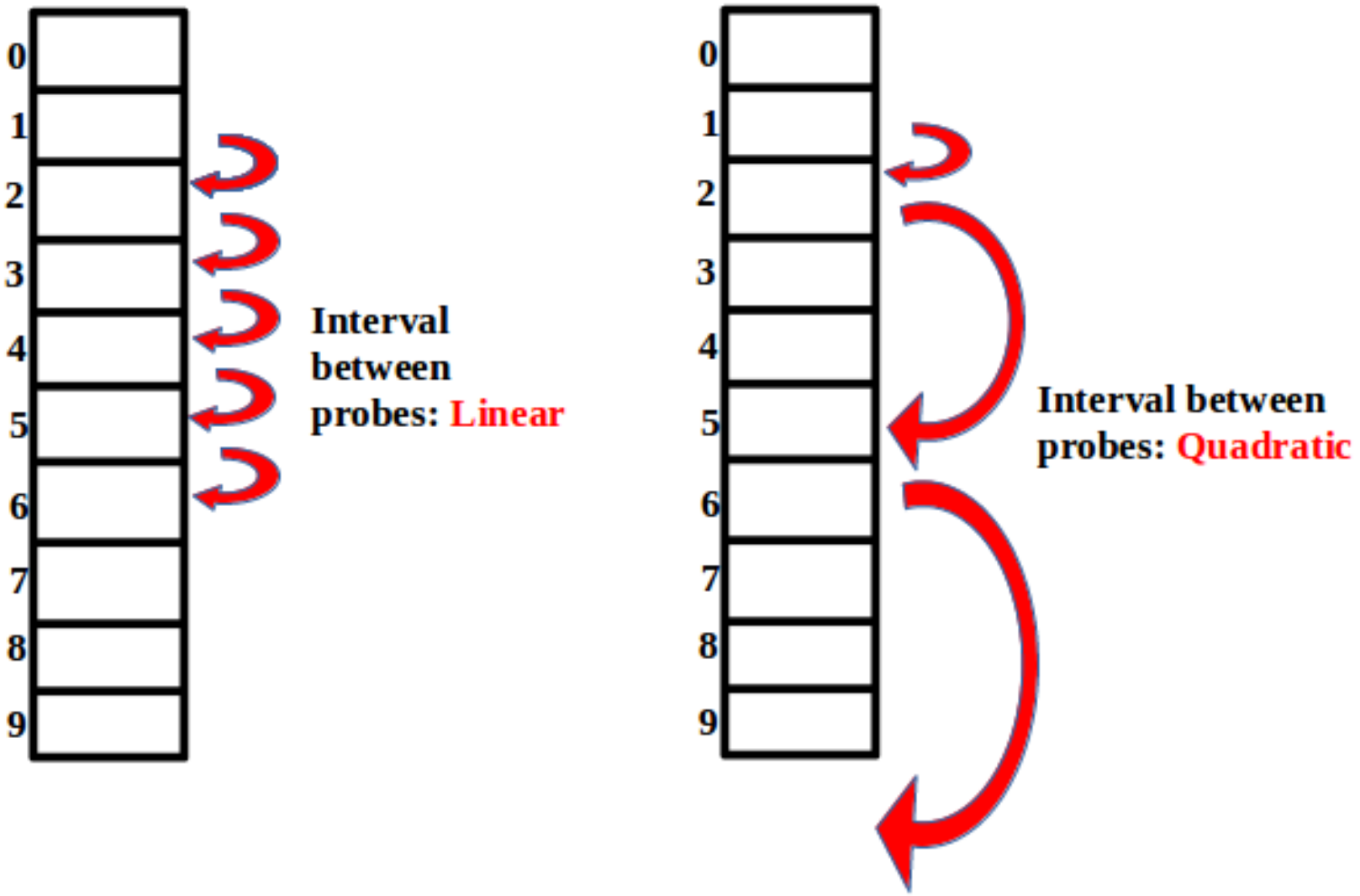
- Quadratic probing caters to the primary clustering problem that exists in linear probing technique. Quadratic probing provides good memory caching because it preserves some locality of reference. But linear probing does this task better and gives better cache performance.

