Data Structures Notes

Contributor: Abhishek Sharma

[Founder at TutorialsDuniya.com]

Computer Science Notes

External Sorting

All the *internal sorting* algorithms require that the input fit into main memory. There are, however, applications where the input is much too large to fit into memory. For those external sorting algorithms, which are designed to handle very large inputs.

Why We Need New Algorithms

Most of the internal sorting algorithms take advantage of the fact that memory is directly addressable. Shell sort compares elements $a[i]$ and $a[i - hk]$ in one time unit. Heap sort compares elements $a[i]$ and $a[i * 2]$ in one time unit. Quicksort, with median-of-three partitioning, requires comparing $a[left]$, $a[center]$, and $a[right]$ in a constant number of time units. If the input is on a tape, then all these operations lose their efficiency, since elements on a tape can only be accessed sequentially. Even if the data is on a disk, there is still a practical loss of efficiency because of the delay required to spin the disk and move the disk head.

The time it takes to sort the input is certain to be insignificant compared to the time to read the input, even though sorting is an $O(n \log n)$ operation and reading the input is only $O(n)$.

Model for External Sorting

The wide variety of mass storage devices makes external sorting much more device dependent than internal sorting. The algorithms that we will consider work on tapes, which are probably the most restrictive storage medium. Since access to an element on tape is done by winding the tape to the correct location, tapes can be efficiently accessed only in sequential order (in either direction).

We will assume that we have at least three tape drives to perform the sorting. We need two drives to do an efficient sort; the third drive simplifies matters. If only one tape drive is present, then we are in trouble: any algorithm will require $O(n^2)$ tape accesses.

The Simple Algorithm

The basic external sorting algorithm uses the *merge* routine from merge sort. Suppose we have four tapes, *Ta*1, *Ta*2, *Tb*1, *Tb*2, which are two input and two output tapes. Depending on the point in the algorithm, the *a* and *b* tapes are either input tapes or output tapes.

Suppose the data is initially on *Ta*1. Suppose further that the internal memory can hold (and sort) *m* records at a time. A natural first step is to read *m* records at a time from the input tape, sort the records internally, and then write the sorted records alternately to *Tb*1 and *Tb*2. We will call each set of sorted records a *run*. When this is done, we rewind all the tapes. Suppose we have the same input as our example for Shell sort.

T_{a1}	81	94	11	96	12	35	17	99	28	58	41	75	15
T_{a2}													
T_{b1}													
T_{b2}													

If $m = 3$, then after the runs are constructed, the tapes will contain the data indicated in the following figure.

T_{a1}							
T_{a2}							
T_{b1}	11	81	94	17	28	99	15
T_{b2}	12	35	96	41	58	75	

Now *Tb*1 and *Tb*2 contain a group of runs. We take the first run from each tape and merge them, writing the result, which is a run twice as long, onto *Ta*1. Then we take the next run from each tape, merge these, and write the result to *Ta*2. We continue this process, alternating between *Ta*1 and *Ta*2, until either *Tb*1 or *Tb*2 is empty. At this point either both are empty or there is one run left. In the latter case, we copy this run to the appropriate tape. We rewind all four tapes, and repeat the same steps, this time using the *a* tapes as input and the *b* tapes as output. This will give runs of 4*m*. We continue the process until we get one run of length *n*.

This algorithm will require log(*n*/*m*) passes, plus the initial run-constructing pass. For instance, if we have 10 million records of 128 bytes each, and four megabytes of internal memory, then the first pass will create 320 runs. We would then need nine more passes to complete the sort. Our example requires log 13/3 = 3 more passes, which are shown in the following figure.

T_{a1}	11	12	35	81	94	96	15						
T_{a2}	17	28	41	58	75	99							
T_{b1}													
T_{b2}													

T_{a1}													
T_{a2}													
T_{b1}	11	12	17	28	35	51	58	75	81	94	96	99	
T_{b2}	15												

T_{a1}	11	12	15	17	28	35	41	58	75	81	94	96	99
T_{a2}													
T_{b1}													
T_{b2}													

Multiway Merge

If we have extra tapes, then we can expect to reduce the number of passes required to sort our input. We do this by extending the basic (two-way) merge to a *k*-way merge.

Merging two runs is done by winding each input tape to the beginning of each run. Then the smaller element is found, placed on an output tape, and the appropriate input tape is advanced. If there are *k* input tapes, this strategy works the same way, the only difference being that it is slightly more complicated to find the smallest of the *k* elements. We can find the smallest of these elements by using a priority queue. To obtain the next element to write on the output tape, we perform a *delete_min* operation. The appropriate input tape is advanced, and if the run on the input tape is not yet completed, we *insert* the new element into the priority queue. Using the same example as before, we distribute the input onto the three tapes.

T_{a1}								
T_{a2}								
T_{a3}								
T_{b1}	11	81	94	41	58	75		
T_{b2}	12	35	96	15				
T_{b3}	17	28	99					

We then need two more passes of three-way merging to complete the sort.

T_{a1}	11	12	17	28	35	81	94	96	99
T_{a2}	15	41	58	75					
T_{a3}									
T_{b1}									
T_{b2}									
T_{b3}									

T_{a1}												
T_{a2}												
T_{a3}												
T_{b1}	11	12	15	17	28	35	41	58	75	81	94	96	99
T_{b2}												
T_{b3}												

After the initial run construction phase, the number of passes required using *k*-way merging is log*k*(*n*/*m*) , because the runs get *k* times as large in each pass. For the example above, the formula is verified, since log3 13/3 = 2. If we have 10 tapes, then *k* = 5, and our large example from the previous section would require log5 320 = 4 passes.

Polyphase Merge

The *k*-way merging strategy developed in the last section requires the use of 2*k* tapes. This could be prohibitive for some applications. It is possible to get by with only *k* + 1 tapes. As an example, we will show how to perform two-way merging using only three tapes.

Suppose we have three tapes, *T*1, *T*2, and *T*3, and an input file on *T*1 that will produce 34 runs. One option is to put 17 runs on each of *T*2 and *T*3. We could then merge this result onto *T*1, obtaining one tape with 17 runs. The problem is that since all the runs are on one tape, we must now put some of these runs on *T*2 to perform another merge. The logical way to do this is to copy the first eight runs from *T*1 onto *T*2 and then perform the merge. This has the effect of adding an extra half pass for every pass we do.

An alternative method is to split the original 34 runs unevenly. Suppose we put 21 runs on *T*2 and 13 runs on *T*3. We would then merge 13 runs onto *T*1 before *T*3 was empty. At this point, we could rewind *T*1 and *T*3, and merge *T*1, with 13 runs, and *T*2, which has 8 runs, onto *T*3. We could then merge 8 runs until *T*2 was empty, which would leave 5 runs left on *T*1 and 8 runs on *T*3. We could then merge *T*1 and *T*3, and so on. The following table below shows the number of runs on each tape after each pass.

	Run Const.	After T**3** + T**2**	After T**1** + T**2**	After T**1** + T**3**	After T**2** + T**3**	After T**1** + T**2**	After T**1** + T**3**	After T**2** + T**3**
*T*1	0	13	5	0	3	1	0	1
*T*2	21	8	0	5	2	0	1	0
*T*3	13	0	8	3	0	2	1	0

The original distribution of runs makes a great deal of difference. For instance, if 22 runs are placed on *T*2, with 12 on *T*3, then after the first merge, we obtain 12 runs on *T*1 and 10 runs on *T*2. Afte another merge, there are 10 runs on *T*1 and 2 runs on *T*3. At this point the going gets slow, because we can only merge two sets of runs before *T*3 is exhausted. Then *T*1 has 8 runs and *T*2 has 2 runs. Again, we can only merge two sets of runs, obtaining *T*1 with 6 runs and *T*3 with 2 runs. After three more passes, *T*2 has two runs and the other tapes are empty. We must copy one run to another tape, and then we can finish the merge.

It turns out that the first distribution we gave is optimal. If the number of runs is a Fibonacci number *Fn*, then the best way to distribute them is to split them into two Fibonacci numbers *Fn*-1 and *Fn*-2. Otherwise, it is necessary to pad the tape with dummy runs in order to get the number of runs up to a Fibonacci number. We leave the details of how to place the initial set of runs on the tapes as an exercise.

We can extend this to a *k*-way merge, in which case we need *k*th order Fibonacci numbers for the distribution, where the *k*th order Fibonacci number is defined as $F^{(k)}(n) = F^{(k)}(n - 1) + F^{(k)}(n - 2) + + F^{(k)}(n - k)$, with the appropriate initial conditions $F^{(k)}(n) = 0,0\ n\ k - 2$, $F^{(k)}(k - 1) = 1$.

Replacement Selection

The last item we will consider is construction of the runs. The strategy we have used so far is the simplest possible: We read as many records as possible and sort them, writing the result to some tape. This seems like the best approach possible, until one realizes that as soon as the first record is written to an output tape, the memory it used becomes available for another record. If

the next record on the input tape is larger than the record we have just output, then it can be included in the run.

Using this observation, we can give an algorithm for producing runs. This technique is commonly referred to as *replacement selection*.

Initially, *m* records are read into memory and placed in a priority queue. We perform a *delete_min*, writing the smallest record to the output tape. We read the next record from the input tape. If it is larger than the record we have just written, we can add it to the priority queue. Otherwise, it cannot go into the current run. Since the priority queue is smaller by one element, we can store this new element in the dead space of the priority queue until the run is completed and use the element for the next run. Storing an element in the dead space is similar to what is done in heapsort. We continue doing this until the size of the priority queue is zero, at which point the run is over. We start a new run by building a new priority queue, using all the elements in the dead space. Figure 7.18 shows the run construction for the small example we have been using, with $m = 3$. Dead elements are indicated by an asterisk.

In this example, replacement selection produces only three runs, compared with the five runs obtained by sorting. Because of this, a three-way merge finishes in one pass instead of two. If the input is randomly distributed, replacement selection can be shown to produce runs of average length $2m$. For our large example, we would expect 160 runs instead of 320 runs, so a five-way merge would require four passes. In this case, we have not saved a pass, although we might if we get lucky and have 125 runs or less. Since external sorts take so long, every pass saved can make a significant difference in the running time.

	3 Elements In Heap Array			Output	Next Element Read
	H[1]	H[2]	H[3]		
Run 1	11	94	81	11	96
	81	94	96	81	12*
	94	96	12*	94	35*
	96	35*	12*	96	17*
	17*	35*	12*	End of Run.	Rebuild Heap
Run 2	12	35	17	12	99
	17	35	99	17	28
	28	99	35	28	58
	35	99	58	35	41
	41	99	58	41	75*
	58	99	75*	58	end of tape
	99		75*	99	
			75*	End of Run.	Rebuild Heap
Run 3	75			75	

Figure: Example of run construction

As we have seen, it is possible for replacement selection to do no better than the standard algorithm. However, the input is frequently sorted or nearly sorted to start with, in which case replacement selection produces only a few very long runs. This kind of input is common for external sorts and makes replacement selection extremely valuable.

K-Way Merging

- The 2-way merge algorithm is almost identical to the merge procedure in figure.
- In general, if we started with m runs, then the merge tree would have $\lceil \log_2 m \rceil + 1$ levels for a total of $\lceil \log_2 m \rceil$ passes over the data file. The number of passes over the data can be reduced by using a higher order merge, i.e., k-way merge for $k \geq 2$. In this case we would simultaneously merge k runs together.
- The number of passes over the data is now 2, versus 4 passes in the case of a 2-way merge. In general, a k-way merge on m runs requires at most $\lceil \log_k m \rceil$ passes over the data. Thus, the input/output time may be reduced by using a higher order merge.

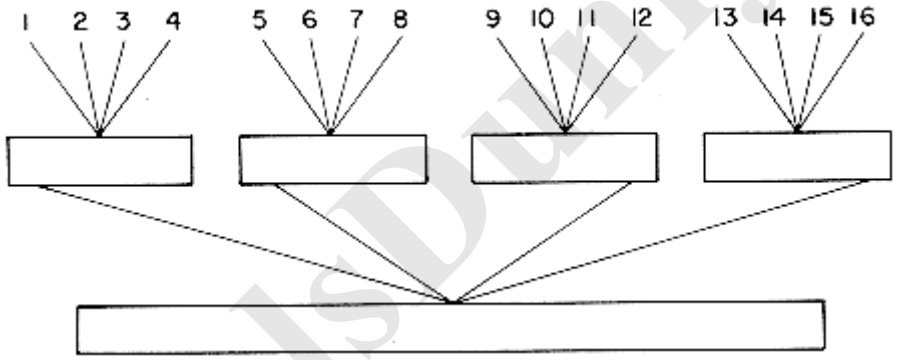


Figure: A 4-way Merge on 16 Runs

- The use of a higher order merge, has some other effects on the sort. To begin with, k-runs of size $S_1, S_2, S_3, \ldots, S_k$ can no longer be merged internally in $O(\sum_1^k S_i)$ time.
- In a k-way merge, as in a 2-way merge, the next record to be output is the one with the smallest key. The smallest has now to be found from k possibilities and it could be the leading record in any of the k-runs.
    - The most direct way to merge k-runs would be to make $k - 1$ comparison to determine the next record to output. The computing time for this would be $O((k-1)\sum_1^k S_i)$. Since $\log_k m$ passes are being made, the total number of key comparisons being made is $\boldsymbol{n(k - 1) \log_k m = n(k - 1) \log_2 m / \log_2 k}$ where n is the number of records in the file. Hence, $(k-1)/\log_2 k$ is the factor by which the number of key comparisons increases. As k increases, the reduction in input/output time will be overweighed by the resulting increase in CPU time needed to perform the k-way merge.

- For large k (say, $k \geq 6$) we can achieve a significant reduction in the number of comparisons needed to find the next smallest element by using the idea of a selection tree. Hence, the total time needed per level of the merge tree is $O(n\log_2 k)$. Since the number of levels in this tree is $\boldsymbol{O(\log_k m)}$, the asymptotic internal processing time becomes $\boldsymbol{O(n \log_2 k \log_k m) = O(n \log_2 m)}$. The internal processing time is independent of k.

- In going to a higher order merge, we save on the amount of input/output being carried out. There is no significant loss in internal processing speed. Even though the internal processing time is relatively insensitive to the order of the merge, the decrease in input/output time is not as much as indicated by the reduction to $\log_k m$ passes.

- This is so because the number of input buffers needed to carry out a k-way merges increases with k. Though $k + 1$ buffer are sufficient. Since the internal memory available is fixed and independent of k, the buffer size must be reduced as k increases. This in turn implies a reduction in the block size on disk. With the reduced block size each pass over the data results in a greater number of blocks being written or read.

- This represents a potential increase in input/output time from the increased contribution of seek and latency times involved in reading a block of data. Hence, beyond a certain k value the input/output time would actually increase despite the decrease in the number of passes being made. The optimal value for k clearly depends on disk parameters and the amount of internal memory available for buffers.

Buffer Handling for Parallel Operation

If k runs are being merged together by a k-way merge, then we clearly need at least k input buffers and one output buffer to carry out the merge. This, however, is not enough if input, output and internal merging are to be carried out in parallel. For instance, while the output buffer is being written out, internal merging has to be halted since there is no place to collect the merged records. This can be easily overcome through the use of two output buffers. While one is being written out, records are merged into the second. If buffer sizes are chosen correctly, then the time to output one buffer would be the same as the CPU time needed to fill the second buffer. With only k input buffers, internal merging will have to be held up whenever one of these input buffers becomes empty and another block from the corresponding run is being read in. This input delay can also be avoided if we have $2k$ input buffers. These $2k$ input buffers have to be cleverly used in order to avoid reaching a situation in which processing has to be held up because of lack of input records from any one run. Simply assigning two buffers per run does not solve the problem. To see this, consider the following example.

Example 1: Assume that a two way merge is being carried out using four input buffers, $IN(i)$, $1 \leq i \leq 4$, and two output buffers, $OU(1)$ and $OU(2)$. Each buffer is capable of holding two records. The first few records of run 1 have key value 1, 3, 5, 7, 8, 9. The first few records of run 2 have key value 2, 4, 6, 15, 20, 25. Buffers $IN(1)$ and $IN(3)$ are assigned to run 1. The remaining two input buffers are assigned to run 2. We start the merging by reading in one

buffer load from each of the two runs. At this time the buffers have the configuration of figure 8.12(a). Now runs 1 and 2 are merged using records from *IN* (1) and *IN*(2). In parallel with this the next buffer load from run 1 is input. If we assume that buffer lengths have been chosen such that the times to input, output and generate an output buffer are all the same then when *OU*(1) is full we have the situation of figure 8.12(b). Next, we simultaneously output *OU*(1), input into *IN*(4) from run 2 and merge into *OU*(2). When *OU*(2) is full we are in the situation of figure 8.12(c). Continuing in this way we reach the configuration of figure 8.12(e). We now begin to output *OU*(2), input from run 1 into *IN*(3) and merge into *OU*(1). During the merge, all records from run 1 get exhausted before *OU*(1) gets full. The generation of merged output must now be delayed until the inputting of another buffer load from run 1 is completed!

Example 1 makes it clear that if 2*k* input buffers are to suffice then we cannot assign two buffers per run. Instead, the buffers must be floating in the sense that an individual buffer may be assigned to any run depending upon need. In the buffer assignment strategy we shall describe, for each run there will at any time be, at least one input buffer containing records from that run. The remaining buffers will be filled on a priority basis. I.e., the run for which the *k*-way merging algorithm will run out of records first is the one from which the next buffer will be filled. One may easily predict which run's records will be exhausted first by simply comparing the keys of the last record read from each of the *k* runs. The smallest such key determines this run. We shall assume that in the case of equal keys the merge process first merges the record from the run with least index. This means that if the key of the last record read from run *i* is equal to the key of the last record read from run *j*, and $i < j$, then the records read from *i* will be exhausted before those from *j*. So, it is possible that at any one time we might have more than two bufferloads from a given run and only one partially full buffer from another run. All bufferloads from the same run are queued together. Before formally presenting the algorithm for buffer utilization, we make the following assumptions about the parallel processing capabilities of the computer system available:

(i) We have two disk drives and the input/output channel is such that it is possible simultaneously to read from one disk and write onto the other.

(ii) While data transmission is taking place between an input/output device and a block of memory, the CPU cannot make references to that same block of memory. Thus, it is not possible to start filling the front of an output buffer while it is being written out. If this were possible, then by coordinating the transmission and merging rate only one output buffer would be needed. By the time the first record for the new output block was determined, the first record of the previous output block would have been written out.

(iii) To simplify the discussion we assume that input and output buffers are to be the same size.

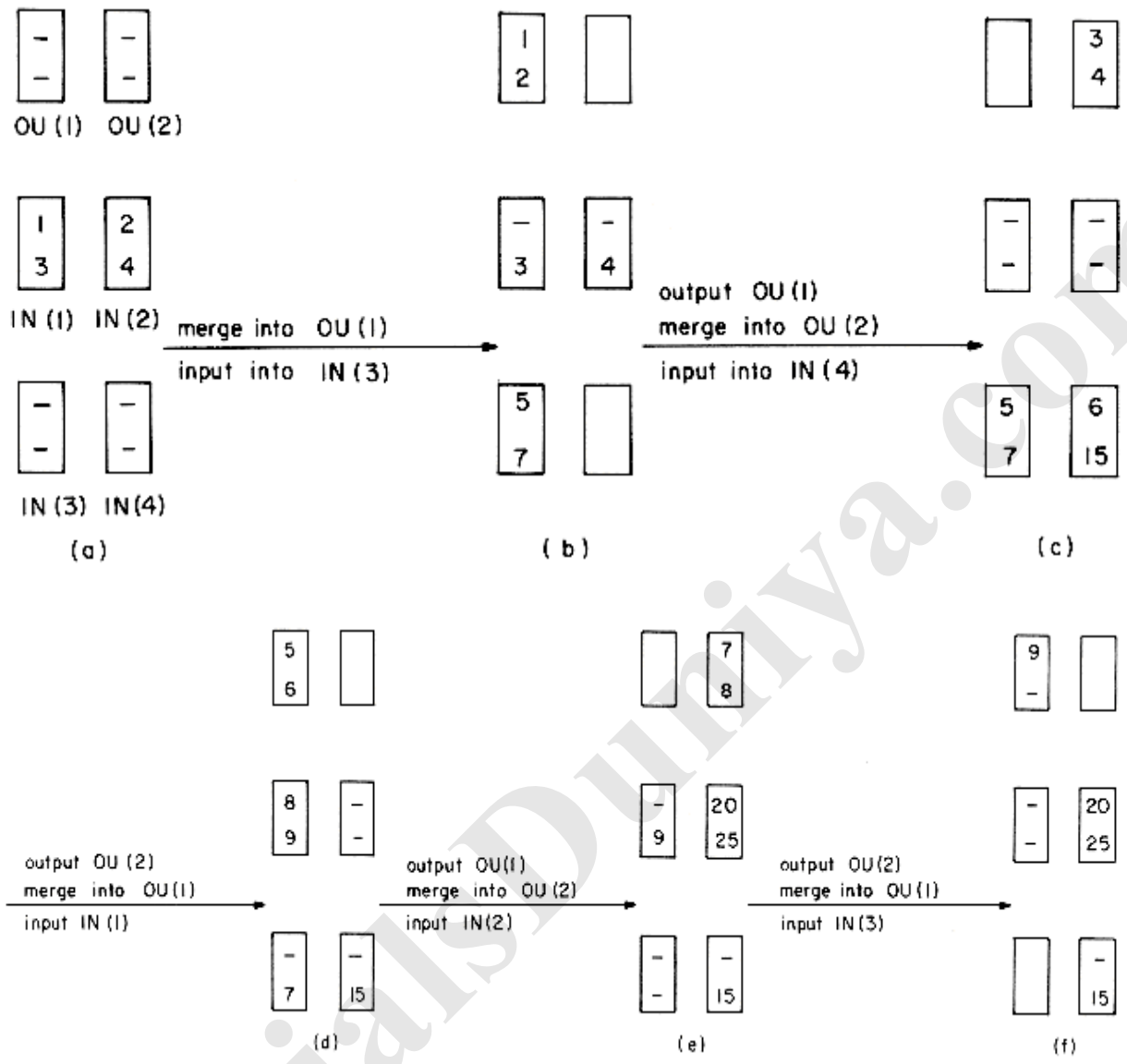


Figure 1 Example showing that two fixed buffers per run are not enough for continued parallel operation

Keeping these assumptions in mind, we first formally state the algorithm obtained using the strategy outlined earlier and then illustrate its working through an example. Our algorithm merges *k*-runs, k>=2, using a *k*-way merge. *2k* input buffers and 2 output buffers are used. Each buffer is a contiguous block of memory. Input buffers are queued in *k* queues, one queue for each run. It is assumed that each input/output buffer is long enough to hold one block of records. Empty buffers are stacked with AV pointing to the top buffer in this stack. The stack is a linked list. The following variables are made use of:

IN (i) ... input buffers, $1 \le i \le 2k$
OUT (i) ... output buffers, $0 \le i \le 1$
FRONT (i) ... pointer to first buffer in queue for run i, $1 \le i \le k$
END (i) ... end of queue for i-th run, $1 \le i \le k$

LINK (i) ... link field for i-th input buffer
in a queue or for buffer in stack $1 \le i \le 2k$
LAST (i) ... value of key of last record read
from run i, $1 \le i \le k$
OU ... buffer currently used for output.

The algorithm also assumes that the end of each run has a sentinel record with a very large key, say $+\infty$. If block lengths and hence buffer lengths are chosen such that the time to merge one output buffer load equals the time to read a block then almost all input, output and computation will be carried out in parallel. It is also assumed that in the case of equal keys the *k*-way merge algorithm first outputs the record from the run with smallest index.

```
   procedure BUFFERING
1  for i <-- 1 to k do     //input a block from each run//
2    input first block of run i into IN(i)
3  end
4  while input not complete do end     //wait//
5  for i <-- 1 to k do     //initialize queues and free buffers//
6    FRONT(i) <-- END(i) <-- i
7    LAST(i) <-- last key in buffer IN(i)
8    LINK(k + i) <-- k + i + 1     //stack free buffer//
9  end
10  LINK(2k)  <-- 0; AV <-- k + 1; OU <-- 0
   //first queue exhausted is the one whose last key read is smallest//
11  find j such that LAST(j) =  min  {LAST(i)}
                             l≤i≤k
12  l <-- AV; AV <-- LINK(AV)     //get next free buffer//
13  if LAST(j) ≠ + ∞ then [begin to read next block for run j into
                           buffer IN(l)]
14  repeat      //KWAYMERGE merges records from the k buffers
              FRONT(i) into output buffer OU until it is full.
              If an input buffer becomes empty before OU is filled, the
              next buffer in the queue for this run is used and the empty
              buffer is stacked or last key = + ∞//
15    call KWAYMERGE
16    while input/output not complete do     //wait loop//
17    end
      if LAST(j) ≠ + ∞     then
18    [LINK(END(j)) <-- l; END(j) <-- l; LAST(j) <-- last key read
                                    //queue new block//
19    find j such that LAST(j) =  min {LAST(i)}
                           l≤i≤k
20    l <-- AV; AV <-- LINK(AV)]     //get next free buffer//
21    last-key-merged <-- last key in OUT(OU)
22    if LAST(j) ≠ + ∞ then [begin to write OUT(OU) and read next block of
                        run j into IN(l)]
23                  else [begin to write OUT(OU)]
24    OU <-- 1 - OU
```

```
25   until last-key-merged = + ∞
26   while output incomplete do     //wait loop//
27   end
28  end BUFFERING
```

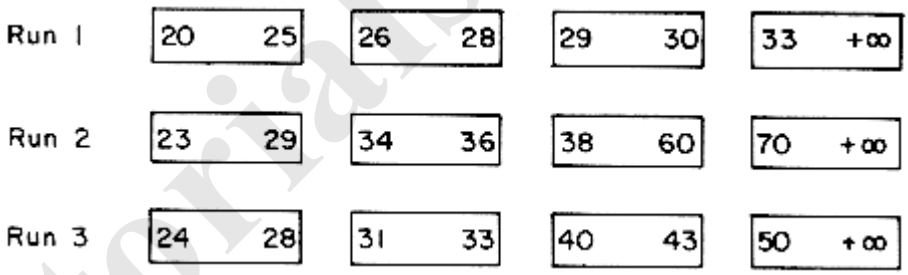
Notes: 1) For large k, determination of the queue that will exhaust first can be made in $\log_2 k$ comparisons by setting up a selection tree for LAST(i), $1 \le i \le k$, rather than making k - 1 comparisons each time a buffer load is to be read in. The change in computing time will not be significant, since this queue selection represents only a very small fraction of the total time taken by the algorithm.

2) For large k the algorithm KWAYMERGE uses a selection tree

3) All input/output except for the initial k blocks that are read and the last block output is done concurrently with computing. Since after k runs have been merged we would probably begin to merge another set of k runs, the input for the next set can commence during the final merge stages of the present set of runs. I.e., when LAST(j) = $+\infty$ we begin reading one by one the first blocks from each of the next set of k runs to be merged. In this case, over the entire sorting of a file, the only time that is not overlapped with the internal merging time is the time for the first k blocks of input and that for the last block of output.

4) The algorithm assumes that all blocks are of the same length. This may require inserting a few dummy records into the last block of each run following the sentinel record $+\infty$.

Example 2: To illustrate the working of the above algorithm, let us trace through it while it performs a three-way merge on the following three runs:



Each run consists of four blocks of two records each; the last key in the fourth block of each of these three runs is $+\infty$. We have six input buffers IN(i), $1 \le i \le 6$, and 2 output buffers OUT(0) and OUT(1). The status of the input buffer queues, the run from which the next block is being read and the output buffer being ouput at the beginning of each iteration of the **repeat-until** of the buffering algorithm.

Theorem : The following is true for algorithm BUFFERING:

(i) There is always a buffer available in which to begin reading the next block; and

(ii) during the *k*-way merge the next block in the queue has been read in by the time it is needed.

Proof: (i) Each time we get to line 20 of the algorithm there are at most $k + 1$ buffer loads in memory, one of these being in an output buffer. For each queue there can be at most one buffer that is partially full. If no buffer is available for the next read, then the remaining k buffers must be full. This means that all the k partially full buffers are empty (as otherwise there will be more than $k + 1$ buffer loads in memory). From the way the merge is set up, only one buffer can be both unavailable and empty. This may happen only if the output buffer gets full exactly when one input buffer becomes empty. But $k > 1$ contradicts this. So, there is always at least one buffer available when line 20 is being executed.

(ii) Assume this is false. Let run R_i be the one whose queue becomes empty during the KWAYMERGE. We may assume that the last key merged was not the sentinel key $+\infty$ since otherwise KWAYMERGE would terminate the search rather then get another buffer for R_i. This means that there are more blocks of records for run R_i on the input file and $\text{LAST}(i) \neq +\infty$. Consequently, up to this time whenever a block was output another was simultaneously read in (see line 22). Input/output therefore proceeded at the same rate and the number of available blocks of data is always k. An additional block is being read in but it does not get queued until line 18. Since the queue for R_i has become empty first, the selection rule for the next run to read from ensures that there is at most one block of records for each of the remaining $k - 1$ runs. Furthermore, the output buffer cannot be full at this time as this condition is tested for before the input buffer empty condition. Thus there are fewer than k blocks of data in memory. This contradicts our earlier assertion that there must be exactly k such blocks of data.

Run Generation

Using conventional internal sorting methods, it is possible to generate runs that are only as large as the number of records that can be held in internal memory at one time. Using a tree of losers it is possible to do better than this. In fact, the algorithm we shall present will on the average generate runs that are twice as long as obtainable by conventional methods. This algorithm was devised by Walters, Painter and Zalk. In addition to being capable of generating longer runs, this algorithm will allow for parallel input, output and internal processing. For almost all the internal sort methods discussed in Chapter 7, this parallelism is not possible. Heap sort is an exception to this. In describing the run generation algorithm, we shall not dwell too much upon the input/output buffering needed. It will be assumed that input/output buffers have been appropriately set up for maximum overlapping of input, output and internal processing. Wherever in the run generation algorithm there is an input/output instruction, it will be assumed that the operation takes place through the input/output buffers. We shall assume that there is enough space to construct a tree of losers for k records, $R(i)$, $0 \leq i < k$. This will require a loser tree with k nodes numbered 0 to $k - 1$. Each node, i, in this tree will have one field $L(i)$. $L(i)$, $1 \leq i < k$ represents the loser of the tournament played at node i. Node 0 represents the overall winner of the tournament. This node will not be explicitly present in the algorithm. Each of the k record positions $R(i)$, has a run number field $RN(i)$, $0 \leq i < k$ associated with it. This field

will enable us to determine whether or not *R* (*i*) can be output as part of the run currently being generated. Whenever the tournament winner is output, a new record (if there is one) is input and the tournament replayed as discussed in section Algorithm RUNS is simply an implementation of the loser tree strategy discussed earlier. The variables used in this algorithm have the following significance:

R(i), 0 ≤ i < k ... the k records in the tournament tree
KEY(i), 0 ≤ i < k ... key value of record R(i)
L(i), 0 < i < k ... loser of the tournament played at node i
RN(i), 0 <= i < k ... the run number to which R(i) belongs
RC ... run number of current run
Q ... overall tournament winner
RQ ... run number for R(Q)
RMAX ... number of runs that will be generated
LAST__KEY ... key value of last record output

The loop of lines 5-25 repeatedly plays the tournament outputting records. The only interesting thing about this algorithm is the way in which the tree of losers is initialized. This is done in lines 1-4 by setting up a fictitious run numbered 0. Thus, we have $RN(i) = 0$ for each of the k records $R(i)$. Since all but one of the records must be a loser exactly once, the initialization of $L(i) \leftarrow i$ sets up a loser tree with R(0) the winner. With this initialization the loop of lines 5-26 correctly sets up the loser tree for run 1. The test of line 10 suppresses the output of these k fictitious records making up run 0. The variable LAST__KEY is made use of in line 13 to determine whether or not the new record input, R(Q), can be output as part of the current run. If KEY(Q) < LAST__KEY then $R(Q)$ cannot be output as part of the current run RCas a record with larger key value has already been output in this run. When the tree is being readjusted (lines 18-24), a record with lower run number wins over one with a higher run number. When run numbers are equal, the record with lower key value wins. This ensures that records come out of the tree in non-decreasing order of their run numbers. Within the same run, records come out of the tree in non-decreasing order of their key values . RMAX is used to terminate the algorithm. In line 11, when we run out of input, a record with run number RMAX + 1 is introduced. When this record is ready for output, the algorithm terminates in line 8. One may readily verify that when the input file is already sorted, only one run is generated. On the average, the run size is almost $2k$. The time required to generate all the runs for an n run file is $O(n \log k)$ as it takes $O(\log k)$ time to adjust the loser tree each time a record is output. The algorithm may be speeded slightly by explicitly initializing the loser tree using the first k records of the input file rather than k fictitious records as in lines 1-4. In this case the conditional of line 10 may be removed as there will no longer be a need to suppress output of certain records.

Optimal Merging Of Runs

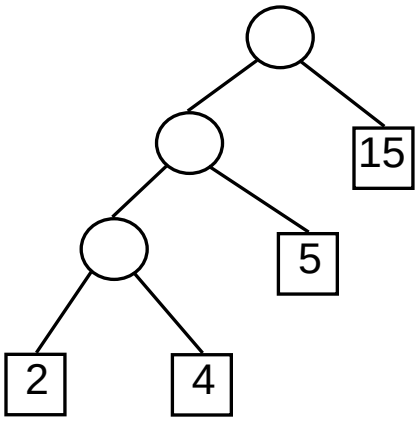
- When we merge by using the first merge tree, we merge some records only once, while others may be merged up to three times.
- In the second merge tree, we merge each record exactly twice.
- We can construct Huffman tree to solve this problem.

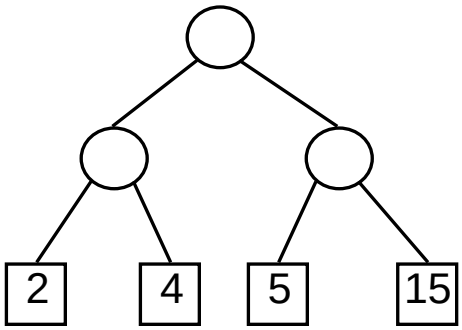
TutorialsDuniya.com

Download FREE Computer Science Notes, Programs, Projects, Books PDF for any university student of BCA, MCA, B.Sc, B.Tech CSE, M.Sc, M.Tech at https://www.tutorialsduniya.com

- Algorithms Notes
- Artificial Intelligence
- Android Programming
- C & C++ Programming
- Combinatorial Optimization
- Computer Graphics
- Computer Networks
- Computer System Architecture
- DBMS & SQL Notes
- Data Analysis & Visualization
- Data Mining
- Data Science
- Data Structures
- Deep Learning
- Digital Image Processing
- Discrete Mathematics
- Information Security
- Internet Technologies
- Java Programming
- JavaScript & jQuery
- Machine Learning
- Microprocessor
- Operating System
- Operational Research
- PHP Notes
- Python Programming
- R Programming
- Software Engineering
- System Programming
- Theory of Computation
- Unix Network Programming
- Web Design & Development

Please Share these Notes with your Friends as well





External path length:

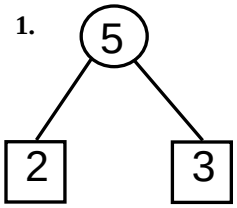
2*3+4*3+5*2+15*1=43

2*2+4*2+5*2+15*2=52

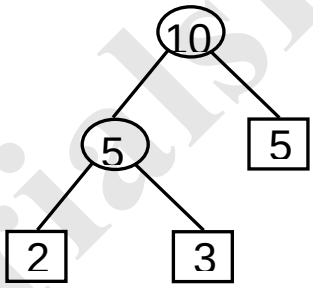
Construction of a Huffman tree

Runs of length : 2,3,5,7,9,13

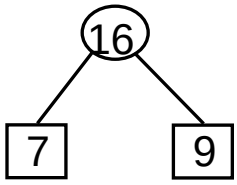
1.

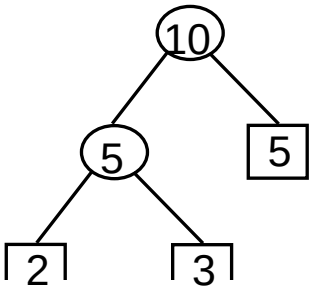


2.

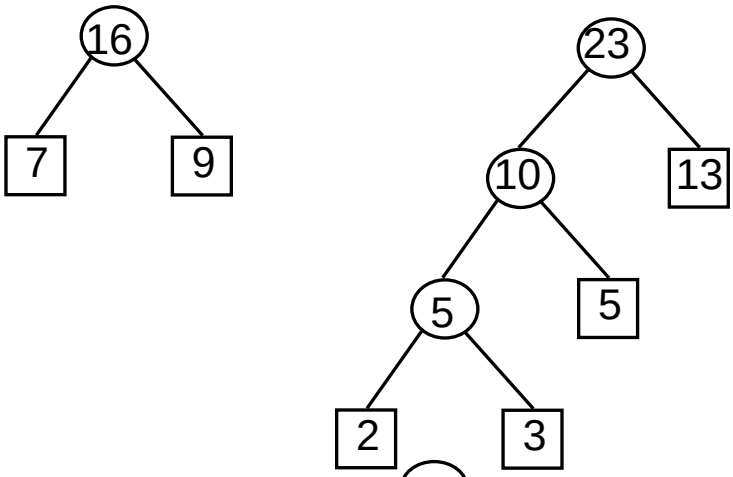


3.

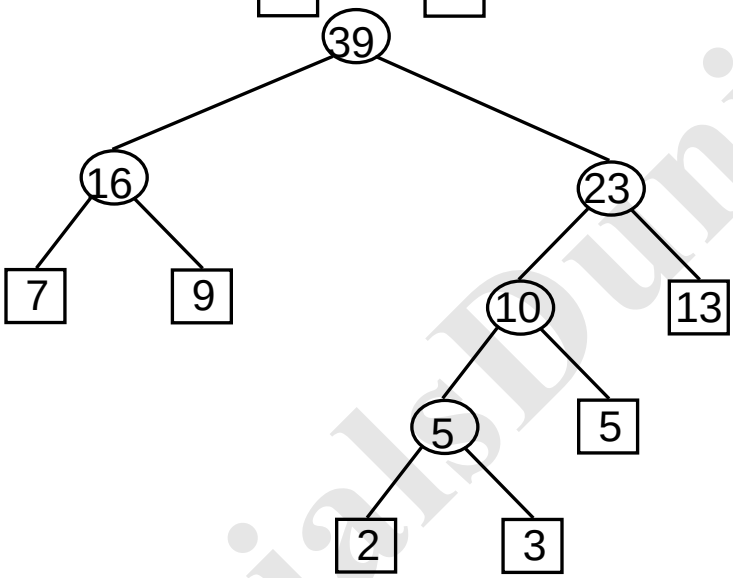




4.



5.



Introduction

Hashing:- The implementation of the hash table is frequency called as hashing.

→ Hashing is a technique which can be used to perform the operations such as insertion, deletion, search / find operations. in average constant time of Big-oh ⇒ O(1)

→ The ideal hash table data structure is a fixed size array by dividing that hash table into the numbers of locations which is equal to the size of the hash table.

→ where each location is capable of storing the given keys into the hash table.

→ The hash table can be indexed from zero to n-1 (0 to n-1) where n is the size of the hash table.

→ The hash table implementation can be done by using hash function only. that is nothing can be done without hash function in hash table

Hashing organisation:-

The organization of the hash table can be done by using → bucket
→ hash function.

Bucket:- A bucket a storage location of the hash table.

→ The hash table can be divided into the no of buckets depending on given hash table size.

bucket is capable of storing the given keys.

→ The hash table operations can be perform only with in the buckets of hash table.

Hash Function:- The hash function is a mapping function which maps the given search keys into the buckets of the hash table.

i.e., it is a function from search keys to hash table locations.

Types of Hashing:-

The Hashing can be divided into two types.

which are 1) static Hashing
2) Dynamic Hashing.

1) static Hashing:-

In static Hashing the size of the hash table is fixed.

i.e., no growing (or) shinking during the implementation of the hash table.

2) Dynamic Hashing:-

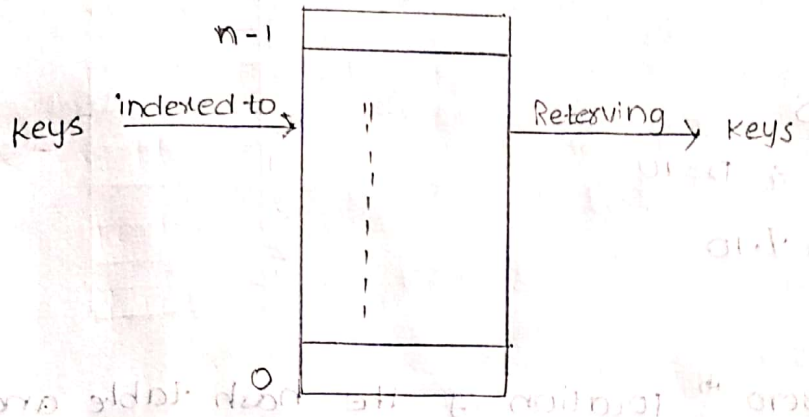
In Dynamic Hashing the size of the hash table is variable i.e., the size of the hash table may grow (or) shink during the implementation of the hash table.

Static Hashing:

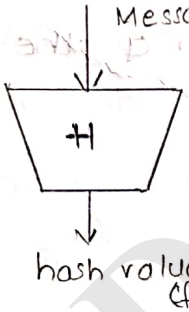
Hashing: Hashing is the process of indexed to and retriving the keys from hash table.

Hash Table: The hash table is of fixed size array in static hashing. The hash table contains the number of buckets ~~dividing~~ divided from 0 to n-1

The keys are inserted into a table by using hash function.



Hash Function: The hash function is a mapping function which maps the given message of arbitary length to the buckets of the hash table by retriving output as hash value of fixed size.



