Module 5-Hashing

Module 5 Syllabus	**HASHING**: Introduction, Static Hashing, Dynamic Hashing **PRIORITY QUEUES**: Single and double ended Priority Queues, Leftist Trees **INTRODUCTION TO EFFICIENT BINARY SEARCH TREES**: Optimal Binary Search Trees

5.0 Hashing

- Hashing enables us to perform the dictionary operations such as search, insert and deleting expected time.
- In a mathematical sense, a map is a relation between two sets. We can define Map M as a set of pairs, where each pair is of the form (key, value), where for given a key, we can find a value using some kind of a "function" that maps keys to values.
- The key for a given object can be calculated using a function called a hash function.
- Hashing technique 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.

5.0.1 Hash Function

- Is a function which is used to put the data in hash table. The integer is returned by the hash function is called hash key.
- A good hash function should satisfy 2 criteria
    1. A hash function should hash address such that keys are distributed as evenly as possible among the various cells of the hash table.
    2. Computation of key should be simple.

5.0.2 Hash Table

- Hash table is a data structure used for storing and retrieving data very quickly.
- Insertion, Deletion or Retrieval operation takes place with help of hash value.
- Hence every entry in the hash table is associated with some key.
- Using the hash key the required piece of data can be searched in the hash table by few or more key comparisons. The searching time is dependent upon the size of the hash table.

VTUSYNC.IN

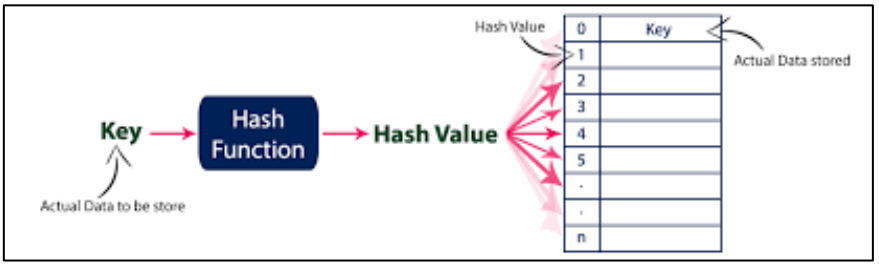


Figure 5.17 Hashing.

5.0.3 Types of Hash Functions

1. **Division Method:** It is the most simple method of hashing an integer x. This method divides x by M and then uses the remainder obtained. In this case, the hash function can be given as h(x) = x mod 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. Generally, it is best to choose M to be a prime number because making M a prime number increases the likelihood that the keys are mapped with a uniformity in the output range of values.

Example: If the record to be stored {54,72, 89,37} is to be placed in the hash table and if the table size is 10

H(Key) = Record % Size i.e:
H (54) =54%10=4
H (72) =72%10=2
H (89) =89%10=9
H (37) =37%10=7

0	
1	
2	72
3	
4	54
5	
6	
7	37
8	
9	89

2. **Mid-Square Method:**

- Here, the key K is squared. A number 'l' in the middle of K^2 is selected by removing the digits from both ends. H(k)=l

VTUSYNC.IN

- Example 1:

  Solution: Let key=2345, Its square is K^2 =574525

  H (2345) =45=>by discarding 57 and 25

- Example 2: Calculate the hash value for keys 1234 using the mid-square method. The hash table has 100 memory locations.

  Solution: Note that the hash table has 100 memory locations whose indices vary from 0 to 99. This means that only two digits are needed to map the key to a location in the hash table.

  When k = 1234, k^2 = 1522756, h (1234) = 27.

3. Folding Method:

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.

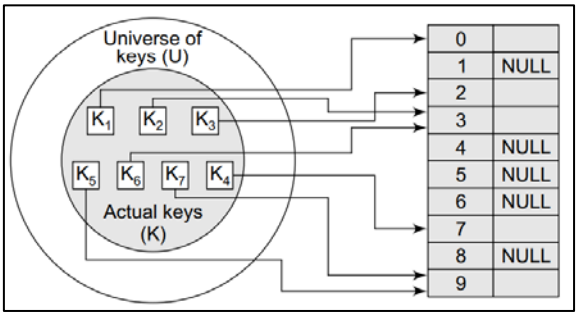
Step 2: Add the individual parts. That is, obtain the sum of k1 + k2 + ... + kn. The hash value is produced by ignoring the last carry, if any.

Note that the number of digits in each part of the key will vary depending upon the size of the hash table.

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

5.0.4 Collision Resolution Techniques

Figure 10.5 shows a hash table in which each key from the set K is mapped to locations generated by using a hash function. Note that keys k2 and k6 point to the same memory location. This is known as collision. That is, when two or more keys map to the same memory location, a collision Figure 5.18 shows a hash table in which each key from the set K is mapped to locations generated by using a hash function. Note that keys k2 and k6 point to the same memory location. This is known as collision. That is, when two or more keys map to the same memory location, a collision.



VTUSYNC.IN

Figure 5.18 Collision Example

A method used to solve the problem of collision, also called collision resolution technique, is applied. The two most popular methods of resolving collisions are:

1. Collision Resolution by Linear Probing (open addressing)

Suppose new record R with key K is to be added to the memory table T, but that memory with H(k) = h is already filled, one way of avoiding collision is to design R to 1st available location following T[h]. We assume that T with m location is circular so T[1] comes after T[m]. according to procedure, we search for record R in table T by linearly searching location T[h], T[h+1]… until we meet an empty location or finding R.

Example: Consider a hash table of size 10. Using linear probing, insert the keys 72, 27, 36, 24, 63, 81, 92, and 101 into the table.

Solution: Let H(k) = k mod m, m = 10

Initially hash table will be

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

H(72) = 72 mod 10 = 2

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

H(27) = 27 mod 10 = 7

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

H(36) = 36 mod 10 = 6

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

H(24) = 24 mod 10 = 4

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

H(63) = 63 mod 10 =3

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

H(81) = 81 mod 10 =1

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

VTUSYNC.IN

H(92) = 92 mod 10 =2

Collision occurred since 2 is already filled. So go to next position – 3, which is also already filled, go to next position – 4 which is also already filled. So go to 5 – which is not filled – so insert the key 92 in position 5.

0	1	2	3	4	5	6	7	8	9
-1	**81**	**72**	**63**	**24**	**92**	**36**	**27**	-1	-1

H(101) = 101 mod 10 = 1

Collision occurred since 1 is already filled. Do linear probing and the next position free is 8, so insert key 101 in position 8.