- One of the major drawbacks with quadratic probing is that a sequence of successive probes may only explore a fraction of the table, and this fraction may be quite small. If this happens then we will not be able to find an empty location in the table despite the fact that the table is by no means full.

- Although quadratic probing is free from primary clustering, but it is still liable to what is known as *secondary clustering*. This means that if there is a collision between two keys then the same probe sequence will be followed for both. (Try to insert key 92 and you will see how this happens). With quadratic probing, potential for multiple collisions increases as the table becomes full. This situation is usually encountered when the hash table is more than full.

- Quadratic probing is widely applied in the Berkeley Fast File System to allocate free blocks.

Quadratic Probing



Double Hashing

- To start with double hashing uses one hash value and then repeatedly steps forward an interval until an empty location is reached. The interval is decided using a second, independent hash function, hence the name double hashing. Therefore, in double hashing we use two hash functions rather a single function. The hash function in case of double hashing can be given as,

  h(k, i) = [h1(k) + i*h2(k)] mod m

- where, m is the size of the hash table, h1(k) and h2(k) are two hash functions given as, h1(k) = k mod m, h2(k) = k mod m', i is the probe number that varies from 0 to m-1 and m' is chosen to be less than m. we can choose m' = m-1 or m-2.

Double Hashing

- When we have to insert a key k in the hash table, we first probe the location given by applying h1(k) mod m because during the first probe, i = 0.

- If the location is vacant the key is inserted into it, else subsequent probes generate locations that are at an offset of h2(k) mod m from the previous location.

- Since the offset may vary with every probe depending on the value generated by second hash function, the performance of double hashing is very close to the performance of the ideal scheme of uniform hashing.

Double Hashing

Double hashing minimizes repeated collisions and the effects of clustering. That is, double hashing is free from problems associated with primary clustering as well secondary clustering.

Example: Consider a hash table with size = 11. Using double hashing insert the keys 72, 27, 36, 24, 63, 81, 92 and 101 into the table. Take h1 = k mod 11 and h2 = k mod 7. ***Let m = 11*** Initially the hash table can be given as,