Ex:- Insert the keys 50, 62, 78, 11, 15, 19, 47 into the hash table of size 10]X

The hash function can be ~~given~~ defined as $h(k) = k \% D$

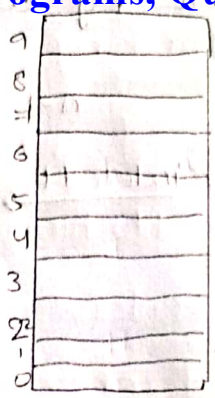
where k : key
D : size of hash table.

→ The hash function always returns a value is called as hash value (or) Message digest.

Ex:- Insert the keys 50, 62, 78, 11, 15, 19, 47 into the hash table of size 10.

→ The hash table can be created with 10 locations by indexing from 0 to 9. as the given size is of 10.

Hash table:



For key 50:

$h(K) = K \% D$

where $K = 50$ & $D = 10$

$h(50) = 50 \% 10$

$= 0$

→ Go to the zero th location of the hash table and insert the key 50.

For key 62:

$h(62) = 62 \% 10$

$= 2$

→ Go to the 2nd location of the hash table and insert the key 62.

For key 78:

$h(78) = 78 \% 10 = 8$

→ Go to the 8th location of the hash table and insert the key 78.

For key 11:

$h(11) = 11 \% 10 = 1$

→ Go to the 1st location of the hash table and insert the key 11.

For key 15:

$h(15) = 15 \% 10 = 5$

→ Go to the 5th location of the hash table and insert the key 5.

For key 19:

$h(19) = 19 \% 10 = 9$

→ Go to the 9th location of the hash table and insert the key 9.

$h(47) = 47 \% 10 = 7$

→ Go to the 7th location of the hash table and insert the key 47.

After inserting all these keys to the hash table. The hash table is.

9	19
8	78
7	47
6	
5	15
4	
3	
2	62
1	11
0	50

Delete the keys 15 and 44 from the above hash table.

For key 15:

$h(15) = 15 \% 10 = 5$

→ Go to 5th location of the hash table and check if the given key is there (or) not.

→ The given key 15 exists in 5th location of the hash table. so delete the key 15 from 5th location of the hash table.

→ After deletion the hash table is.

9	19
8	78
7	47
6	
5	
4	
3	
2	62
1	11
0	50

For key 44:

$h(44) = 44 \% 10 = 4$.

→ Go to 4th location of the hash table and check whether the given key is ~~not~~ there (or) not.

→ In 4th location the given key 44 is not there so, No deletion can be done.

Search the keys 19, 62 & 59 in the above hash table.

For key 19:

$h(19) = 19 \% 10 = 9$

→ Go to the 9th location of the hash table and check whether the given key is there (or) not.

→ In 9th location the given key 19 is there.

so, our search operation is successfull.

For key 62:

$h(62) = 62 \% 10 = 2.$

→ Go to the 2nd location of the hash table and Search the given key is there (or) not.

→ In second location the given key 62 is there.

so, search operation is succesfull.

For key 59:

$h(59) = 59 \% 10 = 9$

→ Go to the 9th location of the hash table and search whether the given key is there (or) not.

→ In 9th location the given key 59 is not there.

so, our search operation is unsuccessfull.

Ex: Insert the keys 17, 41, 110, 7, 123, 148

Delete the keys 110, 140.

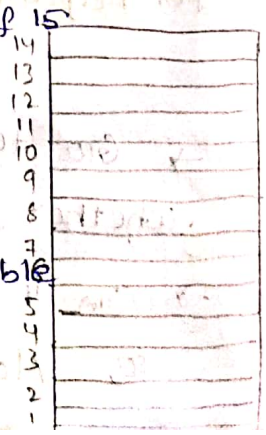
Search the keys 7, 75 in the hash table of size 15.

→ The hash table can be created with 15 locations by indexing from 0 to 14 as the given size of 15 hash table:-

For key 17:

$h(K) = K \% D = 17 \% 15 = 2$

→ Go to the 2nd location of the hash table and insert the key 17.



$h(41) = 41 \% 15 = 11$

→ Go to the 11th location of the hash table and insert the key 41.

For key 110:

$h(110) = 110 \% 15 = 5$

→ Go to the 5th location of the hash table and insert the key 110.

For key 7:

$h(7) = 7 \% 15 = 7$

→ Go to the 7th location of the hash table and insert the key 7.

For key 123:

$h(123) = 123 \% 15 = 3$

→ Go to the 3rd location of the hash table and insert the key 3.

For key 148:

$h(148) = 148 \% 15 = 13$.

→ Go to the 13th location of the hash table and insert the key 148.

After inserting all the keys to the hash table. The hash table will be

14	
13	148
12	
11	41
10	
9	
8	
7	7
6	
5	110
4	
3	123
2	17
1	
0	

Deleting the keys:

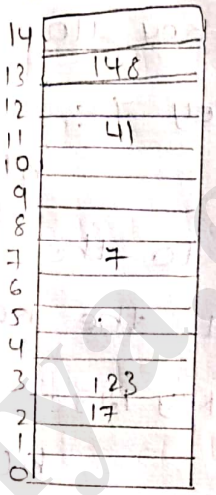
For key 110:

$$h(110) = 110 \% 15 = 5$$

→ Go to 5th location of hash table and check if the given key is there or not

→ The given 110 is exists in 5th location of the hash table. so delete the key 110 from 5th location of the hash table.

→ After deletion the hash table is



For key 140:

$$h(140) = 140 \% 15 = 5$$

→ Go to the 5th location of the hash table and check if the given key is there (or) not

→ The given 140 is not exists. so, no deletion can be done.

Search the keys:

For key 7:

$$h(7) = 7 \% 15 = 7$$

→ Go to the 7th location of hash table and check whether the given key is there (or) not.

→ In 7th location the given key 7 is there. so, our search operation is succesfull.

For key 75:

$$h(75) = 75 \% 15 = 0$$

→ Go to the 0th location of hash table & search the given key is there (or) not.

so, our search operation is unsuccessful.

Feauters of Hash function:-

1. The Hash function genesates fixed length output data (Hash value)
2. The Hash function Converts orbitary length data to fixed length data, this process can be called as hashing the data
3. The hash functions can also be called as Compressed functions. As it compress the given large data into smaller representation.
4. The hash value generated by the hash function is the smaller representation of the keys that's why the hash value can be called as digest.
5. If the hash value is of n bit then those many n bits hash functions can be used to generate hash values.

→ The hash value ranges from 160 - 512 bits / 20 - 64 bytes.

Secure Hash Function (SHA):

While inserting the keys into the buckets of the hash table, if more than one key wants to enter into the same location then it uses secure hash functions to provide security by avoiding collisions.

→ The SHA Family available in four types of algorithms which are

- SHA-0
- SHA-1
- SHA-2
- SHA-3

SHA-0:

SHA-0 is the original ~~fun~~ version of secured hash function is ~~one~~ 160 bit and proposed by National Institue OF Standard Technologies (NIST).

* It had few weakness and didn't became the much popular.
* In this situation the **NIST** called for a new secure hash function in 1995 to correct the weakness of SHA-0.

SHA-1:

The SHA-1 algorithm designs a new secure hash function. in the year 1995. to release all the weakness occured in SHA-0.

* The SHA-1 hash function widely employable in applications and protocalls by including Secure Socket layer (SSL)
* The long term employability of SHA-1 is doubtfull.

so, the next version SHA-2 invited in the year 2003.

SHA-2:

SHA-2 secure hash function is the ~~storg~~ strongest hash function which can offers four variance of hash values.

they are

SHA - 224

SHA - 256

SHA - 384

SHA - 512 by depending the no of bits.

SHA-2 algorithm is not became that much popular as it uses still the design of SHA-0.

* In this situation the NIST develop a new secure hash function in October 2012.

SHA-3:

The SHA-3 secure Hash function develop by using Keccak algorithm as it offers many benfits such as efficient performance at good resistance from attacks.

** Collision Resolution / overflow Handling Techniques:-

Collision: When more than one key wants to enter into the same location of the hash table, then that situation can be called as Collision.

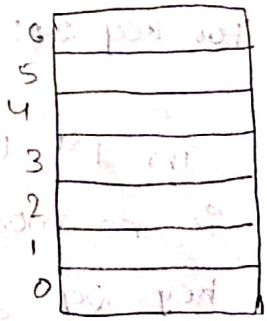
Overflow: If the key insertion exceeds the level of defined bucket capacity then it results in over flow condition.

Ex:- Insert the keys 43, 51, 15, 16, 29 of table size D=7

→ For key 43:

$h(k) = k \% D$

$h(43) = 43 \% 7 = 1$



→ Go to the 1st location and store 43

For key 51:

$h(51) = 51 \% 7 = 2$

→ Go to the 2nd location and ~~store~~ 51. check whether that location is empty (or) not.

For key 15:

~~h(15) = 15 % 7 = 1~~

The second location is empty. so, insert the key 51 into the table.

For key 15:

$$h(15) = 15 \% 7 = 1$$

[→ ~~Go to the 1st location~~]

Here the 1st location of the hash table was already occupied by key 43 and the key 15 also wants to enter into the same location of the hash table. so, it result in collision.

* consider the capacity of each bucket of the hash table is 2.
* When collision occur at 1st location some more space available to insert another key, so, insert key ~~15~~ 15 in 1st location of the hash table.

For key 16:

$$h(16) = 16 \% 7 = 2.$$

In 2nd location already the key 51 is there. so, it results in collision to insert another key. under this condition only insert key 16 as free space is available for that.

For key 29:

$$h(29) = 29 \% 7 = 1$$

In 1st location already 2 keys are there and there is no more space to insert another key but the key 29 ~~also~~ also wants the same location to insert.

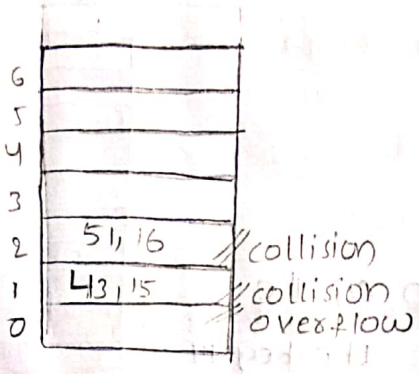
* Here if the insertion can be done it exceeds the defined capacity level by resulting in overflow condition.

TutorialsDuniya.com

Download FREE Computer Science Notes, Programs, Projects, Books PDF for any university student of BCA, MCA, B.Sc, B.Tech CSE, M.Sc, M.Tech at https://www.tutorialsduniya.com

- Algorithms Notes
- Artificial Intelligence
- Android Programming
- C & C++ Programming
- Combinatorial Optimization
- Computer Graphics
- Computer Networks
- Computer System Architecture
- DBMS & SQL Notes
- Data Analysis & Visualization
- Data Mining
- Data Science
- Data Structures
- Deep Learning
- Digital Image Processing
- Discrete Mathematics
- Information Security
- Internet Technologies
- Java Programming
- JavaScript & jQuery
- Machine Learning
- Microprocessor
- Operating System
- Operational Research
- PHP Notes
- Python Programming
- R Programming
- Software Engineering
- System Programming
- Theory of Computation
- Unix Network Programming
- Web Design & Development

Please Share these Notes with your Friends as well



To avoid collision / overflow there are two types of hashing techniques.

They are 1) closed Hashing
2) open Hashing.

1) closed Hashing: If any collision occured in closed hashing then those collisions can be resolve by using Seperate chaining method.

→when collision occured in closed hashing then that location only can be used to resolve that collision closedly. i.e., The collide bucket only extended to resolve the collision.

Seperate chaining Method: In seperate chaining method a chain can be created through single linked list which can be connected (or) linked to each and every bucket of the hash table.

→ When collision occured then the keys are inserted in the considered linked chain to that bucket.

→ If More keys are there to insert into the same location then the chain will be extended by assigning a new node to the next pointer variable.

→ If No more keys are there to insert then assign the next pointer to null.

Q- Insert the keys 17, 54, 27, 39, 4, 59, 9 into the hash table of size 10.

→ For key 17:

$h(k) = k \% 10$

$h(17) = 17 \% 10 = 7$

→ Go to the 7th location of the hash table and insert the key 17 in data field of the node.

For key 54:

$h(54) = 54 \% 10 = 4$

→ Go to the 4th location of the hash table and insert the key 54 in data field of the node.

For key 27:

$h(27) = 27 \% 10 = 7$

→ Go to the 7th location of the and insert the key 27 by creating a new node which linked to the next pointer of previous node.

For key 39:

$h(39) = 39 \% 10 = 9$

→ Go to the 9th location of the hash table and insert the key 39 in data field of the node.

For key 4:

$h(4) = 4 \% 10 = 4$

→ Go to the 4th location and insert the key 4 by creating a new node which linked to the next pointer of previous node.

For key 59:

$h(59) = 59 \% 10 = 9$.

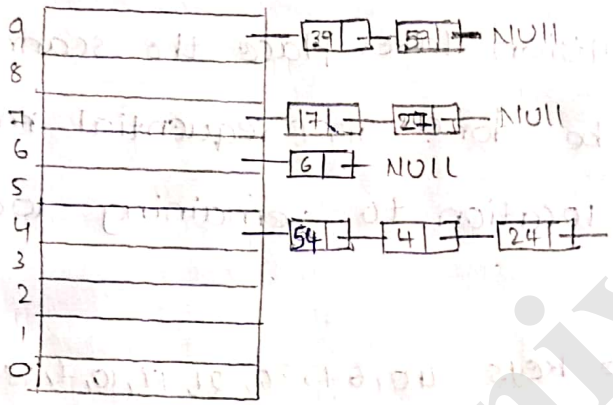
→ Go to the 9th location and insert the key 9 by creating a new node which linked to the next pointer of previous node.

For key 24: $h(24) = 24 \% 10 = 4$.

→ Go the 4th location and insert the key 24 by creating a new node which linked to the next pointer of previous node.

For key 6: $h(6) = 6 \% 10 = 6$

→ Go the 6th location of the hash table and insert the key 6 in data field of the node.



Drawbacks of seperate chaining Method:

1. The maintance of linked list for each and every bucket of the hash table is burden of the user.
2. It is very time consuming process and requires more space for the creation of nodes.

2) Open Hashing: In open hashing if any collision occures then we have to search for an open slot (empty location) to insert that collide key into the hash table.

→ The open hashing can be done by using three techniques

- Linear probing
- Quadratic probing
- Double probing.

Linear Probing:

Probe: Probe is an attempt to find out an empty location to insert the collide key into the hash table.

→ In linear probing the probing can be done in ~~search~~ sequential order to find out an empty location when collision occured.

i.e., when collision takes place the scanning of the locations can be done in sequential manner from that collide location to remaining locations in hash table

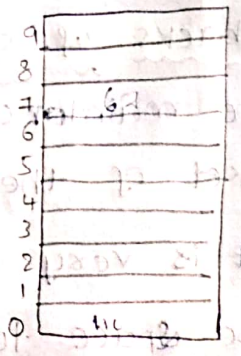
Ex:- Insert the keys 40, 67, 30, 21, 17, 10, 7, 28 into the hash table of size 10.

→ Consid

For key 40:

$h(K) = K \% D$

$= 40 \% 10 = 0$



→ Go to the 0^{th} location and insert the key 40 into hash table.

For key 67: $67 \% 10 = 7$

→ Go to the 7^{th} location and insert the key 67 into the hash table.

For key 30: $h(30) = 30 \% 10 = 0$.

→ Here the 0^{th} location already occupied by the key 40. so, it results in collision.

This collision can be resolve by finding out an empty location using linear probing to insert that collide key.

empty slot from that collide place to remaining locations from the hash table.

→ From collide location 0, the next location 1 is empty. so, the insert the collide key 30 in 1st location of the hash table.

For key 21: $h(21) = 21 \% 10 = 1$

→ Here the 1st location already occupied by the key 30. so, it results in collision.

→ From that 1st location the next open slot is 2. so, insert that collide key into 2nd location of the hash table.

For key 17: $h(17) = 17 \% 10 = 7$.

→ At 7th location collision occured. so, the next free slot from 7th location is location 8. so, insert that collide key into 8th location of the hash table.

For key 10: $h(10) = 10 \% 10 = 0$

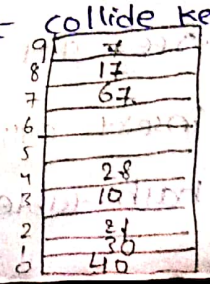
→ At 0th location collision occured. so the next free slot from 0th location is location 3. so, insert that collide key into 3rd location of the hash table.

For key 7: $h(7) = 7 \% 10 = 7$

→ At 7th location collision occured. so the next free slot location is 9. so insert that collide key into 9th location of the hash table.

For key 28: $h(28) = 28 \% 10 = 8$.

→ Here the 8th location collision occured. so, the next free slot location is 4. so insert that collide key into 4th location of the hash table.



After linear probing the hash table is

The number of probs required for the collided key values

h(30) = 1 probe

h(21) = 1 probe

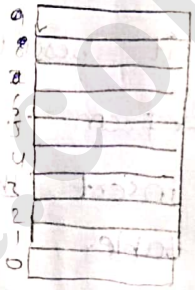
h(17) = 1 probe

h(10) = 3 probes

h(7) = 2 probes

h(28) = 6 probes

Ex:- Insert the keys 50, 35, 15, 37 into hash table of size 7.



→ For key 50: h(k) = k % D

h(50) = 50 % 7

= 1

→ Go to the 1st location of the hash table and insert the key 50 into hash table.

For key 35: h(35) = 35 % 7

= 0

→ Go to the 0th location of the hash table and insert the key 35 into hash table.

For key 15: h(15) = 15 % 7 = 1

→ Here 1st location already occupied by the key 50 so, it results collision.

So, the next free slot from 1st location is location 2. So, insert collide key 15 into 2nd location of hash table.

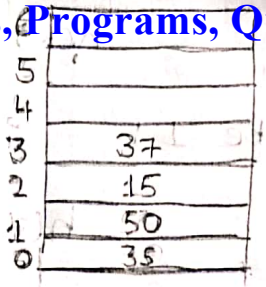
For key 37: h(37) = 37 % 7 = 2

→ Here 2nd location already occupied. So, the next free slot from 2nd location is location 3. So, insert collide key 37 into 3rd location of hash table.

the collides keys are

$h(15) = 1$ probe

$h(37) = 1$ probe.



Quadratic probing:

In Quadratic probing the number of probes to find out an empty location will be reduced by using a new hash function.

$$h(k) = (h(k) + i^2) \% D$$

where $i = 1, 2, 3, 4$

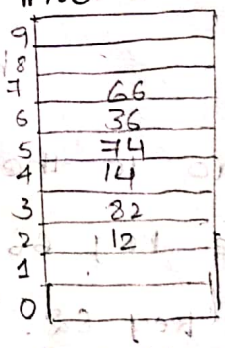
This Hash function can be used when Collision occured in the Hash table otherwise the normal hash function $h(k) = k \% D$

Can be used to insert the given keys into the hash table.

Ex:- Insert the keys 14, 36, 74, 12, 66, 82 into the hash table of size 10

→ For key 14:

$$h(14) = 14 \% 10 = 4$$



→ Go to the 4th location, and insert the key 14 in 4th location of the hash table.

For key 36: $h(36) = 36 \% 10 = 6$

→ Go to the 6th location and insert the key 36 in 6th location of the hash table.

For key 74: $h(74) = 74 \% 10 = 4$,

→ Here the collision occured at 4th location.

→ Resolve this collision using the hash function

$h(K,i) = (h(K) + i^2)$

Probe 1: $i = 1$

$h(74) = h(4 + 1^2) \% 10 = 5$

→ Go to 5th location and if it empty insert the key 74 into the hash table

For key 12:

$h(12) = 12 \% 10 = 2$

→ Go to 2nd location and if insert the key 12 into the 2nd location of hash table.

For key 66:

$h(66) = 66 \% 10 = 6$

→ Here the collision occured at 6th location.

→ Resolve this collision using hash function.

$h(K) = (h(K) + i^2) \% D$

probe 1: $i = 1$

$h(66) = (6 + 1^2) \% 10 = 7 \% 10 = 7$

→ Go to the 7th location and if it is empty insert the key 66 into the hash table.

For key 82:

$h(82) = 82 \% 10 = 2$

X

[→ Go to the 2nd location and insert the key 82 into 2nd location of hash table.] X

→ Here the collision occured at 2nd location,

→ Resolve this collision using hash function.

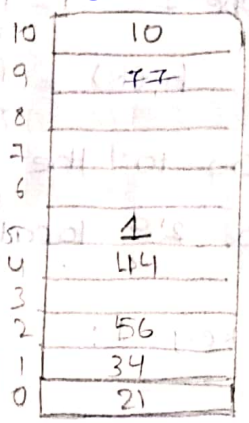
$h(K) = (h(K) + i^2) \% D$

probe 1: $i = 1$

$h(82) = (2 + 1^2) \% 10 = 3 \% 10 = 3$

→ Go to 3rd location and insert the key 82 in 3rd location of hash table.

Ex:- Insert the key 10, 21, 34, 44, 56, 77 into hash table of size 11.



For key 10:

$$h(10) = 10 \% 11 = 10$$

→ Go to the 10th location and insert the key 10 into 10th location of hash table.

For key 21:

$$h(21) = 21 \% 11 = 10$$

→ Here the collision occured at 10th location.

→ Resolve this collision using hash function.

$$i=1 \Rightarrow h_i(k) = (h(k)+i^2) \% D = (10+1^2) \% 11 = 11 \% 11 = 0$$

→ Go to the 0th location and insert the key 21 into 0th location of hash table.

For key 34:

$$h(34) = 34 \% 11 = 1$$

→ Go to the 1st location and insert the key 34 into 1st location of hash table.

For key 44:

$$h(44) = 44 \% 11 = 0$$

→ Here the collision occured at 0th location.

→ Resolve this collision using hash function.

Probe 1: $i = 1$

$$h_i(k) = (h(k) + i^2) \% D = (0+1) \% 11 = 1 \% 11 = 1$$

Probe 2: $i = 2$

$$h_i(k) = (h(k) + i^2) \% D = (0 + 2^2) \% 11 = 4 \% 11 = 4$$

→ Go to the 4th location and insert the key 44 into 4th location of hash table.

For key 56:

$$h(56) = 56 \% 11 = 1$$

→ Here the collision occured at 1st location.

→ Resolve this collision using hash function.

Probe 1: i=1

$h_i(k) = (h(k)+i^2) \% 11 = (1+1^2) \% 11 = 2 \% 11 = 2$

→ Go to the 2nd location and insert the key 56 into 2nd location of hash table.

For key 1:

$h(1) = 1 \% 11 = 1$

→ Here the collision occured at 1st location

→ Resolve this collision using hash table.

probe 1: i=1

$h_i(k) = (1+1^2) \% 11 = 2 \% 11 = 2$

probe 2: i=2

$h_i(1) = (1+2^2) \% 11 = 5 \% 11 = 5$

→ Go to the 5th location and insert the key 1 into 5th location of the hash table.

For key 77:

$h(77) = 77 \% 11 = 0.$

→ Here the collision occured at 0th location.

→ Resolve this collision using hash table.

probe 1: i=1

$h_i(k) = (0+1^2) \% 11 = 1 \% 11 = 1$

probe 2: i=2

$h_i(k) = (0+2^2) \% 11 = 4 \% 11 = 4$

probe 3: i=3

$h_i(k) = (0+3^2) \% 11 = 9 \% 11 = 9$

→ Go to 9th location and insert the key 77 into the 9th location of the hash table.

* In quadratic probing also there is no gasante to finding out an empty location when collision occured.

i.e., this technique gives only four chances to find out an empty location when collision takes place.

Double Hashing: In double hashing two hash functions are used to avoid the collisions while inserting the given keys into the hash table.

Initially all keys are inserted into the hash table using the first hash function

$$h_1(k) = k \% D$$

where K = key
D is the size of the table.

If there is any collision the double hashing provides second hash function

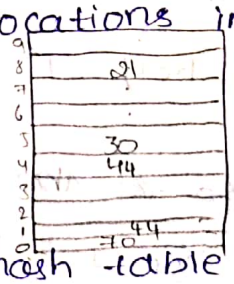
$$h_2(k) = R - (k \% R)$$

where R : The first prime < table size

To avoid the collisions.

Ex:- Insert the keys 70, 44, 30, 14, 21, 65 into the hash table of size 10.

→ Consides the hash table with 10 locations indexing from 0 to 9. and



For key 70: $h_1(70) = 70 \% 10 = 0$

→ Go to the 0^{th} location of the hash table and insert the key 70 into 0^{th} location of hash table.

For key 44: $h_1(44) = 44 \% 10 = 4$

→ Go to the 4^{th} location of the hash table and insert the key 44 into 4^{th} location of hash table.

For key 30: $h_1(30) = 30 \% 10 = 0$

→ The 0th location already occupied by the key 70. So, it results in collision.

→ To avoid this collision use 2nd hash function.

$$h_2(K) = R - (K \% R)$$

where R = 7 which is the first prime < table size

$$h_2(30) = 7 - (30 \% 7) = 7 - 2 = 5$$

→ Go 5 locations away from the collide location 0 which is location 5. So, insert the key in 5th location of the hash table.

For key 14: $h_1(14) = 14 \% 10 = 4$

→ The 4th location already occupied by the key 44 so, it results in collision.

→ To avoid this collision use 2nd hash function

$$h_2(14) = 7 - (14 \% 7) = 7 - 0 = 7$$

→ Move 7 locations away from 4th location which is location 1. So, insert the key 1st location of the hash table.

For key 21: $h_1(21) = 21 \% 10 = 1$

→ The 1st location already occupied by the key 14, so, it results in collision,

$$h_2(21) = 7 - (21 \% 7) = 7$$

→ Move 7 locations away from 1st location which is location 7. So, insert the key 7th location of the hash table.

→ The 5th location already occupied by the key 30 so, it results in collision.

$$h_2(k) = 7 - (65 \% 7) = 7 - 2 = 5$$

→ Move 5 locations away from the collide location 5 which is location 0 this is not empty.

→ so, again move 5 locations away from 0th location and check the collision is empty or not to insert collide key into hash table.

→ continue this process until an empty location will found otherwise determined that these is no chance to insert the key into the hash table

→ For key 65 these is no empty location. i.e., no chance to insert the key into the hash table.

Ex:- Insert the keys 77, 36, 14, 5, 26, 62 of size 12.

→ consider the hash table

For key 77:

$$h_1(k) = 77 \% 12 = 5$$

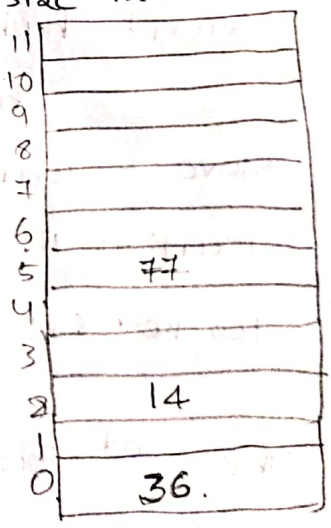
→ Go to the 5th location and insert the key 77 into 5th location of the hash table.

For key 36: $h_1(36) = 36 \% 12 = 0$

→ Go to the 0th location and insert the key 36 into 0th location of the hash table.

For key 14: $h_1(14) = 14 \% 12 = 2$

→ Go to the 2nd location and insert the key 14 into 2nd location of the hash table.



For key 5: $h_1(5) = 5 \% 12 = 5$

↳ Go to the 5th location check whether it is empty or not. Here 5th location was already occupied by the key 77, go to the next hash function.

$$h_2(5) = 7 - (5 \% 7) = 7 - 5 = 2.$$

Move 2 locations away from location 5 which is location 7. So, insert the key 5 into 7th location.

For key 26:

$$h(26) = 26 \% 12 = 2$$

The second location is already occupied by the key 14. So, go to the next hash function.

$$h_2(26) = 7 - (26 \% 7) = 7 - 5 = 2.$$

Move 2 locations above from the collide location.

For key 29:

$$h(29) = 29 \% 12 = 5$$

TO location 5 is already occupied with key 17. So, go to second hash function.

$$h_2(29) = 7 - (29 \% 7) = 7 - 1 = 6.$$

Move 6 locations away from the 5th location which is location 11. So, insert the key 29 at 11th location.

For key 62:

$$h(62) = 62 \% 12 = 2$$

The 2nd location is already occupied with the key 77 and go to next hash function.

$$h_2(62) = 7 - (62 \% 4) = 7 - 6 = 1$$

Move 1 location away from the collide location which is location 3 and insert the key 62 at 3rd location.

TutorialsDuniya.com

Download FREE Computer Science Notes, Programs, Projects, Books PDF for any university student of BCA, MCA, B.Sc, B.Tech CSE, M.Sc, M.Tech at https://www.tutorialsduniya.com

- Algorithms Notes
- Artificial Intelligence
- Android Programming
- C & C++ Programming
- Combinatorial Optimization
- Computer Graphics
- Computer Networks
- Computer System Architecture
- DBMS & SQL Notes
- Data Analysis & Visualization
- Data Mining
- Data Science
- Data Structures
- Deep Learning
- Digital Image Processing
- Discrete Mathematics
- Information Security
- Internet Technologies
- Java Programming
- JavaScript & jQuery
- Machine Learning
- Microprocessor
- Operating System
- Operational Research
- PHP Notes
- Python Programming
- R Programming
- Software Engineering
- System Programming
- Theory of Computation
- Unix Network Programming
- Web Design & Development

Please Share these Notes with your Friends as well

Drawbacks:

1. Finding of an empty location to insert a new element into the hash table when collision occur is difficult.

If more than half of the table gets full.

Rehashing:

When a new element inserted into the hash table it is very difficult in finding of an empty location by using open/closed hashing techniques.

* This problem can be achived by reconstructing a new hashtable with large size.
* In Rehashing the given keys are remapped into new constructed hash table with new hash function, of it's new increasing size.

The table can Rehash when

1) when the table gets more than half full.
2) when an insertion fails and finding of empty location.
3) when the table capacity reaches it defined load factor.

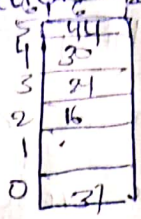
* In rehashing a new hash table can be constructed by doubling the previous size to its next prime number This is the size of new constructed hash table.

→ In this newly constructed hash-table the given keys are remapped with new table size. If any collision occured then use any open/closed hashing techniques to resolve the collision.

Collisions.

Ex:- Insert the keys 16, 24, 30, 44, 6, 31 into the hashtable of size 7.

Consider the hash-table with 7 location. Indexing from 0-6.



For key 16: $h(16) = 16 \% 7 = 2$

Go to the 2nd location insert the key 16 into the hashtable.

For key 24: $h(24) = 24 \% 7 = 3$

Go to the 3rd location insert the key 24 into the hash-table.

For key 30: $h(30) = 30 \% 7 = 2$.

→ In 2nd location already key 16 is there, it results in collision.

→ TO resolve this collision we linear probing. By using linear probing the next empty slot to insert key 30 is location 4, so insert the collide key into 4 location of hash table

For key 44: $h(44) = 44 \% 7 = 2$

In 2nd location already key 16 is there, it results in collision;

TO resolve this collision use linear probing.

By using linear probing the next empty slot to insert key 44 is location 5. so, insert the collide key into 5 location of hash-table.

For key 6: $h(6) = 6 \% 7 = 6$.

→ Go to the 6th location insert the key 6 into the hashtable.

For key 31: $h(31) = 31 \% 7 = 3$.

In 3rd location already key 24 is there, it results in collision.

To resolve this collision use linear probing. By using linear probing the next empty slot to insert key 31 into 0th location.

Here, the table is more than half full. Now reconstruct the hashtable by doubling the size with its next prime number is 17.

Table size: $7 + 7 = 14$ Prime 17

The new hash function is

$h(k) = k \% 17$, Now, re map the keys

For key 16: $h(k) = k \% 17 = 16 \% 17 = 16$.

Go to the 16th location of the hashtable & insert the key 16.

For key 24: $h(24) = 24 \% 17 = 7$

Go to the 7th location of the hashtable and insert the key 24.

For key 30: $h(30) = 30 \% 17 = 13$

Go to the 13th location of the re-hashed table and insert the key 30.

For key 44: $h(44) = 44 \% 17 = 10$

Go to the 10th location of the re-hashed table and insert the key 44.

For key 6: $h(6) = 6\%17 = 6$.

Go to 6th location and insert the key 6.

For key 31: $h(31) = 31\%17 = 11$

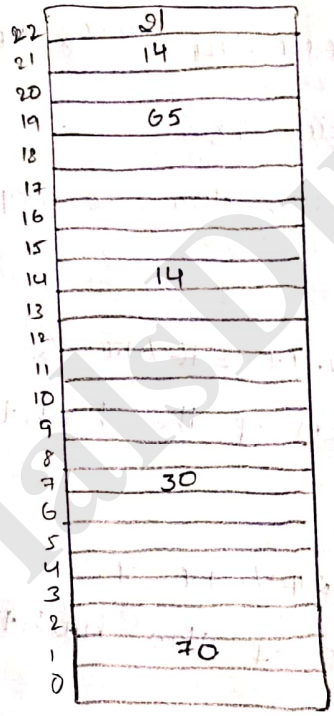
Go to the 11th location of the re-hash table and insert the key 31.

Insert the keys 70, 44, 30, 14, 21, 65 into the hash table of size 10.

As we done the insertion by double hashing previous there is no empty location for the key 65.

Now, re-construct the hash table by doubling the size with it's next prime number is 23.

Construct hash table with size of 23.



For key 70: $h(70) = 70\%23 = 1$

Go to the 1st location of the hash table and insert the key 23.

For key 44: $h(44) = 44\%23 = 21$

Go to the 21st location of the hash table and insert the key 44.

for key 30: $h(30) = 30 \% 23 = 7$.

Go to the 7th location of the hash table and insert the key 30.

for key 14: $h(14) = 14 \% 23 = 14$

Go to the 14th location of the hash table and insert the key 14.

for key 21: $h(21) = 21 \% 23 = 21$.

The 21st location is already occupied by the key so, by linear probing the next empty slot is location 22. Insert at 22nd location.

for key 65: $h(65) = 65 \% 23 = 19$.

Go to the 19th location of the re-hashed table and insert the key 65.

Drawback:-

It is very expensive to construct a new Hashtable.

Extendible Hashing: In open (or) closed hashing techniques to perform any operation it requires several access on hash table.

→ To minimize the number of access a new technique will be used which is extendible hashing

→ In extendible hashing all operations are performed with few number of disk access.

→ In extendible hashing it requires two disk access only for find operation and very few number of access for insert and delete operations.

→ This extendible hashing can be done by using directorys and keys are inserted in binary format

→ Here the directory can be divided into 2^D sub directries are roots. For each root a leaf directory required to connect. This leaf can be called as disk.

→ The capacity of each leaf or disk is also 2^D where D is the size of the directory.

→ When the insertion of binary format keys reaches if load factor of leaf then it requires to split the root into two directorys / roots by increasing its size by 1

→ After splitting the separate leafes are attached to the roots which requires the split operation and remaining all indicating with the same leafs of previous one.

Ex:- Insert the keys 110110, 010101, 111010, 111011, 101100, 110011, 111101. into directory of size 2.

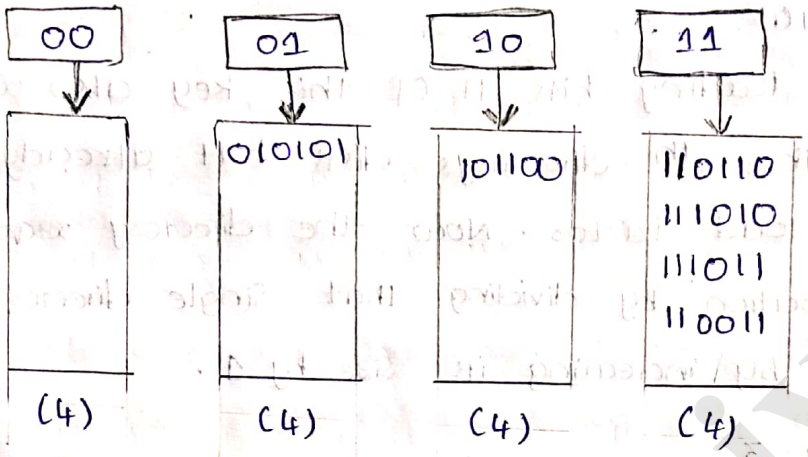
Given D=2

So, divide the directory into 2^D subdirectorys / roots which are 00, 01, 10, 11

where $2^2 = 4$

The capacity of each leaf / disk is also $2^D = 2^2 = 4$

Now the extendible hashing divides the directory as



For the insertion of given keys into the directorys match appropriate directory / root with first two leading bits with keys where the match will be found with the root insert the given key into its appropriate disk / leaf.

→ For key 110110:

The first two leading bits 11 of this key matches with the fourth directory

So, insert the key into its disk.

→ For key 010101 :

The first two leadings bits 01 of this key matches with the second directory. so, insert the key into its disk.

→ For key 111010 :

The first two leading bits 11 of this key matches with the fourth directory. so, insert the key into its disk.

→ For key 111011 :

The first two leading bits 11 of this key matches with the fourth directory. so, insert key into the disk.

→ For key 101100:

The first two leading bits 10 of this key matches with the 3rd directory. so, insert the key into the disk

→ For key 110011:

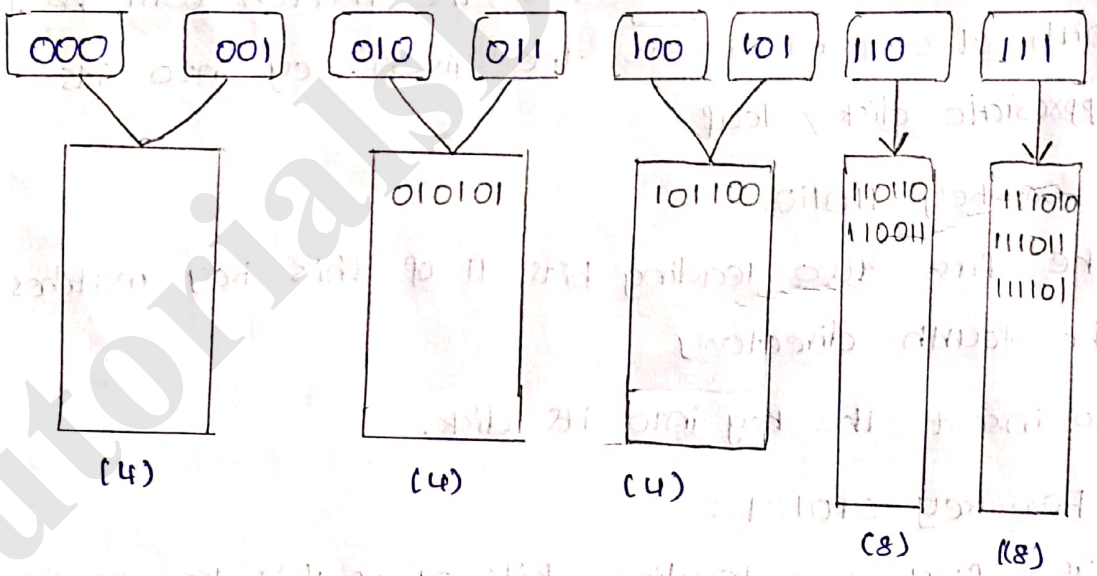
The first two leading bits 11 of this key matches with 4th directory. so, insert the key into the disk.

→ For key 111101:

The first two leading bits 11 of this key also wants to occupy the 4th directorys disk but already it reaches its load factor. Now the directory requires the split operation by dividing that single directory into two directorys by increasing its size by 1.

Now D = 3

The directory can be divided into 2^D (8) subdirectorys/ roots.



The capacity of directorys 00, 01, 10 is same of four elements and directory 11 increases its capacity from 4 to 8;

→ The given keys are remapped into the extended into extended directorys as shown in above table.

Dynamic Hashing: In Dynamic hashing the size of the hash table may grow or shrink during the performance of its operations.

Motivation of Dynamic Hashing:

To ensure good performance it requires to increase the size of the hash table where it reaches its load density of pre specified threshold

Example:

Consider the hash table of size 10 and pre specified threshold value is 4.

→ If insertion of the elements into the hash table reaches its pre specified threshold value 4 then increase the size of the hash table for good performance to insert remaining all elements without collision.

** The dynamic hashing can be done in two ways.

1. Directory less Dynamic Hashing
2. Dynamic Hashing with Directories.

1. Directory less Dynamic Hashing:

In Directory less Dynamic Hashing the hash table can be used to insert the given elements into its buckets.

→ Here the size of the hash table may grow when it reaches pre specified threshold (load factor) value (or) if any insertion fails (or). if table gets more than half full

Ex: Rehashing.

2. Dynamic Hashing with Directories:

In this technique the directories will be used instead of using the hash tables.

→ The directory is a storage location where the data can be inserted.

→ Here the directory can be divided into 2^D roots and the keys are inserted into its seperately connected ~~leafs~~ leaves where the capacity of each leave is also 2^D.

Ex:- Extendible Hashing.

Unit-II

Priority Queue (Heaps)

Priority Queues

→ Basic Model of priority Queue

→ Simple Implementations of priority Queue

→

Basic Model of priority Queue

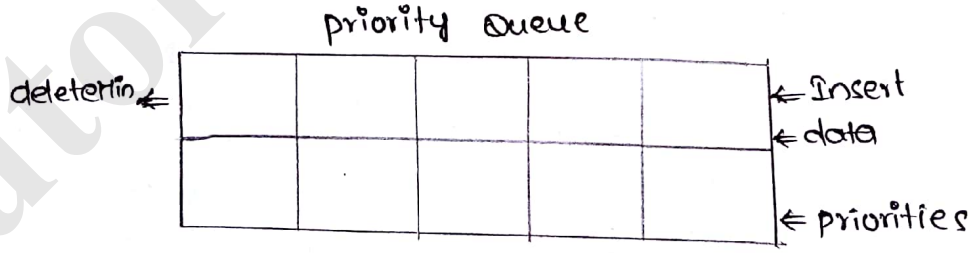
A Priority Queue performs Two operations

1) Insert
2) deletion

→ A priority Queue is a special Queue Based data structure in which the elements are inserted in order as they arrive along with the priority and the elements will be deleted Based on the priority

→ The Basic Model of priority Queue can Only perform two operations they are Insert, deleteMin.