0	1	2	3	4	5	6	7	8	9
-1	**81**	**72**	**63**	**24**	**92**	**36**	**27**	**101**	-1

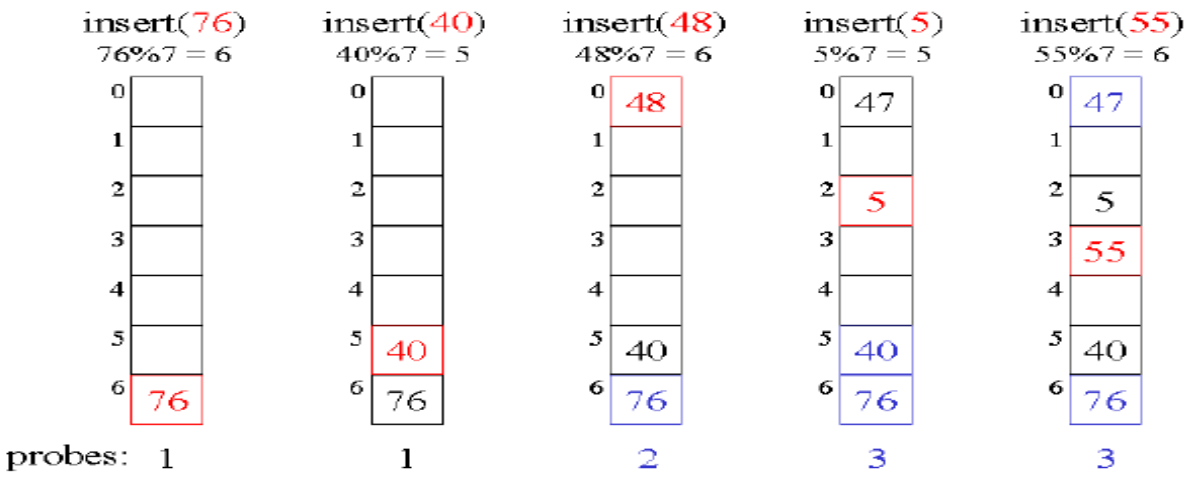
Key	***Home address***	***Actual Address***	***Search length***
		0	
81	1	1	1
72	2	2	1
63	3	3	1
24	4	4	1
92	2	5	4
36	6	6	1
27	7	7	1
101	1	8	8
		9	

Average search length = (1+1+1+1+4+1+1+8)/ 8 = 2.25

2. Quadratic Probing

A variation of the linear probing idea is called **quadratic probing**. Instead of using a constant "skip" value, if the first hash value that has resulted in collision is *h*, the successive values which are probed are *h*+1, *h*+4, *h*+9, *h*+16, and so on. In other words, quadratic probing uses a skip consisting of successive perfect squares.

VTUSYNC.IN



Probe sequence :h,h+1^2,h=2^2,h=3^2.....h+i^2

H(k)=(h+i$^{2)}$%Tablesize

3. Double Hashing

In double hashing, we use two hash functions rather than a single function. Double hashing uses the idea of applying a second hash function to the key when a collision occurs. The result of the second hash function will be the number of positions form the point of collision to insert.There are a couple of requirements for the second function:

- it must never evaluate to 0
- must make sure that all cells can be probed
- A popular second hash function is: Hash2(key) = M - (key % M) where M is a prime number that is smaller than the size of the table. But any independent hash function may also be used.

Example: 37,90,45,22,17,49,55

H_1(key)=key%10

H_2(key)=7-(key%7)

After collision

(H_1(key)+1*H_2(key))%10

90
17
22
45
55

VTUSYNC.IN

37
49

4. Rehashing

When the hash table becomes nearly full, the number of collisions increases, thereby degrading the performance of insertion and search operations. In such cases, 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.

This is done 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.

Consider the hash table of size 5 given below. The hash function used is h(x) = x % 5. Rehash the entries into to a new hash table.

0	1	2	3	4
	26	31	43	17

Note that the new hash table is of 10 locations, double the size of the original table.

0	1	2	3	4	5	6	7	8	9

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
	31		43			26	17		

5. Chaining

- Chaining technique avoids collision using an array of liked lists (run time).If more than one key has same hash value, then all the keys will be inserted at the end of the list (insert rear) one by one and thus collision is avoided.
- Example: Construct a hash table of size and store the following words: **like, a, tree, you, first, a, place, to**
- Let H(str)=P0+P1+P2+……+Pn-1; where Pi is position of letter in English alphabet series.
- Then calculate the **hash address = Sum % 5**

VTUSYNC.IN

H(like) = 12 + 9 + 11 + 5 = 37 % 5 = 2

H(a) = 1 %5 =1

H(tree) = 20 + 18 + 5 + 5 = 48 % 5 = 3

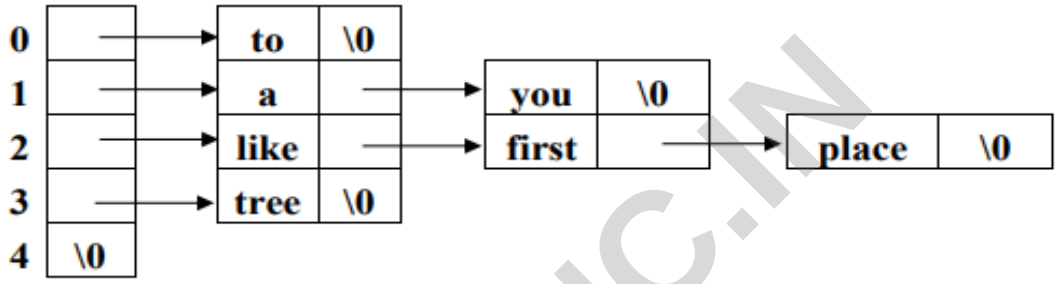
H(you) = 25 + 15 + 21 = 61 % 5 = 1

H(first) = 6 + 9 + 18 + 19 + 20 = 72 % 5 = 2

H(a) = 1 % 5 = 1

H(place) = 16 + 12 + 1 + 3 + 5 = 37 % 5 =2

H(to) = 20 + 15 = 35 % 5 = 0



5.0.5 Types of Hashing

1. Static Hashing:

- Is a hashing technique in which the table(bucket) size remains the same (Fixed during compilation time) is called static hashing.
- Various techniques of static hashing are linear probing and chaining
- As the size is fixed, this type of hashing consist handling overflow of elements (Collision) efficiently.