0	1	2	3	4	5	6	7	8	9	10
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

We have,

h(k, i) = [h1(k) + ih2(k)] mod m

Step1: Key = 72

h(72, 0) = [72 mod 11 + (0 X 72 mod 7] mod 11

= [6+ (0 X 2)] mod 11

= 6 mod 11 = 6

Since, T[6] is vacant, insert the key 72 in T[6]. The hash table now becomes,

0	1	2	3	4	5	6	7	8	9	10
-1	-1	-1	-1	-1	-1	72	-1	-1	-1	-1

Double Hashing

Step2: Key = 27

h(27, 0) = [27 mod 11+ (0 X 27 mod 7)] mod 11

= [5 + (0)] mod 11

= 5 mod 11

= 5

Since, T[5] is vacant, insert the key 27 in T[5]. The hash table now becomes,

0	1	2	3	4	5	6	7	8	9	10
-1	-1	-1	-1	-1	27	72	-1	-1	-1	-1

Step3: Key = 36

h(36, 0) = [36 mod 11 + (0 X 36 mod 7)] mod 11

= [3 + (0)] mod 11

= 3 mod 11

= 3

Since, T[3] is vacant, insert the key 36 in T[3]. The hash table now becomes,

0	1	2	3	4	5	6	7	8	9	10
-1	-1	-1	36	-1	27	72	-1	-1	-1	-1

Double Hashing

Step4 Key = 24

h(24, 0) = [24 mod 11+ (0 X 27 mod 7)] mod 11

= [2 + (0)] mod 11

= 2 mod 11

= 2

Since, T[2] is vacant, insert the key 27 in T[2]. The hash table now becomes,

0	1	2	3	4	5	6	7	8	9	10
-1	-1	24	36	-1	27	72	-1	-1	-1	-1

Step5: Key = 63

h(63, 0) = [63 mod 11 + (0 X 63 mod 7)] mod 11

= [8 + (0)] mod 11

= 8 mod 11

= 8

Since, T[8} is vacant, insert the key 63 in T[8]. The hash table now becomes,

0	1	2	3	4	5	6	7	8	9	10
-1	-1	-1	36	-1	27	72	-1	63	-1	-1

Double Hashing

Step 6: Key = 81

h(81, 0) = [81 mod 11 + (0 X 81 mod 7)] mod 11

= [4 + (0)] mod 11

= 4 mod 11

= 4

Since, T[4] is vacant, insert the key 81 in T[4]. The hash table now becomes,

0	1	2	3	4	5	6	7	8	9	10
-1	-1	-1	36	81	27	72	-1	63	-1	-1

Step 7: Key = 92

h(92, 0) = [92 mod 11 + (0 X 92 mod 7)] mod 11

= [4 + (0 X 1)] mod 11

= (4 + 0) mod 11

= 4 mod 11

= 4

Now, T[4] is occupied, so we cannot store the key 92 in T[4]. Therefore, try again for next location

Double Hashing

Thus probe, i = 1, this time.

Key = 92

h(92, 1) = [92 mod 11 + (1 X 92 mod 7)] mod 11
= [4 + (1 X 1)] mod 11
= (4 + 1) mod 11
= 5 mod 11
= 5

Now, T[5] is occupied, so we cannot store the key 92 in T[5]. Therefore, try again for next location. Thus probe, i = 2, this time.

Key = 92

h(92, 2) = [92 mod 11 + (2 X 92 mod 7)] mod 11
= [4 + (2 X 1)] mod 11
= (4 + 2) mod 11
= 6 mod 11
= 6

Now, T[6] is occupied, so we cannot store the key 92 in T[6]. Therefore, try again for next location.

0	1	2	3	4	5	6	7	8	9	10
-1	-1	-1	36	81	27	72	-1	63	-1	-1

Double Hashing

Thus probe, i = 3, this time.

Key = 92

h(92, 3) = [92 mod 11 + (3 X 92 mod 7)] mod 11

= [4 + (3 X 1)] mod 11

= (4 + 3) mod 11

= 7 mod 11

= 7

Since, T[7] is vacant, insert the key 92 in T[7]. The hash table now becomes,

0	1	2	3	4	5	6	7	8	9	10
-1	-1	-1	36	81	27	72	92	63	-1	-1

Rehashing

- When the hash table becomes nearly full, the number of collisions increases, thereby degrading the performance of insertion and search operations.
- A better option is to create a new hash table with size double of the original hash table.
- All the entries in the original hash table will then have to be moved to the new hash table by taking each entry, computing its new hash value, and then inserting it in the new hash table.
- Though rehashing seems to be a simple process, it is quite expensive and must therefore not be done frequently.

Rehashing

- Consider the hash table of size 5 given below. The hash function used is h(x) = x % 5. Insert 26,31,43,17.

0	1	2	3	4
-1	26	31	43	17

- Rehash the entries into to a new hash table as the hash table is almost full.
- Note that the new hash table is of 10 locations, double the size of the original table.
- Now, rehash the key values from the old hash table into the new one using hash function h(x) = x % 10.

0	1	2	3	4	5	6	7	8	9
-1	31	-1	43	-1	-1	26	17	-1	-1

Closed Addressing(Open hashing)

- Closed addressing is one of the collision resolution technique, in which we cannot store keys at everywhere in the hash table, other than hash value.
- If keys are generating same hash code, then collision occurs. In that case we store all keys in linked list, in the same level
- There is only one technique, that is **separate chaining,** that resolve the collisions.

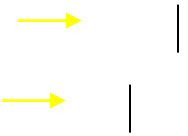
Separate Chaining

- In chaining, each location in the hash table stores a pointer to a linked list that contains the all key values that were hashed to the same location.
- That is, location l in the hash table points to the head of the linked list of all the key values that hashed to l.
- However, if no key value hashes to l, then location l in the hash table contains NULL.
- Figure shows how the key values are mapped to a location l in the hash table and stored in a linked list that corresponds to l.