Structure of Basic Model of priority Queue



Insert:- This operation inserts a new element into the priority queue along with priority

DeleteMin:- This operation finds and deletes Minimum element first from the priority Queue

→ The Basic Model priority Queue should support operations like deleteMax, getHighest priority, getlowestpriority etc--- So that later the Basic Model priority Queue has been Implimented with new operations

Types of priority queues

The priority Queues have been divided into the two types 1) Max priority queue

2) Min priority queue

Max priority Queue

In the Max priority queue, the elements are Inserted in the order as they arrive along with priority and the maximum element will be deleted first from the priority queue

	0	1	2	3
Data	60	90	150	100
priorities	1	2	4	3

deletion Order :-
150 100 90 60

Min priority Queue

In the Min priority queue, the elements are Inserted in the order as they arrive along with priority and the minimum element will be deleted first from the priority queue.

	0	1	2	3
Data	10	20	30	15
priorities	5	4	3	4

FCFS

Deletion order :- 10 15 20 30

Data	50	80	50	90
priorities	5	4	5	3

Deletion order :- 50 50 80 90

Simple Implementations of priority Queue

Usually the priority queues can be Implemented into four different ways

1) Arrays
2) Linked lists
3) BST
4) Binary Heap

1) Arrays Implementation of priority Queue :-

In the Arrays the elements of priority Queue can be Inserted in sequential order and deletion can be performed based on priority

Ex:- Max priority Queue

Arrival	Element	priority
1	1999	2
2	3998	5
3	2997	3
4	1996	1
5	3995	4

priority - Queue

Arrays representation

	1	2	3	4	5
Indexes →	1999	3998	2997	1996	3995
	2	5	3	1	4

Insertion order :- 1999 3998 2997 1996 3995

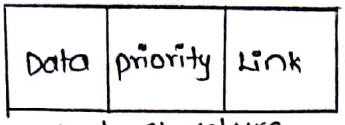
Deletion order :- 3998 3995 2997 1999 1996

Time complexity

Insertion => O(1)

Deletion => O(n)

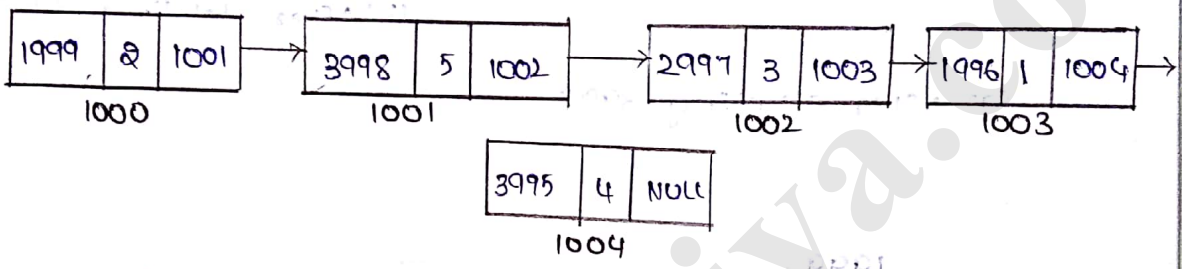
Linked list Implementation of priority Qu



Node structure

→ In this Implementation, especially we use doubly linked list in which every node contains three fields such as Data, priority, link fields

Linked List representation

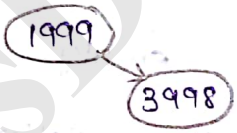


Insertion order :- 1999, 3998, 2997, 1996, 3995

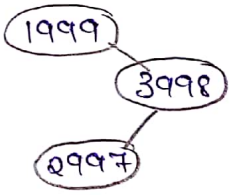
3) Binary Search Tree Implementation of priority Queues

Insertion :-

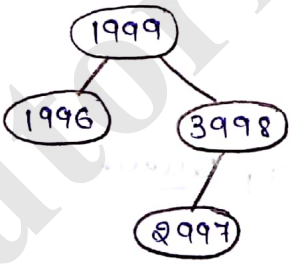
(a) Insert 1999



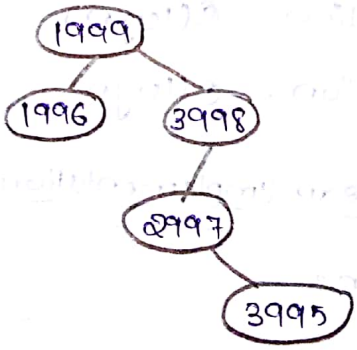
(b) Insert 3998



(c) Insert 2997



(d) Insert 1996



(e) Insert 3995

Deletion

1) Deleting the leaf node

2) Deleting the node having only one child

TutorialsDuniya.com

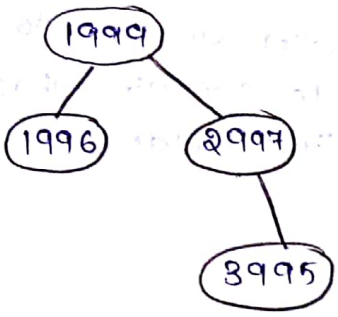
Download FREE Computer Science Notes, Programs, Projects, Books PDF for any university student of BCA, MCA, B.Sc, B.Tech CSE, M.Sc, M.Tech at https://www.tutorialsduniya.com

- Algorithms Notes
- Artificial Intelligence
- Android Programming
- C & C++ Programming
- Combinatorial Optimization
- Computer Graphics
- Computer Networks
- Computer System Architecture
- DBMS & SQL Notes
- Data Analysis & Visualization
- Data Mining
- Data Science
- Data Structures
- Deep Learning
- Digital Image Processing
- Discrete Mathematics
- Information Security
- Internet Technologies
- Java Programming
- JavaScript & jQuery
- Machine Learning
- Microprocessor
- Operating System
- Operational Research
- PHP Notes
- Python Programming
- R Programming
- Software Engineering
- System Programming
- Theory of Computation
- Unix Network Programming
- Web Design & Development

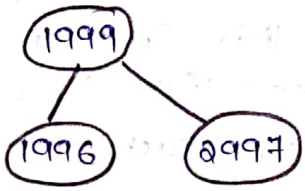
Please Share these Notes with your Friends as well

3) Deleting the node having two childs

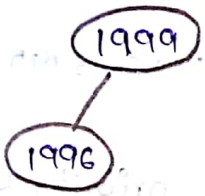
Deletion



(a) After deleting 3998



(b) After deleting 3995



(c) After deleting 2997

(d) After deleting 1999

(e) After deleting 1996

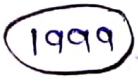
Time complexity

Insertion - O(log n)

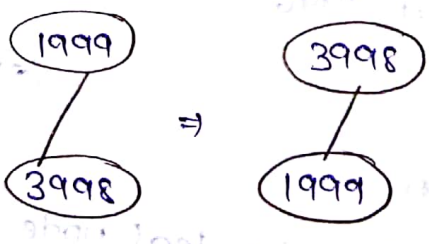
deletion - O(log n)

Binary heap implementation of priority Queue

Insertion:

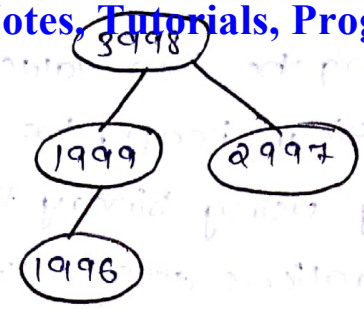


a) Insert 1999



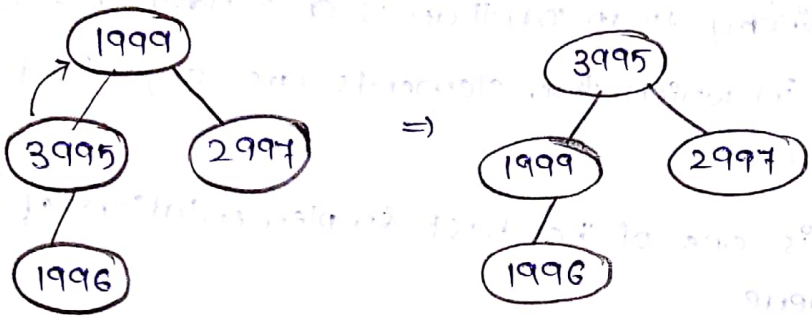
b) Insert 3998

c) Insert 2997

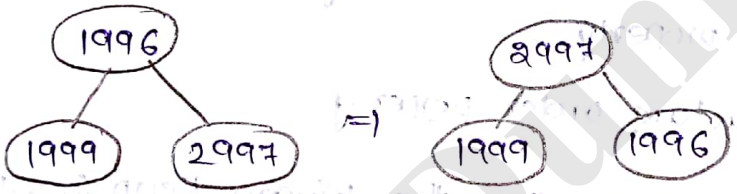


(d) Insert 1996

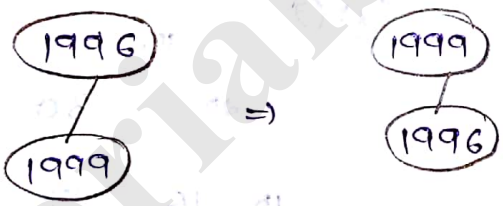
Deletion



a) After deleting 3998



b) After deleting 3995



(c) After deleting 2997

(d) After deleting 1999

e) After deleting 1996

Time complexity

1) Insertion - O(logn)

2) delelion - O(logn)

Note:- Among the four datastructures such as arrays, linked list, BST, Binary heap. the priority queue can be implimented efficiently using Binary Heap because it can perform more operations efficiently than the remaining data structures

Binary Heap

A Binary Heap (or) Heap is a Binary Tree based datastructure in which the elements are organised in Hirarechal order

→ Binary Heap is one of the best implementation of the priority queue.

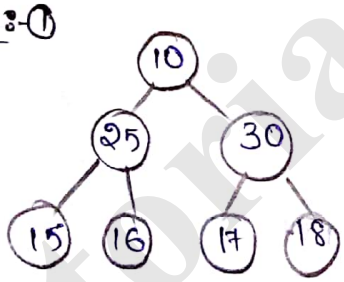
→ usually, the Binary Heap follows or posses two properties

1) Shape (or) structure order property
2) Heap order property.

Shape (or) structure order property

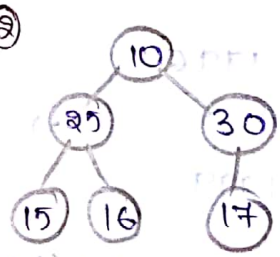
→ This property states that the Binary Heap should be either Complete (or) almost Complete Binary Tree.

Ex:-①



Complete Binary Tree

Ex:-②



Almost complete Binarytree

Heap order property

→ This property states that Every [Parent] node in the Heap is [less than (or) Equal to] (or) [greater than (or) Equal to] its children

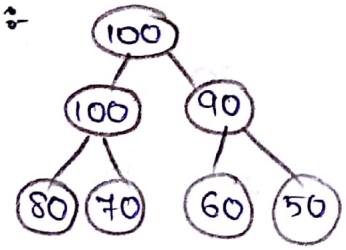
→ Depends upon the heap property, the Binary Heaps are classified into two types they are

1) Max Heap
2) Min Heap

Max Heap

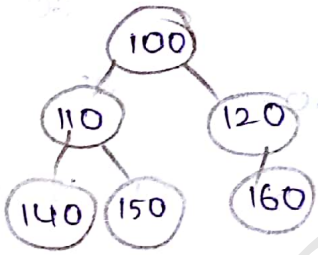
In the Max Heap Every parent node is greater than (or) Equal to its children

Eg:-



Min Heap

In the Minheap Every parent node is less than or Equal to its children.



Basic operations of Binary Heap

→ The Binary Heap has Basically two operations

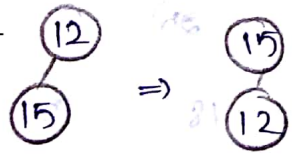
1) Insert
2) Delete

Insert :- this operation inserts a new element into the heap

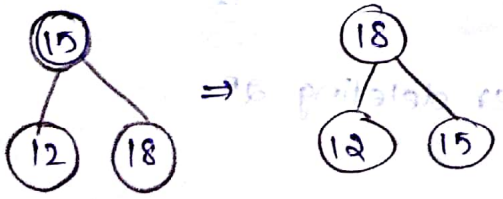
Ex :- construct a Max heap using following elements

12, 15, 18, 25, 30, 60

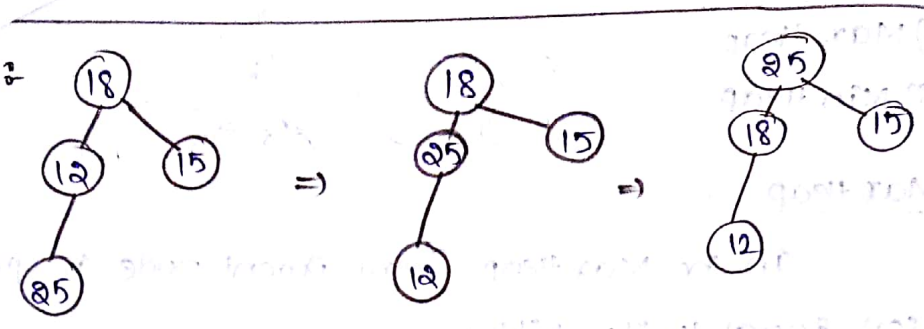
Step① :- 12

Step② :- 

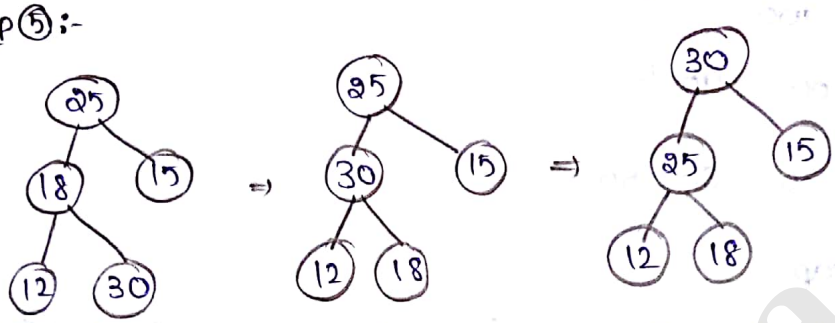
Step③ :-



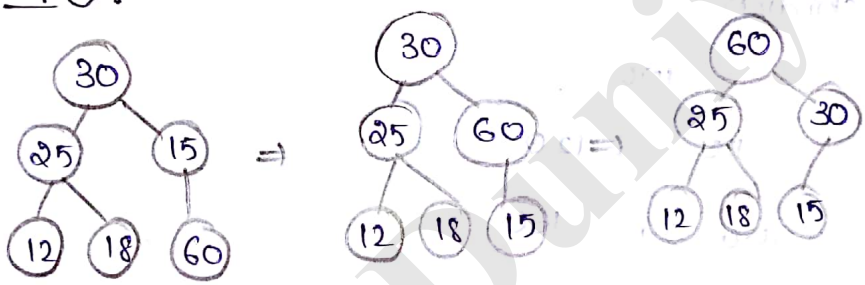
Step④ :-



Step⑤ :-



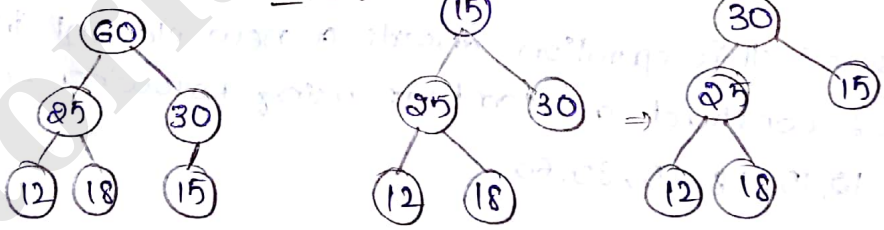
Step⑥ :-



Delete :-

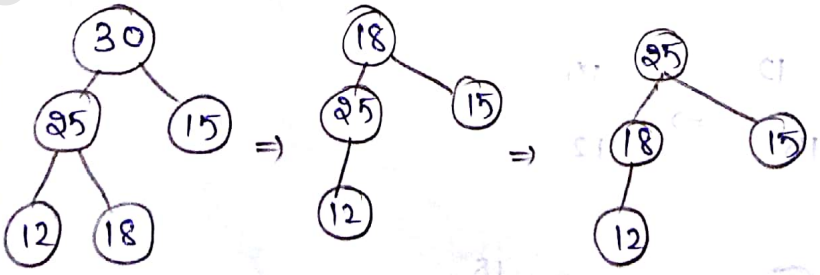
This operation deletes the ~~root~~ specific element by replacing with last element in the binary heap.

Step① :-



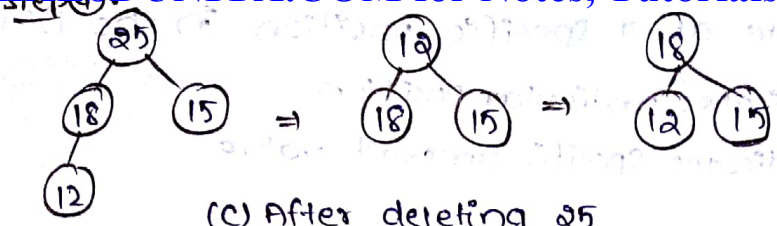
(a) After deleting 60

Step② :-



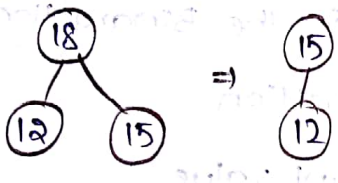
(b) After deleting 25

Step(3):-



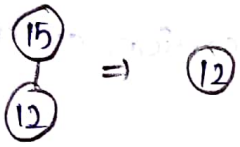
(c) After deleting 25

Step(4):-



(d) After deleting 18

Step(5):-



(e) After deleting 15

Step(6):-

empty

(f) After deleting 12

other operations of Binary Heap

Binary Heap has some other operations they are 1) getMin()

2) getMax()
3) Increase key (p, d)
4) Decrease key (p, d)
5) Build Heap()
6) deleteMin()
7) deleteMax()
8) display()

getMin() :- This operation returns minimum element in the binary heap.

getMax() :- This operation returns maximum element in the Binary heap

Increase key (p, d) :- This operation increases the value

by some value at a specific position in the Binary Heap.

where, p indicates particular position
d indicates specific amount value

(p, d)

DecreaseKey (p,d) :- This operation decreases the value by some value at a specific position in the Binary Heap.

where, p indicates particular position
d indicates specific amount value

BuildHeap() :- This operation can be used to reconstruct the binary Heap inorder to obtain Maxheap (or) Minheap while we are performing either insertion or deletion operation

deleteMin() :- This operation deletes minimum element from the Binary Heap.

deleteMax() :- This operation deletes maximum element from the Binary Heap.

Display:- this operation displays all the elements in the Binary heap

Program to illustrate Binary Heap

```
#include <iostream.h>
#include <conio.h>
int heap[100], n=0;
int insert(int,int);
int delet(int);
void display();
void main()
{
  clrscr();
  int choice, num;
  do
  {
    cout<<"\n 1. insert() "<<endl;
```

```
cout<< "2. delete()"<<endl;
cout<< "3. display()"<<endl;
cout<< "4. Exit()"<<endl;
cout<< "\n Enter your choice :";
cin>> choice;
Switch(choice)
{
case1 : cout<<"\n Enter the element to be inserted to
                the heap:";
  cin>>num;
  insert(num, n);
  n=n+1;
  break;
  case2: cout<<"\n Enter the element to be deleted
                from the heap:";
   cin>>num;
   delet(num);
   break;
   case3:
   display();
   break;
   case4: cout<<"Exit"<<endl;
   break;
   default:
   cout<<"enter a valid choice:";
   }
   }
   while(choice!=4);
   getch();
  }
 // Insertion
int insert(int num, int loc)
{
  int parent;
  while(loc>0)
  {
   parent = (loc-1)/2;
   IF(num<=heap[parent])
```

```
{
 heap[loc]=num;
 return 0;
}
 heap[loc]=heap[parent];
 loc=parent;
}
 heap[loc]=num;
}
// deletion
int delet(int num)
{
 int left, right, i, temp, parent;
 for(i=0; i<num; i++)
 {
  if(num==heap[i])
  break;
 }
 if(num!=heap[i])
 {
  cout<<"the element "<<num<<" is not found in
        the heap"<<endl;
  return 0;
 }
  heap[i]=heap[n-1];
  n=n-1;
 /* parent=(i-1)/2;
 if(heap[i]>heap[parent])
 {
  insert(heap[i], i);
  return 0;
 } */
```

```
left = 2*i+1;
right = 2*i+2;
while(right<n)
{
if (heap[i]>=heap[left] && heap[i]>=heap[right])
return 0;
if (heap[right]<=heap[left])
{
  temp=heap[i];
   heap[i]=heap[left];
   heap[left]=temp;
   i= left;
  }
else
{
  temp=heap[i];
   heap[i]= heap[right];
   heap[right]=temp;
    i=right;
  }
   left=2*i+1;
   right= 2*i+2;
   if (left==n-1)&& heap[i]
   {
     temp=heap[i];
      heap[i]=heap[left];
      heap[left]=temp;
   } */
  }
 }
 // display
   void display()
```

```
{
  int i;
  if (n==0)
  {
  cout <<"\n heap is empty"<<endl;
  }
  for (i=0; i<n; i++)
  {
    cout << heap[i] <<" ";
  }
}
```

Output :-

```
1. Insert()
2. delete()
3. display()
4. exit()
Enter your choice : 1
Enter the element to be inserted to the heap : 30
1. Insert()
2. delete()
3. display()
4. exit()
Enter your choice : 1
Enter the element to be inserted to the heap: 50
1. Insert()
2. delete()
3. display()
4. exit()
enter your choice : 1
enter the element to be inserted to the heap : 40
1. Insert()
2. delete()
3. display()
```

Output:

```
Enter your choice:1
Enter the element to be inserted to the heap: 20
1. Insert()
2. delete()
3. display()
4. Exit()
Enter your choice: 3
50 40 30 20
1. Insert()
2. delete()
3. display()
4. Exit()
Enter your choice: 2
Enter the element to be deleted from the heap: 40
1. Insert()
2. delete()
3. display()
4. exit()
Enter your choice: 3
50 30 20
1. Insert()
2. delete()
3. display()
4. exit()
Enter your choice: 4
Exit()
```

Binomial queue or heap:-

A Binomial queue is an extension of Binary heap. the Binomial queue is able to perform union or merge operation between two binomial queues very fast together with the other operations provided by the binary heap.

Structure of Binomial Queue

The Binomial queue can be structured or formed by a set of Binomial trees. A binomial tree is a general tree with special shape and also has certain properties

→ usually, the Binomial queue can be denoted by 'H' and binomial tree can be denoted by Bh where h represents height of the binomial tree

properties of Binomial tree

1) The Binomial tree (Bh) of height h has exactly 2^h no of nodes

ex:- The Binomial tree B_0 has $2^0 = 1$ node
B_1 has $2^1 = 2$ nodes
B_2 has $2^2 = 4$ nodes

2) There is atmost one Binomial tree for each height including height = 0

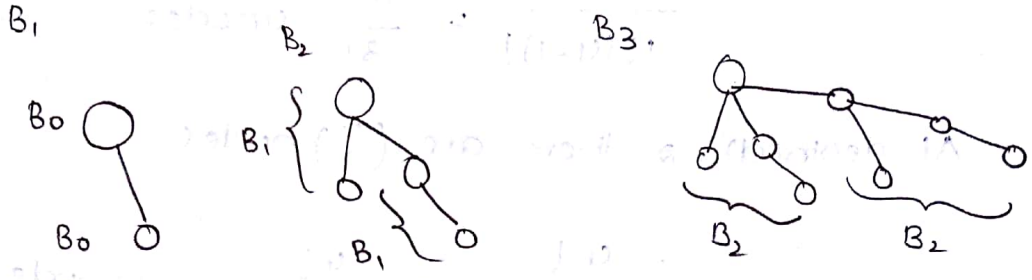
ex:-

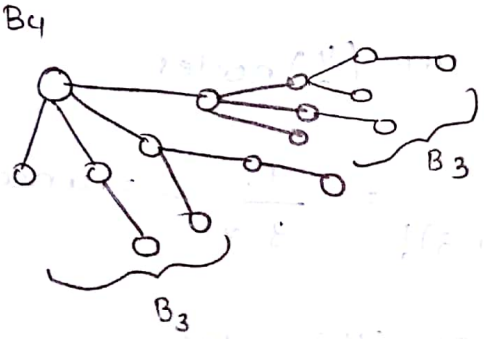
B_0 (Node)

3) The Binomial tree (Bh) can be formed by attaching one Binomial tree (Bh-1) to the root of another

Binomial tree (Bh-1)

Ex :-





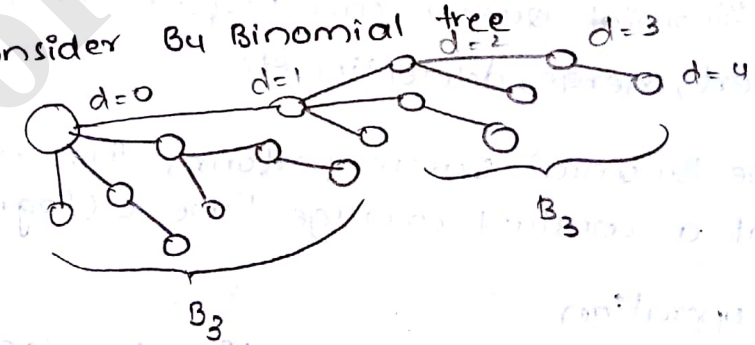
4) There are $\binom{h}{d}$ nodes at every depth(d) level of Binomial tree (Bh)

where d is the depth
h is height

$\binom{h}{d}$ is binomial coeffecient

$$\binom{h}{d} = \frac{h!}{d!(h-d)!}$$

Eg :- consider B4 Binomial tree



depth(d) = 0, 1, 2, 3, 4

At depth(d) = 0, there are $\binom{4}{0}$ nodes

$$= \frac{4!}{0!(4-0)!} = \frac{4!}{1 \times 4!} = 1 \text{ node}$$

TutorialsDuniya.com

Download FREE Computer Science Notes, Programs, Projects, Books PDF for any university student of BCA, MCA, B.Sc, B.Tech CSE, M.Sc, M.Tech at https://www.tutorialsduniya.com

- Algorithms Notes
- Artificial Intelligence
- Android Programming
- C & C++ Programming
- Combinatorial Optimization
- Computer Graphics
- Computer Networks
- Computer System Architecture
- DBMS & SQL Notes
- Data Analysis & Visualization
- Data Mining
- Data Science
- Data Structures
- Deep Learning
- Digital Image Processing
- Discrete Mathematics
- Information Security
- Internet Technologies
- Java Programming
- JavaScript & jQuery
- Machine Learning
- Microprocessor
- Operating System
- Operational Research
- PHP Notes
- Python Programming
- R Programming
- Software Engineering
- System Programming
- Theory of Computation
- Unix Network Programming
- Web Design & Development

Please Share these Notes with your Friends as well

At depth(d)=1, there are $\binom{4}{1}$ nodes

$$= \frac{4!}{1!(4-1)!} = \frac{4!}{3!} = 4 \text{ nodes}$$

At depth(d) = 2, there are $\binom{4}{2}$ nodes

$$= \frac{4!}{2!(4-2)!} = \frac{4!}{2! \times 2!} = 6 \text{ node}$$

At depth(d) = 3, there are $\binom{4}{3}$ nodes

$$= \frac{4!}{3!(4-3)!} = \frac{4!}{3! \times 1!} = 4 \text{ nodes}$$

At depth(d) = 4, there are $\binom{4}{4}$ nodes

$$= \frac{4!}{4!(4-4)!} = \frac{4!}{4! \times 1!} = 1 \text{ node}$$

5) Every Binomial tree, usually follows minheap Order property.

Operations of Binomial Queue

→ The Binomial queue can perform various operations insert, merge, deleteMin etc---

→ The Binomial Queue performs the three operations at a constant average time O(logn)

Insert operation

The Insert operation can construct a Binomial Queue by inserting a single node Binomial tree into the Binomial queue. If there are any two Binomial trees of same height in the Binomial queue then merge operation will be performed between them.

By attaching the larger root Binomial tree as a child to the smaller root Binomial tree

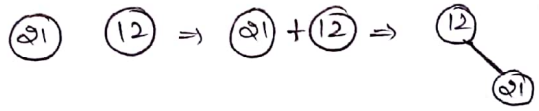
Ex:- construct a Binomial queue with the following elements.

12, 21, 24, 65, 16, 18

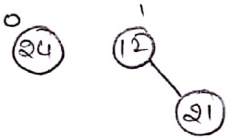
Step①:- Insert 12

H_1 : 

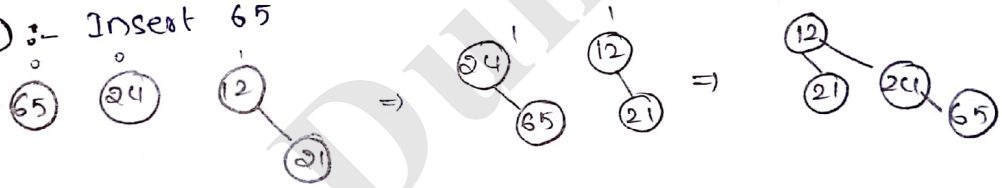
Step②:- Insert 21

H_1 : 

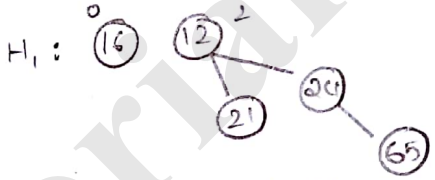
Step③:- Insert 24

H_1 : 

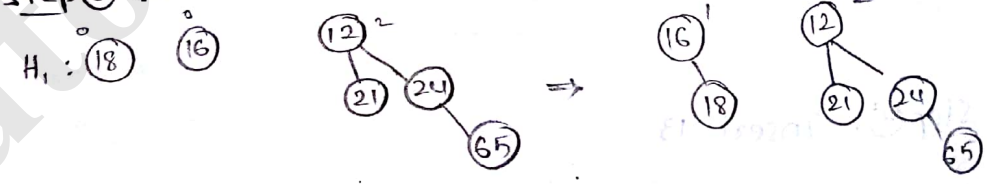
Step④:- Insert 65

H_1 : 

Step⑤:- Insert 16

H_1 : 

Step⑥:- Insert 18

H_1 : 

Ex:- construct a Binomial queue H_2 with following elements

23, 51, 24, 65, 14, 26, 13

Step①:-

H_2 : 

Step ③ :- Insert 51

H_2 : 23 51 → 23 / 51

Step ③ :- Insert 24

H_2 : 24 23 / 51

Step ④ :- Insert 65

H_2 : 65 24 23 / 51 ⇒ 24 / 65 23 / 51 ⇒ 23 / 51 24 / 65

Step ⑤ :- Insert 14

14 23 / 51 24 / 65

Step ⑥ :- Insert 26

14 26 23 / 51 24 / 65

⇓

14 / 26 23 / 51 24 / 65

Step ⑦ :- Insert 13

13 14 / 26 23 / 51 24 / 65

Merge operation:

→ The Merge Operation not Only merges the Binomial Trees of same height in a single Binomial Queue but also merges Binomial Trees of same height in two different Binomial Queues.

→ Merging two different Binomial queues is very easy operation.

Procedure steps

1) Add the corresponding Binomial trees of same height in two Binomial Queues and place the resulting binomial tree into new Binomial queue.

2) Compare the binomial trees height from "0 to max height" in both binomial queues.

case(i) :- If neither binomial Queue has the binomial tree of same height then skip merge operation and moves to compare next height.

case(ii) :- If one binomial queue has a binomial tree of height 'h' and other does not have them leave it and place it into new binomial queue

case(iii) :- If there are two binomial trees of same height in both binomial queues then the larger root binomial tree is attached as child to the smaller root binomial tree and then place the resulting binomial tree into the new binomial queue.

3) Repeat the merge process untill the max height binomial trees are encountered in both binomial queues.

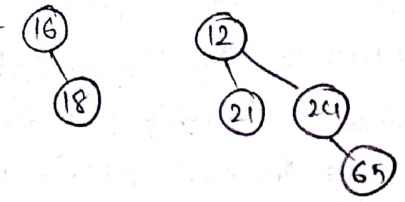
Ex :- consider two binomial queues H_1 & H_2 with six & seven trees are encountered in both binomial queues.

Let H_3 be the new Binomial queue to store the result of merge operation between H_1 & H_2

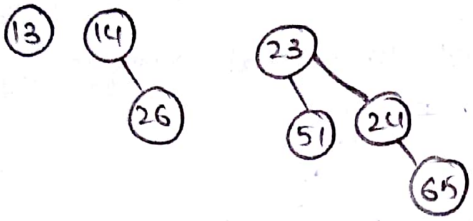
H_1 :- 12, 21, 24, 65, 16, 18

H_2 :- 23, 51, 24, 65, 14, 26, 13

H_1 :-



H_2 :-

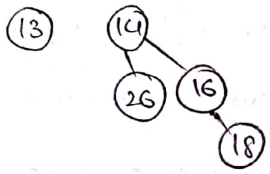


Step(1) :-

H_3 :-

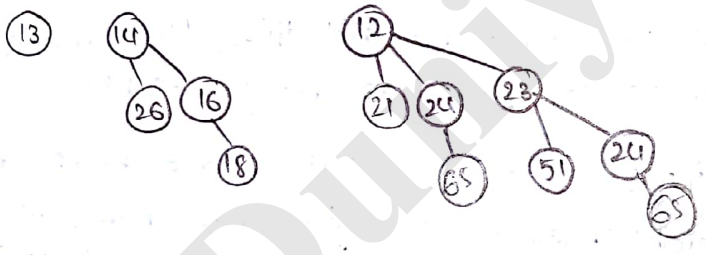
Step(2) :-

H_3 :-



Step(3) :-

H_3 :-

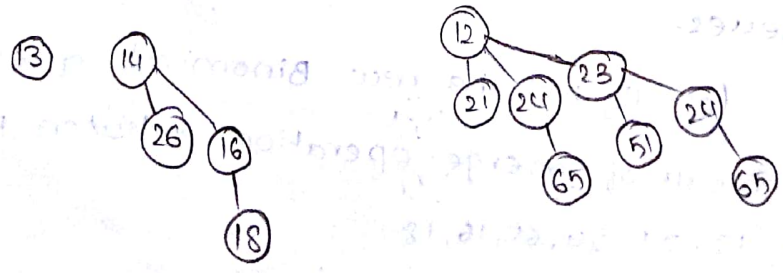


deleteMin operation

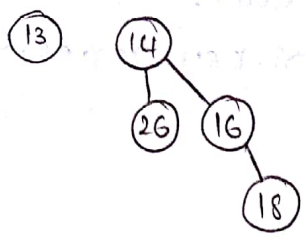
→ The deleteMin operation deletes the minimum element from the Binomial queue by breaking the original Binomial queue into two temporary Binomial queues and then merge operation will be performed between the temporary Binomial queues and the resultant Binomial trees are placed into original Binomial queue.

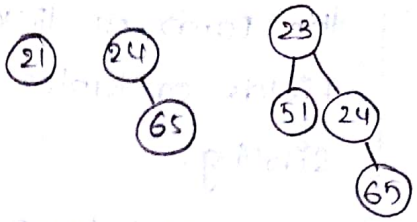
Eg:- let H_3 be original Binomial Queue and H_3', H_3'' are the two temporary Binomial Queues of H_3

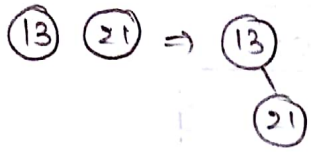
H_3 :

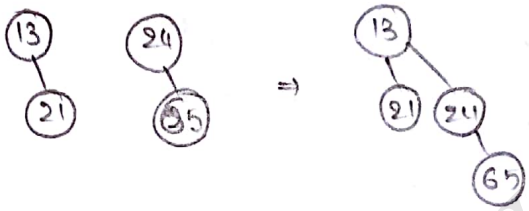


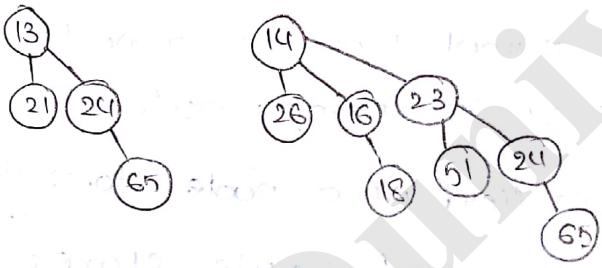
Step ①:- delete 12

H_3' :- 

H_3'' :- 

Step ②:- H_3 :- 

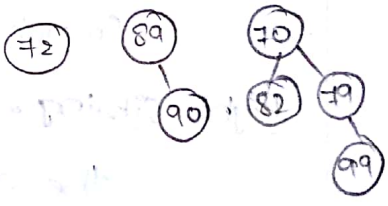
Step ③:- H_3 :- 

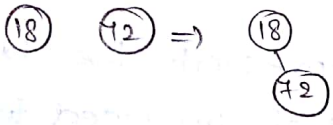
Step ④:- 

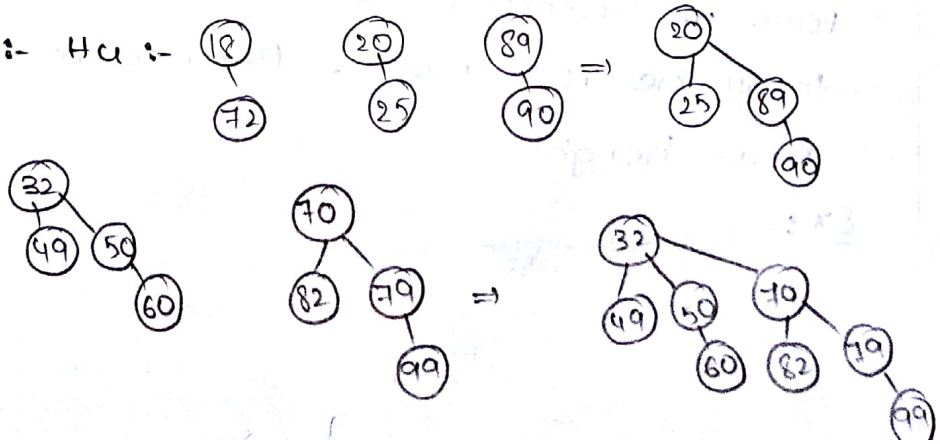
H_4 :- 

Step ①:- delete 15

H_4' :- 

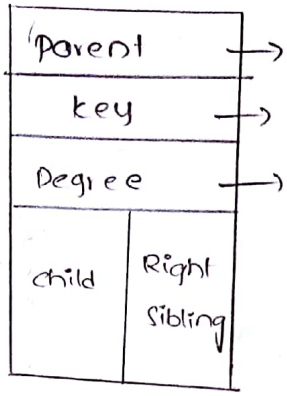
H_4'' :- 

Step ②:- H_4 :- 

Step ③:- H_4 :- 

→ Usually the binomial queue can be represented in the form of linked list where each node contains 5 fields or tuples such as parent, key, Degree, child, Right sibling.

Node Structure



Parent:- The parent field of a node points are stores the address of its parent node.

key:- the key field of a node stores the actual element

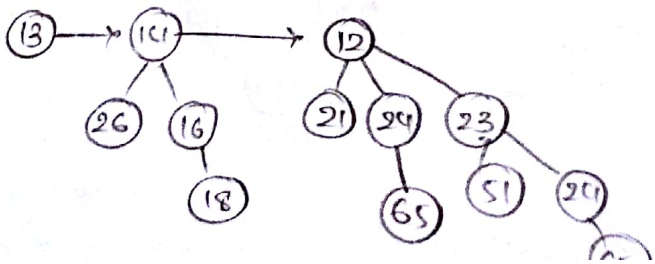
degree:- the degree of a node stores the total no. of children of that specific node.

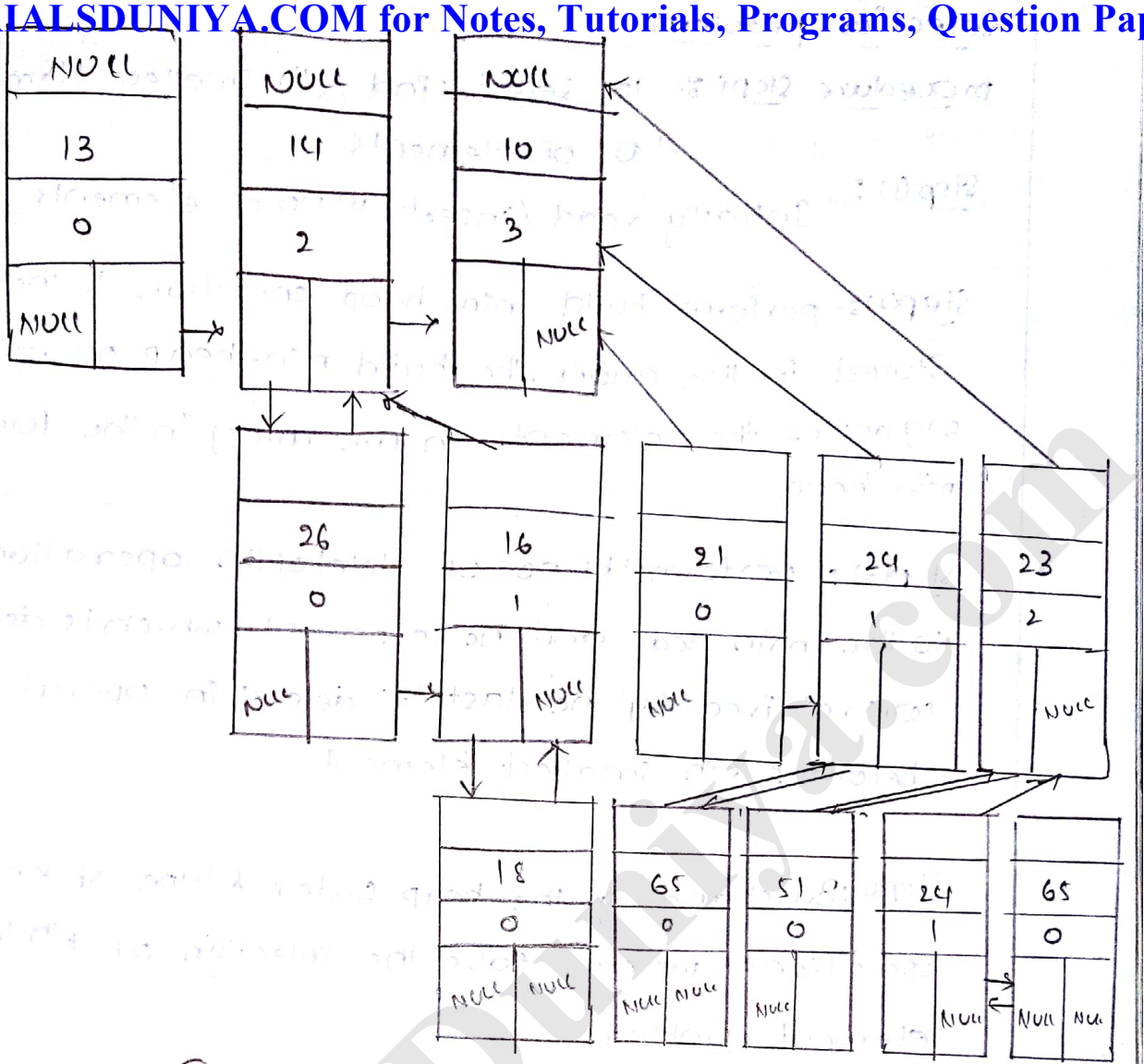
Child:- The child field points are stores the address of any of child node of specific node (most probably from the left child)

Right sibling:- the Right sibling field points or stores the address of right sibling node of specific node.