Example:
Elements to be stored: 24, 93, 45
H (24) = 24 % 10 = 4
H (93) = 93 % 10 = 3
H (45) = 45 % 10 = 5

0	
1	
2	
3	93
4	24
5	45
6	
7	
8	
9	

Drawbacks of static hashing

1. Table size is fixed and hence cannot accommodate data growth.

VTUSYNC.IN

2. Collisions increases as data size grows.

2. Dynamic Hashing:

Dynamically increases the size of the hash table as collision occurs. There are two types:

1) Dynamic hashing using directory or (Extendible hashing) : uses a directory that grows or shrinks depending on the data distribution. No overflow buckets

2) Directory less Dynamic hashing or (Linear hashing): No directory. Splits buckets in linear order, uses overflow buckets.

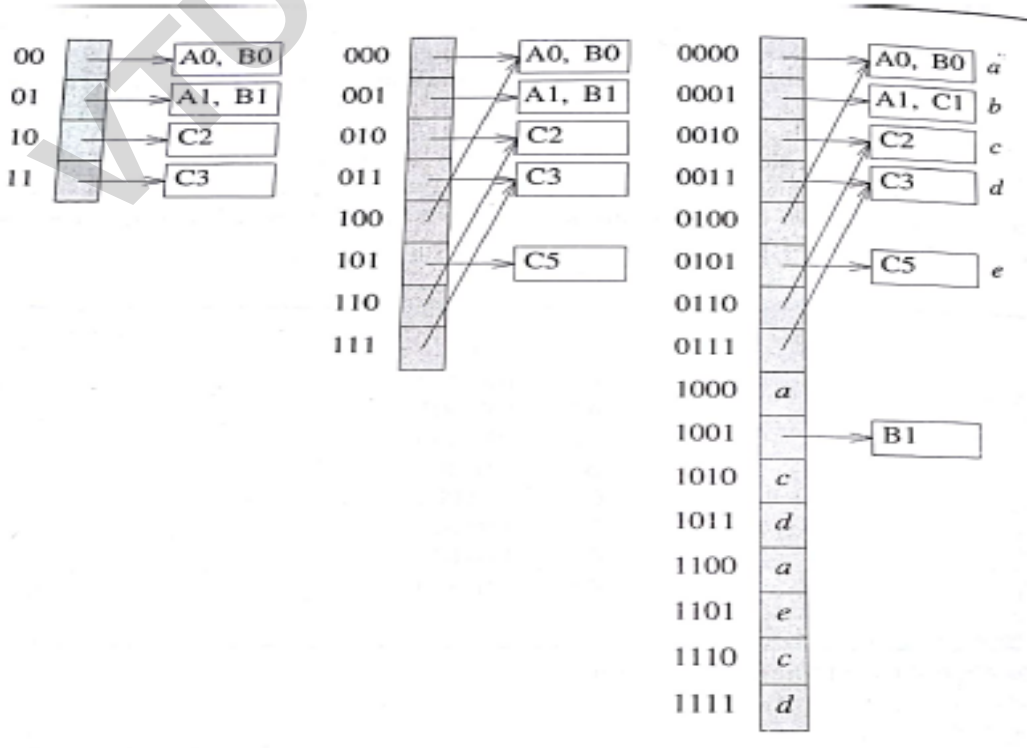
Dynamic hashing using directory

• Uses a directory of pointers to buckets/bins which are collections of records

• The number of buckets are doubled by doubling the directory, and splitting just the bin that overflowed.

• Directory much smaller than file, so doubling it is much cheaper.

k	$h(k)$
A0	100 000
A1	100 001
B0	101 000
B1	101 001
C1	110 001
C2	110 010
C3	110 011
C5	110 101



(a) depth = 2 (b) depth = 3 (c) depth = 4

VTUSYNC.IN

Explain directoryless dynamic hashing (Appeared in Dec. 2016/Jan 2017)- 5 Marks

Directory less Dynamic hashing

Basic Idea:

• Pages are split when overflows occur – but not necessarily the page with the overflow.

• Directory avoided in Linear hashing by using overflow pages. (chaining approach)

• Splitting occurs in turn, in a round robin fashion.one by one from the first bucket to the last bucket.

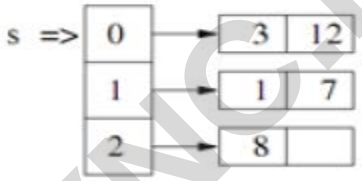
• Use a family of hash functions **h**0, **h**1, **h**2, ...

- Each function's range is twice that of its predecessor.

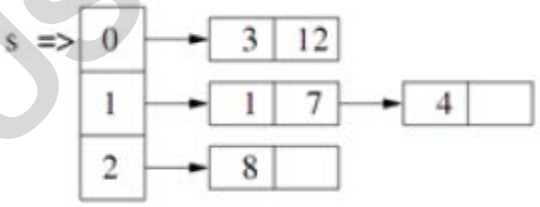
• When all the pages at one level (the current hash function) have been split, a new level is applied.

• Insert in Order using linear hashing: 1,7,3,8,12,4,11,2,10

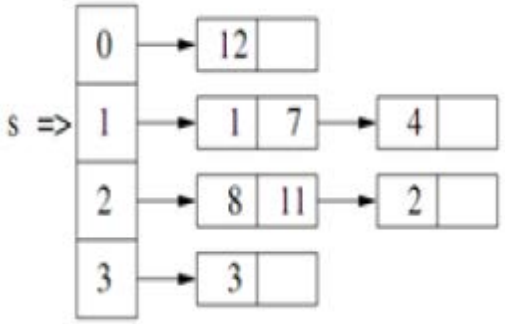
• After insertion till 12:



• When 4 inserted overflow occurred. So we split the bucket (no matter it is full or partially empty). And increment pointer.

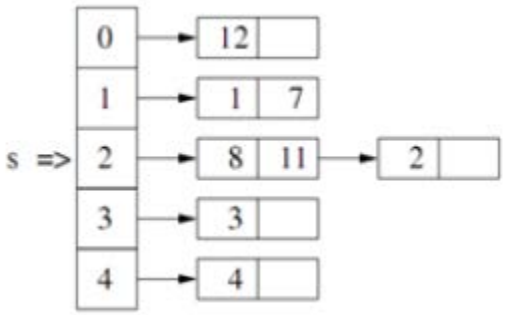


• So we split bucket 0 and rehashed all keys in it. Placed 3 to new bucket as h1 (3 mod 6 = 3) and (12 mod 6 = 0). Then 11 and 2 are inserted. And now overflow. *s* is pointing to bucket 1, hence split bucket 1 by re- hashing it.