Separate Chaining

Example: Insert the keys 7, 24, 18, and 52 in a chained hash table of 9 memory locations. Use h(k) = k mod m

0	NULL
1	NULL
2	NULL
3	NULL
4	NULL
5	NULL
6	NULL
7	NULL
8	NULL

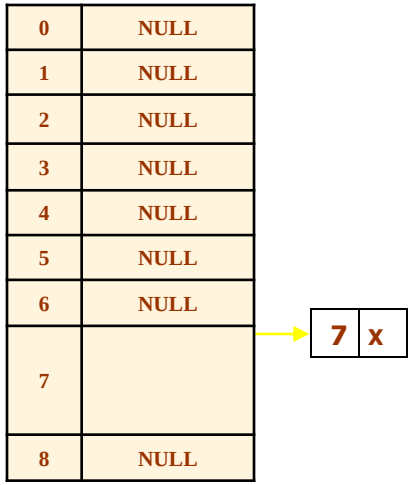


Separate Chaining

Example: Insert the keys 7, 24, 18, and 52 in a chained hash table of 9 memory locations. Use h(k) = k mod m

In this case, m=9. Initially, the hash table can be given as

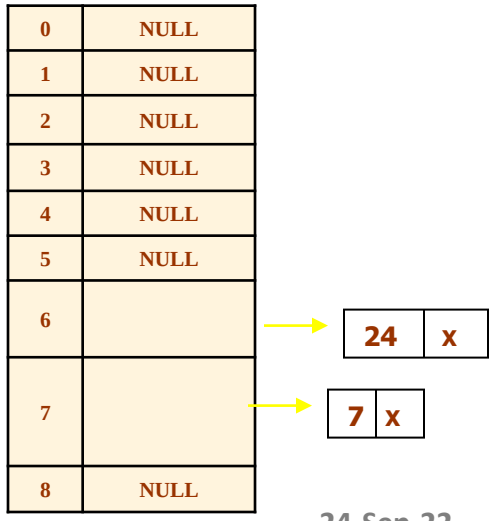
Step 1: Key = 7



Step 2: Key = 24

h(k) = 24 mod 9

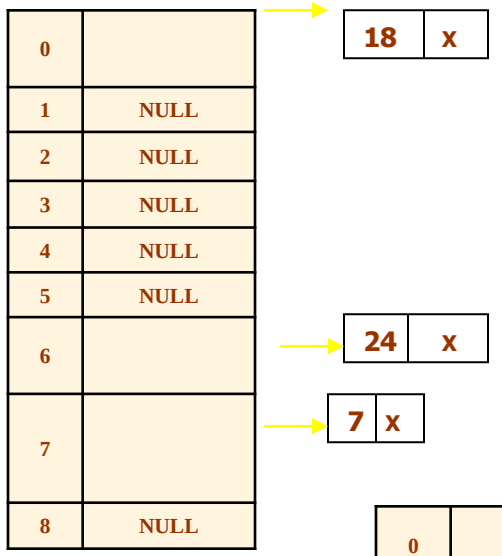
= 6



Separate Chaining

Step 3: Key = 18

h(k) = 18 mod 9 = 0

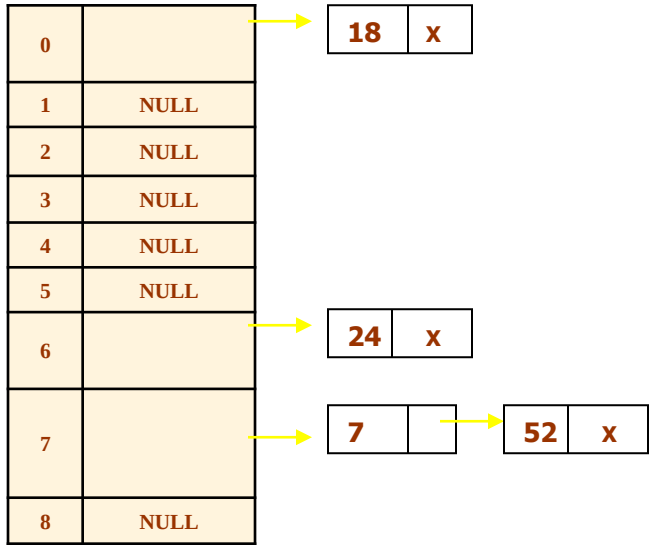


Step 4: Key = 52

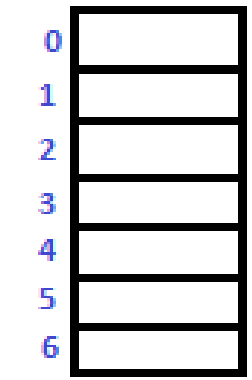
h(k) = 52 mod 9

= 7

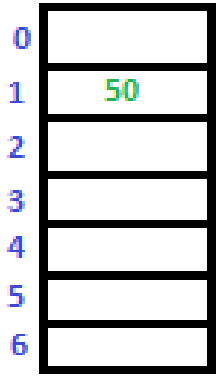
Insert 52 in the beginning of the linked list of location 7



