UNIT-II

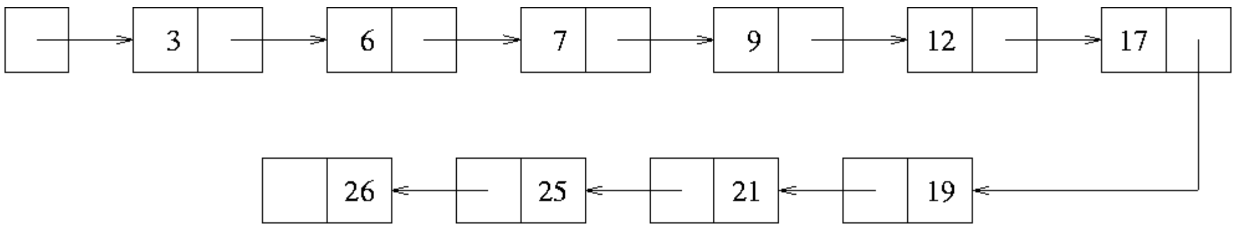
SKIP LISTS

Skip list is a type of data structure that can be used as an alternative to balanced (binary) trees or B-Trees. As compared to a binary tree, skip lists allow quick search, insertions and deletions of elements with simple algorithms. This is achieved by using probabilistic balancing rather than strictly enforce balancing as done in B-trees.

Skip lists are also significantly faster than equivalent algorithms for B-Trees.
A skip list is basically a linked list with additional pointers that allow intermediate nodes to be skipped, hence the name skip list.

In a simple linked list that consists of 'n' elements, to perform a search 'n' comparisons are required in the worst case.

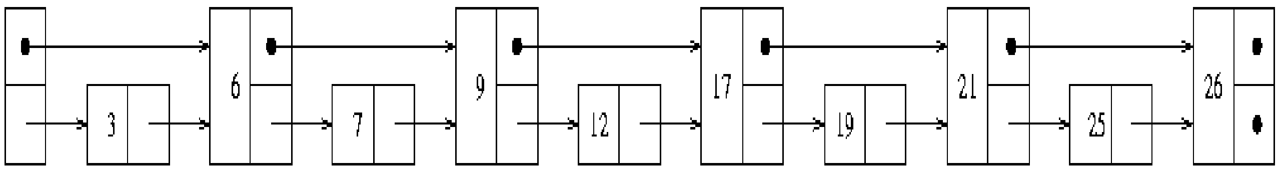
For example:



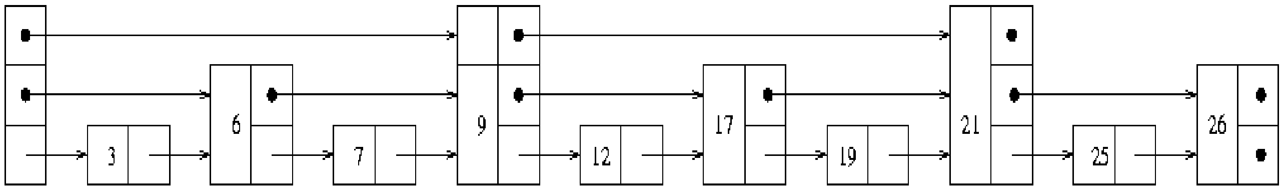
If a second pointer pointing two nodes ahead is added to every node, the number of comparisons goes down to n/2+1 in the worst case.

Consider a stored list where every other node has an additional pointer, to the node two a head of it in the list.

Here, every other node has an additional pointer.



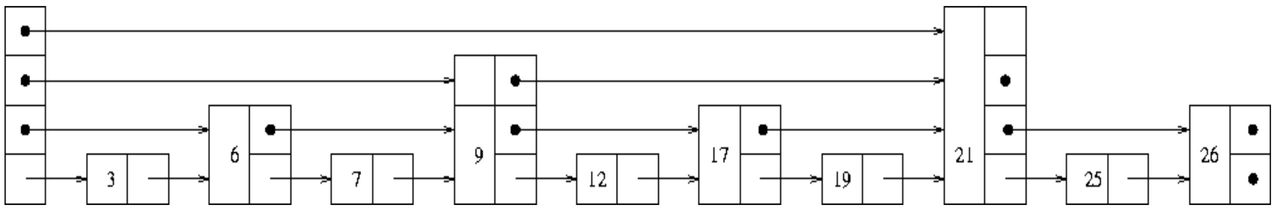
Next, every second node has a pointer two ahead of it



In the list of above figure, every second node has a pointer two ahead of it; every fourth node has a pointer four ahead if it. Here we need to examine no more than $\lceil \frac{n}{4} \rceil$ + 2 nodes.

In below figure, (every $(2^i)^{th}$ node has a pointer (2^i) node ahead (i = 1, 2,...); then the number of nodes to be examined can be reduced to $\lceil \log_2 n \rceil$ while only doubling the number of pointers.

Here, Every $(2^i)^{th}$ node has a pointer to a node (2^i)nodes ahead (i = 1, 2,...)