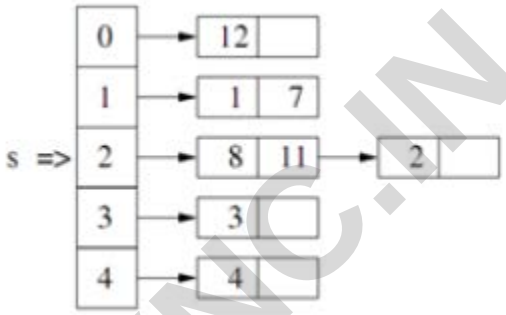


VTUSYNC.IN

After split:



Insertion of 10 : as (10 mod 3 = 1) and bucket 1 < s, we need to hash 10 again using h1(10) = 10 mod 6 = 4th bucket.



VTUSYNC.IN

5.0.6 Priority Queues

1 Single and Double- ended priority queues

The single ended priority queues may be categorized as min and max priority queues.The operations supported by a min priority queue are:

SP1: Return an element with minimum priority.

SP2: Insert an element with an arbitrary priority.

SP3: Delete an element with minimum priority.

The operations supported by a max priority queue are the same as those supported by a min priority queue except that in SP1 and SP3 we replace minimum by maximum. The heap structure is a classical data structure for the representation of a priority queue. Using a min (max) heap, the minimum (maximum) element can be found in O(1) time. Meldable (single- ended) priority queue, augments the operations SP1 through SP3 with a meld operation that melds together two priority queues.

A double-ended priority queue (DEPQ) is a data structure that supports the following opertions on a collection of elements.

DP1: Return an element with minimum priority.

DP2: Return an element with maximum priority.

DP3: Insert an element with an arbitrary priority.

DP4: Delete an element with minimum priority.

DP5: Delete an element with maximum priority.

Example: A DEPQ may be used to implement a network buffer. This buffer holds packets that are waiting their turn to be sent out over a network link; each packet has an associated priority. When the network link becomes available, a packet with the highest priority is transmitted. This corresponds to a DeleteMax operation. When a packet arrives at the buffer from elsewhere in the network, it is added to this buffer. This corresponds to an Insert operation. If the buffer is full, we must drop a packet with the minimum priority before we can insert one. This is achieved using a DeleteMin operation.

5.0.7 Leftlist Trees

Let n be the total number of elements in the two priority queues that are to be combined. If heaps are used to represent priority queues, then the combine operation takes O(n) time. Using a leftist tree, the combine operation as well as the normal priority queue operations take logarithmic time.

Leftist trees, are defined using the concept of an extended binary tree. An extended binary tree is a binary tree in which all empty binary sub trees have been replaced by a square node. The square nodes in an extended binary tree are called external nodes. The original (circular) nodes of the binary tree are called internal nodes

Let X be a node in an extended binary tree. Let left-child (x) and right-child (x), respectively, denote the left and right children of the internal node x. Define shortest (x) to be the length of a shortest path from x to an external node. It is easy to see that shortest (x) satisfies the following recurrence:

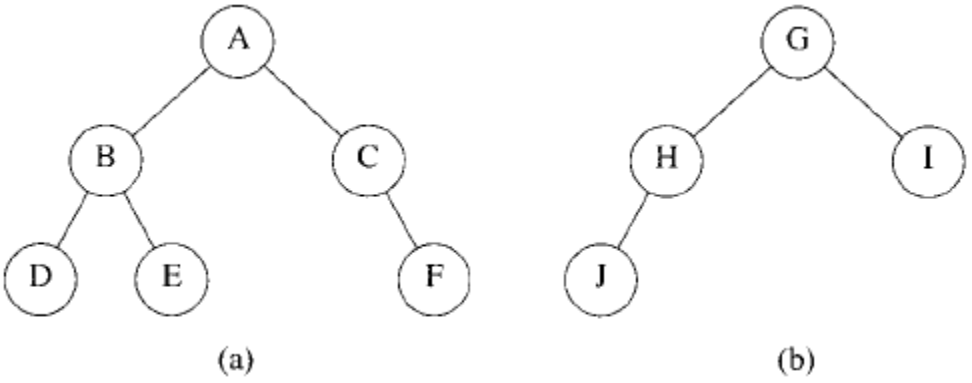


Figure : Two Binary Trees

VTUSYNC.IN

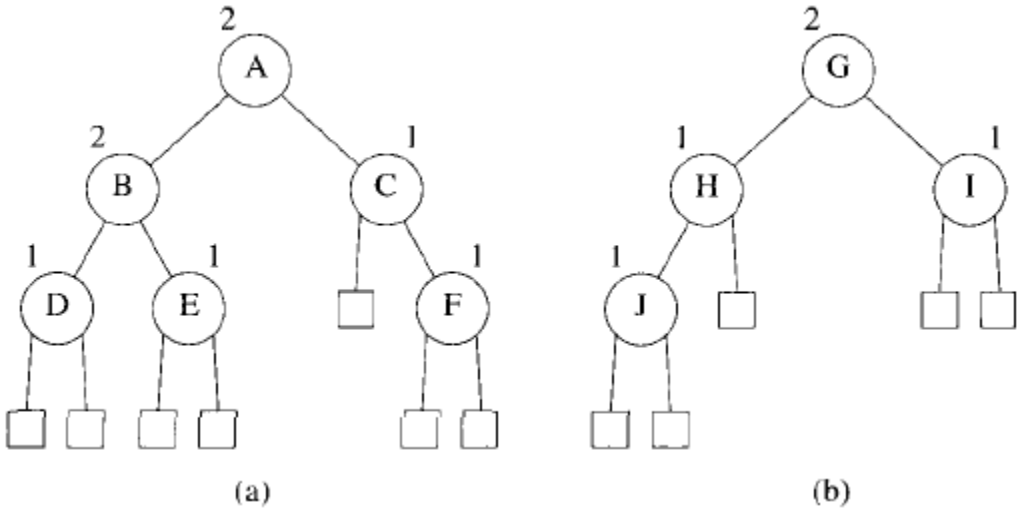


Figure : Extended binary trees corresponding to above binary Trees

1) Height-Biased Leftist Trees.