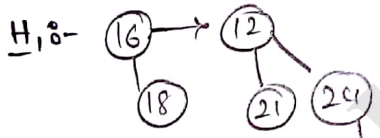
→ In order to represent the binomial queue in the form of linkedlist we need to make corrections to all the root nodes from zero height to max height.

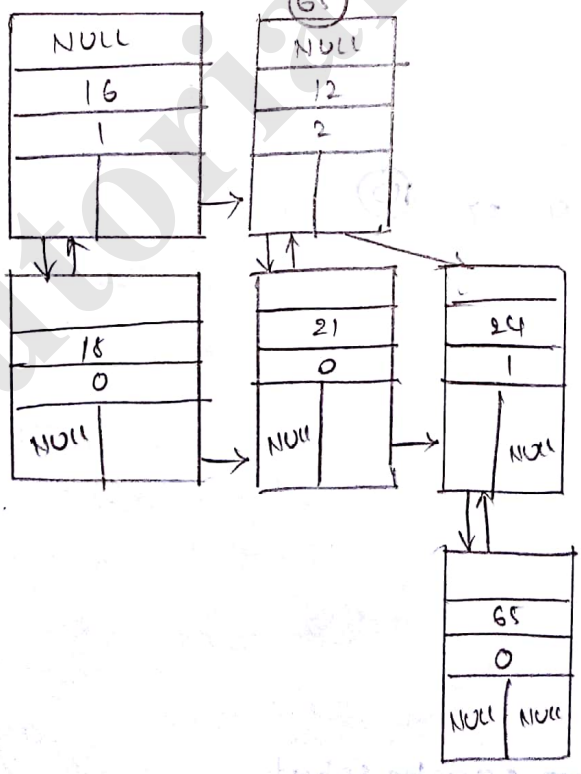
Ex:- H_3:





H,8-





Applications of priority queue

In General, priority queue can be used in various applications there are 6 applications

1) Graph algorithm (Prims, kruskal)
2) heap sort
3) operating system
4) Huffman coding
5) selection problem
6) event simulation problem

Selection problem

procedure steps :- To select/find kth smallest element in 'N' of elements

Step(1) :- Intially read / Insert 'N' no of elements into array

Step(2) :- perform build min-heap operation to the elements stored in the array. the build min-heap operation organizes the elements in the array in the form of min-heap.

Step(3) :- perform 'k' no of deleteMin operations to the min heap and the element which is deleted (or) retrived by the last 'k' deleteMin operation becomes kth smallest element

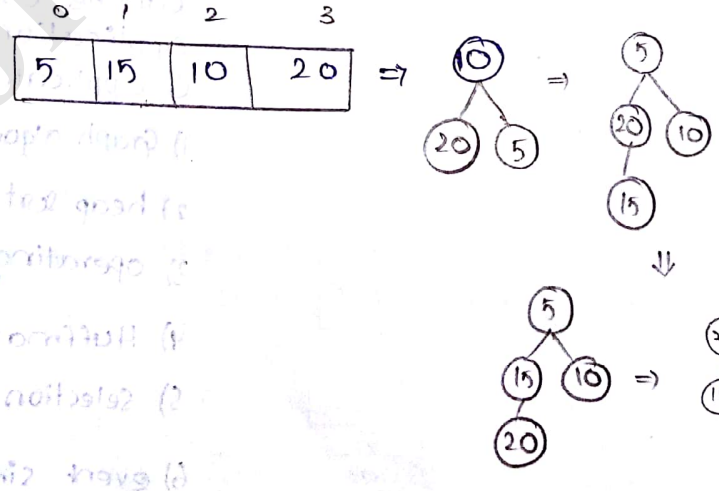
Step(4) :- By changing the heap order k type of k no. of operations. we can solve the selection of k^{th} largest element problem.

Example :-

N = 4

k = 2

10, 20, 5, 15



→ The selection problem can be solved

Time complexity O(klog(n))

Efficient Binary Search Trees

AVL TREE

An Avl tree is "height Balanced Binary Search Tree" which was invented by two Foreign mathematicians Gm Adelson-velski and Em Landis in year 1962

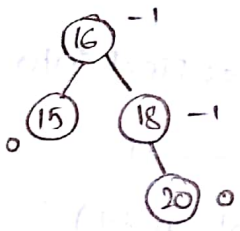
→ A BST is said height Balanced search tree if and only if the Balance at every node in tree is either -1 (or) 0 (or) 1

→ The Balance factor of a node in BST can be determine the height difference between left subtree and right subtree

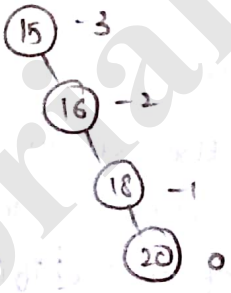
→ **Balanced Factor = height of left subtree − height of right subtree**

L-R

Ex 8-



AVL Tree



not AVL Tree

Time complexity :-

Best case → O(1)
Average case → O(logn)
Worst case → O(logn)

} Insertion deletion search operations

→ In other words, an AVL Tree is height balanced binary search Tree in which the balanced factor of every node is -1 (or) 0 (or) 1

→ The AVL tree may become imbalance while performing insertion and deletion operation

→ In order to balance the imbalanced AVL Tree we perform certain operations which are known as rotations.

Rotation :-

The rotation is the process of moving imbalanced nodes either left side or right side in AVL Tree

Basically the rotations have been classified into two catogeries they are 1) Single rotation

2) Double rotation < LR rotation / RL rotation

1) Single rotation

The Single rotation is classified into two types

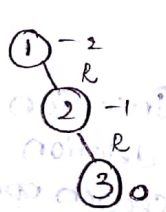
They are → 1) LL rotation (left-left)

2) RR rotation (right-right)

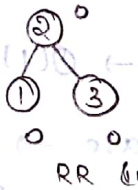
LL-Rotation

LL rotation is applied on AVL Tree when RR imbalance is existed usually the RR imbalance is occured when a node is inserted as the right child to the right subtree of a specific node in the AVL Tree.

Ex :-



RR imbalance occurred. so LL is applied

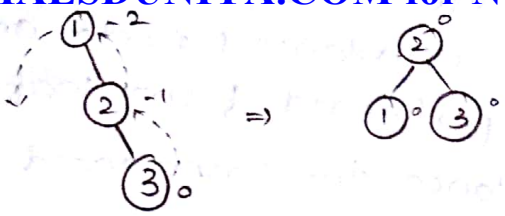


RR imbalance

Using LL rotation it becomes Balanced Binary Tree

Imbalanced Binary Search Tree

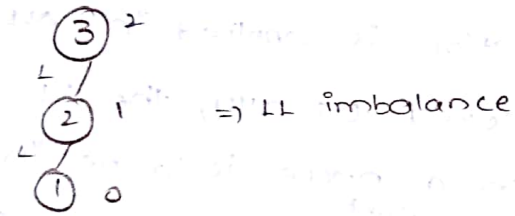
→ The LL rotation moves the nodes in the imbalance part of AVL Tree one position left from the current position (anti clock wise direction)



RR rotation

RR rotation is applied on AVL Tree when LL imbalance is existed usually the LL imbalance is occured when a node is inserted as the left child to the left subtree of a specific node in AVL Tree

Ex :-



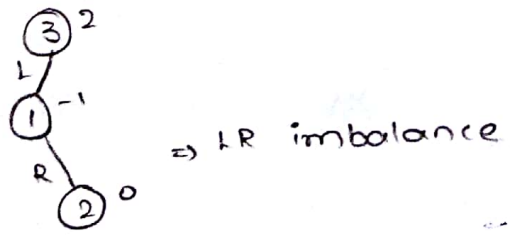
→ The RR rotation moves the nodes in the imbalance part of AVL Tree one position right from the current position (clock wise direction)

LR rotation

The LR rotation is applied on the AVL Tree when LR imbalance is existed usually the LR imbalance is occured when a node is inserted as the right child to the left subtree of a specific node in AVL Tree

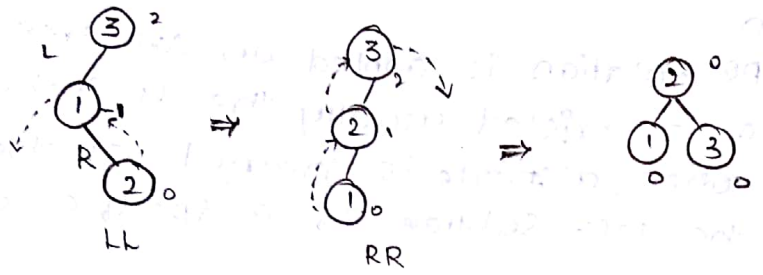
16 days

Ex :-



→ the LR rotation performs two operations successively such as LL Rotation and RR rotation

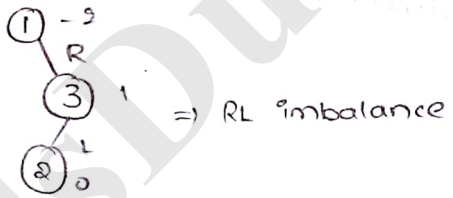
→ In the LR rotation, first the LL rotation is applied from leaf node to child node of unbalanced node and then the RR rotation can be performed from leaf node to imbalanced node. To balance the imbalanced AVL Tree



RL rotation

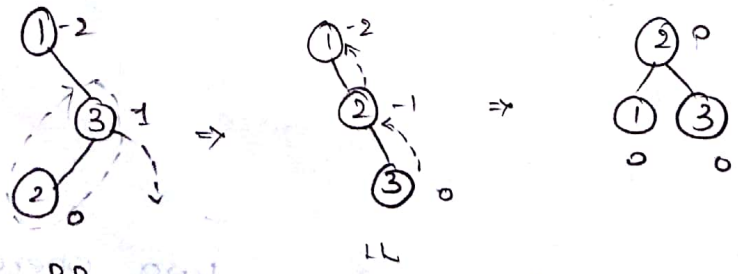
The RL rotation is applied in AVL Tree when RL imbalance is occured usually, the RL imbalance may be occured when a node is to be inserted as the ~~right~~ left child to the right subtree of a specific node in AVL Tree

Example :-



→ The RL rotation performs two operations successively such as RR rotation and LL rotation

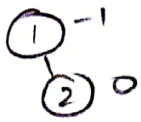
→ In the RL rotation, first the LL rotation is applied from leaf node to child node of unbalanced node and then the RR rotation can be performed from leaf node to imbalanced node. To balance the imbalanced AVL Tree



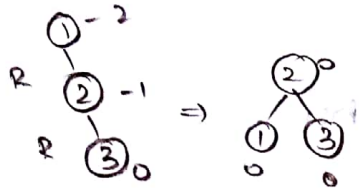
Design an AVL tree by using 1 to 9 elements.

Insert 1 :-

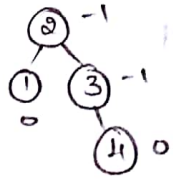
Insert 2 :-



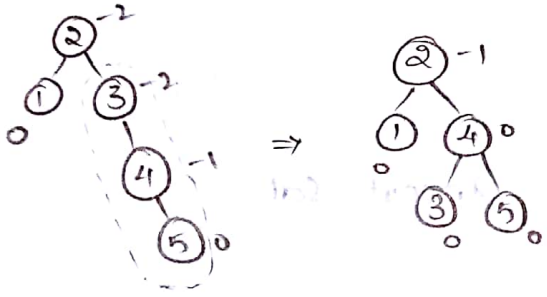
Insert 3 :-



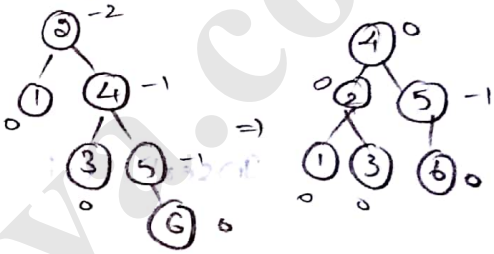
Insert 4 :-



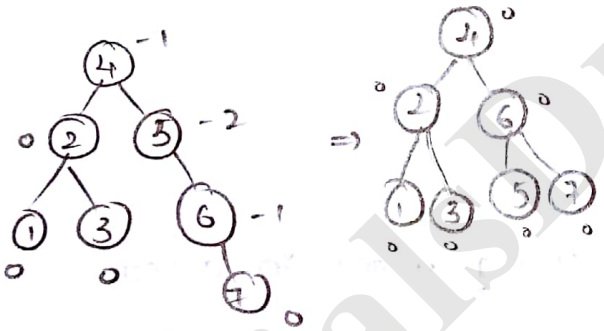
Insert 5 :-



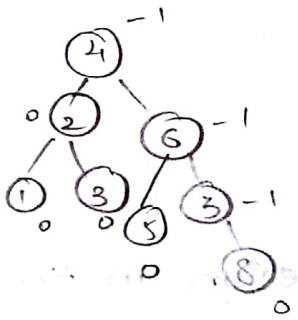
Insert 6 :-



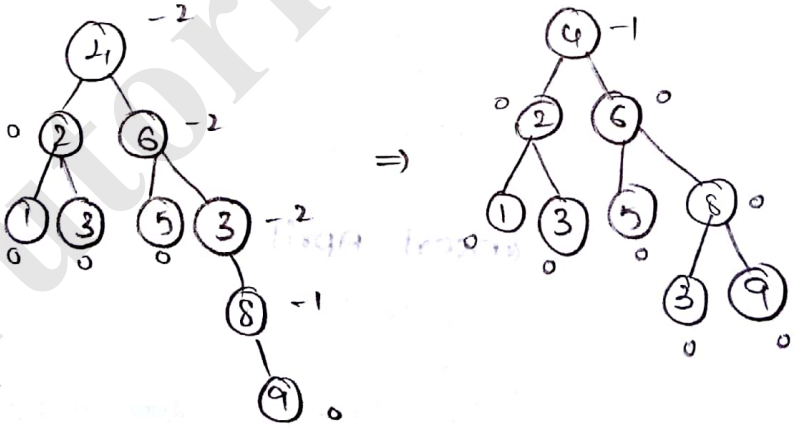
Insert 7 :-



Insert 8 :-



Insert 9 :-



TutorialsDuniya.com

Download FREE Computer Science Notes, Programs, Projects, Books PDF for any university student of BCA, MCA, B.Sc, B.Tech CSE, M.Sc, M.Tech at https://www.tutorialsduniya.com

- Algorithms Notes
- Artificial Intelligence
- Android Programming
- C & C++ Programming
- Combinatorial Optimization
- Computer Graphics
- Computer Networks
- Computer System Architecture
- DBMS & SQL Notes
- Data Analysis & Visualization
- Data Mining
- Data Science
- Data Structures
- Deep Learning
- Digital Image Processing
- Discrete Mathematics
- Information Security
- Internet Technologies
- Java Programming
- JavaScript & jQuery
- Machine Learning
- Microprocessor
- Operating System
- Operational Research
- PHP Notes
- Python Programming
- R Programming
- Software Engineering
- System Programming
- Theory of Computation
- Unix Network Programming
- Web Design & Development

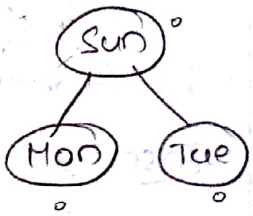
Please Share these Notes with your Friends as well

Design an AVL Tree by using week (7days)

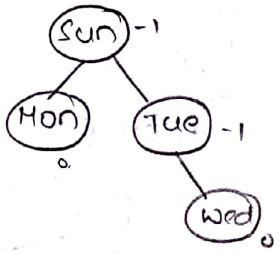
Insert sun

Insert Mon

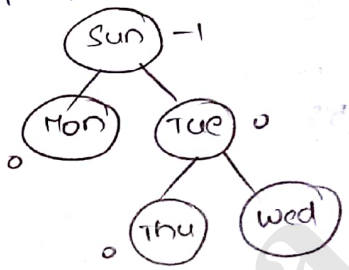
Insert Tue



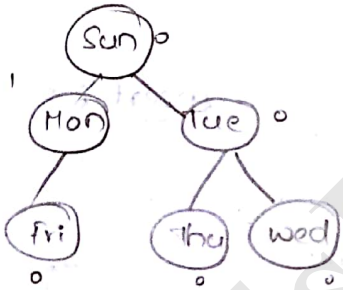
Insert wed



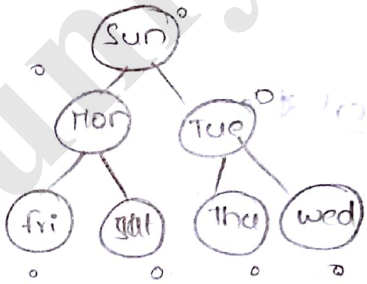
Insert Thu



Insert fri



Insert Sat

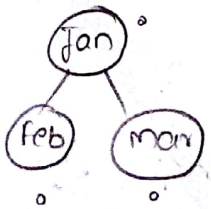


Design an AVL Tree by using moths in a year

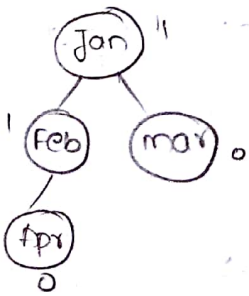
Insert Jan

Insert Feb

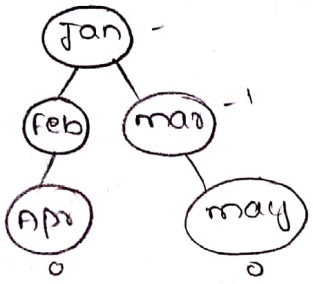
Insert Mar



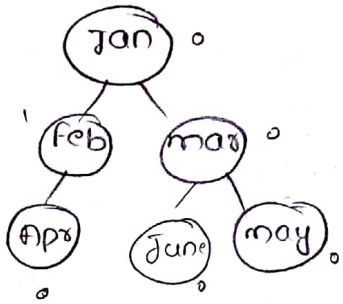
Insert April



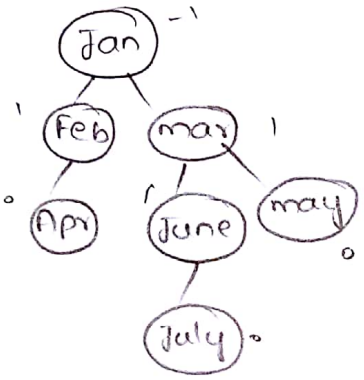
Insert may



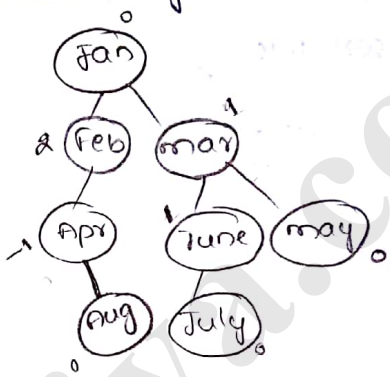
Insert June



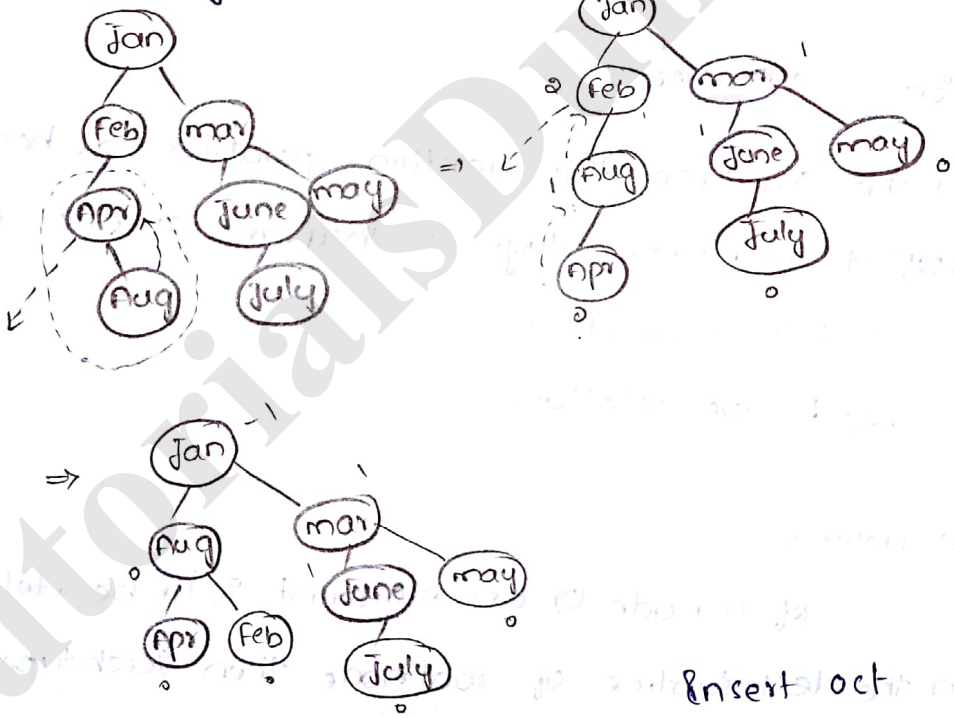
Insert July



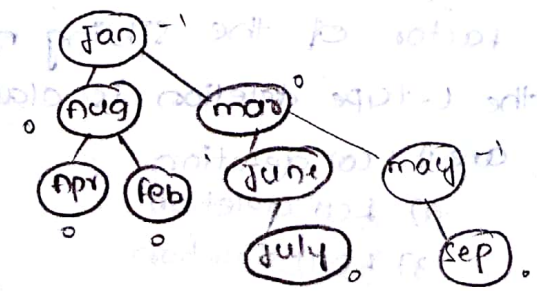
Insert Aug



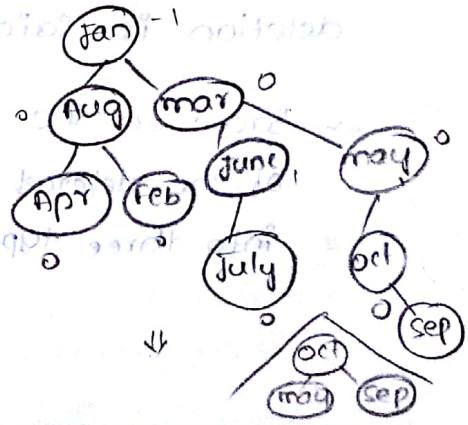
Insert Sep



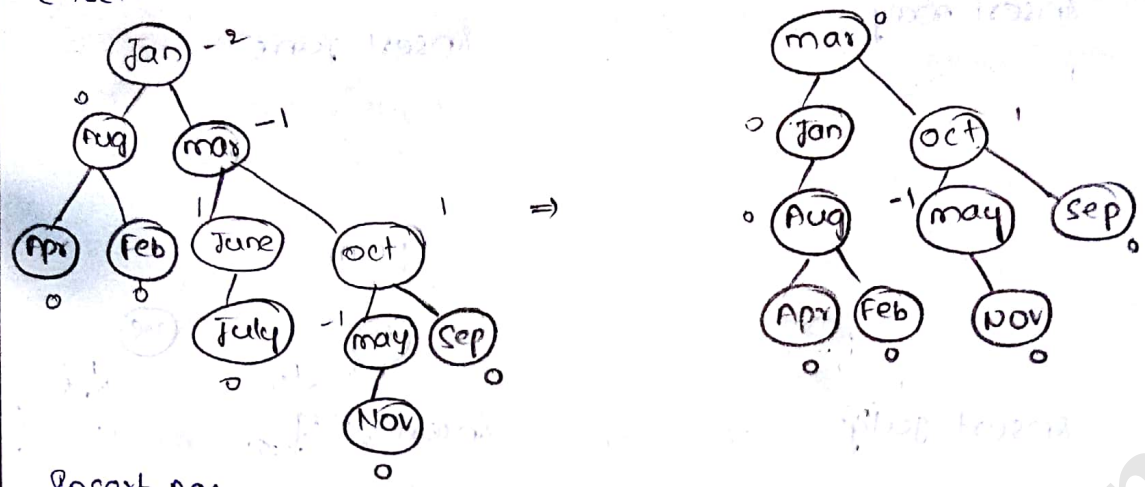
Insert sep



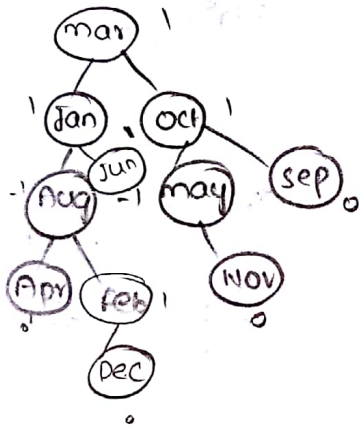
Insert oct



Insert Nov



Insert Dec



Deletion in AVL Tree

→ In the AVL Tree, the deletion operation has been classified into two catogeries they are as follows

(1) L-Type deletion

(2) R-type deletion

L-Type deletion

If a node or an element is to be deleted from the left subtree of root node then Such type of deletion is said to be L-Type deletion in the AVL Tree.

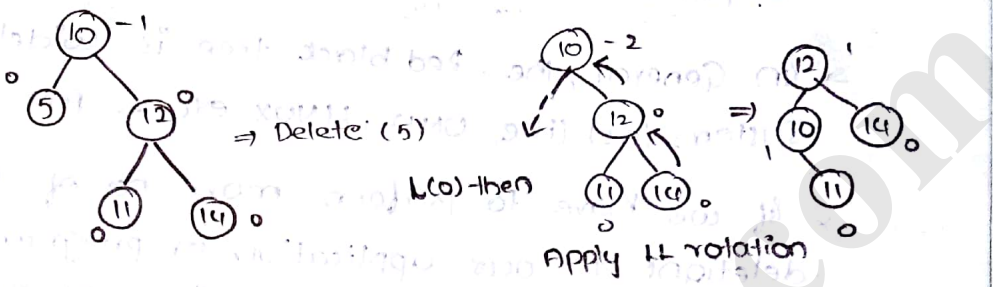
→ Based on the Balance factor of the sibling node of the deleted node, the L-type deletion is classified into three types they are 1) L(0) deletion
2) L(1) deletion
3) L(-1) deletion

L(0) deletion ⇒ Apply LL Rotation

L(1) deletion ⇒ Apply RL Rotation

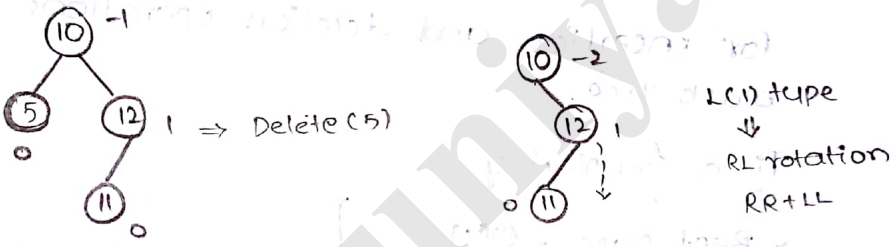
L(-1) deletion ⇒ Apply LL Rotation

Example (1) L(0) type



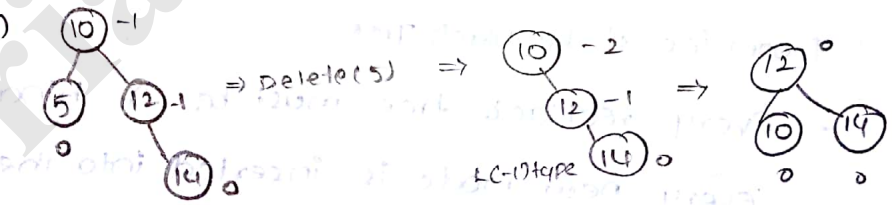
Example (2)

L(1) type



Example (3)

L(-1) type



R-Type deletion

If a node or an element is to be deleted from a right subtree of root node then such type of deletion is said to be R-Type deletion in AVL Tree

→ Based on the Balance factor of the sibling node of the deleted node, the R Type deletion is classified in three types 1) R(0) deletion ⇒ Apply RR rotation

2) R(1) deletion ⇒ RR rotation

3) R(-1) deletion ⇒ LR rotation

Redblack Tree

→ Redblack Tree is another efficient balanced binary Search Tree in which nodes are coloured either red or black

→ The Redblack Tree was invented by one of the German Computer Scientist Rudelf Bayer in the year 1972

→ In General, the Red black tree is widely used in Operating System (OS) like UNIX, LINUX etc--- for Sheduling process

→ If we have to perform more no of Insertions and deletions in our application or program, we can prefer to use redblack tree. Otherwise we can use AVL Tree because AVL Tree may contain more no of Iterations for Insertion and deletion operations than the Red black Tree.

Time Complexity

→ Best case - O(1)
Worst case - O(logn)
average case - O(logn)
} for Insertion, deletion, Search operations

→ Based on properties of Redblack Tree the node can be coloured either red or Black in the Redblack Tree

Properties of Redblack Tree

→ Every redblack tree must be a binary Search Tree

→ every new node is inserted into the redblack tree with red colour

→ the root node of redblack tree should be coloured as Black

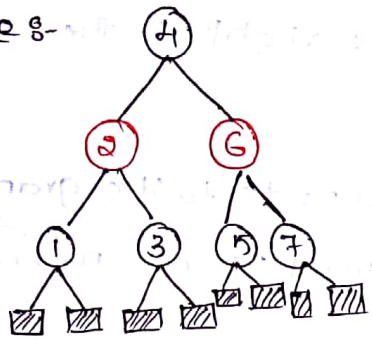
→ The children of every red node are Black

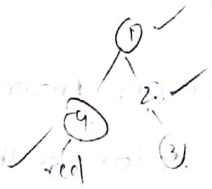
→ There should be same (or) equal no of black nodes in each path of the redblack tree

→ There should not be two consecutived red nodes but there may be two consecutive black nodes in every path of redblack tree

→ The null nodes in the redblack tree should be colored as black.

Example :-





Insertion operation in Redblack Tree

Case(i) :- Inorder to insert node (or) element into redblack Tree there are 5 possible cases widely followed.

case (i) :- If new node's parent's sibling node colour is red.

(1) change the colour of parent, grandparent & parent's sibling node of new node

case (ii) :- If new node's parent's sibling colour is black (or) null node and the new node is left to its parent then (newnode's parent is left to newnode's grandparent)

① Apply RR rotation from new node to its grand's parent

② change the colour to the grand parent, parent of new node.

Case (iii) :- If new node's parent sibling color is black (or) null node and new node is right child to its parent & newnode's parent is right to newnode's grandparent

① Apply LL rotation from new node to its grand parent node

② change the color to the grand parent, parent of new node.

case (iv) :- If new node's parent sibling node colour is black (or) null node and the new node is right to its parent & new node's parent is left to the newnodes grand parent

① Apply LR rotation from new node to its grand parent

② change the color to the grand parent and parent of new node

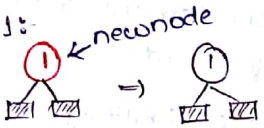
Case (5) :- If new node's parent's sibling node color is black (or) null node and the newnode is left to its parent and now node's parent is right to the new node's grand parent

① Apply RL rotation from newnode to it's grand parent

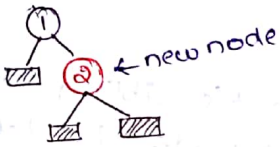
② Change the color to the grand parent, and new node

Construct a Red-black tree using 1 to 9 numbers

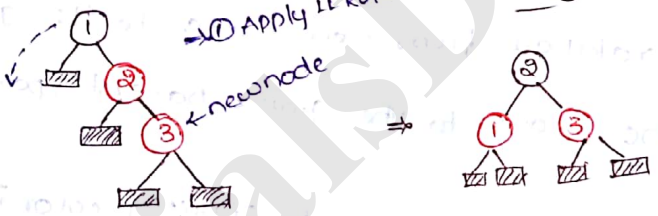
Insert 1 :-



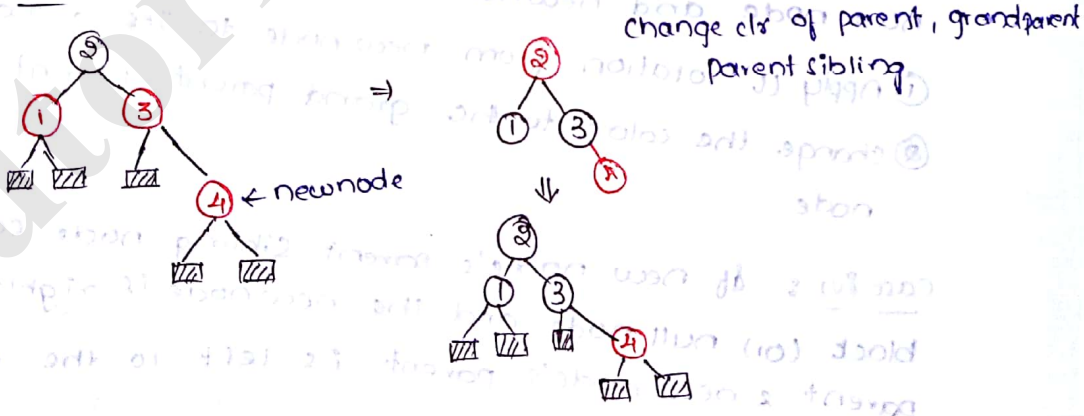
Insert 2 :-



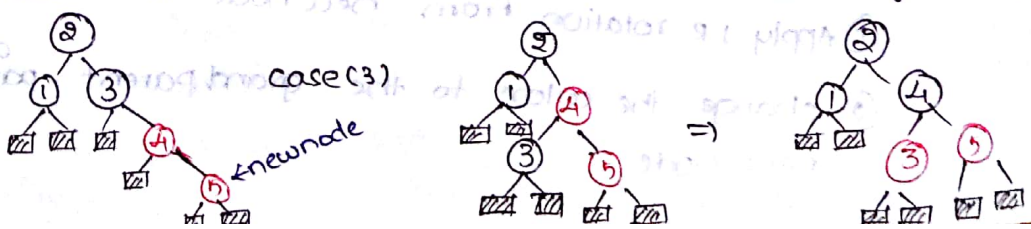
Insert 3 :-



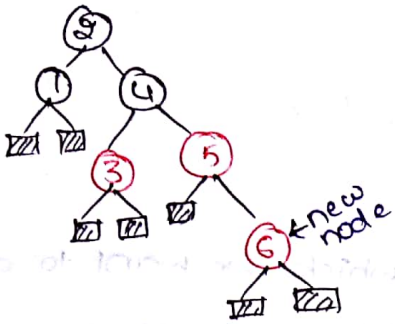
Insert 4 :-



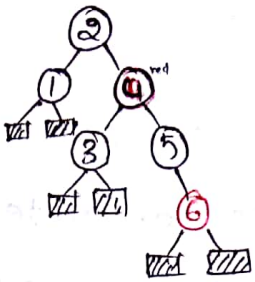
Insert 5 :-



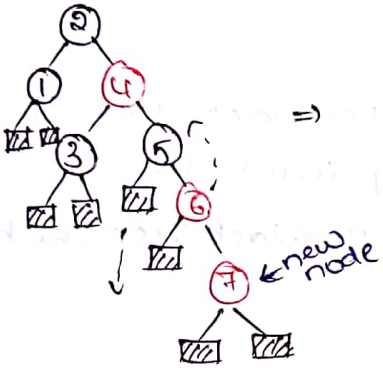
Insert 6 :-



case (i)

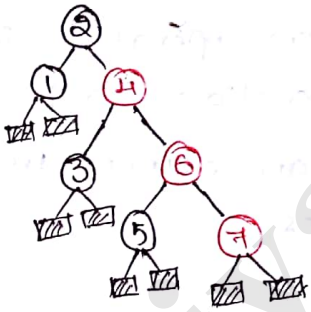


Insert 7 :-

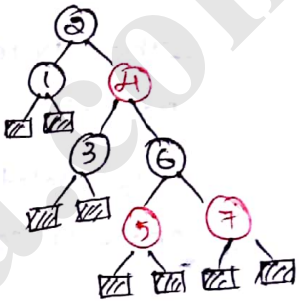


case (iii)

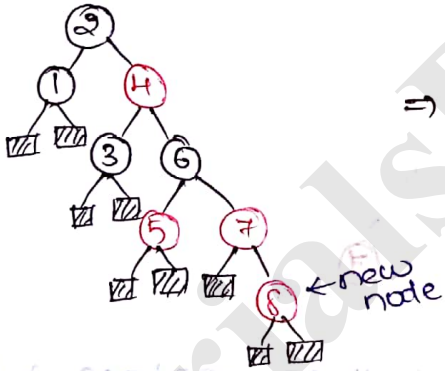
(i) LL rotation



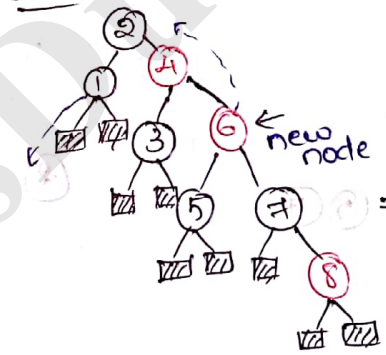
2) colour change



Insert 8 :-

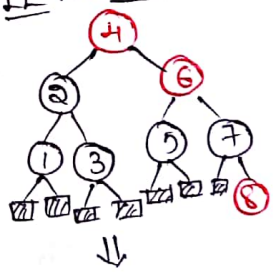


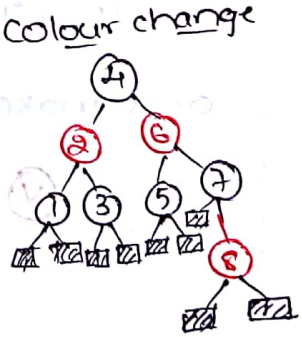
case (i)



case (iii)

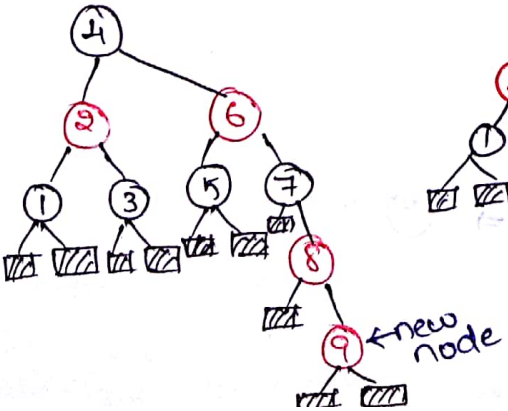
LL rotation



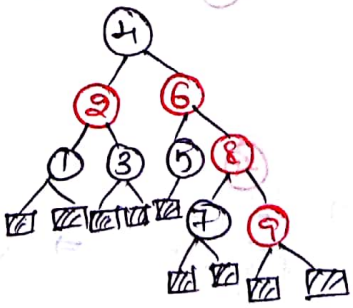


Colour change

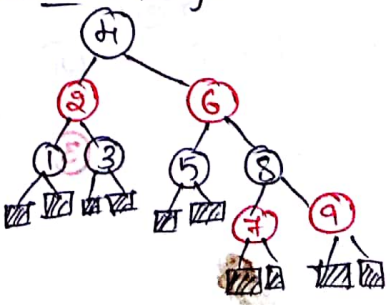
Insert 9 :-



case (iii)



Colour change



Construct a redblack tree by the following elements

4, 2, 1, 3, 6, 5, 7, 9, 8

Deletion operation in the redblack Tree

Consider the parameters.

v is the element which we want to delete

u is the child which replaces v

p is the parent of u

s is sibling of u,

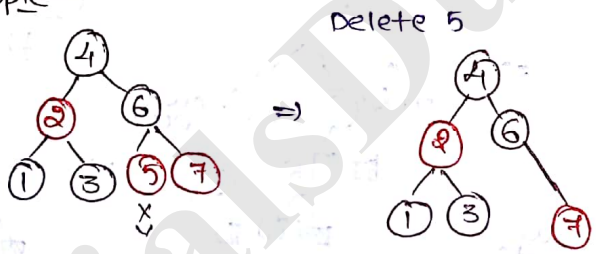
→ The Deletion operation in Redblack tree is similar to Deletion operation in Binary search Tree.

→ To Delete an element from Redblack Tree, we have to follow certain cases

Case (i)

If v is red leafnode then delete it & no operation can be performed the redblack Tree.