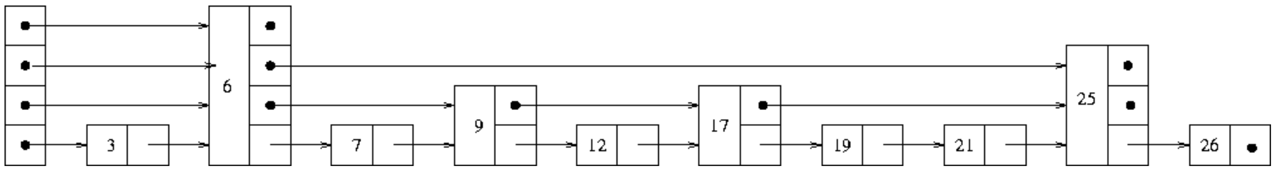
- A node that has k forward pointers is called a *level k* node. If every $(2^i)^{th}$ node has a pointer (2^i) nodes ahead, then
  # of level 1 nodes 50 %
  # of level 2 nodes 25 %
  # of level 3 nodes 12.5 %
- Such a data structure can be used for fast searching but insertions and deletions will be extremely cumbersome, since levels of nodes will have to change.
- What would happen if the levels of nodes were randomly chosen but in the same proportions (below figure)?
  - level of a node is chosen randomly when the node is inserted
  - A node's i^{th} pointer, instead of pointing to a node that is 2^{i-1} nodes ahead, points to the next node of level i or higher.
  - In this case, insertions and deletions will not change the level of any node.

  - Some arrangements of levels would give poor execution times but it can be shown that such arrangements are rare.

Such a linked representation is called a skip list.

- Each element is represented by a node the level of which is chosen randomly when the node is inserted, without regard for the number of elements in the data structure.
- A level *i* node has *i* forward pointers, indexed 1 through *i*. There is no need to store the level of a node in the node.
- Maxlevel is the maximum number of levels in a node.
  - Level of a list = Maxlevel
  - Level of empty list = 1
  - Level of header = Maxlevel

It is a skip list



Initialization:

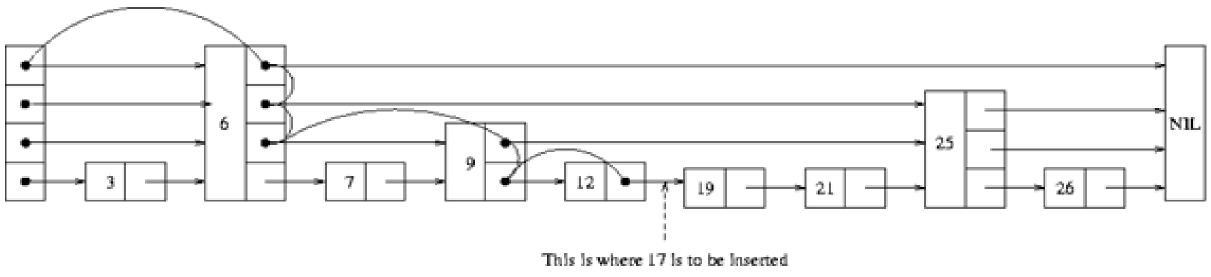
An element NIL is allocated and given a key greater than any legal key. All levels of all lists are terminated with NIL. A new list is initialized so that the level of list = maxlevel and all forward pointers of the list's header point to NIL

Search:

We search for an element by traversing forward pointers that do not overshoot the node containing the element being searched for. When no more progress can be made at the current level of forward pointers, the search moves down to the next level. When we can make no more progress at level 1, we must be immediately in front of the node that contains the desired element (if it is in the list).

Insertion and Deletion:

- Insertion and deletion are through search and splice
- update [*i*] contains a pointer to the rightmost node of level *i* or higher that is to the left of the location of insertion or deletion.
- If an insertion generates a node with a level greater than the previous maximum level, we update the maximum level and initialize appropriate portions of update list.
- After a deletion, we check to see if we have deleted the maximum level element of the list and if so, decrease the maximum level of the list.
- Below figure provides an example of Insert and Delete. The pseudo - code for Insert and Delete is shown below.



Analysis of Skip lists:

In a skiplist of 16 elements, we may have

- 9 elements at level 1
- 3 elements at level 2
- 3 elements at level 3
- 1 element at level 6

One important question is:

Where do we start our search? Analysis shows we should start from level $L(n)$ where

$$L(n) = \log_2 n$$

In general if p is the probability fraction,

$$L(n) = \log_{\frac{1}{p}} n$$

where p is the fraction of the nodes with level i pointers which also have level $(i + 1)$ pointers.

- However, starting at the highest level does not alter the efficiency in a significant way.
- Another important question to ask is:
  What should be MaxLevel? A good choice is

$$MaxLevel = L(N) = \log_{\frac{1}{p}} N$$

where *N* is an upperbound on the number of elements is a skiplist.

- Complexity of search, delete, insert is dominated by the time required to search for the appropriate element. This in turn is proportional to the length of the search path. This is determined by the pattern in which elements with different levels appear as we traverse the list.

- Insert and delete involve additional cost proportional to the level of the node being inserted or deleted.