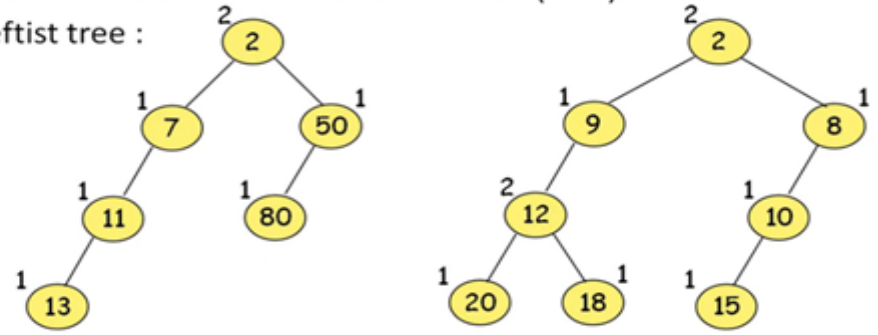
$$shortest(x) = \begin{cases} 0 \text{ if } x \text{ is an external node} \\ 1 + \min\{shortest(left_child(x)), shortest(right_child(x))\} \text{ otherwise} \end{cases}$$

Definition: A *leftist tree* is a binary tree such that if it is not empty, then

$$shortest(left_child(x)) \geq shortest(right_child(x))$$

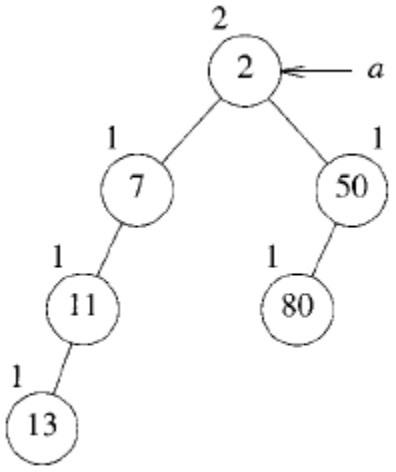
for every internal node x.

- Definition of A Min (Max) Leftist Tree
  - A **min leftist tree** is a leftist tree in which the key value in each node is smaller than the key value in its children (if any).
  - A **max leftist tree** is a leftist tree in which the key value in each node is greater than the key value in its children (if any).
  - A min(max) leftist tree is a leftist tree that is also a min(max) tree
  - Example of min leftist tree :

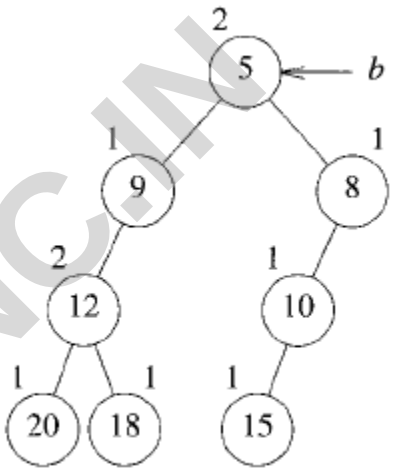


VTUSYNC.IN

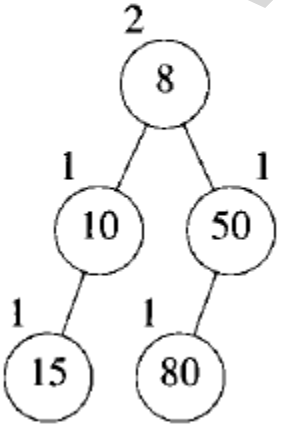
- The operations on the min (max) leftist trees are insert, delete, and meld(combine).
- The **insert** and **delete-min** operations can both be done by using the combine operation.
  - To **insert** an element x into a min leftist tree, we first create a min leftist tree that contains the single element x. Then we combine the two min leftist trees.
  - To **delete** the min element from a nonempty min leftist tree, we combine the min leftist trees root->LeftChild and root->RightChild and delete the node root.
- To **meld**(combine) two min-leftist trees
  1. Choose minimum root of the two trees, A and B.
  2. Leave the left subtree of smaller root (suppose A) unchanged and combine the right subtree of A with B. Back to step 1, until no remaining vertices.
  3. Compare shortest(x) and swap to make it satisfy the definition of leftist trees.



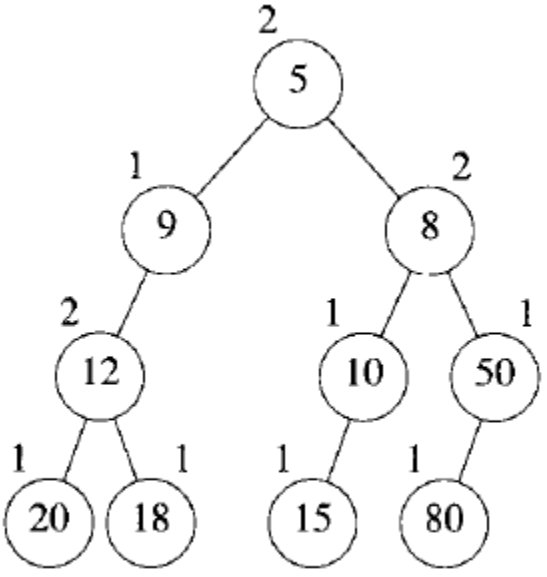
(a)



(b)

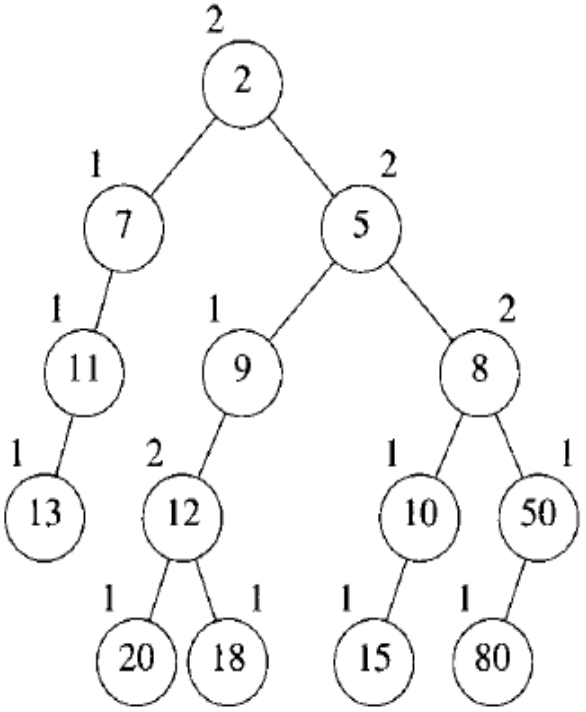


(a)

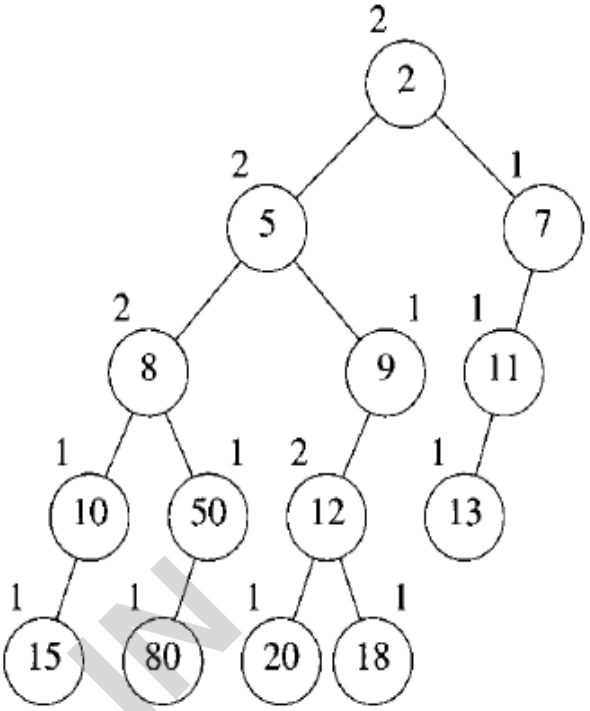


(b)