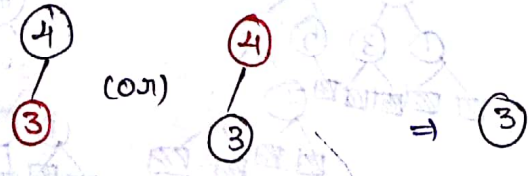
Example:



Case (ii)

If either v (or) u is red then replace v by u and make u's color as v black

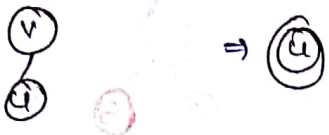
Ex (2):-



case(ii)

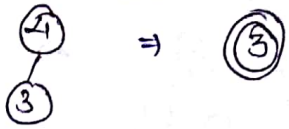
If both v and u are the black then replace v by u & make u as double black colour.

Ex:-(1)



Ex:-(2)

delete 4



case(iv)

If u becomes rootnode then we can convert double black into single black colour because the redblack Tree shouldnot accept double black nodes

→ If u becomes rootnode then we have to follow 4 subcases under case(3)

→ When the Tree becomes balanced we can convert the double black node into single black

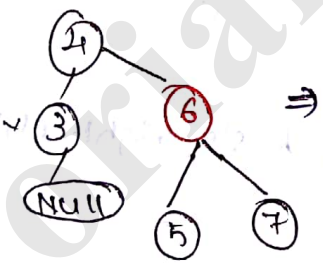
case (3.i)

If u's siblings is red then

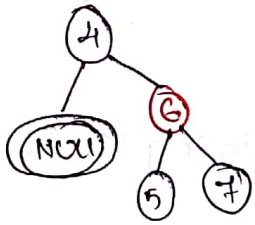
1) Interchange the colours of p & s.

2) Apply LL rotation P.

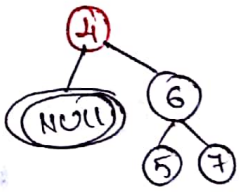
Ex:-



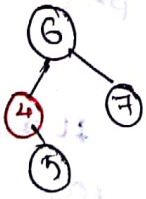
delete 3



(1) Colour change



(2) LL rotation

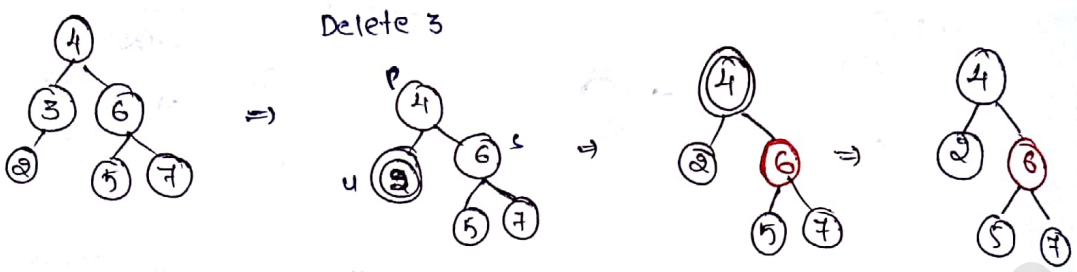


case(3.ii)

If u's siblings 's' is black & s's both children are black then

1) Remove one black colour from u's and also add one black colour to p

Ex :-



Case (3-iii)

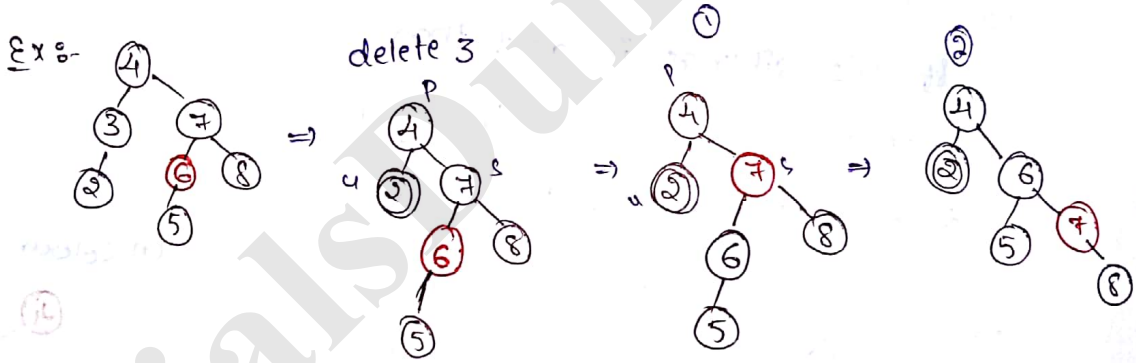
If u's sibling 's' is black & s's leftchild is red and right child may be either black (or) red then

1) Interchange the colours of s & s's leftchild

2) Apply RR rotation on s

3) Reapply suitable case if the double black still Exists

Ex :-



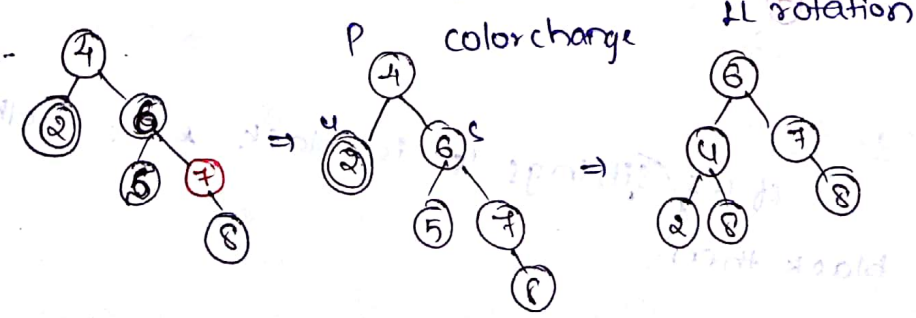
Case (3-iv)

If u's sibling 's' is black & s's right child is red then

① change the red colour of s's right child into black and interchange the colors of p & s

② Apply LL rotation on p

Ex :-



Usually, the search operation can be used to find whether the given element is present or not in the tree

→ The search operation in Redblack Tree is exactly similar to the search operation in Binary Search Tree.

→ In the Redblack Tree, the search operation initially checks whether the tree is Empty (or) non empty.

→ If the Redblack Tree is Empty, then the given element will not be found

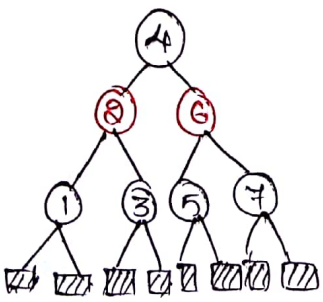
→ If the Redblack Tree is nonempty then, the search operation begins at the root node

→ If the given element is matched with the rootnode in the redblack Tree then the element is found at rootnode.

→ If the given element is less than the rootnode then the search operation will be performed recursively on the leftsubtree of the rootnode → If any element is matched with the given element then the element is found at leftsubtree of the rootnode.

→ If the given element is greater than the rootnode then the search operation will be Perform recursively on the rightsubtree of the rootnode. If any element is matched with the given element then the element is found at right subtree of the rootnode

→ If the given element is not matched with any element in redblack tree then that states the given element is not present in the entire redblack tree



Search the element : 2

It is found at the left sub tree of rootnode

If a binary search tree provides minimum total search time than its other Binary search tree organisation.

→ While performing search operations for the nodes then such Binary search tree is known as optimal Binary search Tree.

→ It is also known as weight balanced Binary search tree

→ The total search time of BST can be computed by the following formula

Total search Time = $Ma \times Na + Mb \times Nb$ --- $Mz \times Nz$

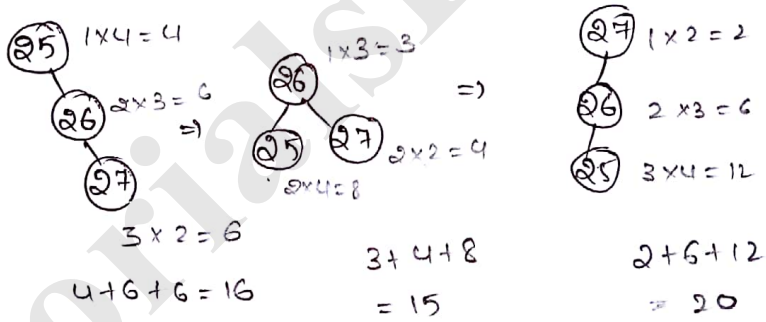
where,

m = no of memory accesses

n = no of times the elements to be searched

Ex :-

elements	search frequency
25	4
26	3
27	2



→ The optimal Binary search Tree basically classified into two types 1) static OBST

2) Dynamic OBST

Static OBST

In static OBST, the tree cannot be changed once it is built or constructed

Dynamic OBST

Unit - V

Multi-way search Tree

→ A multiway Search Tree is a multiway Tree in which every node stores atmost "(m-1)" no of values and "m" no of children, where m is degree of Tree

Ex:- B-tree, B+tree, 2-3 tree etc--

Usage:- Usually, the multiway Search Trees can be used widely in Secondary Storage devices, (Especially harddisk drive) to reduce the disk access time while accessing the data.

Properties:-

① A m-way Search Tree is a multiway tree where as the Binary Search Tree is a two-way tree, hence the m-way Search Tree may (or) may not a binary Search Tree.

② The degree (or) order of Tree (m) defines the no of values & no of subtrees per node.

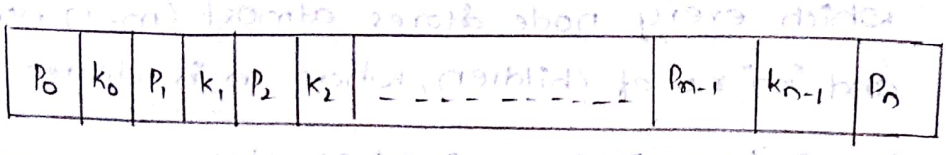
③ It is not compulsory that every node has exactly (m-1) no of values of 'm' subtrees per node.

④ The no of values may be from 1 to (m-1) & the no of subtrees may be from 0 to (i+1) per node, where 'i' is the no of values in that particular node.

⑤ The values in the left subtree are smaller than to their parent & the values in the right subtree are greater than to their parent

⑥ All the values in everynode should be in the

Node Structure

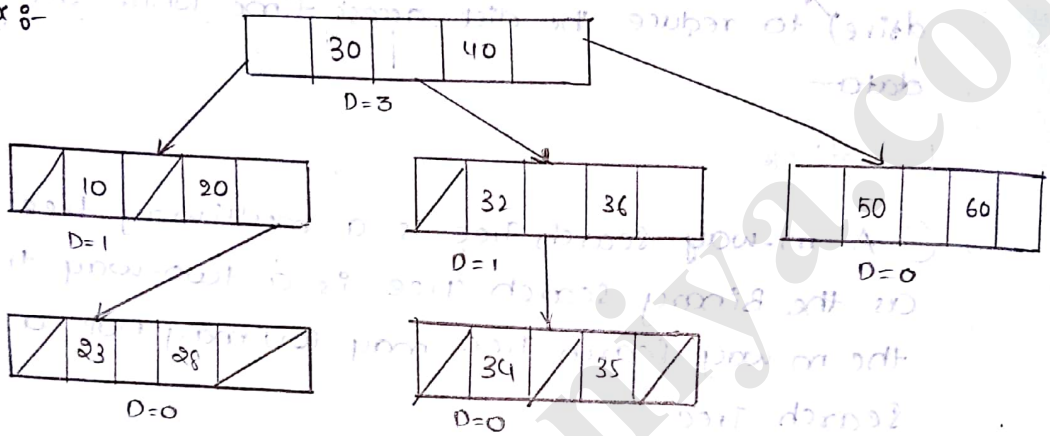


Where,

$P_0, P_1, P_2 \dots P_n$ are the pointer fields

$k_0, k_1, k_2 \dots k_n$ are the data (or) key fields

Ex :-



B-Tree

A BTree is one of the multiway search Tree in which every node stores atmost m-1 number of values and atmost m number of children where m is the degree or order of tree

→ The B-tree was developed by 2 scientists Rudolf Bayer and M.M.C creight in the year 1972.

Properties

→ All the nodes except root node store atleast $(m/2)-1$ no of values & atmost $(m-1)$ no of values. The root may contain minimum one value.

→ All the nodes Except root & leaf node must have atleast m/2 . no of children.

→ Rootnode must have atleast two children if it is not leaf node

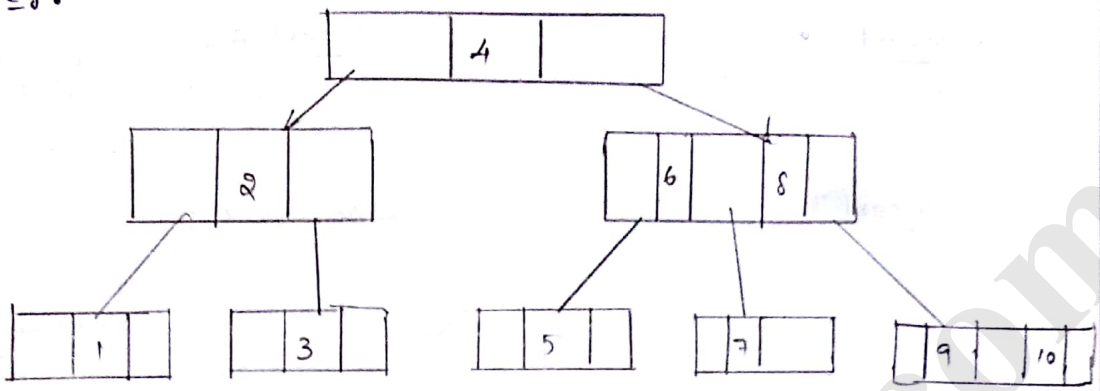
→ If a non-leaf node has 'm' no of children then it must have $(m-1)$ no of values

→ All the leaf nodes should be in same level.

→ All the nodes (BTree) except rootnode must have at least (m/2) values and atmost (m-1) values. The root node may contain min one value - Both insertion & deletion

→ The values in everynode should be in ascending order
The order of BTree always should be odd number

Eg :-



Operations of B-Tree :-

Usally three operations performed on B-Tree they are

1) Insertion
2) Deletion
3) search

Insertion operation in BTree

In Btree, every new value is inserted at leaf node

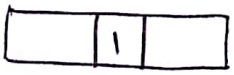
procedure steps :-

1) Check whether the tree is empty (or) not
2) If the tree is Empty then create a newnode & insert new value into it and that becomes rootnode
3) If the tree is not empty then find the specific leaf node
   a) If the leafnode has empty datafield then insert new value by following Binary Search Tree logic & ascending order.
   b) else, split it the tree by sending middle value as parent Repeat the process untill the sent value is fixed into a suitable node.
   c) If the spliting is occured at root node then the sending value becomes newroot and the height of tree is also enhanced.

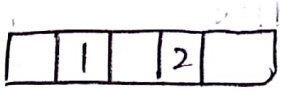
Example :-

construct B-tree of order 1 to 10

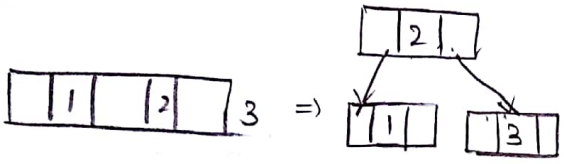
Insert 1 :-



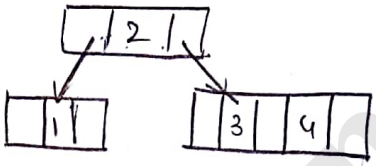
Insert 2 :-



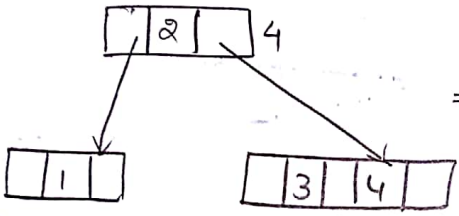
Insert 3 :-



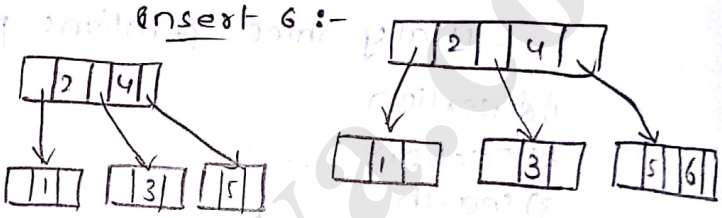
Insert 4 :-



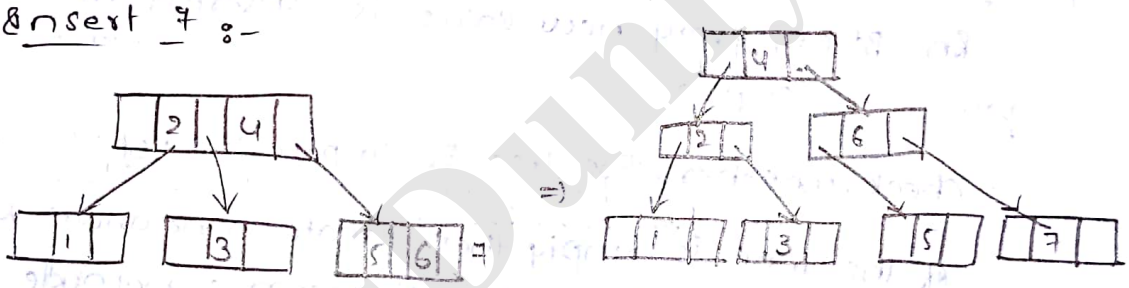
Insert 5 :-



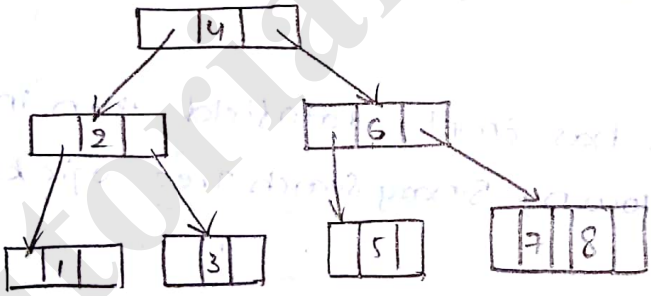
Insert 6 :-



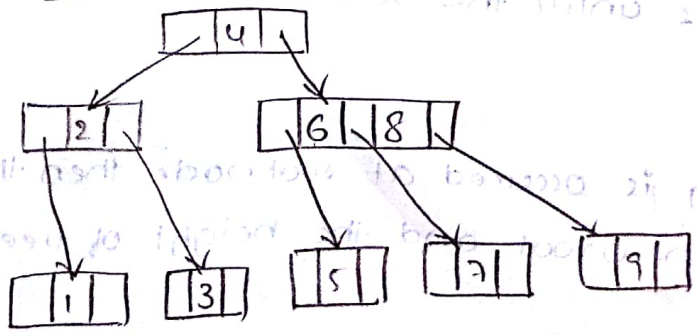
Insert 7 :-



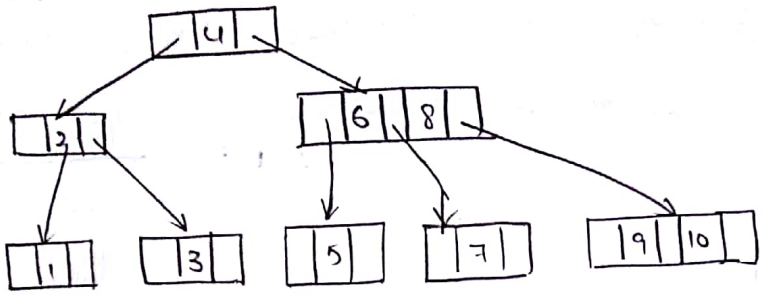
Insert 8 :-



Insert 9 :-



Insert 10:-



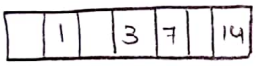
Construct BTree of order m=5 of following elements

3, 14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26, 4, 16, 18, 24, 25 & 19

max = 4, min = 2

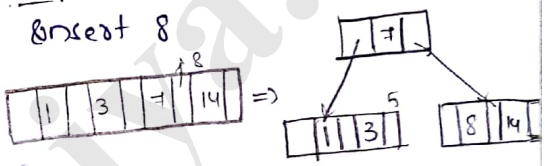
Step ①:-

Insert 3, 14, 7, 1



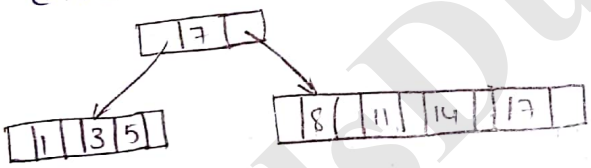
Step ②:-

Insert 8



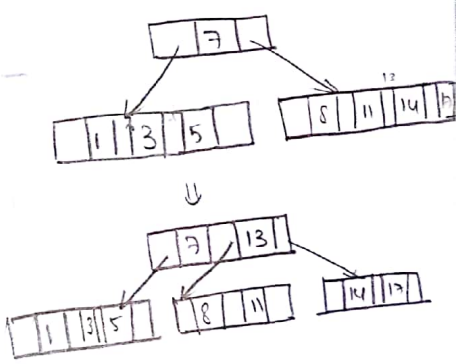
Step ③:-

Insert 5, 11, 17



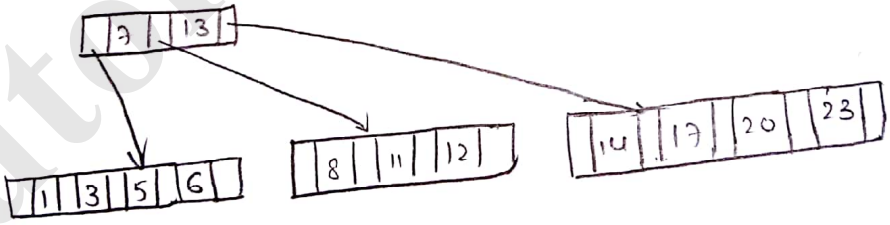
Step ④:-

Insert 13

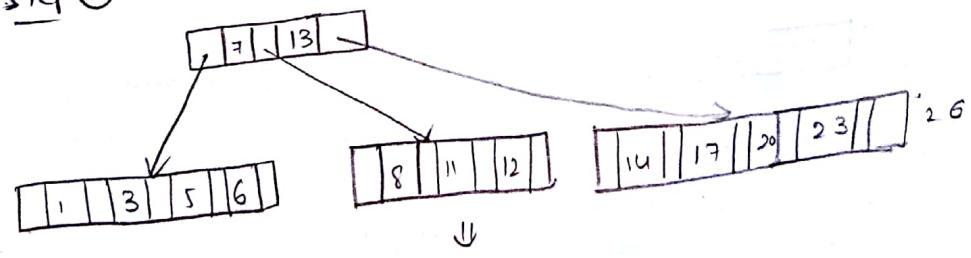


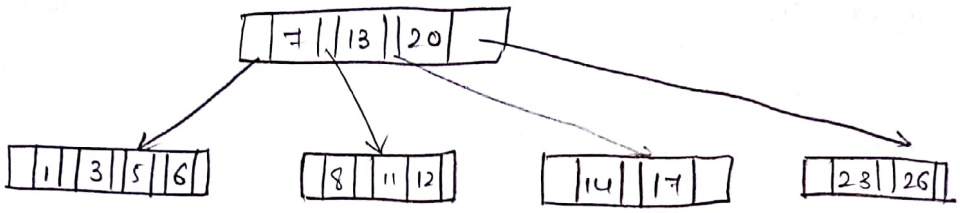
Step ⑤:-

Insert 6, 23, 12, 20

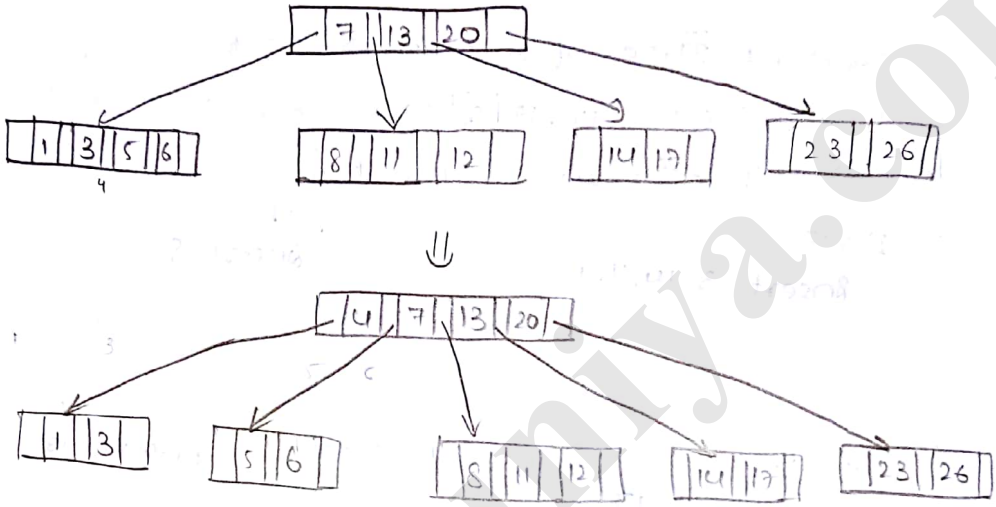


Step ⑥:- Insert 26

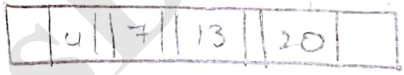


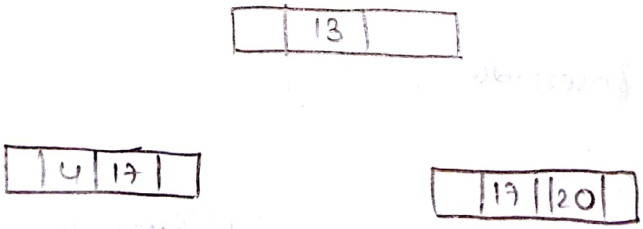


Step (7) :- Insert 4



Step (8) :-





TutorialsDuniya.com

Download FREE Computer Science Notes, Programs, Projects, Books PDF for any university student of BCA, MCA, B.Sc, B.Tech CSE, M.Sc, M.Tech at https://www.tutorialsduniya.com

- Algorithms Notes
- Artificial Intelligence
- Android Programming
- C & C++ Programming
- Combinatorial Optimization
- Computer Graphics
- Computer Networks
- Computer System Architecture
- DBMS & SQL Notes
- Data Analysis & Visualization
- Data Mining
- Data Science
- Data Structures
- Deep Learning
- Digital Image Processing
- Discrete Mathematics
- Information Security
- Internet Technologies
- Java Programming
- JavaScript & jQuery
- Machine Learning
- Microprocessor
- Operating System
- Operational Research
- PHP Notes
- Python Programming
- R Programming
- Software Engineering
- System Programming
- Theory of Computation
- Unix Network Programming
- Web Design & Development

Please Share these Notes with your Friends as well

Eg:- Construct a B-tree of order 3 by following elements:

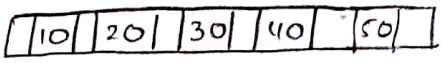
10, 20, 30, 40, 50, 60, 70, 80, 90, 100

m = 3

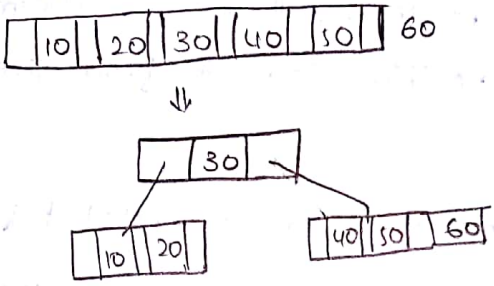
min = (m-1)
= 3-1 = 2

max = (2m-1)
= (6-1)
= 5

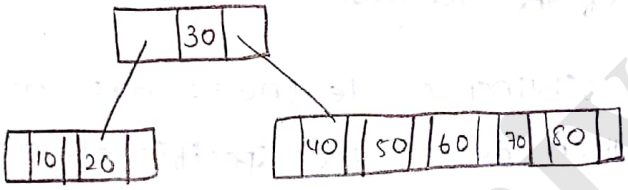
Insert :- 10, 20, 30, 40, 50



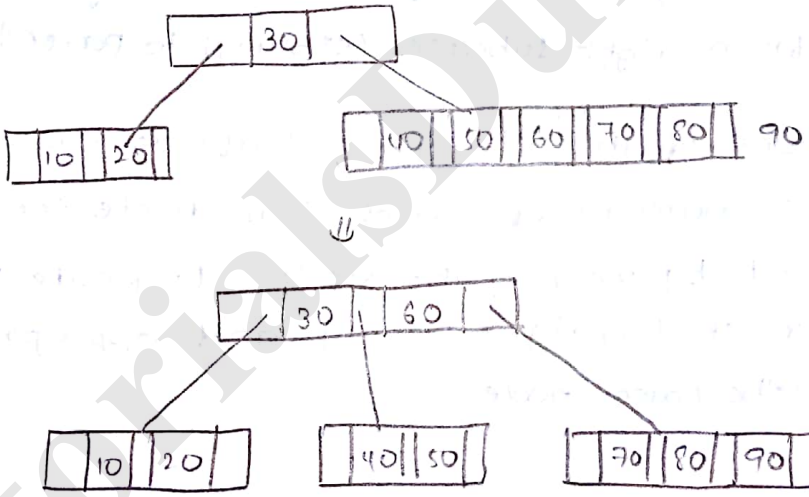
Insert :- 60



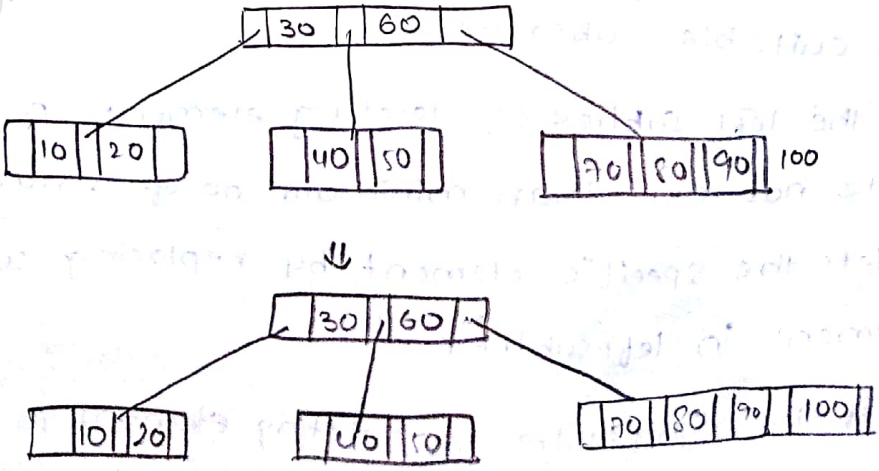
Insert :- 70, 80



Insert :- 90



Insert :- 100



Deletion in the B-Tree

To delete element from B-tree we must follow certain cases

Case(i) :-

If an element is to be deleted from the leafnode and the leafnode has more than the minimum no of elements then delete the specific element from the leafnode.

Case(ii) :-

If an element is to be deleted from the leafnode and the leafnode has only minimum no of values then follow one of the suitable subcases.

a) If either sibling of leafnode has more than minimum no of values then delete the specific element by moving appropriate value from parent node to specific leafnode and move largest value from left sibling (or) smallest value from right subcases (sibling) to parent node

b) If none of the siblings of leafnode has more than the minimum no of values then delete the specific element by merging the specific leafnode with either left (or) right sibling and appropriate value from the parent node.

Case(iii)

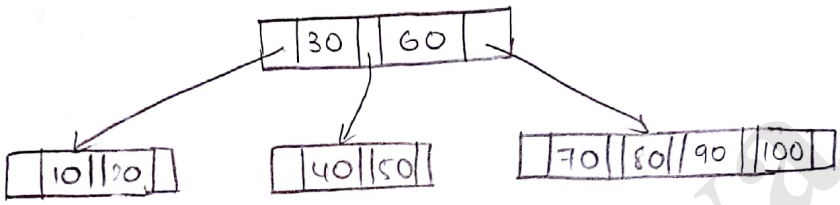
If an element is to be deleted from the internal node (including root node) then follow one of the three suitable subcases

a) If the left subtree of deleting element in the internal node has more than minimum no of values then delete the specific element by replacing with largest element in leftsubtree

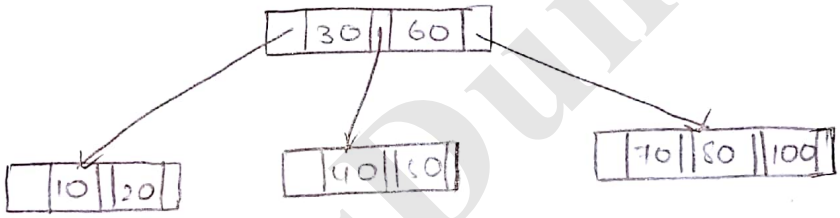
b) If the right subtree of deleting element in internal

node has more than the minimum no of values then delete the specific element by replacing with smallest element from the right subtree.

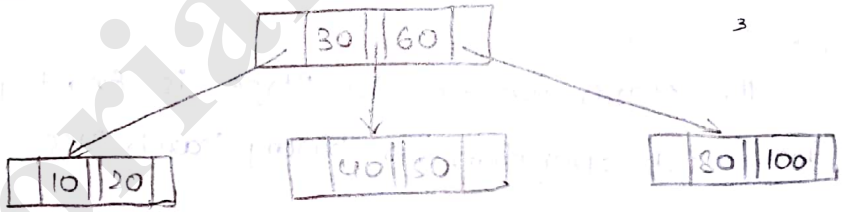
c) If both left & right subtrees of deleting element in the internal node no of values then delete the specific element by merging both left & right subtreer of deleting element.



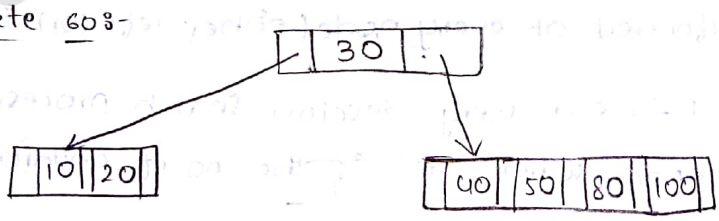
Delete 90 :-



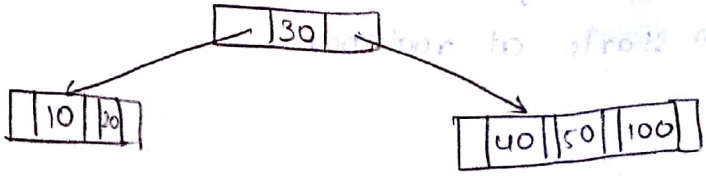
Delete 70 :-



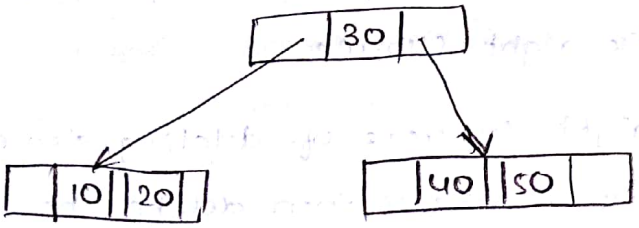
Delete 60 :-



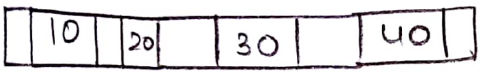
Delete 80 :-



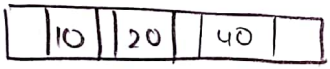
delete 100:-



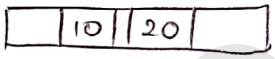
delete 50:-



delete 30 :-



delete 40:-



delete 20:-

Search Operation

→ The search operation in Btree is exactly similar to the search operation in Binary search Tree

→ In the Binary search Tree, two way decision search process will be performed at every node (either left (or) right subtree)

→ In the B-Tree, n-way decision search process will perform at every node where 'n' is the no of children of specific node.

→ In both Binary search Tree and BTree, the search operation starts at rootnode.

procedure steps

→ check whether the tree is empty (or) not

→ If the tree is Empty then the Search element will not be Found. Otherwise,

→ If the tree is not empty, the Search process starts with Rootnode. Initially, the Search element is compared with first element in the rootnode. If the search element is matched with first element then the element is found at rootnode

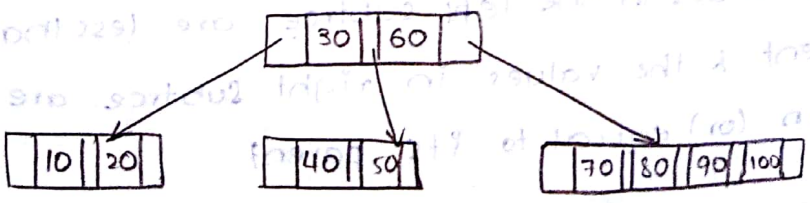
→ If the Search element is lessthan first element in the Rootnode then the search operation will be performed recursively on the leftsubtree of first element in the Root node

→ If the Search element is greater than the first element in the rootnode then the search element is compared with next element in the rootnode. If both are matched then the Search element is found at rootnode.

→ If the rearch element is less than the next element in the rootnode then Search operation will be performed recursively on the leftsubtree of the next element in the rootnode.

→ If the search element is greater than next element in the rootnode then the Search element is compared with another element in the rootnode if any. otherwise the Search operation will be performed recursively on the right subtree of next element in rootnode

→ If the Search element is not matched with anyelement in the B-tree then that indicates the Search element is not available in the entire B-tree.