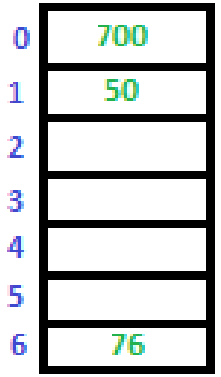
Separate Chaining



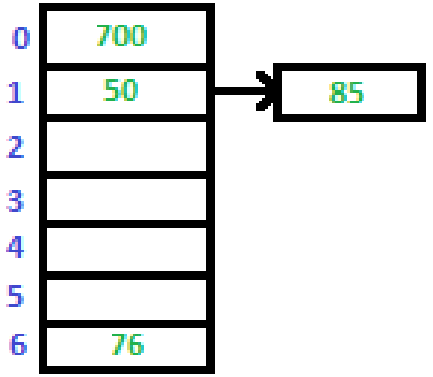
Initial Empty Table



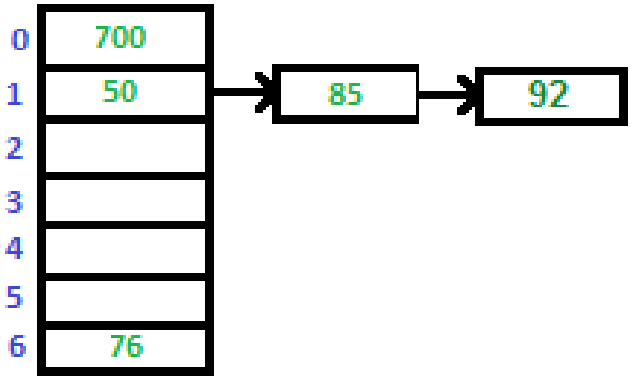
Insert 50



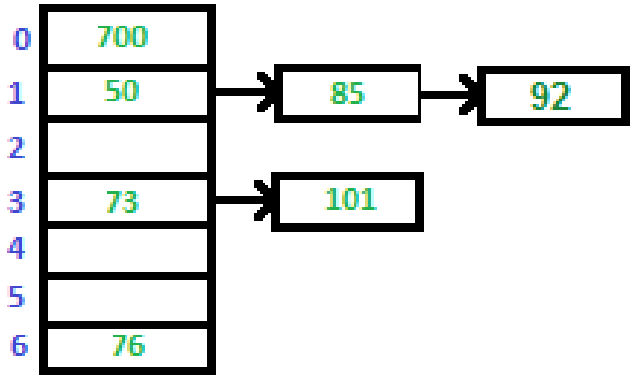
Insert 700 and 76



Insert 85: Collision
Occurs, add to chain



Inser 92 Collision
Occurs, add to chain



Insert 73 and 101

Separate Chaining

Advantages:

- Simple to implement.
- Hash table never fills up, we can always add more elements to the chain.
- Less sensitive to the hash function
- It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

Disadvantages:

- Cache performance of chaining is not good as keys are stored using a linked list. Open addressing provides better cache performance as everything is stored in the same table.
- Wastage of Space (Some Parts of hash table are never used
- If the chain becomes long, then search time can become O(n) in the worst case.
- Uses extra space for links.

Differences between open hashing and closed hashing techniques

Open Hashing	Closed Hashing
1. It is also known as closed addressing.	1. It is also known as open addressing.
2. Keys are stored in linked lists attached to cells of a hash table.	2. Keys are stored directly at an index in the hash table. It does not uses any linked list.
3. Each index maintains a list of keys that are to be stored at the same index and the keys are linked to each other.	3. The index at which the key is actually stored may vary depending on the keys already stored in the hash table.
4. For example, if two keys hash to the same index, then they both are inserted in the list maintained at the same index entry.	4. For example, if element 12 is hashed to index 5 and if the index already has another value stored in it, then 12 will be inserted at any other free location.
5. Efficient for large number of records.	5. Efficient for small number of records.