VTUSYNC.IN



(c)



(d)

```
void min_combine(leftist_tree *a, leftist_tree *b)
{
/* combine the two min leftist trees *a and *b.  The
resulting min leftist tree is returned in *a, and *b
is set to NULL */
   if (!*a)
      *a = *b;
   else if (*b)
      min_union(a,b);
   *b = NULL;
}
```

VTUSYNC.IN

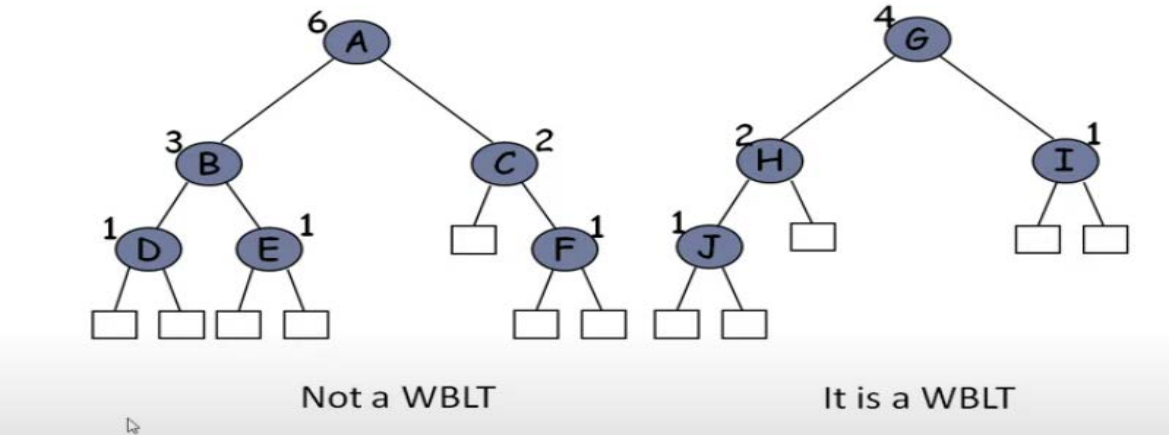
```
void min_union(leftist_tree *a, leftist_tree *b)
{
/* recursively combine two nonempty min leftist trees */
   leftist_tree temp;
   /* set a to be the tree with smaller root */
   if ((*a)->data.key > (*b)->data.key)
      SWAP(*a,*b,temp);
   /* create binary tree such that the smallest key in each
   subtree is in the root */
   if (!(*a)->right_child)
      (*a)->right_child = *b;
   else
      min_union(&(*a)->right_child, b);
   /*leftist tree property */
   if (!(*a)->left_child) {
      (*a)->left_child = (*a)->right_child;
      (*a)->right_child = NULL ;
   }
   else if ((*a)->left_child->shortest <
   (*a)->right_child->shortest)
      SWAP((*a)->left_child,(*a)->right_child, temp);
   (*a)->shortest = (!(*a)->right_child) ? 1 :
   (*a)->right_child->shortest + 1;
}
```

- The operations on the min (max) leftist trees are insert, delete, and meld(combine).
- The **insert** and **delete-min** operations can both be done by using the combine operation.
  - To **insert** an element x into a min leftist tree, we first create a min leftist tree that contains the single element x. Then we combine the two min leftist trees.
  - To **delete** the min element from a nonempty min leftist tree, we combine the min leftist trees root->LeftChild and root->RightChild and delete the node root.
- To **meld**(combine) two min-leftist trees
  1. Choose minimum root of the two trees, A and B.
  2. Leave the left subtree of smaller root (suppose A) unchanged and combine the right subtree of A with B. Back to step 1, until no remaining vertices.
  3. Compare shortest(x) and swap to make it satisfy the definition of leftist trees.

VTUSYNC.IN

2 Weight-Biased Leftist Trees.

- Here we consider number of nodes in the subtree rather than the length of a shortest root to external node path
- **w(x)** be the weight of a node x
- $w(x) = \begin{cases} 0 & \text{if } x \text{ is an external node} \\ 1 + \text{sum of the weights of the children} & \text{otherwise} \end{cases}$
- Definition : A binary tree is a Weight Biased Leftist Tree (WBLT) iff at every internal node the w value of the left child is greater than or equal to the w value of the right child.
- A max(min) WBLT is a max(min) tree that is also a WBLT

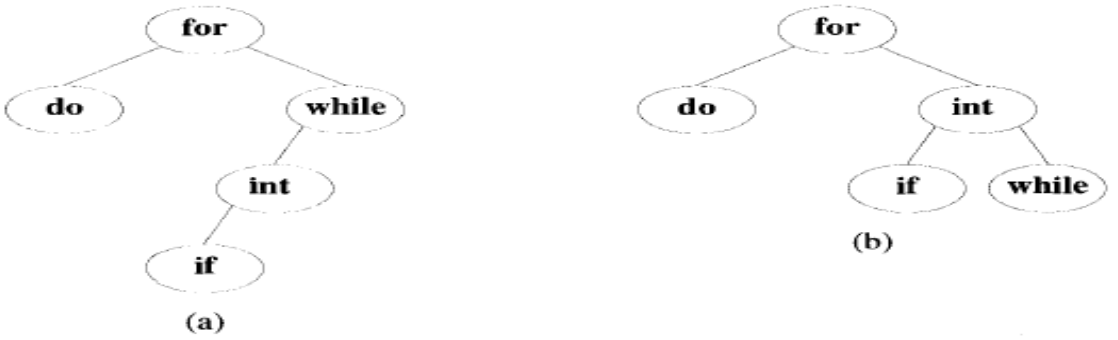


- Here we consider number of nodes in the subtree rather than the length of a shortest root to external node path
- **w(x)** be the weight of a node x
- $w(x) = \begin{cases} 0 & \text{if } x \text{ is an external node} \\ 1 + \text{sum of the weights of the children} & \text{otherwise} \end{cases}$
- Definition : A binary tree is a Weight Biased Leftist Tree (WBLT) iff at every internal node the w value of the left child is greater than or equal to the w value of the right child.
- A max(min) WBLT is a max(min) tree that is also a WBLT
- **Insert and delete operations are analogous to HBLT operations**
- **The meld operation can be done in a single top-to-bottom pass. On the way down, we can update w-values and swap subtrees as necessary.**
- WBLT is faster than HBLT