B+ tree

→ The B+ tree is another multiway search tree in which everynode also contains m no. of children, where 'm' is degree (or) order of tree

→ The B+ tree is an advanced tree of B-tree

→ The B+ tree was developed by one of the foriegn scientist Douglas Comer in the year 1973

→ In General, the B+ trees can be used mostly in the implementation of databases. and also used in secondary storage devices. (Especially in Harddisk drive)

→ The operations like insertion, deletion, and retrival can be performed effeciently on the B+ tree. than B+tree

→ usually, the B+ tree occupies more space than B-tree

→ the B+ tree follows the combination of ISAM (Index sequential Access method) & Btree properties

Properties of B+ tree

→ All the properties of B-tree are inherited into B+ tree

→ The elements stored in the leaf nodes are treated as actual elements and the elements stored in the internal nodes are treated as index values which are used to identify the actual elements in leaf nodes

→ B+ tree may store redundant data

→ All the leafnodes are connected to one another in the form of linked list for faster data access

→ The values in the left subtree are less than to its parent & the values in right subtree are greater than (or) equal to its parent

→ the operations of B+ tree are 1) Insertion

2) deletion

3) search

Ex:- 1)

m = 3

min = m - 1

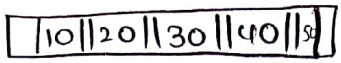
= 3 - 1 = 2

max = 2m - 1

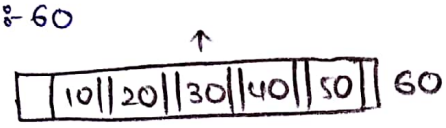
= 2(3) - 1

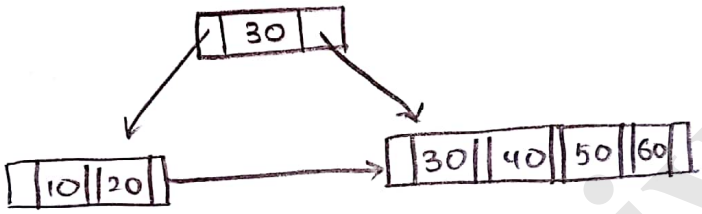
= 5

Insert :- 10, 20, 30, 40, 50

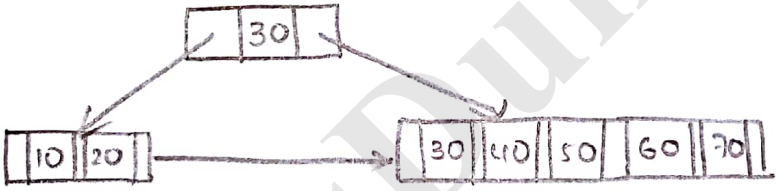


Insert :- 60

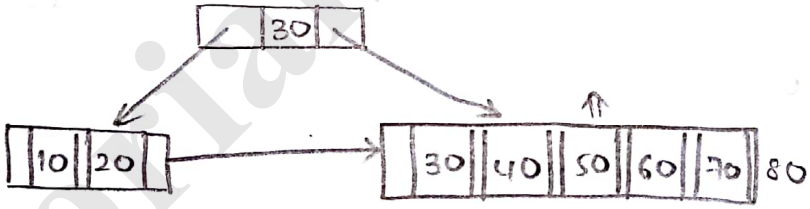


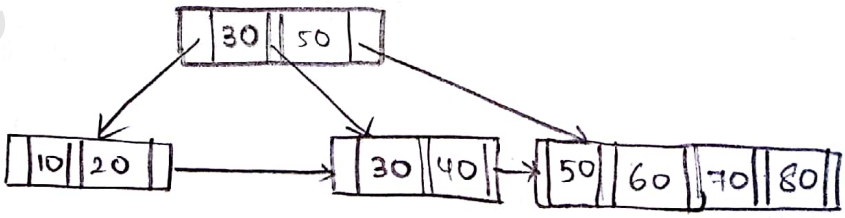


Insert :- 70

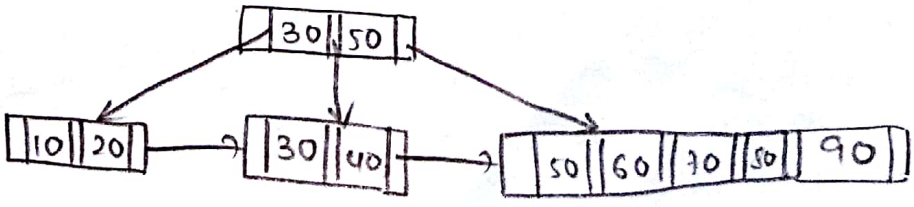


Insert :- 80

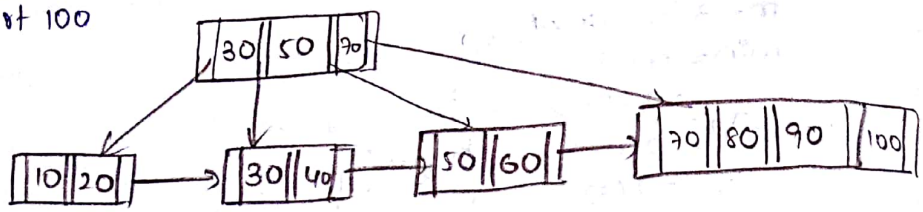




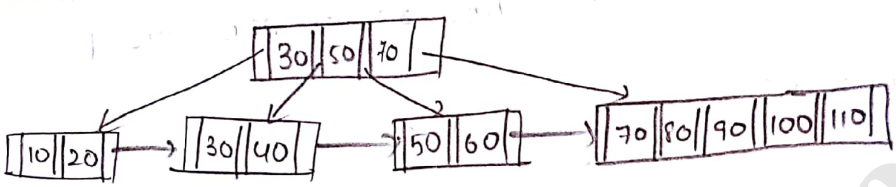
Insert :- 90



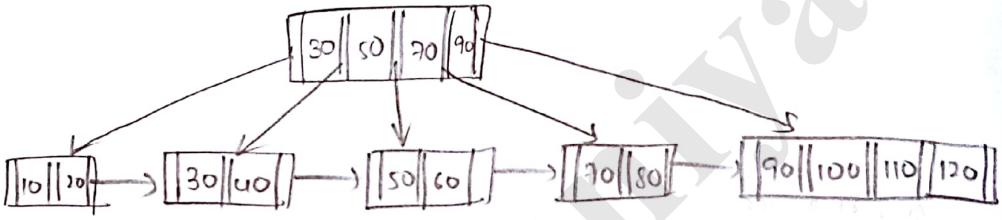
Insert 100



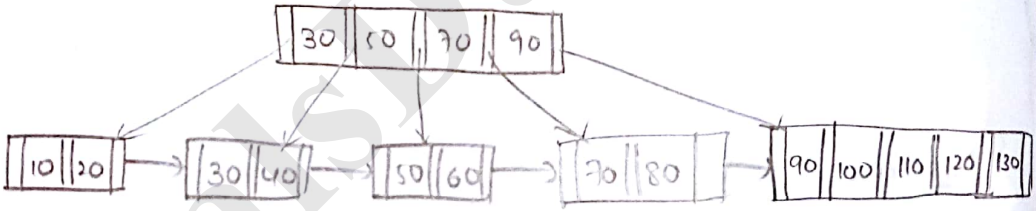
Insert 110



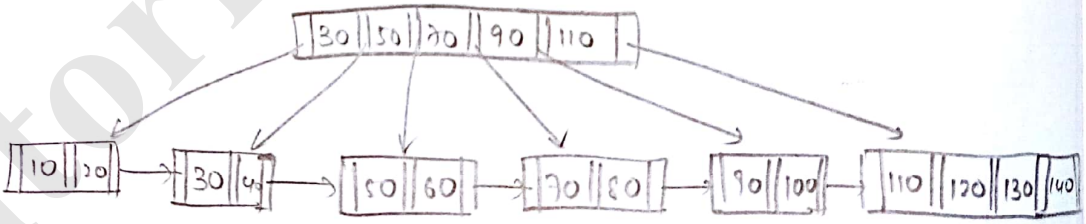
Insert 120



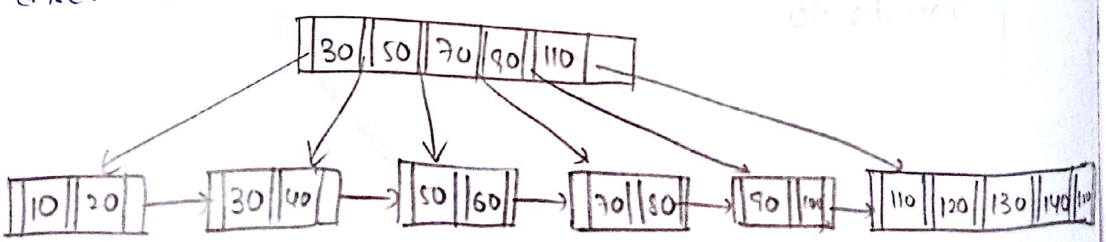
Insert 130

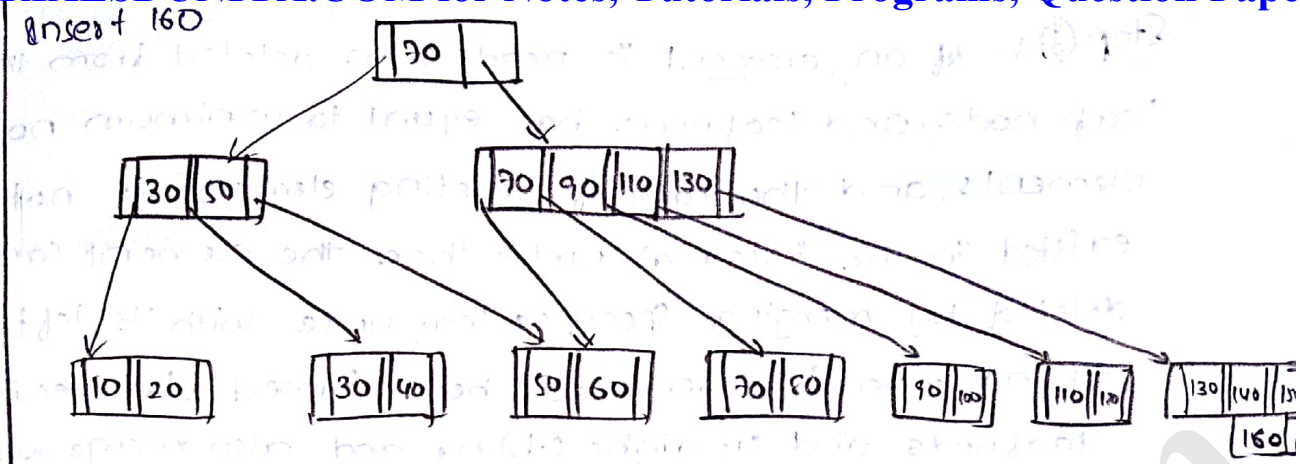


Insert 140



Insert 150





Note:- During insertion operation in Bt tree, when a node is to be splited into two parts the middle value is stored into second splitted part and also stored into its parent node

Deletion operation in Bt tree:-

In Bt tree the deletion is performed at leaf node and copy of deleting element in internal node will be removed by default

To performed, the deletion operation in Bt tree we follow certain cases

Step①:- If an element to be deleted from leafnode and the leafnode has more than minimum number of elements and the copy of deleting element is not existed in internal nodes then the element can be directly from specific leafnode

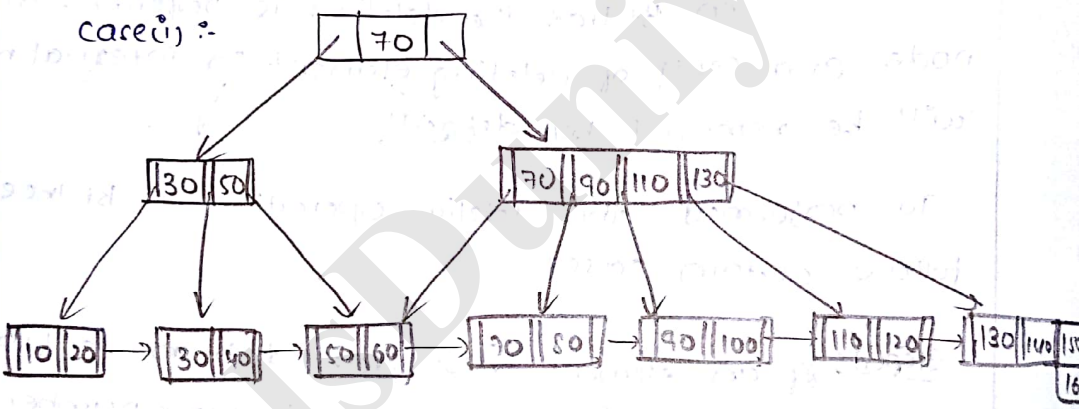
Step②:- If an element is need to be deleted from the leafnode and the leafnode has more than minimum number of elements (or) equal to minimum no of elements and copy of deleting element is existed in internal nodes then the element can be deleted from specific leafnode by merging specific leafnode with its leaf sibling and the internal nodes should be adjusted (merge will perform b/w internal nodes if require)

Step ③ :- If an element is need to be deleted from the leaf node and leafnode has equal to minimum no of elements and the copy of deleting element is not existed in the internal nodes then the element can be deleted by merging specifice leaf node with its left sibling otherwise merge will be performed b/w specific leafnode and its right sibling and also merge will be performed b/w the internal nodes, (if required) to adjust the B+ tree

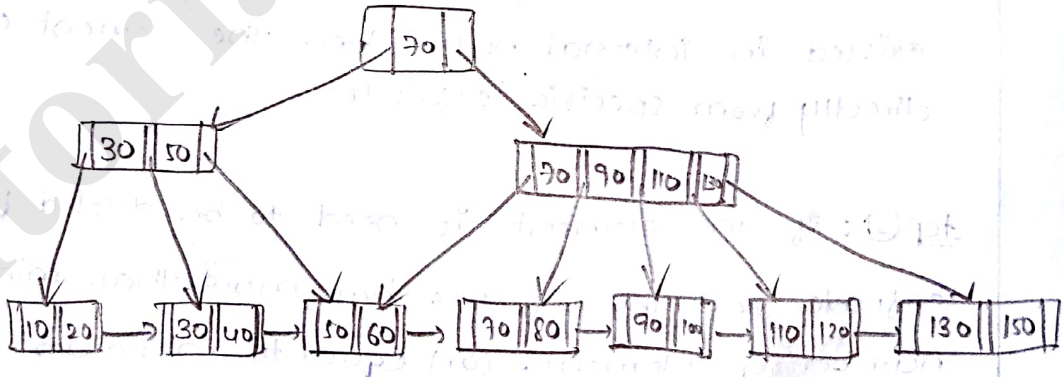
Eg :-

Step ① :-

Case (i) :-

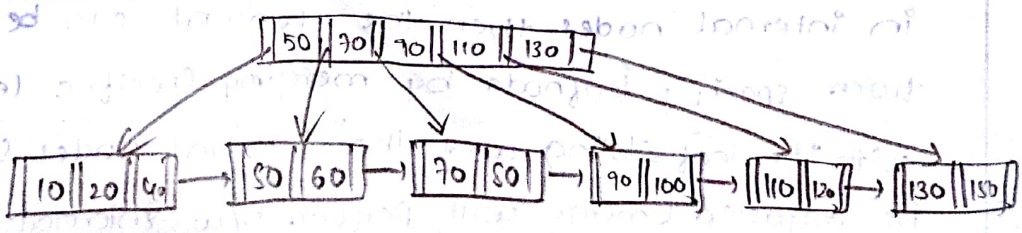


Delete 140 & 160

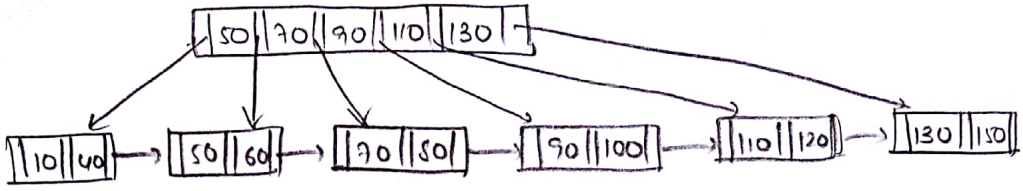


Step ② :-

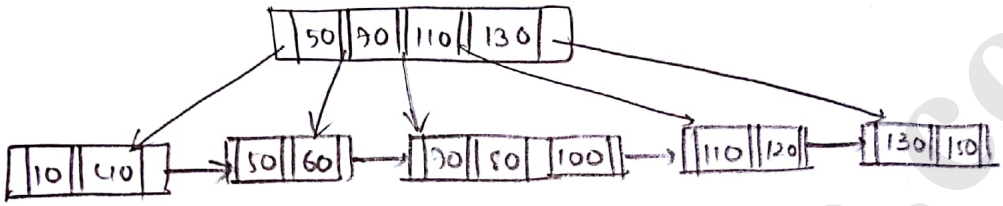
Delete 30



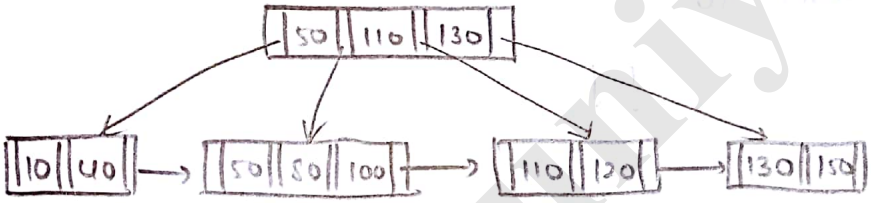
Step③ :- delete 20



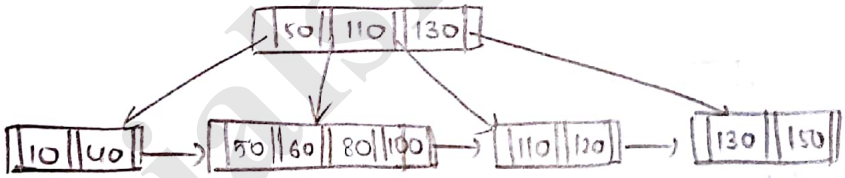
delete 90



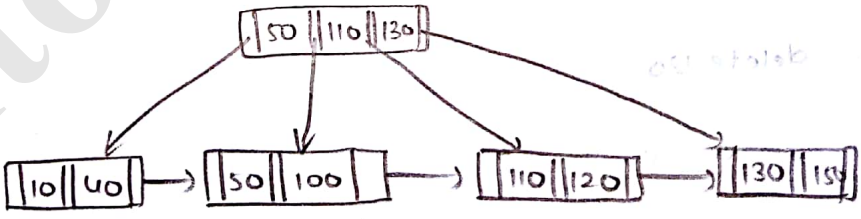
4. delete 60



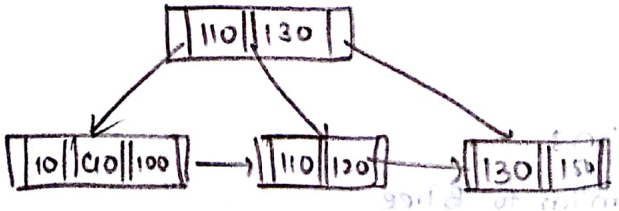
70 delete 70



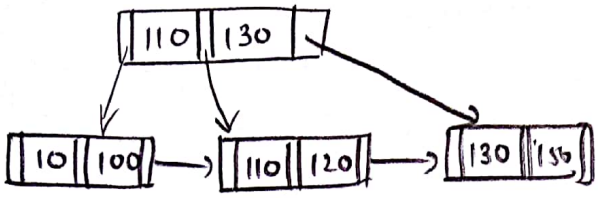
80 delete 80



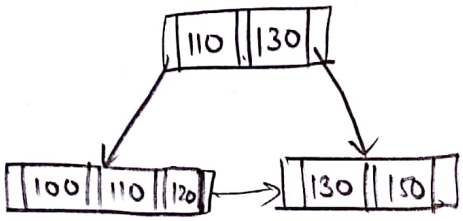
50 delete 50



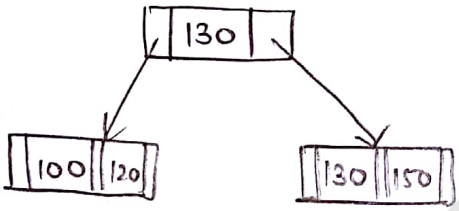
delete 40



10 delete 10

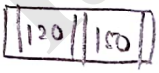


110 delete 110

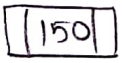


130 delete 130

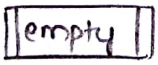
100 delete 100



120 delete 120



150 delete 150



Search operation :-

Similar to B Tree

Digital Search Tree (DST)

→ A DST is special type of Binary tree in which every node stores Binary number (or) Binary element

→ In general the DST can used mostly in two applications
1) IP routing
2) Security systems (Firewalls)

→ The operations like insertion, deletion & search can be performed very efficiently in DST than AVL Tree & Binary Search Tree

→ The DST occupies less space than AVL & Binary Search Tree

Time complexity

Best case ⇒ O(1)

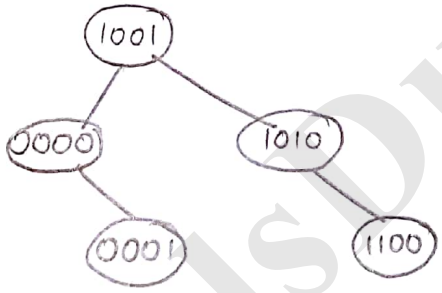
Average case ⇒ O(logN)

'N' is height of DST

worst case ⇒ O(b)

'b' is no of bits of an element

Ex :-



Operations

Three Basic operations performed on DST

1. Insertion
2. Deletion
3. search

→ To insert an element in DST we need to follow the procedure steps

Step① :- check whether the tree is empty or not

Step②:- If the tree is empty then new element is inserted into the DST and which will be treated as rootnode

Step③:-
→ If the tree is not empty then the new element is inserted either left subtree (or) right subtree to the rootnode by checking first bit in the new element. If the

first bit of new element is zero then the new element is to be inserted as the left subtree to the rootnode. Otherwise (1st Bit = 1) the new element is to be inserted as the right subtree to the rootnode.

Step ④:- If the node is already existed in either LST (or) RST (Right Subtree) to the rootnode then the next new element is to be inserted either (left subtree (or) Right Subtree to the existed node by comparing next Bit by Bit with the new element

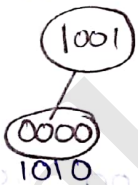
⑤ this procedure will be continued at every node in the DST recursively.

Ex:- construct a DST with four bits by the following elements. 1001, 0000, 1010, 0001, 1000, 0011, 1100, 0100, 1011, 0101, 1101, 0110, 1111, 0111

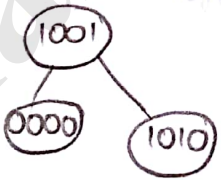
0 - left
1 - right

Insert 1001

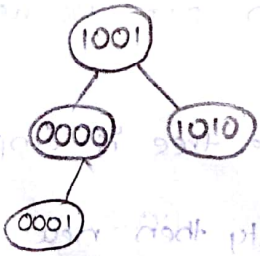
Insert 0000



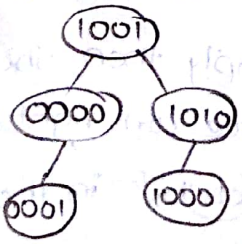
Insert 1010



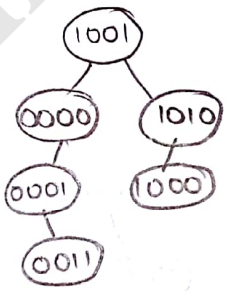
Insert 0001



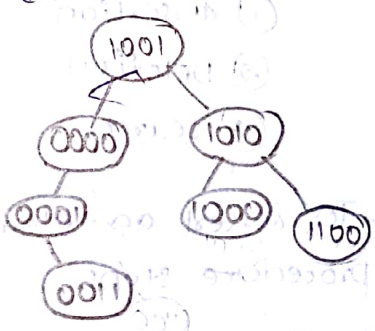
Insert 1000



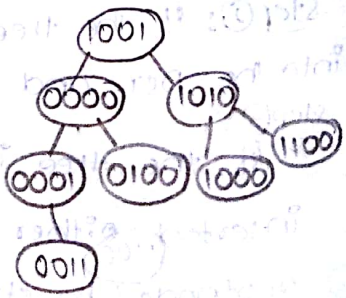
Insert 0011



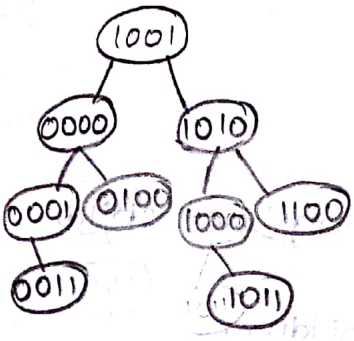
Insert 1100



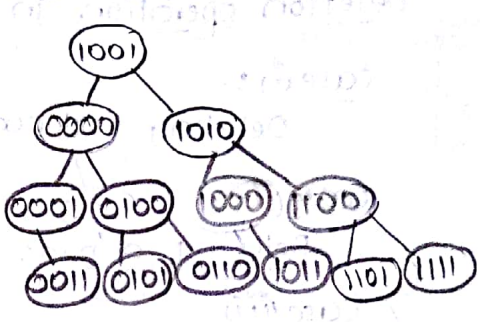
Insert 0100



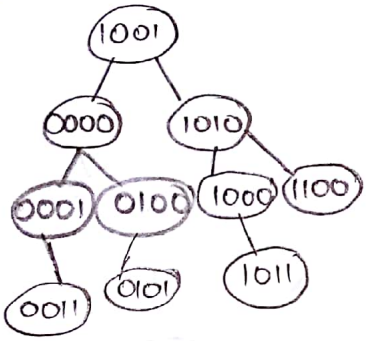
Insert 1011 :-



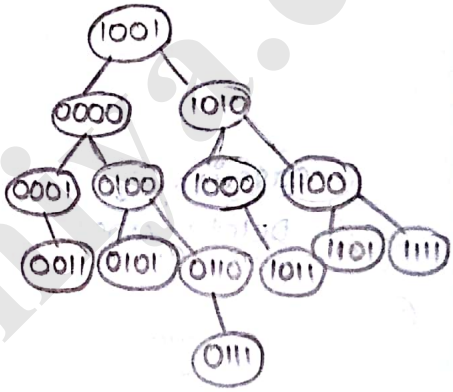
Insert 1111:-



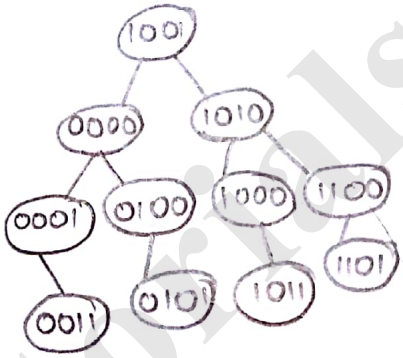
Insert 0101 :-



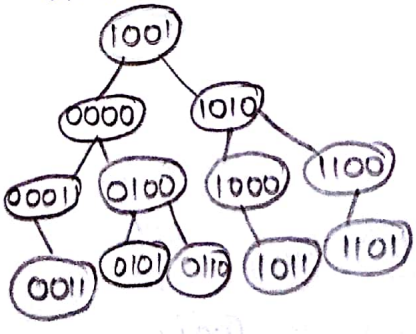
Insert 0111



Insert 1101 :-



Insert 0110 :-



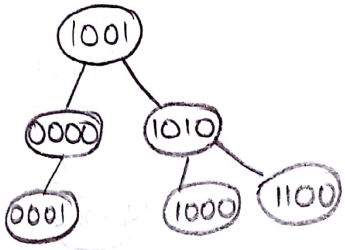
Deletion operation in DST :-

case(i) :-
Deleting leaf node

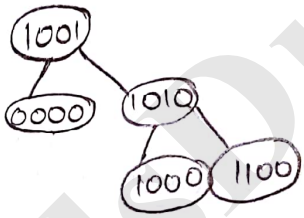
case(ii)
Deleting a node which is having one child

case(iii)
Deleting a node having 2 children

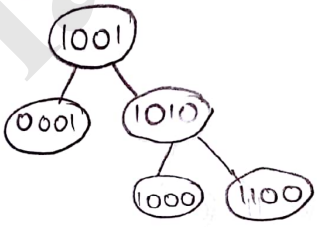
Ex :- Delete the elements from DST



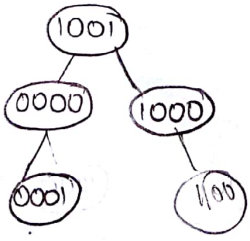
case(i) Eg :-↓
Delete 0001



Case(ii) Eg :-↓
Delete 0000



case(iii) Eg :-↓
Delete 1010



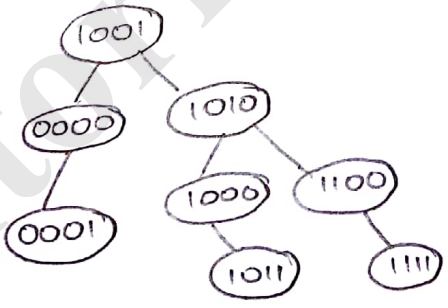
Search operation in DST

To Search an element in the DST we have to follow the following procedure steps

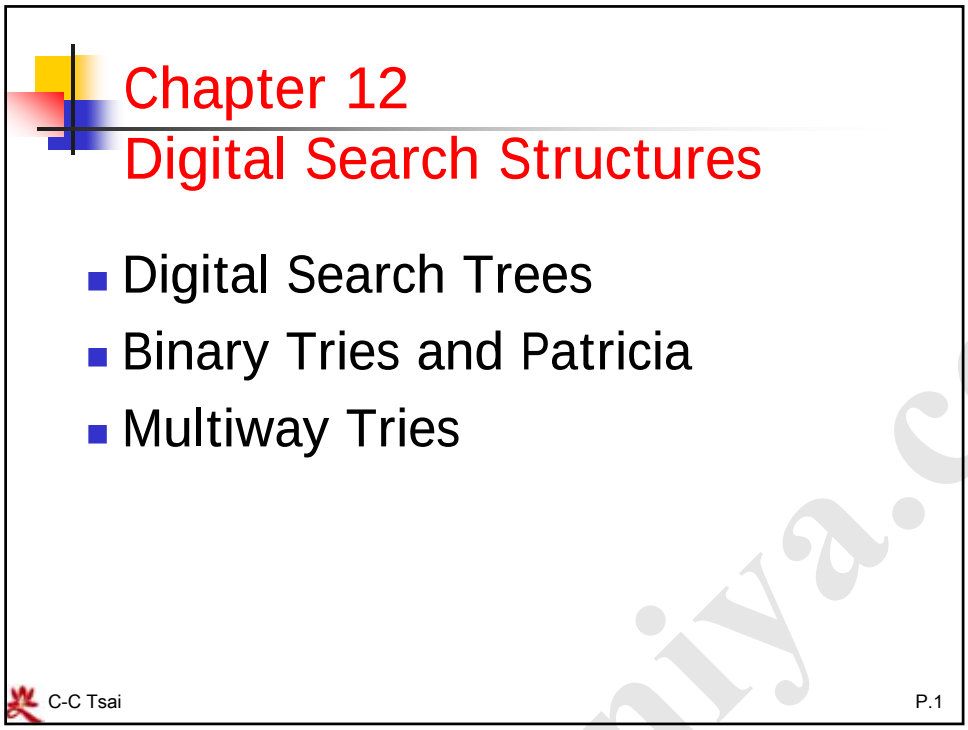
procedure steps

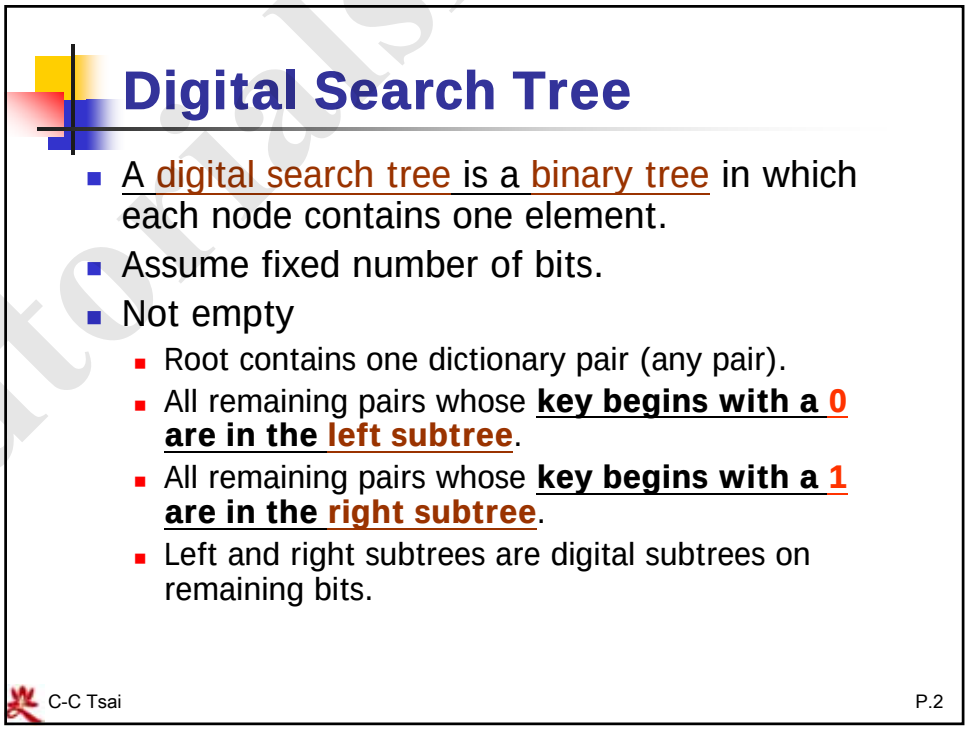
1) check whether the tree is empty or not
2) If the tree is empty then the search element cannot be found
3) If the tree is not empty then the search process starts with rootnode onwards
4) If the search element is matched with the rootnode element then Search element is found at rootnode
5) If the search element is not matched with the rootnode element then the 1st bit of search element will be checked if it is zero, then the search operation willbe performed recursively on the leftsubtree of the rootnode If it is one, the search operation will be performed recursively on the rightsubtree of the rootnode
6) This process will be continued untill the leafnode of either leftsubtree (or) right subtree of the rootnode

Ex :-



Search the element : 1011
the element 1011 is found at right subtree of rootnode

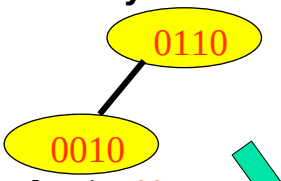




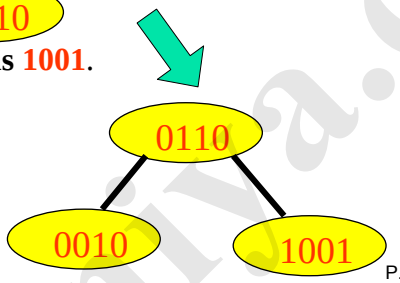
Example of Digital Search Tree

- Start with an empty digital search tree and **insert a pair whose key is 0110**.

- Now, **insert a pair whose key is 0010**.



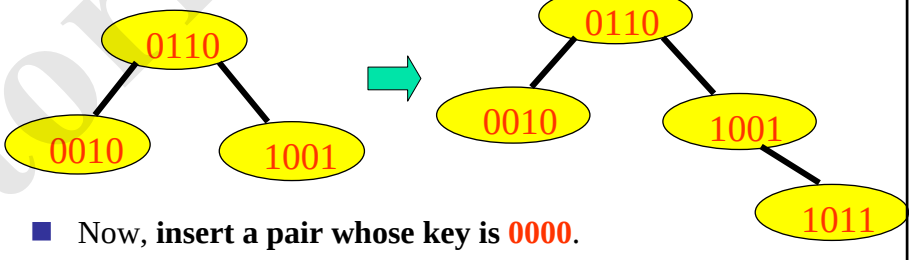
- Now, **insert a pair whose key is 1001**.



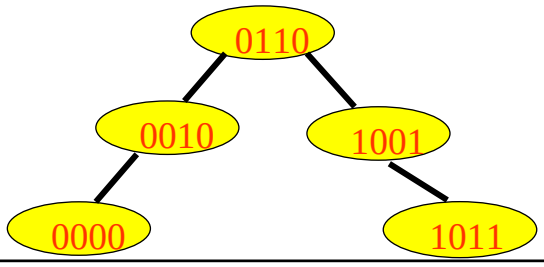
C-C Tsai P.3

Example

- Now, **insert a pair whose key is 1011**.



- Now, **insert a pair whose key is 0000**.



C-C Tsai P.4

TutorialsDuniya.com

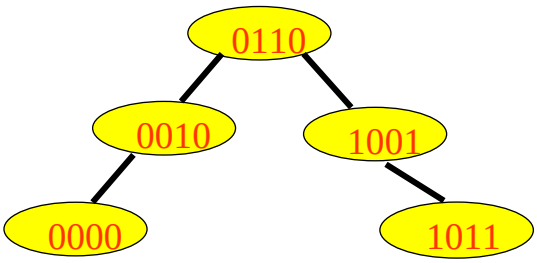
Download FREE Computer Science Notes, Programs, Projects, Books PDF for any university student of BCA, MCA, B.Sc, B.Tech CSE, M.Sc, M.Tech at https://www.tutorialsduniya.com

- Algorithms Notes
- Artificial Intelligence
- Android Programming
- C & C++ Programming
- Combinatorial Optimization
- Computer Graphics
- Computer Networks
- Computer System Architecture
- DBMS & SQL Notes
- Data Analysis & Visualization
- Data Mining
- Data Science
- Data Structures
- Deep Learning
- Digital Image Processing
- Discrete Mathematics
- Information Security
- Internet Technologies
- Java Programming
- JavaScript & jQuery
- Machine Learning
- Microprocessor
- Operating System
- Operational Research
- PHP Notes
- Python Programming
- R Programming
- Software Engineering
- System Programming
- Theory of Computation
- Unix Network Programming
- Web Design & Development

Please Share these Notes with your Friends as well

Search/Insert/Delete

- Complexity of each operation is **O(#bits in a key).**
- #key comparisons = **O(height).**
- Expensive when keys are very long.



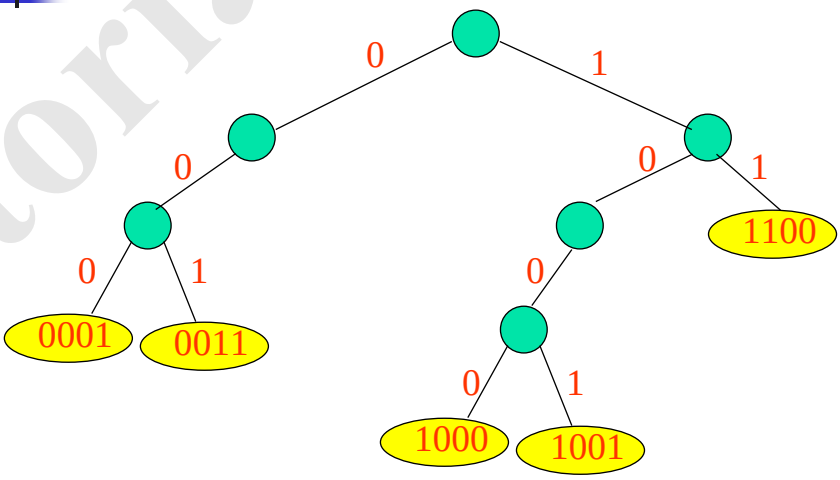
Applications of Digital Search Trees

- Analog of radix sort to searching.
- Keys are binary bit strings.
  - **Fixed length** – 0110, 0010, 1010, 1011.
  - **Variable length** – 01, 00, 101, 1011.
- Application – IP routing, packet classification, firewalls.
  - IPv4 – 32 bit IP address.
  - IPv6 – 128 bit IP address.

Binary Trie

- Information Retrieval.
- At most one key comparison per operation, **search / insert / delete**.
- A Binary trie (pronounced *try*) is a binary tree that has two kinds of nodes: branch nodes and element nodes. For fixed length keys,
  - **Branch nodes**: Left and right child pointers. No data field(s).
  - **Element nodes**: No child pointers. Data field to hold dictionary pair.

Example of Binary Trie

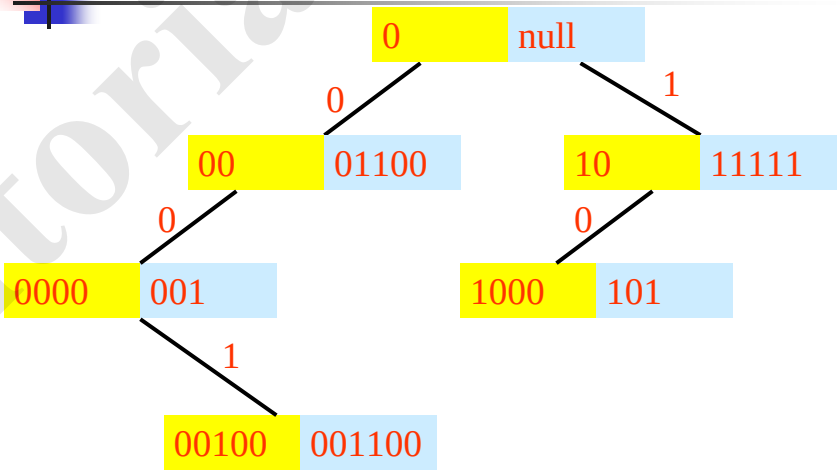


At most one key comparison for a search.

Variable Key Length

- Left and right child fields.
- Left and right pair fields.
  - Left pair is pair whose key terminates at root of left subtree or the single pair that might otherwise be in the left subtree.
  - Right pair is pair whose key terminates at root of right subtree or the single pair that might otherwise be in the right subtree.
  - Field is null otherwise.

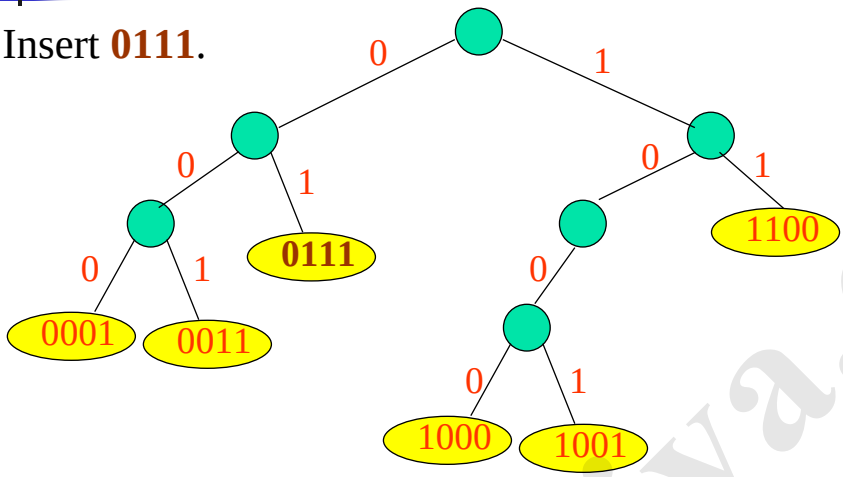
Example of Variable Key Length



At most one key comparison for a search.

Fixed Length Insert

Insert **0111**.

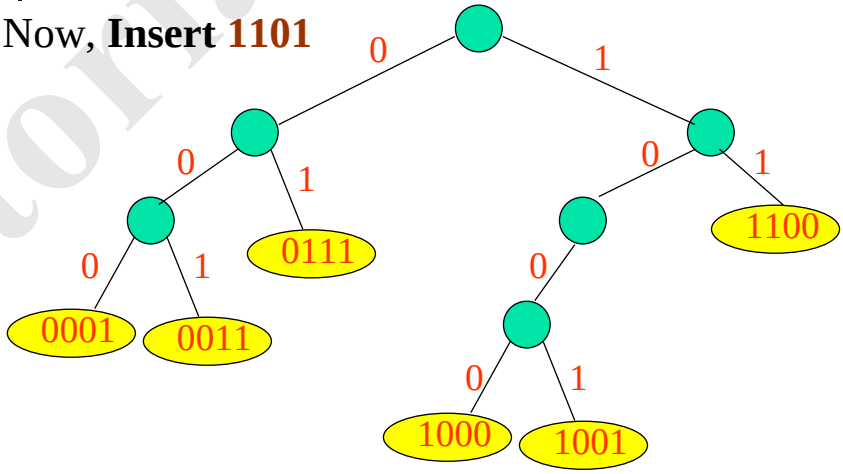


Zero compares.