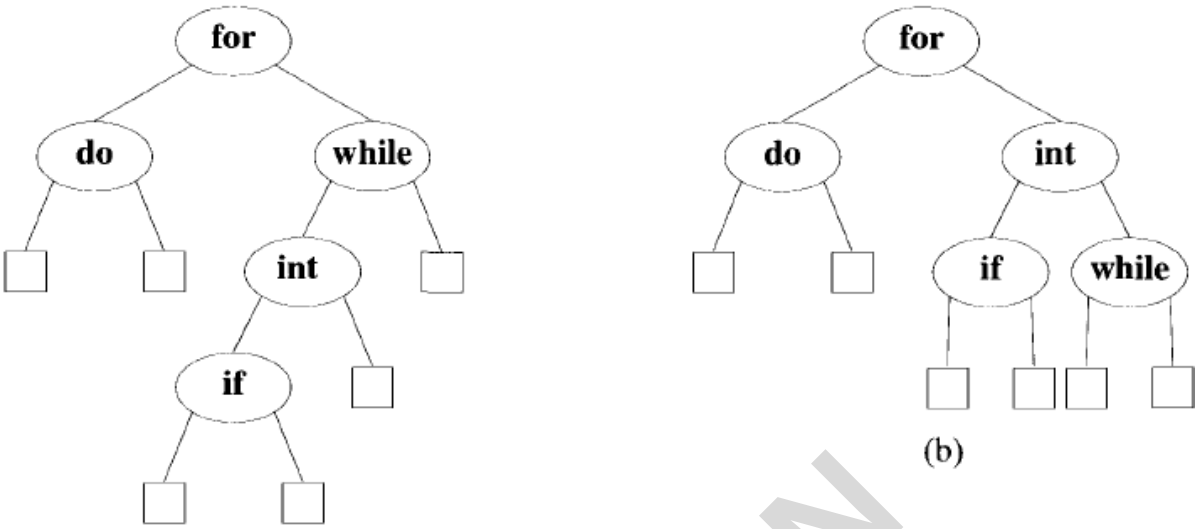
VTUSYNC.IN

5.0.8 Optimal Binary Search Tree:

In computer science, an optimal binary search tree (Optimal BST), sometimes called a weight-balanced binary tree, is a binary search tree which provides the smallest possible search time (or expected search time) for a given sequence of accesses (or access probabilities).

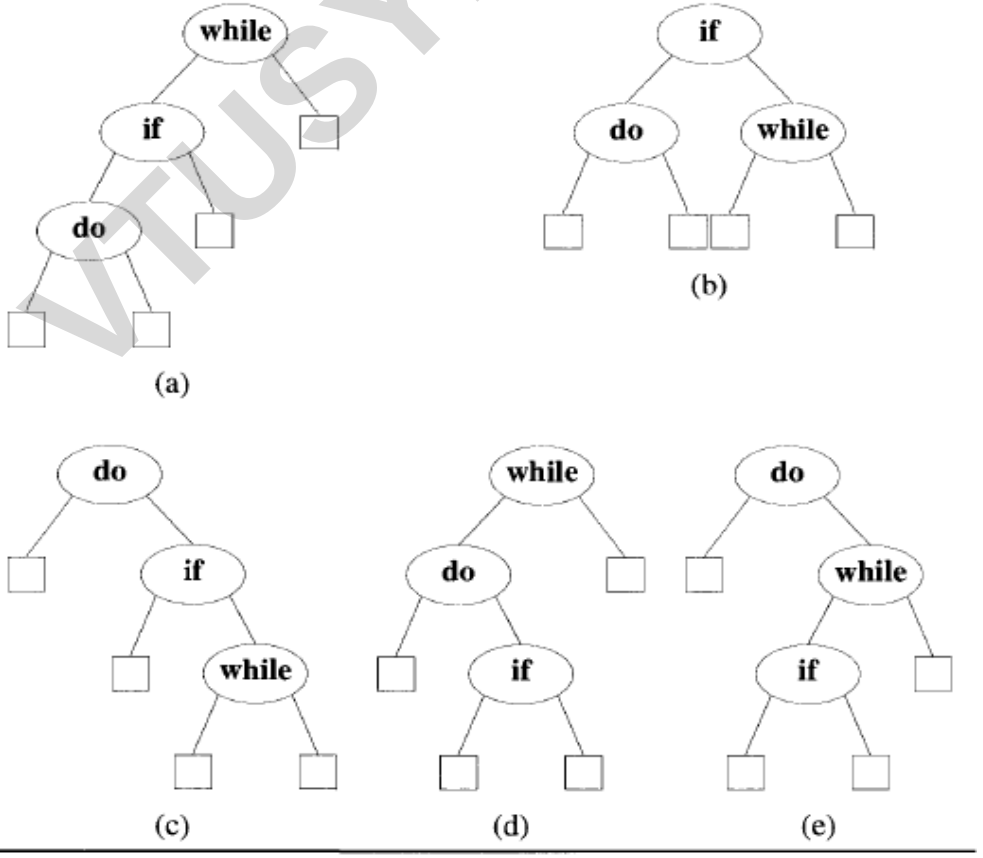


Two possible binary search trees



Binary search trees of Figure with external nodes added

The no of external nodes are same in both trees.



Possible binary search trees for the identifier set {**do**, **if**, **while**}

VTUSYNC.IN

The C (i, J) can be computed as:

$$C(i, J) = \min_{i<k\leq J} \{C(i, k-1) + C(k, J) + P(K) + w(i, K-1) + w(K,J)\}$$

$$= \min_{i<k\leq J} \{C(i, K-1) + C(K, J)\} + w(i, J) \quad -- \quad (1)$$

Where $W(i, J) = P(J) + Q(J) + w(i, J-1)$ -- (2)

Initially $C(i, i) = 0$ and $w(i, i) = Q(i)$ for $0 \leq i \leq n$.

C (i, J) is the cost of the optimal binary search tree 'T_{ij}' during computation we record the root R (i, J) of each tree 'T_{ij}'. Then an optimal binary search tree may be constructed from these R (i, J). R (i, J) is the value of 'K' that minimizes equation (1).