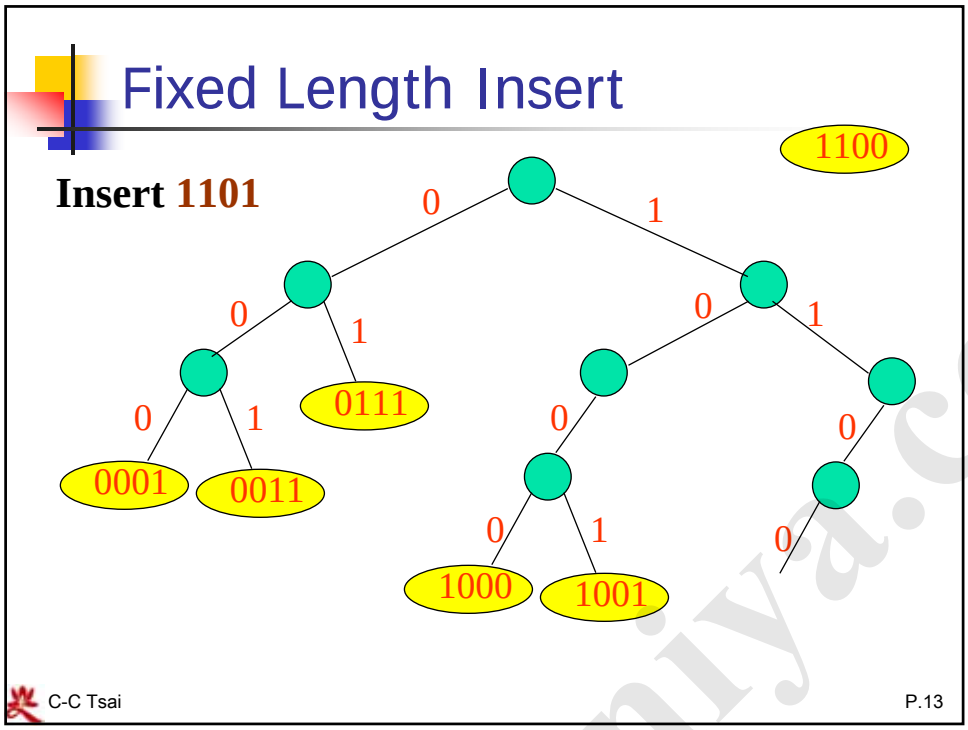
C-C Tsai P.11

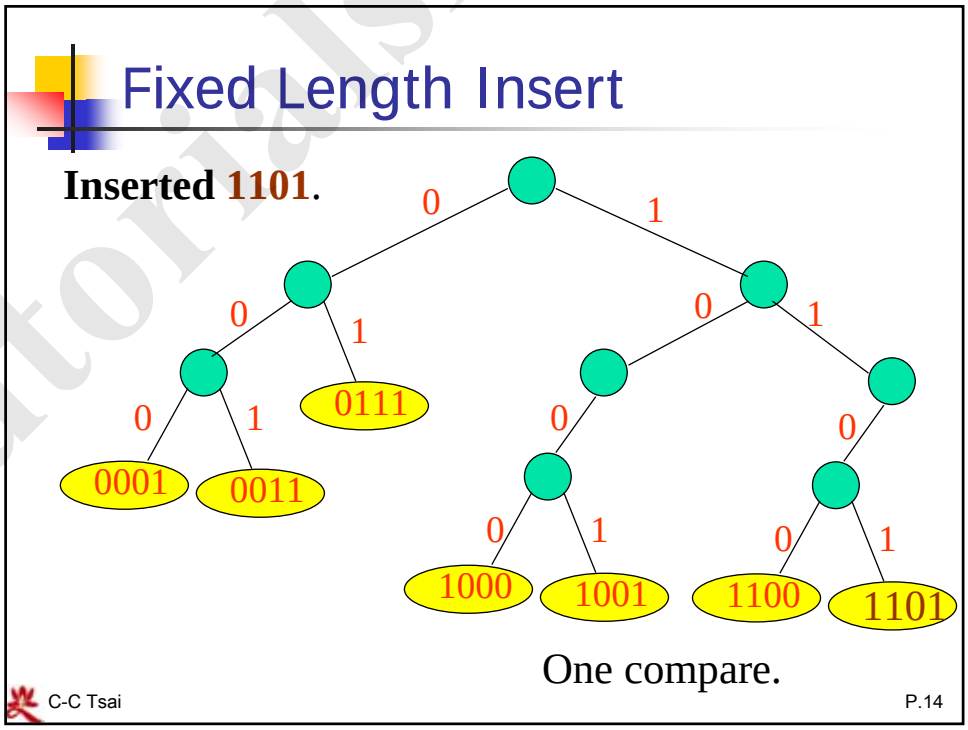
Fixed Length Insert

Now, **Insert 1101**



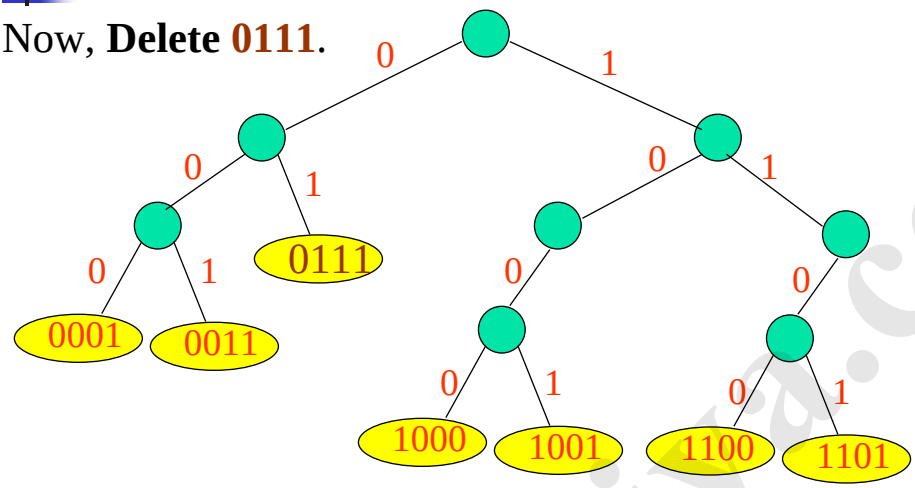
C-C Tsai P.12





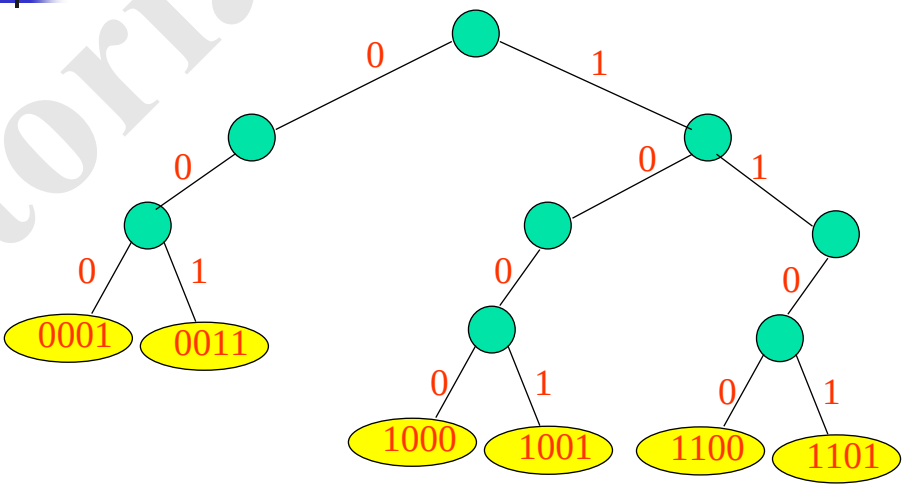
Fixed Length Delete

Now, **Delete 0111**.



C-C Tsai P.15

Fixed Length Delete

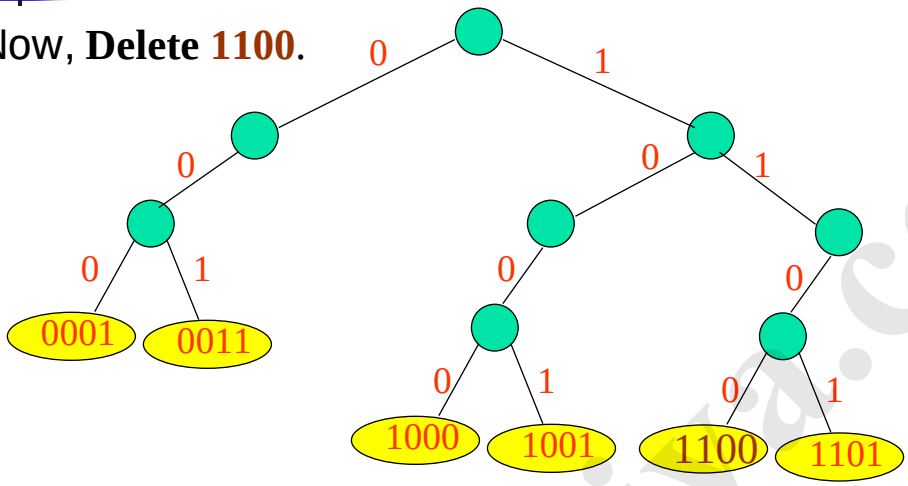


Delete 0111. One compare.

C-C Tsai P.16

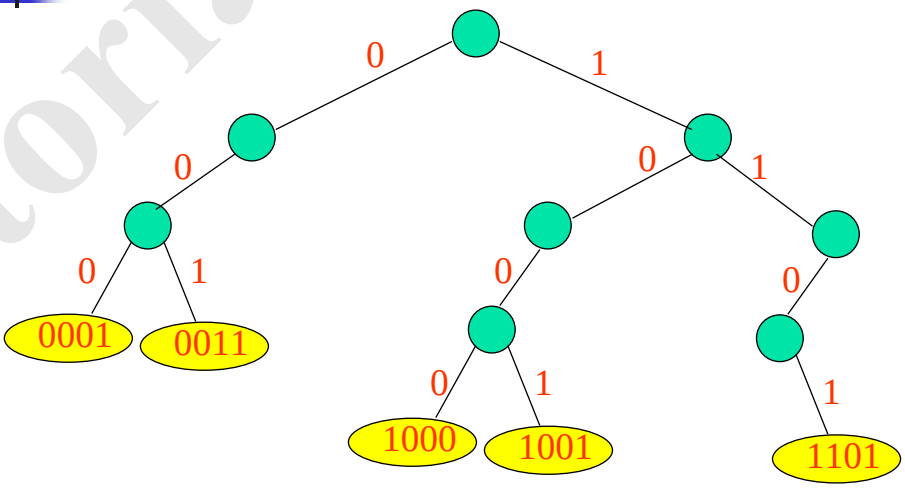
Fixed Length Delete

Now, **Delete 1100**.



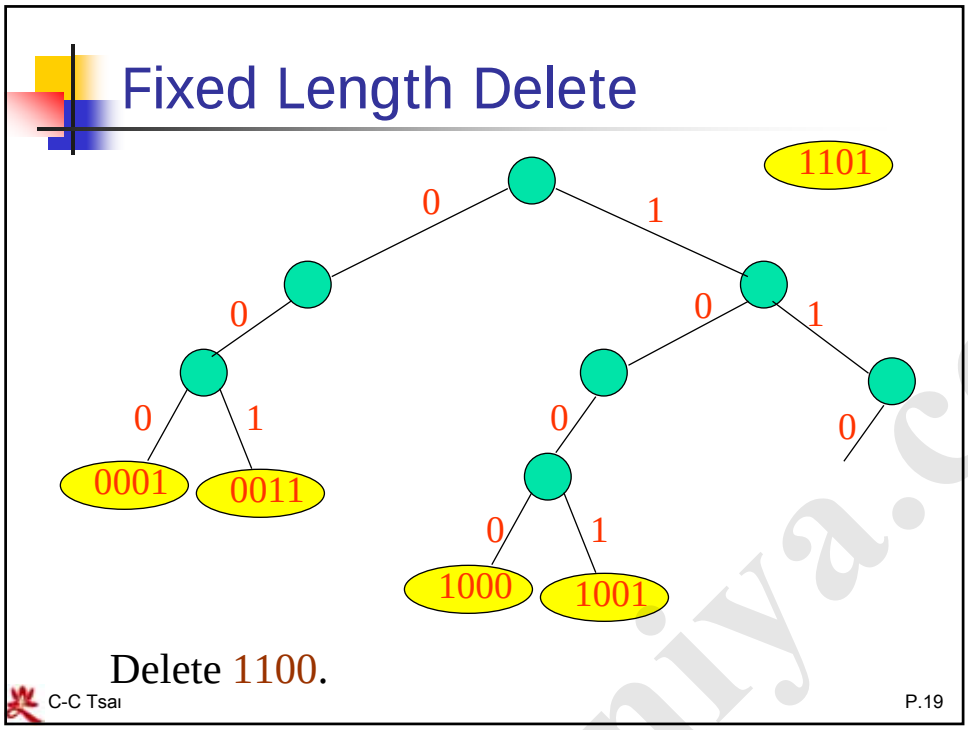
C-C Tsai P.17

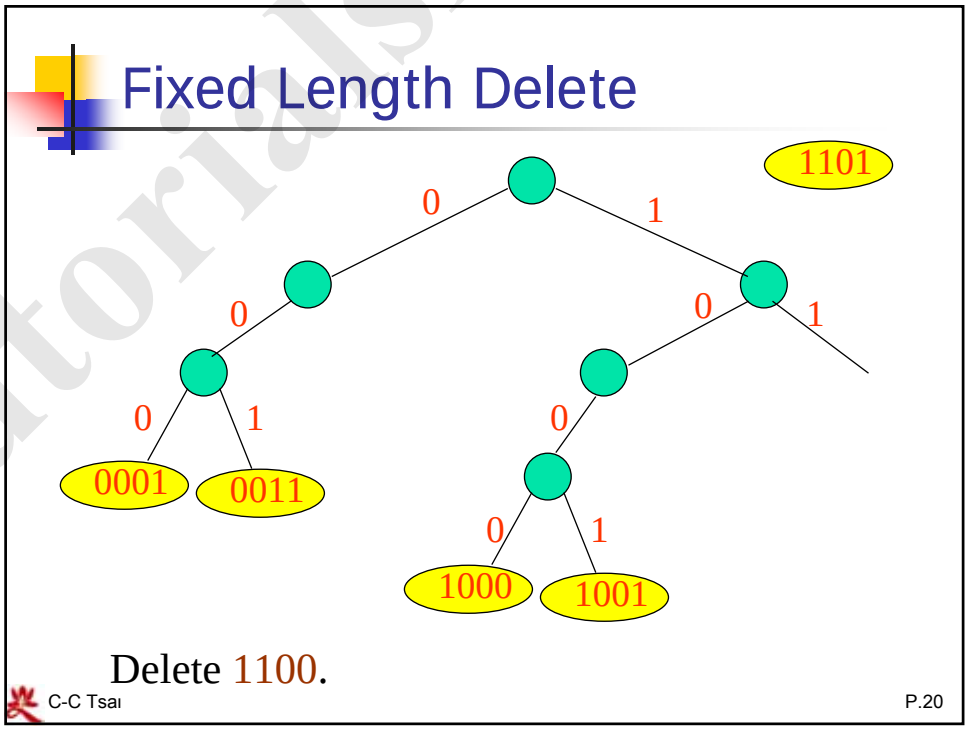
Fixed Length Delete



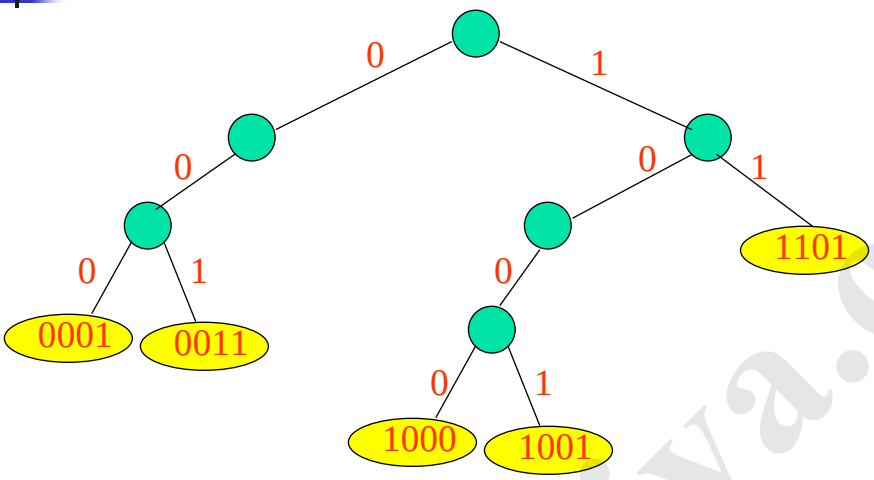
Delete 1100.

C-C Tsai P.18





Fixed Length Delete



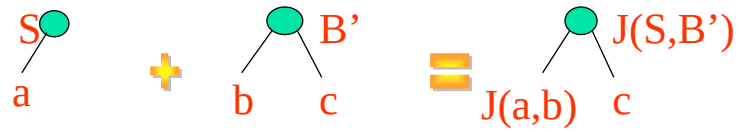
Delete 1100. One compare.

Fixed Length Join(S,m,B)

- Insert m into B to get B'.
- S empty => B' is answer; done.
- S is element node => insert S element into B'; done;
- B' is element node => insert B' element into S; done;
- If you get to this step, the roots of S and B' are branch nodes.

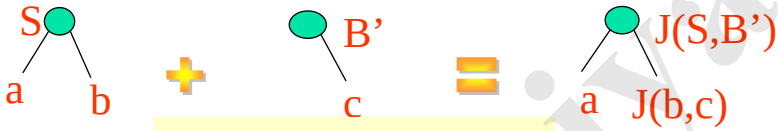
Fixed Length Join(S,m,B)

- S has empty right subtree.



J(X,Y) ⇨join X and Y, all keys in X < all in Y.

- S has nonempty right subtree.
  - Left subtree of B' must be empty, because all keys in B' > all keys in S.

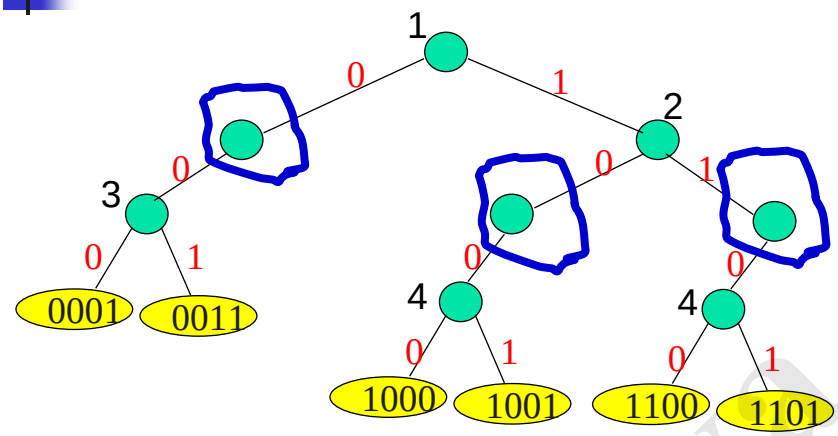


Complexity = O(height).

Compressed Binary Tries

- No branch node whose degree is 1.
- Add a bit# field to each branch node.
- bit# tells you which bit of the key to use to decide whether to move to the left or right subtrie.

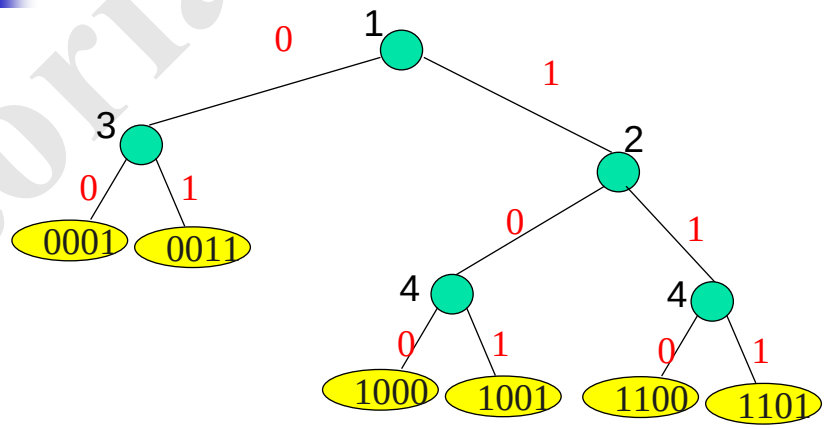
Example: Binary Trie



- **bit#** field shown in black outside branch node.

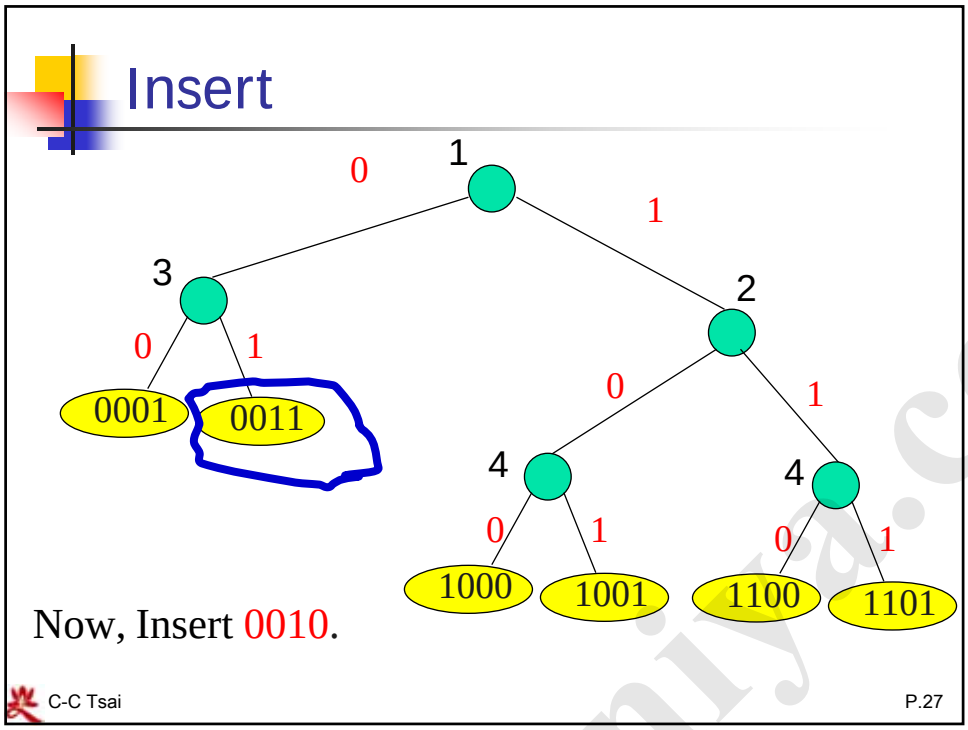
C-C Tsai P.25

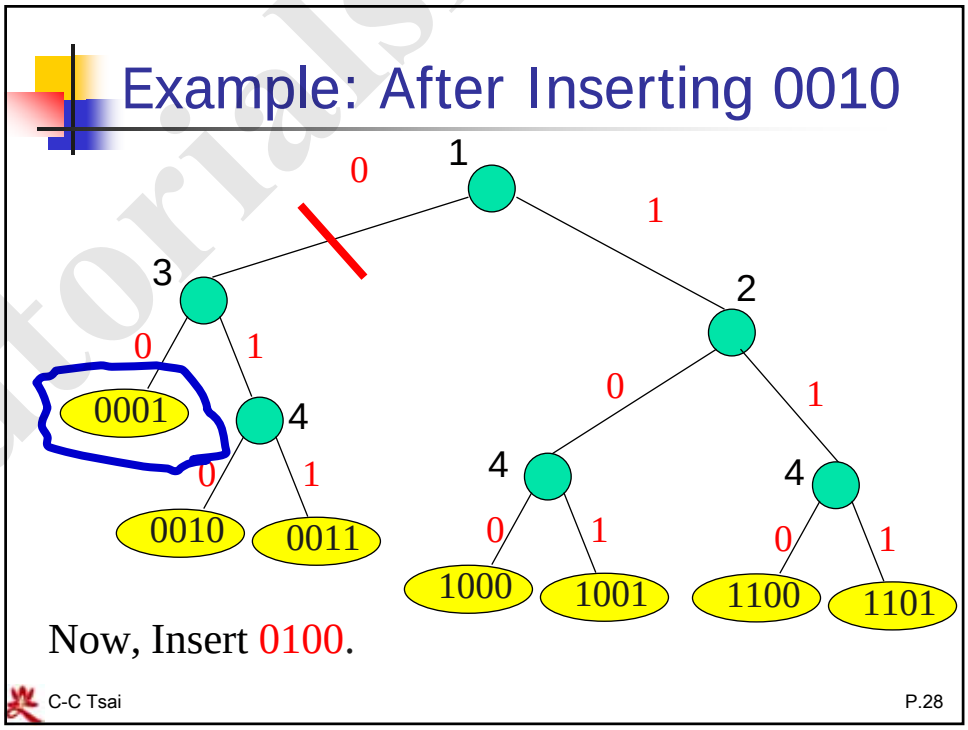
Example: Compressed Binary Trie

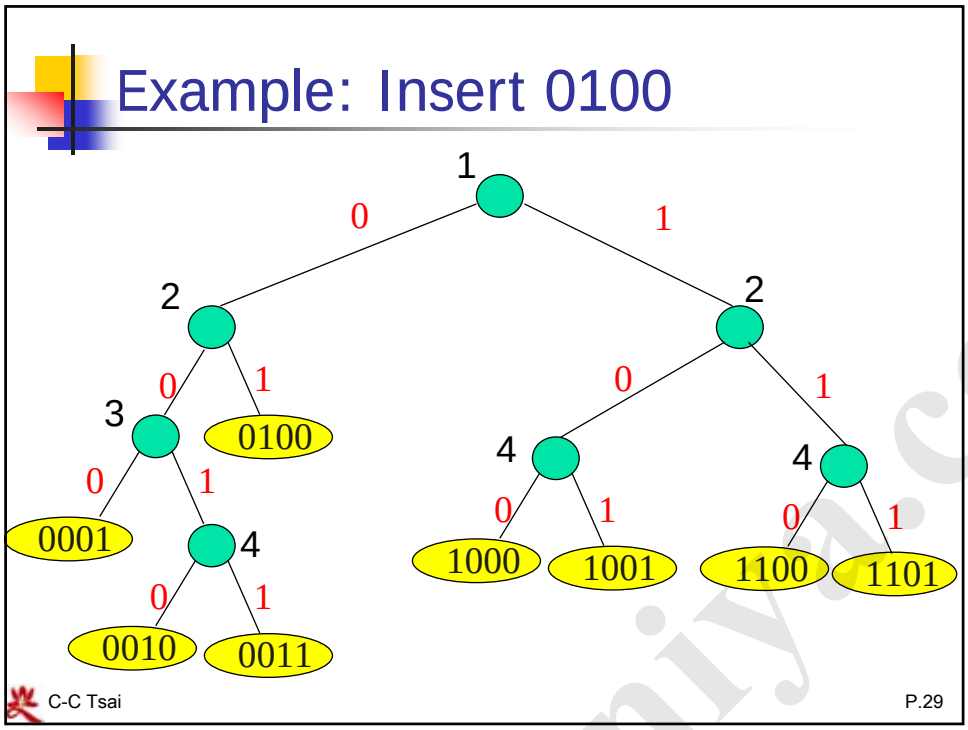


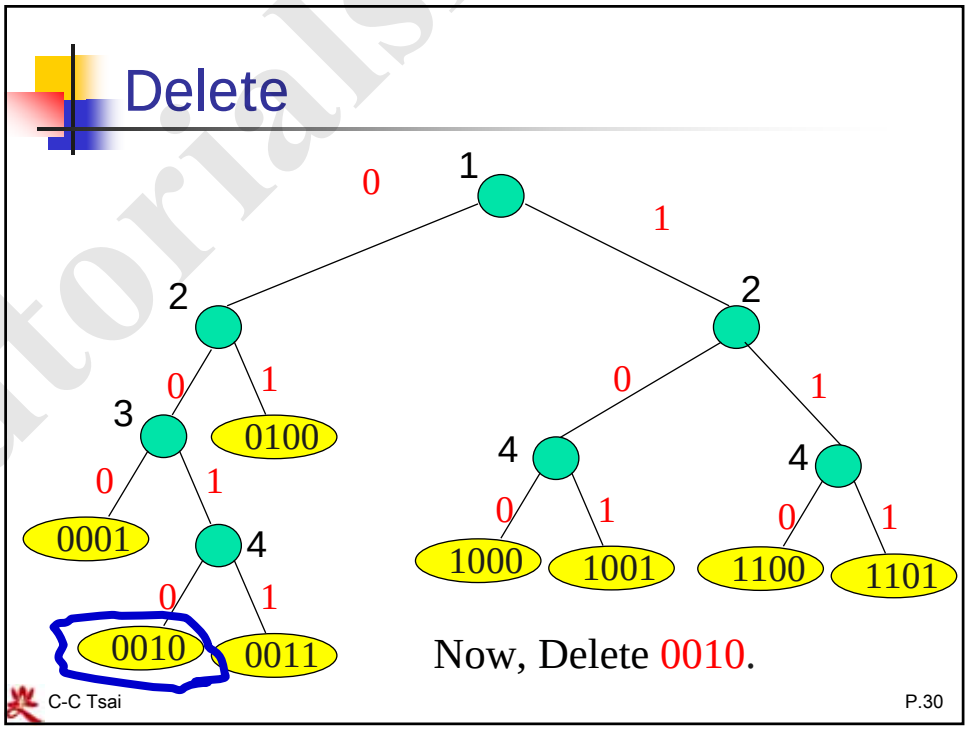
- bit# field shown in black outside branch node.
- #branch nodes = **n – 1**.

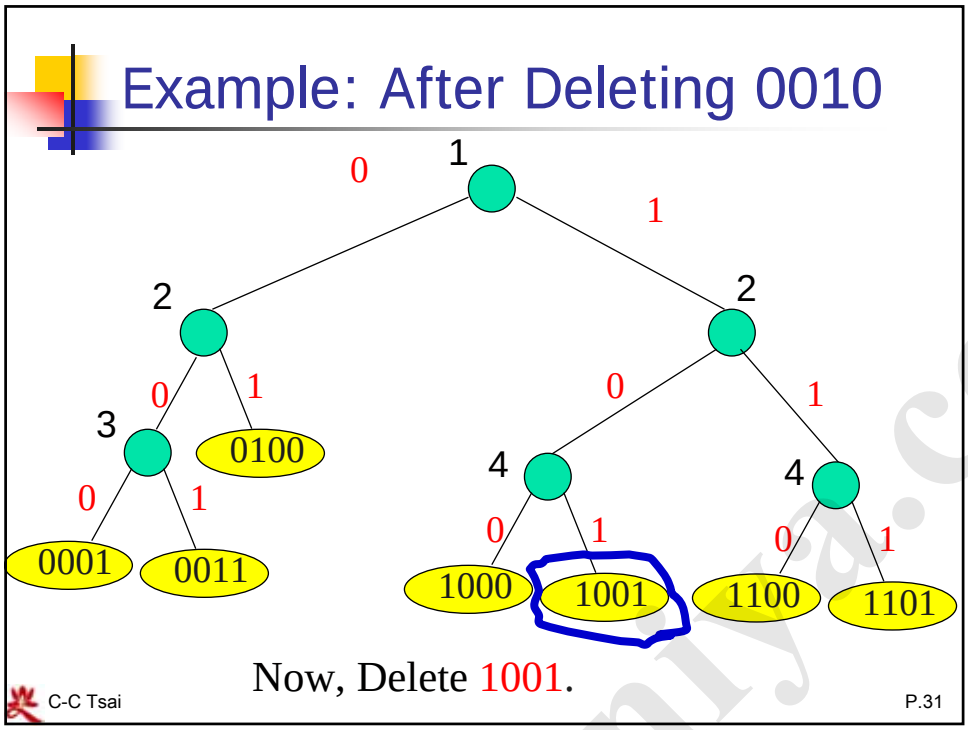
C-C Tsai P.26

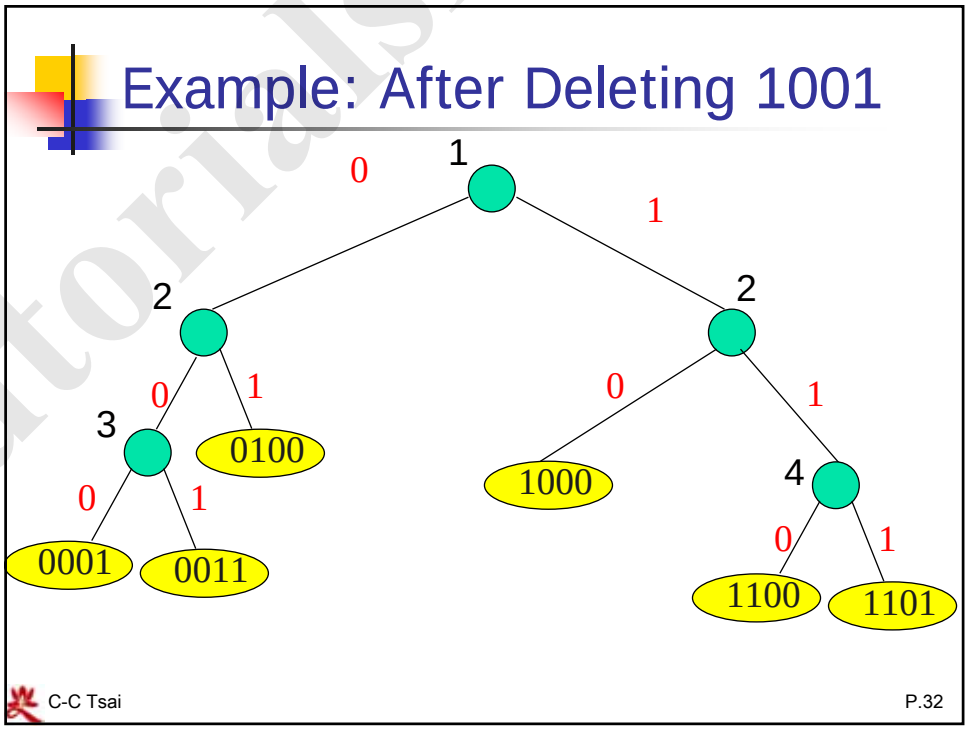


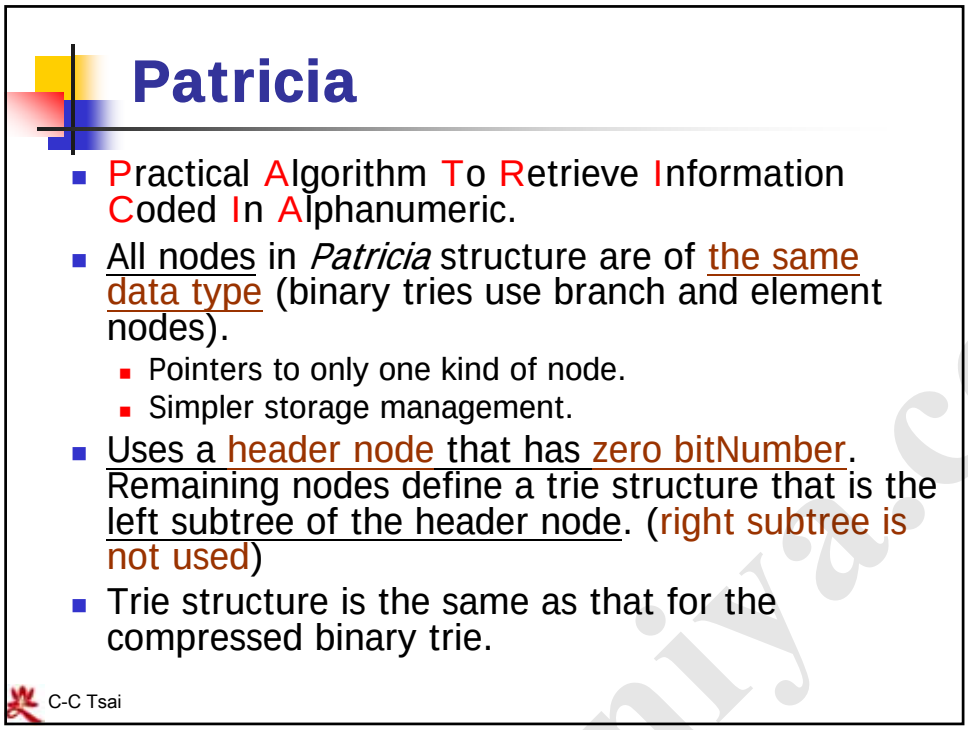


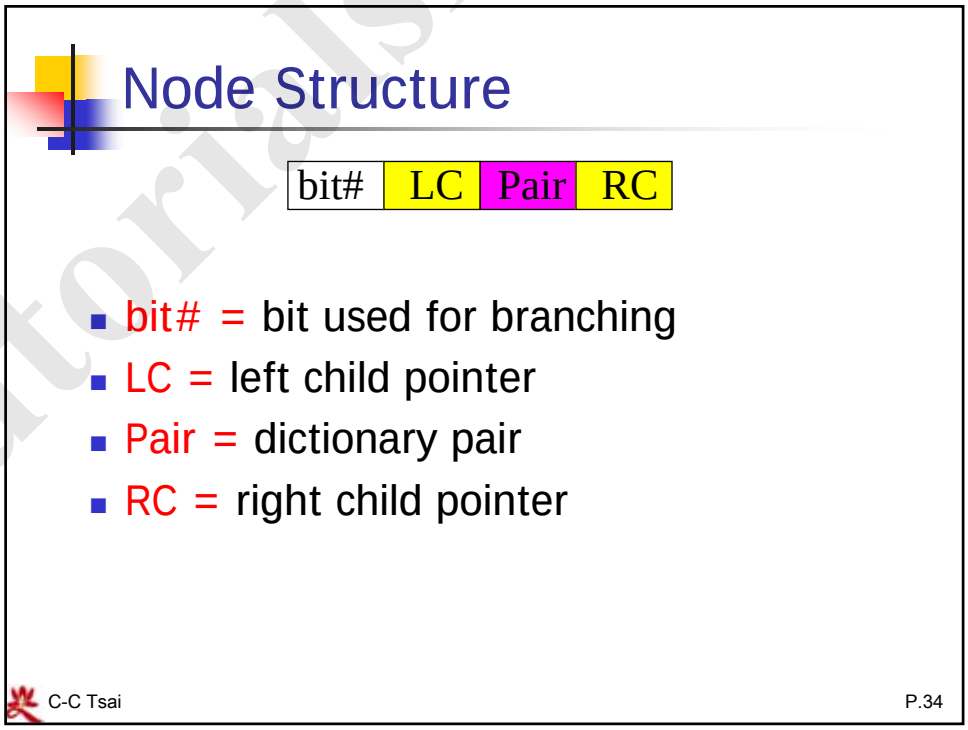




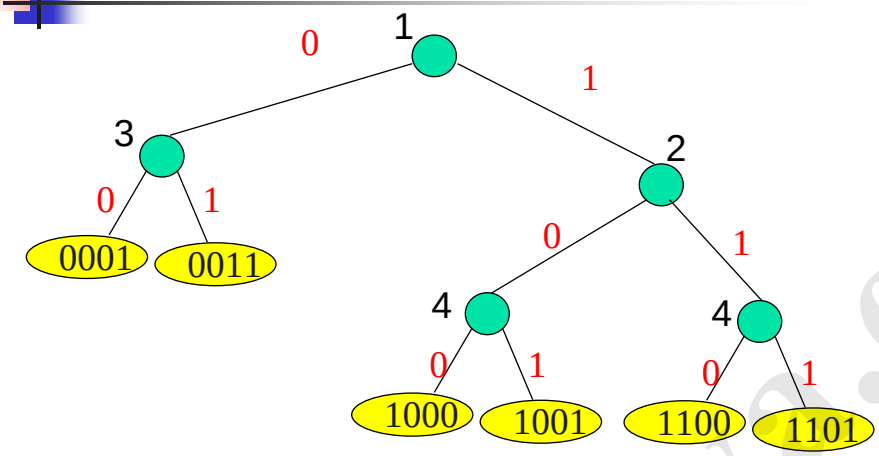






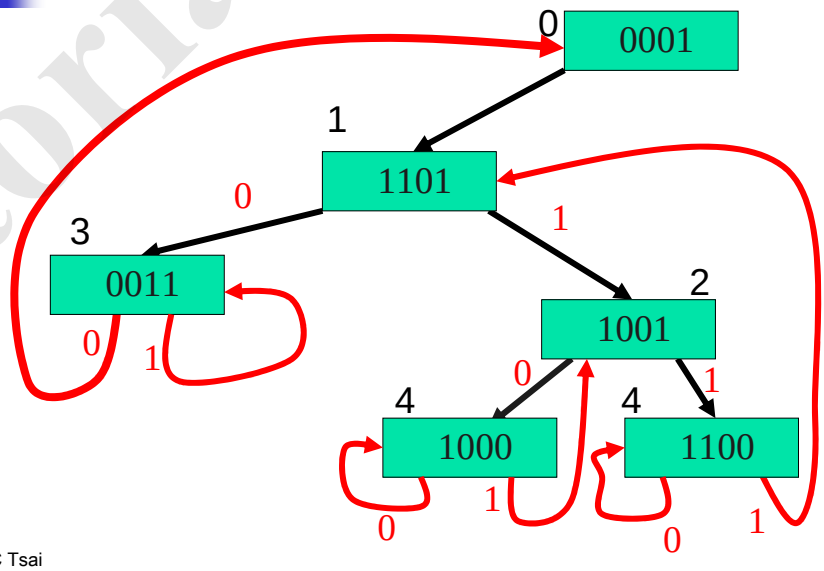


Compressed Binary Trie To Patricia



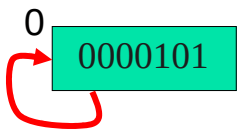
- Move each element into an ancestor or header node.

Example: Compressed Binary Trie To Patricia

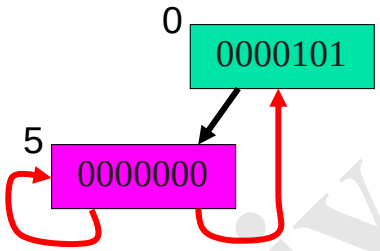


Insert

- Insert 0000101



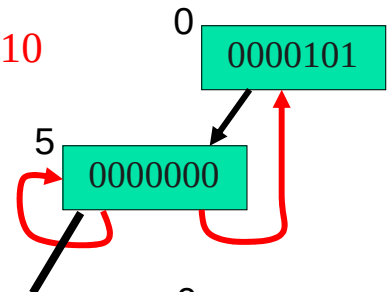
- Insert 0000000



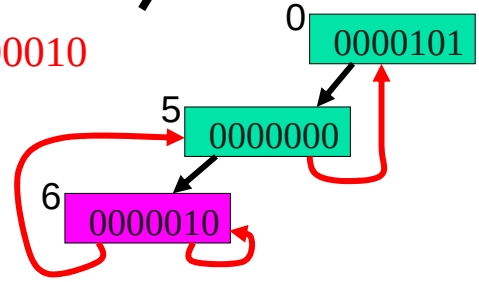
C-C Tsai P.37

Insert

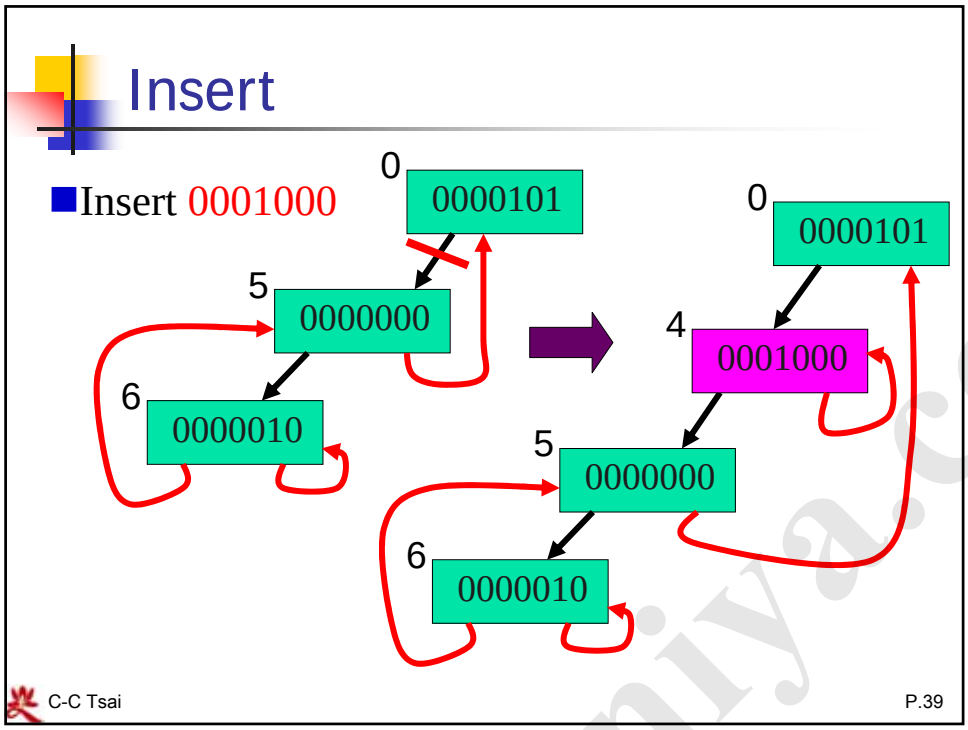
- Now, Insert 0000010

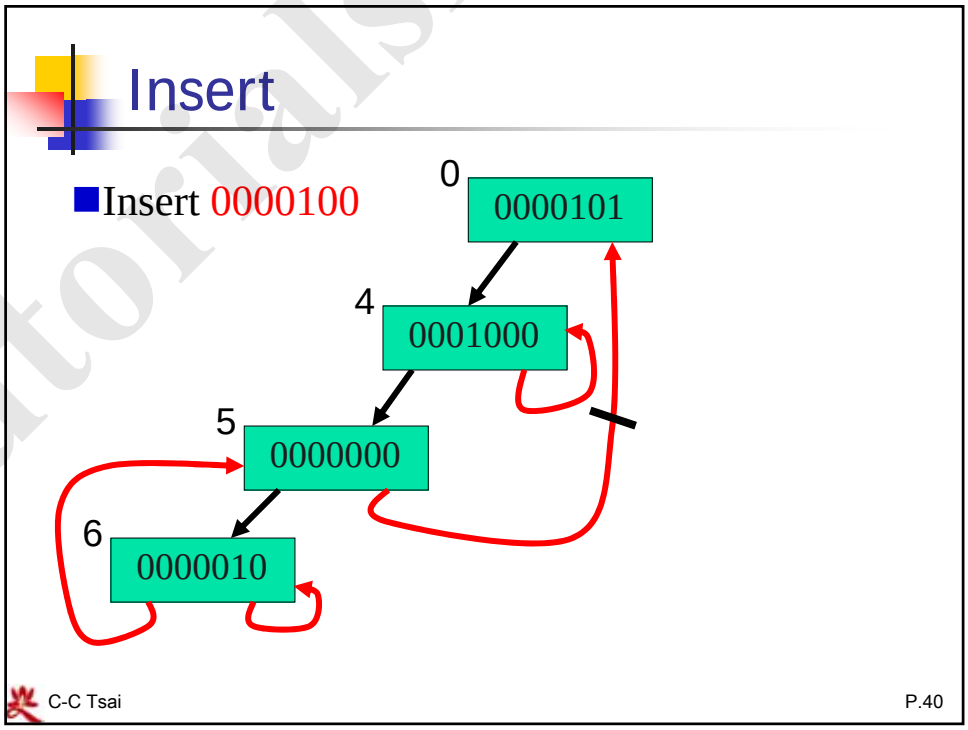


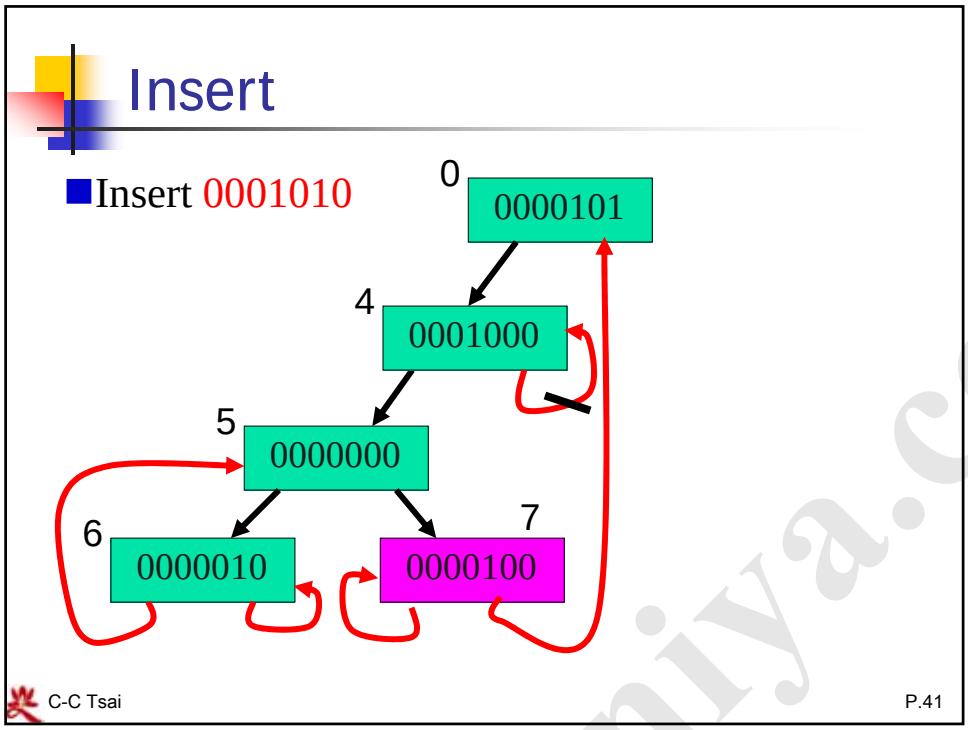
- Inserted 0000010

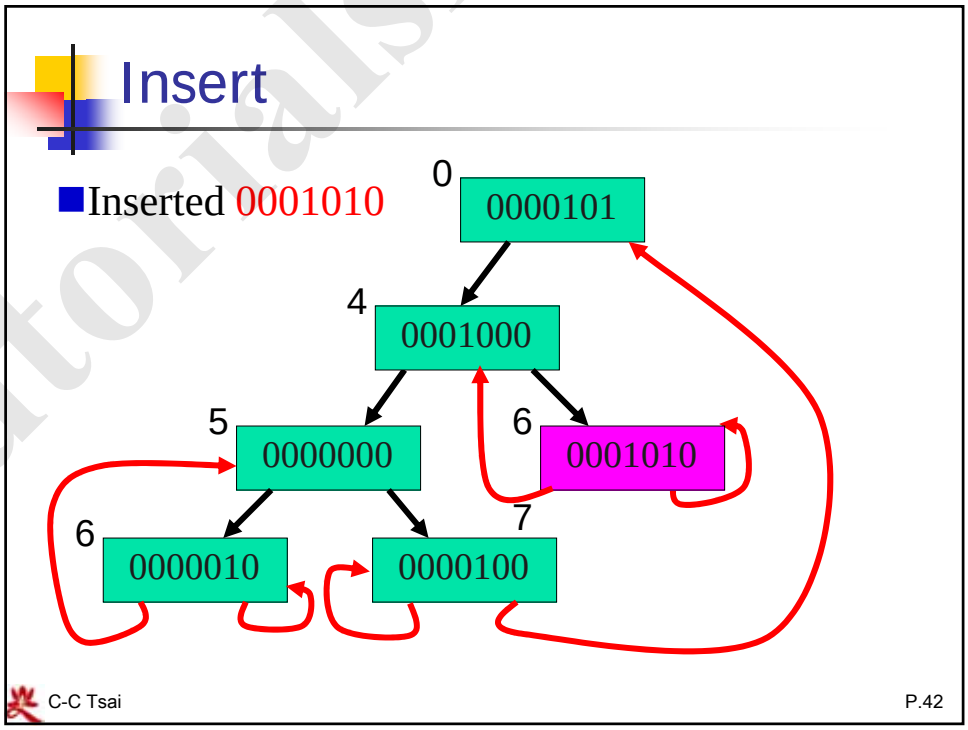


C-C Tsai P.38









TutorialsDuniya.com

Download FREE Computer Science Notes, Programs, Projects, Books PDF for any university student of BCA, MCA, B.Sc, B.Tech CSE, M.Sc, M.Tech at https://www.tutorialsduniya.com

- Algorithms Notes
- Artificial Intelligence
- Android Programming
- C & C++ Programming
- Combinatorial Optimization
- Computer Graphics
- Computer Networks
- Computer System Architecture
- DBMS & SQL Notes
- Data Analysis & Visualization
- Data Mining
- Data Science
- Data Structures
- Deep Learning
- Digital Image Processing
- Discrete Mathematics
- Information Security
- Internet Technologies
- Java Programming
- JavaScript & jQuery
- Machine Learning
- Microprocessor
- Operating System
- Operational Research
- PHP Notes
- Python Programming
- R Programming
- Software Engineering
- System Programming
- Theory of Computation
- Unix Network Programming
- Web Design & Development

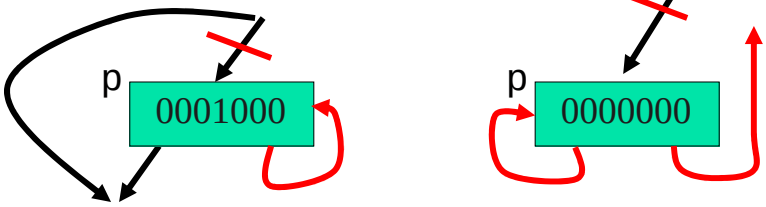
Please Share these Notes with your Friends as well

Delete

- Let p be the node that contains the dictionary pair that is to be deleted.
- Case 1: p has one self pointer.
- Case 2: p has no self pointer.

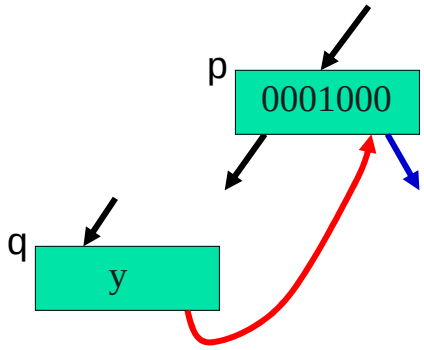
p Has One Self Pointer

- p = header => trie is now empty.
  - Set trie pointer to null.
- p != header => remove node p and update pointer to p.



p Has No Self Pointer

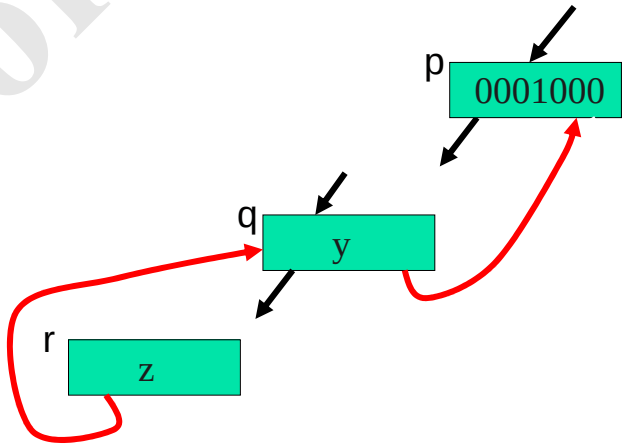
- Let q be the node that has a back pointer to p.
- Node q was determined during the search for the pair with the delete key k.



Blue pointer could be red or black.

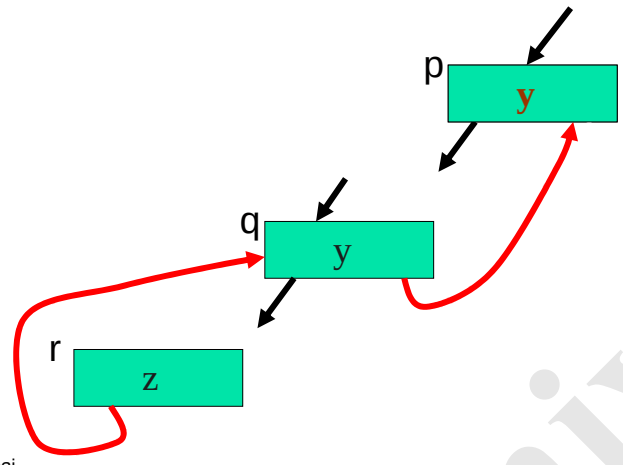
p Has No Self Pointer

- Use the key y in node q to find the unique node r that has a back pointer to node q.



p Has No Self Pointer

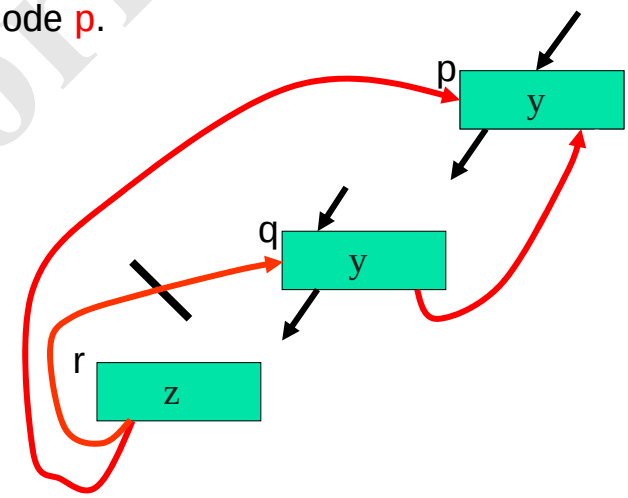
- Copy the pair whose key is y to node p.



C-C Tsai P.47

p Has No Self Pointer

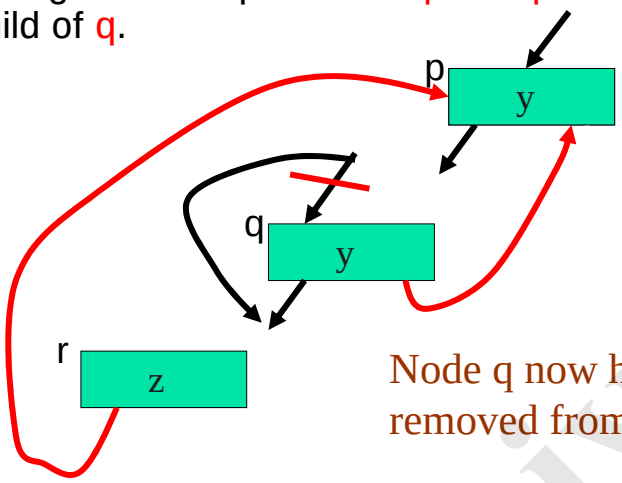
- Change back pointer to q in node r to point to node p.



C-C Tsai P.48

p Has No Self Pointer

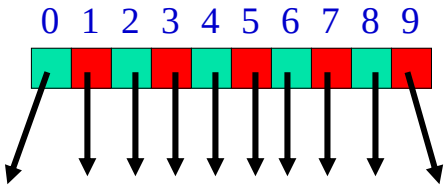
- Change forward pointer to q from parent(q) to child of q.



Node q now has been removed from trie.

Multiway Tries

- Key = Social Security Number.
  - 441-12-1135
  - 9 decimal digits.
- 10-way trie (order 10 trie).



Height <= 10.

Social Security Trie

- 10-way trie
  - Height <= 10.
  - Search: <= 9 branches on digits plus 1 compare.
- 100-way trie
  - 441-12-1135
  - Height <= 6.
  - Search: <= 5 branches on digits plus 1 compare.

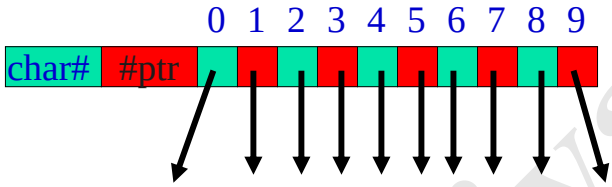
Social Security AVL & Red-Black

- Red-black tree
  - Height <= $2\log_2 10^9 \sim 60$.
  - Search: <= 60 compares of 9 digit numbers.
- AVL tree
  - Height <= $1.44\log_2 10^9 \sim 40$.
  - Search: <= 40 compares of 9 digit numbers.
- Best binary tree.
  - Height = $\log_2 10^9 \sim 30$.

Compressed Social Security Trie

- char# = character/digit used for branching.
  - Equivalent to bit# field of compressed binary trie.
- #ptr = # of nonnull pointers in the node.

Branch Node Structure

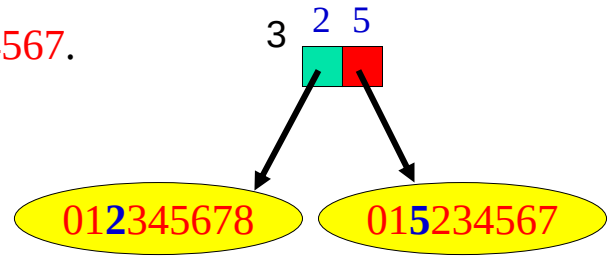


C-C Tsai P.53

Insert

- Insert 012345678.

- Insert 015234567.



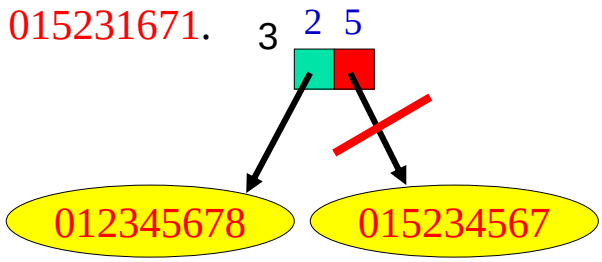
3: The 3rd digit is used for branching

Null pointer fields not shown.

C-C Tsai P.54

Insert

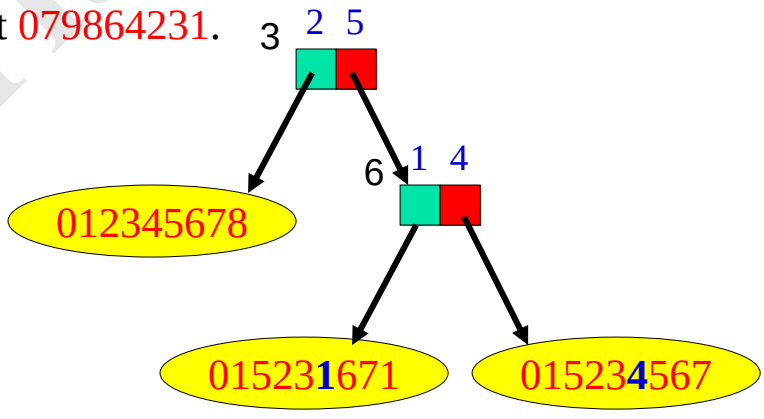
- Insert 015231671.



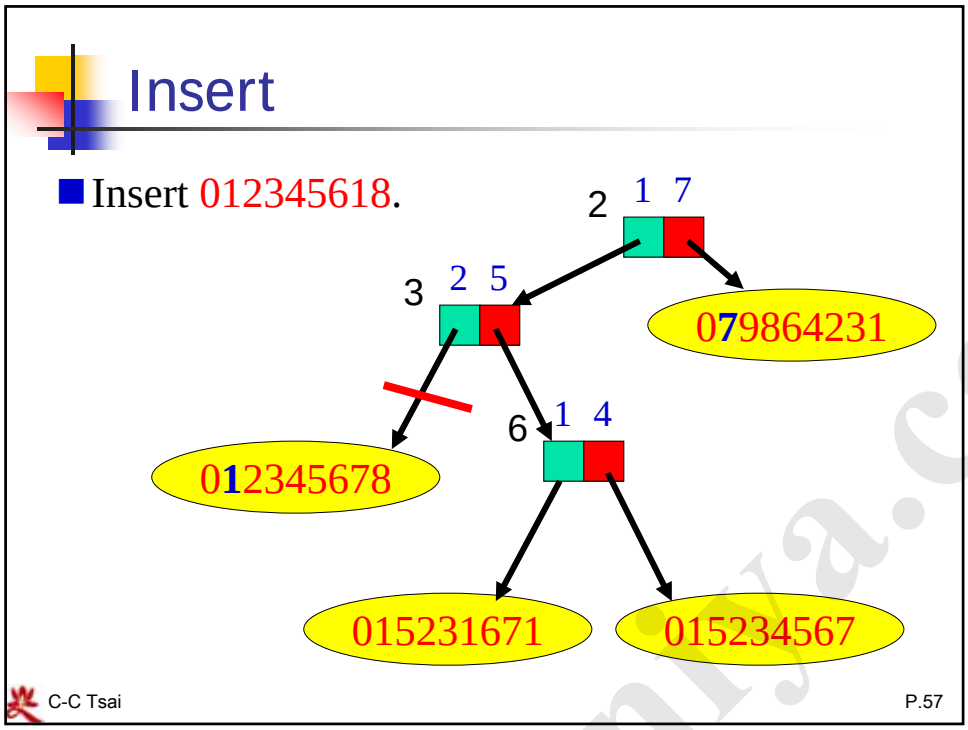
C-C Tsai P.55

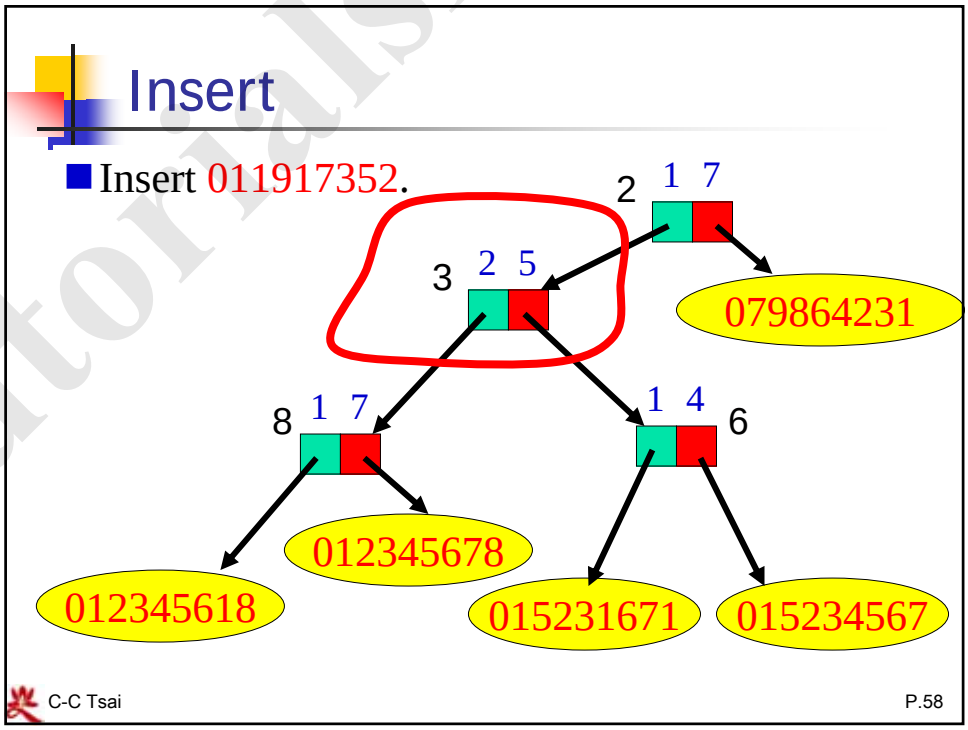
Insert

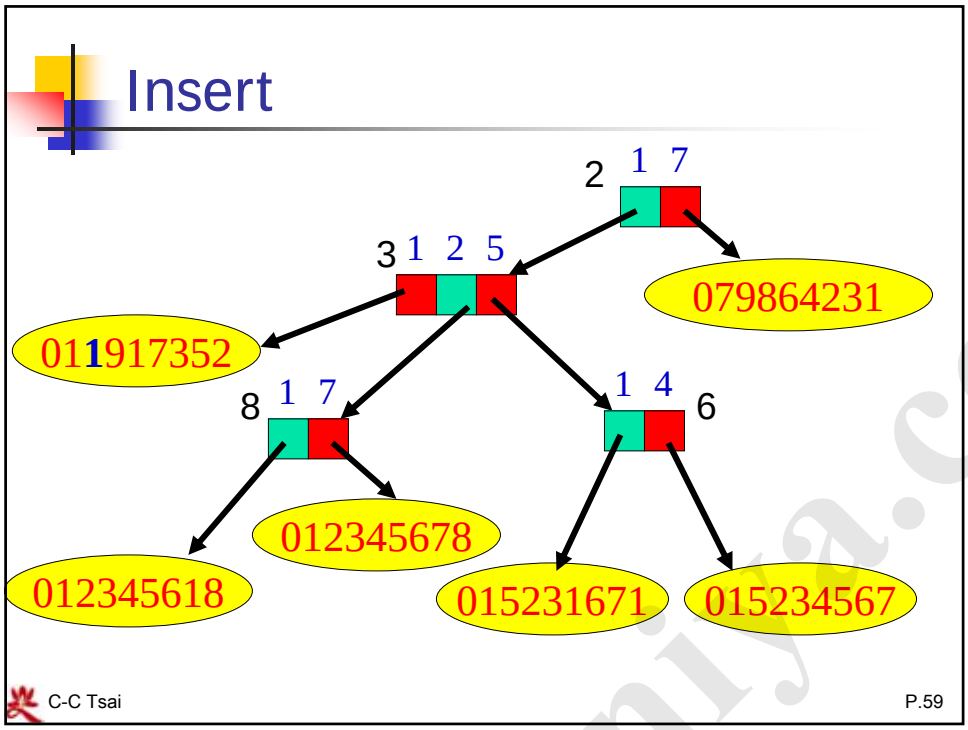
- Insert 079864231.

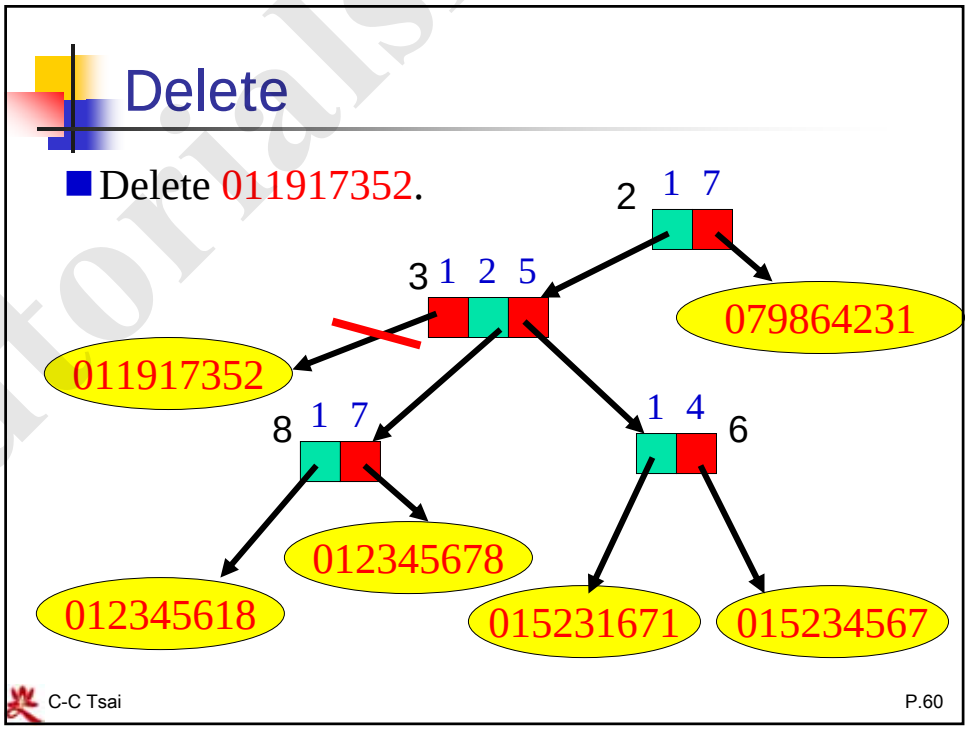


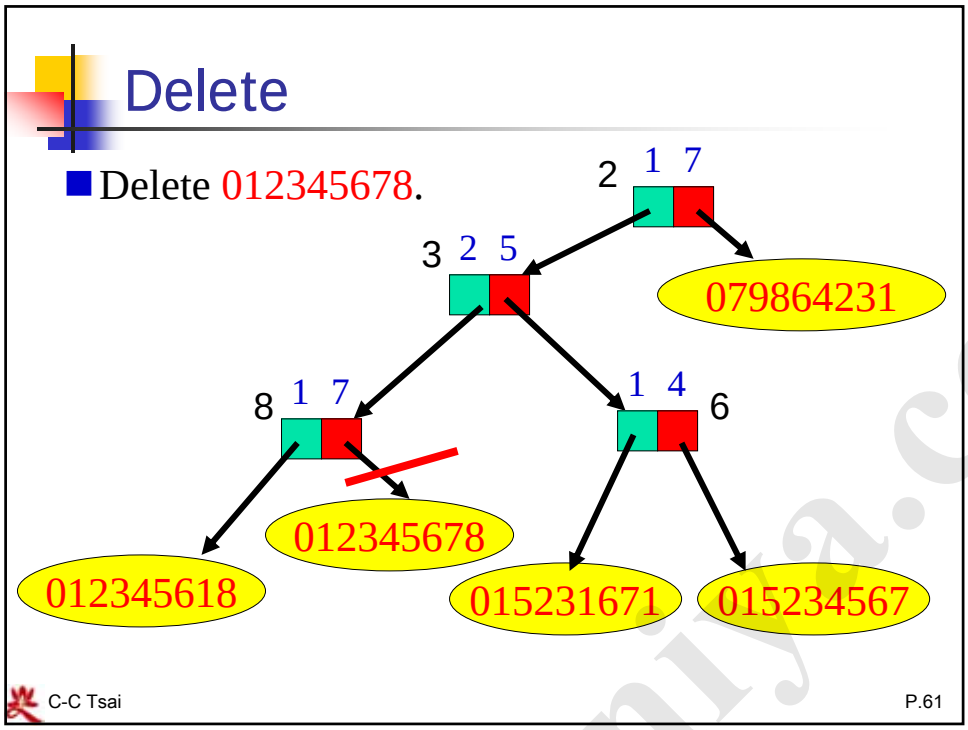
C-C Tsai P.56

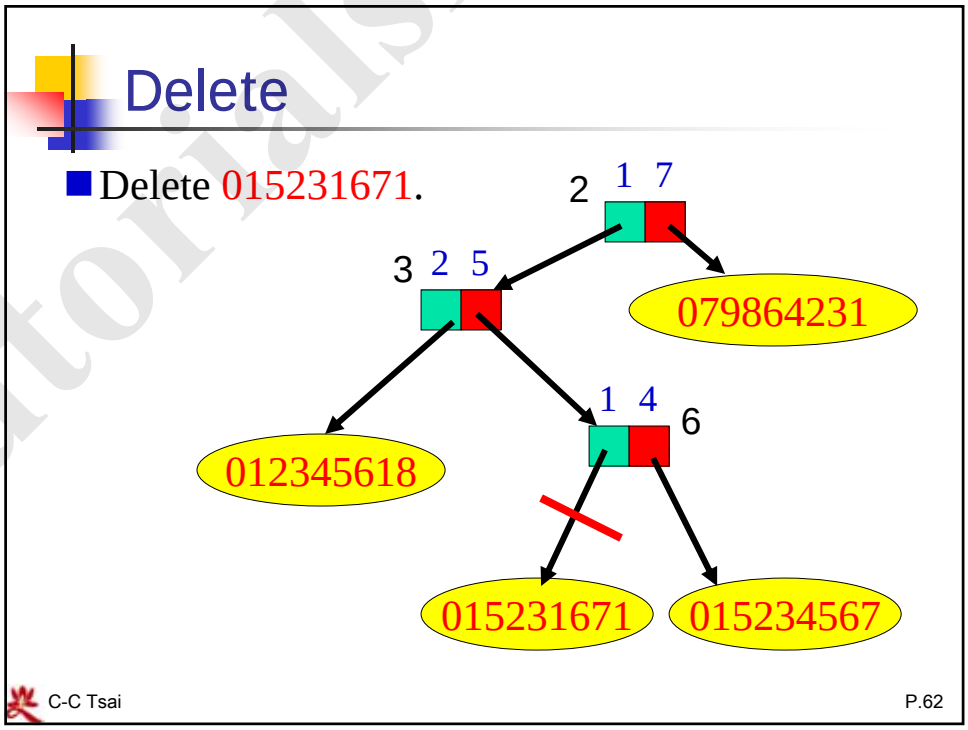




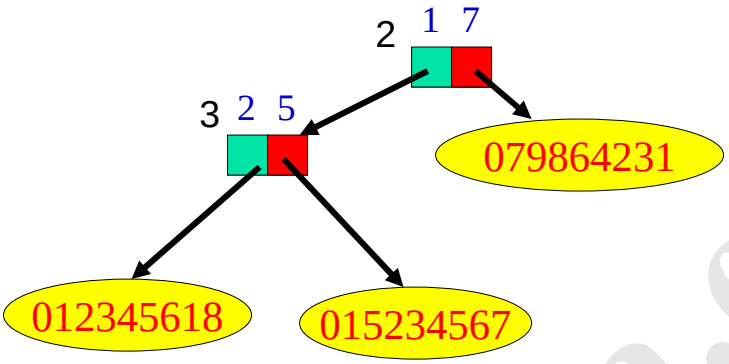








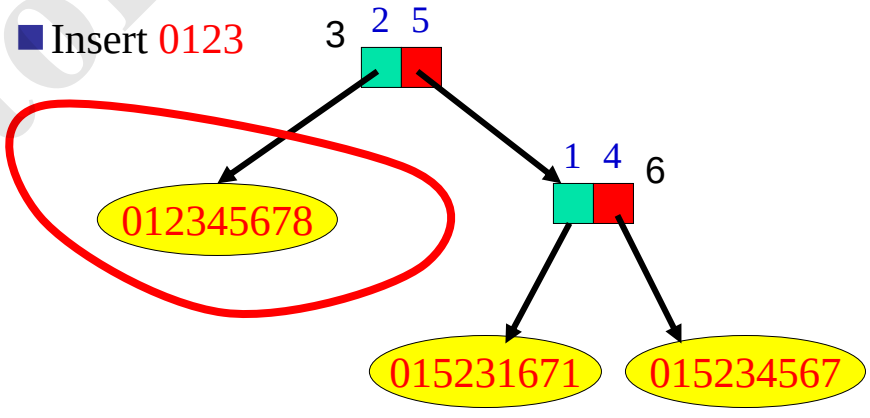
Delete



C-C Tsai P.63

Variable Length Keys

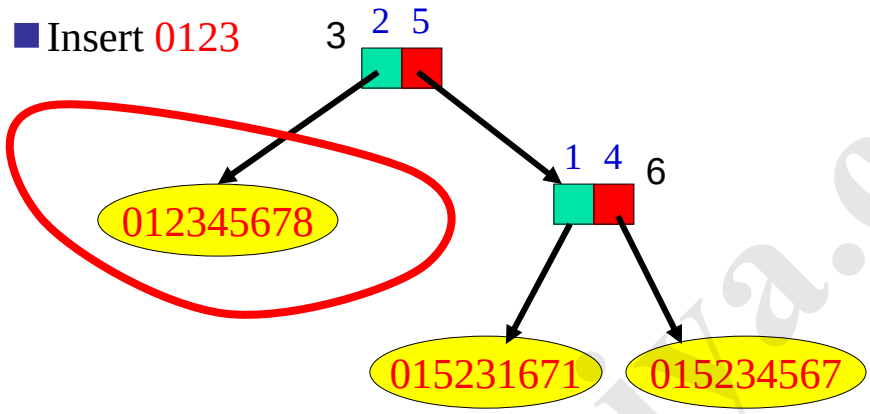
- Problem arises only when one key is a (proper) prefix of another.
- Insert 0123



C-C Tsai P.64

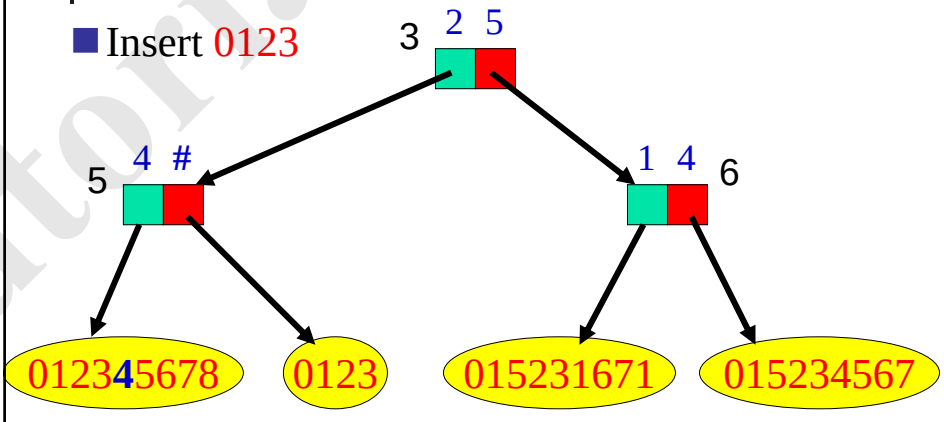
Variable Length Keys

- Add a special end of key character (#) to each key to eliminate this problem.
- Insert 0123



Variable Length Keys

- Insert 0123

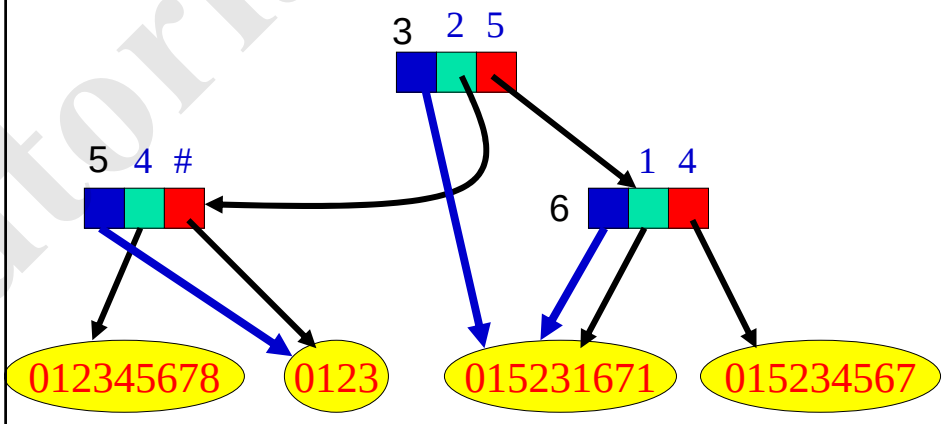


End of key character (#) not shown.

Tries With Edge Information

- Add a new field (element) to each branch node.
- New field points to any one of the element nodes in the subtree.
- Use this pointer on way down to figure out skipped-over characters.

Example



- element field shown in blue.

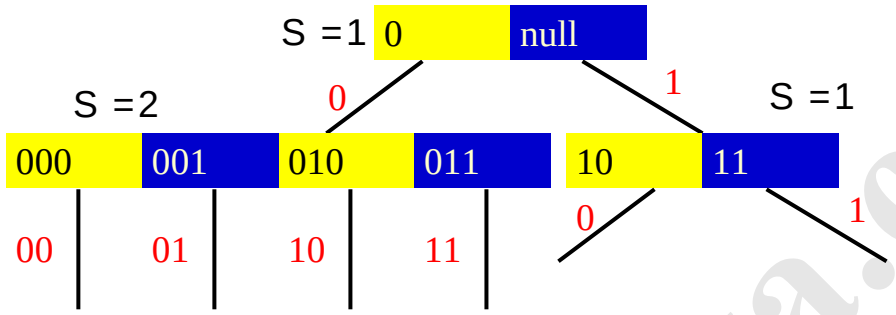
Trie Characteristics

- Expected height of an order m trie is ~$\log_m n$.
- Limit height to h (say 6). Level h branch nodes point to buckets that employ some other search structure for all keys in subtrie.
- Switch from trie scheme to simple array when number of pairs in subtrie becomes <= s (say s=6).
  - Expected # of branch nodes for an order m trie when n is large and m and s are small is n/(s ln m).
- Sample digits from right to left (instead of from left to right) or using a pseudorandom number generator so as to reduce trie height.

Multibit Tries

- Variant of binary trie in which the number of bits (stride) used for branching may vary from node to node.
- Proposed for Internet router applications.
  - Variable length prefixes.
  - Longest prefix match.
- Limit height by choosing node strides.
  - Root stride = 32 => height = 1.
  - Strides of 16, 8, and 8 for levels 1, 2, and 3 => only 3 levels.

Multibit Trie Example



C-C Tsai P.71

Multibit Tries

- Node whose stride is s uses s bits for branching.
  - Node has 2^s children and 2^s element/prefix fields.
  - Prefixes that end at a node are stored in that node.
  - Short prefixes are expanded to length represented by node.
  - When root stride is 3, prefixes whose length is < 3 are expanded to length 3.
  - P = 00* expands to P0 = 000* and P1 = 001*.
  - If Q = 000* already exists P0 is eliminated because Q represents a longer match for any destination.

C-C Tsai P.72

TutorialsDuniya.com

Download FREE Computer Science Notes, Programs, Projects, Books PDF for any university student of BCA, MCA, B.Sc, B.Tech CSE, M.Sc, M.Tech at https://www.tutorialsduniya.com

- Algorithms Notes
- Artificial Intelligence
- Android Programming
- C & C++ Programming
- Combinatorial Optimization
- Computer Graphics
- Computer Networks
- Computer System Architecture
- DBMS & SQL Notes
- Data Analysis & Visualization
- Data Mining
- Data Science
- Data Structures
- Deep Learning
- Digital Image Processing
- Discrete Mathematics
- Information Security
- Internet Technologies
- Java Programming
- JavaScript & jQuery
- Machine Learning
- Microprocessor
- Operating System
- Operational Research
- PHP Notes
- Python Programming
- R Programming
- Software Engineering
- System Programming
- Theory of Computation
- Unix Network Programming
- Web Design & Development

Please Share these Notes with your Friends as well