We solve the problem by knowing W (i, i+1), C (i, i+1) and R (i, i+1), $0 \leq i < 4$; Knowing W (i, i+2), C (i, i+2) and R (i, i+2), $0 \leq i < 3$ and repeating until W (0, n), C (0, n) and R (0, n) are obtained.

Example Let $n = 4$ and $(a_1, a_2, a_3, a_4) =$ **(do, if, int, while)**. Let $p(1:4) = (3, 3, 1, 1)$ and $q(0:4) = (2, 3, 1, 1, 1)$. The p's and q's have been multiplied by 16 for convenience. Initially, we have $w(i, i) = q(i), c(i, i) = 0$ and $r(i, i) = 0, 0 \leq i \leq 4$. the observation $w(i, j) = p(j) + q(j) + w(i, j-1)$, we get

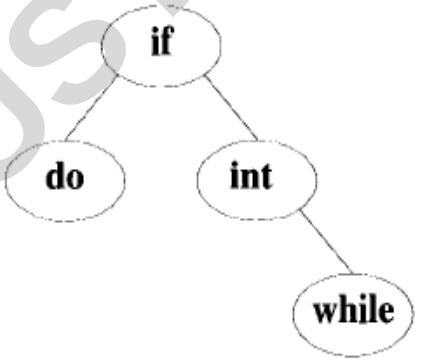
$$\begin{aligned}
w(0,1) &= p(1) + q(1) + w(0,0) = 8 \\
c(0,1) &= w(0,1) + \min\{c(0,0) + c(1,1)\} = 8 \\
r(0,1) &= 1 \\
w(1,2) &= p(2) + q(2) + w(1,1) = 7 \\
c(1,2) &= w(1,2) + \min\{c(1,1) + c(2,2)\} = 7 \\
r(0,2) &= 2 \\
w(2,3) &= p(3) + q(3) + w(2,2) = 3 \\
c(2,3) &= w(2,3) + \min\{c(2,2) + c(3,3)\} = 3 \\
r(2,3) &= 3 \\
w(3,4) &= p(4) + q(4) + w(3,3) = 3 \\
c(3,4) &= w(3,4) + \min\{c(3,3) + c(4,4)\} = 3 \\
r(3,4) &= 4
\end{aligned}$$

Knowing $w(i, i+1)$ and $c(i, i+1)$, $0 \leq i < 4$, we can again use Equation 5.12 to compute $w(i, i+2)$, $c(i, i+2)$, and $r(i, i+2)$, $0 \leq i < 3$. This process can be repeated until $w(0, 4)$, $c(0, 4)$, and $r(0, 4)$ are obtained.

VTUSYNC.IN

	0	1	2	3	4
0	$w_{00} = 2$ $c_{00} = 0$ $r_{00} = 0$	$w_{11} = 3$ $c_{11} = 0$ $r_{11} = 0$	$w_{22} = 1$ $c_{22} = 0$ $r_{22} = 0$	$w_{33} = 1$ $c_{33} = 0$ $r_{33} = 0$	$w_{44} = 1$ $c_{44} = 0$ $r_{44} = 0$
1	$w_{01} = 8$ $c_{01} = 8$ $r_{01} = 1$	$w_{12} = 7$ $c_{12} = 7$ $r_{12} = 2$	$w_{23} = 3$ $c_{23} = 3$ $r_{23} = 3$	$w_{34} = 3$ $c_{34} = 3$ $r_{34} = 4$	
2	$w_{02} = 12$ $c_{02} = 19$ $r_{02} = 1$	$w_{13} = 9$ $c_{13} = 12$ $r_{13} = 2$	$w_{24} = 5$ $c_{24} = 8$ $r_{24} = 3$		
3	$w_{03} = 14$ $c_{03} = 25$ $r_{03} = 2$	$w_{14} = 11$ $c_{14} = 19$ $r_{14} = 2$			
4	$w_{04} = 16$ $c_{04} = 32$ $r_{04} = 2$				

Computation of $c(0,4)$, $w(0,4)$, and $r(0,4)$



Optimal search tree for Example

VTUSYNC.IN

Program : Finding an optimal binary search tree

```
void obst(double *p, double *q, int n)
{
    int i, j, k, m;
    for (i = 0; i < n; i++) {/* initialize */
        /* 0-node trees */
        w[i][i] = q[i]; r[i][i] = c[i][i] = 0;
        /* one-node trees */
        w[i][i+1] = q[i] + q[i+1] + p[i+1];
        r[i][i+1] = i + 1;
        c[i][i+1] = w[i][i+1];
    }
    w[n][n] = q[n]; r[n][n] = c[n][n] = 0;

    /* find optimal trees with m > 1 nodes */
    for (m = 2; m <= n; m++)
        for (i = 0; i <= n- m; i++)
        {
            j = i + m;
            w[i][j] = w[i][j-1] + p[j] + q[j];
            k = KnuthMin(i,j);
            /* KnuthMin returns a value k in the range
               [r[i][j-1], r[i+1][j]] minimizing
               c[i][k-1]+c[k][j] */
            c[i][j] = w[i][j] + c[i][k-1] + c[k][j];
                                    /* Eq. (10.3) */
            r[i][j] = k;
        }
}
```

VTUSYNC.IN

Question Bank

1. Explain open addressing and chaining used to handle overflows in hashing. (Appeared in Dec. 2016/Jan 2017)- 5 Marks
2. Explain directoryless dynamic hashing (Appeared in Dec. 2016/Jan 2017)- 5 Marks
3. Explain Hashing in detail. (Appeared in Dec. 2017/Jan 2018)- 8 Marks
4. What is collision? What are the methods to resolve collision? Explain linear probing with an example. (Appeared in June/July 2017)- 8 Marks
5. Write a short note on hashing-Explain any 3 popular HASH functions. (Appeared in June/July 2018)- 8 Marks
6. . Explain in detail about static and dynamic hashing. (Appeared in Dec.2018/Jan 2019)- 10 Marks
7. Explain Hashing and Collision. What are the methods used to resolve collision. (Appeared in June/July 2019)- 8 Marks
8. What is hashing? Explain with example hash following Hashing function (Appeared in June/July 2019)- 6 Marks
9. Consider the following 4- digit employee number 9614 , 5882 , 67 13 , 4409 , 1825.
10. Find the 2 - digit hash address of each number using (Appeared in June/July 2019)- 8 Marks
    i) The division method with =97 .
    ii) The midsquare method.
    iii) The folding method without reversing
    iv) The folding method with reversing.
11. Explain directory dynamic hashing with example.
12. Explain two types of Leftlist Trees.
13. Write a C-function to meld two min-leftist tree .
14. Write a short note on optimal binary search tree.

VTUSYNC.IN