Advanced List Structures – Skip Lists

Introduction

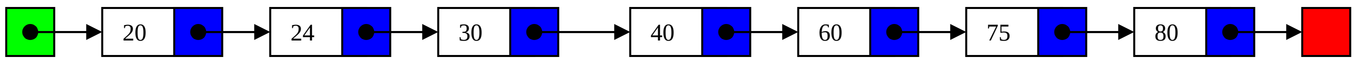
Linked lists have one serious drawback: They require sequential scanning to locate an element during a search. The search must start from the beginning of the list and terminates when either the search element is found or the end of the list is encountered without finding the search element. Ordering the elements in the list can speed up the searching, but a linear (sequential) search is still required.

There have been several hybrid list structures that have been developed which allow searching in a list to occur in other than sequential fashion. Most commonly these techniques allow the search to “skip over” certain nodes in the list to avoid the cost of the sequential search. The **skip list** is one such variant of the ordered list which make nonsequential searching possible.

[*Reference:* Pugh, William, “Skip Lists: A Probabilistic Alternative to Balanced Trees,” *Communications of the ACM 33*(6), (1990), 668-676.]

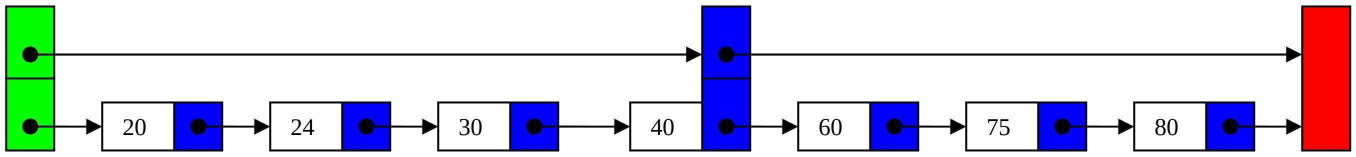
A Simple Skip List

Can you think of a simple way to reduce the worst case “cost” (in terms of the potential number of comparisons that need to be made against the search element) of a search in an ordered linear list of *n* nodes from *n* comparisons to *n/2+1* comparisons? The way to do this is to maintain one additional pointer (reference) to the middle element in the list. With this single additional reference, all searches begin with a comparison against this middle list element. If the search is for an element smaller than the middle element, the only the left-half of the list need to be searched and if the search element is larger than the middle element, only the right-half of the list will be searched. This technique is basically the simplest form of a skip list. In general, skip lists will reduce the cost of a given search much more dramatically, as we will shortly discover. Figures 1 and 2 shown below, illustrate the simple skip list principle.



(Figure 1) A simple ordered linked list with head and tail nodes

Now consider the simple ordered list shown above with the addition of a single pointer to the middle of the list, which in this case is the node containing the element 40. This new list is shown in the Figure 2.

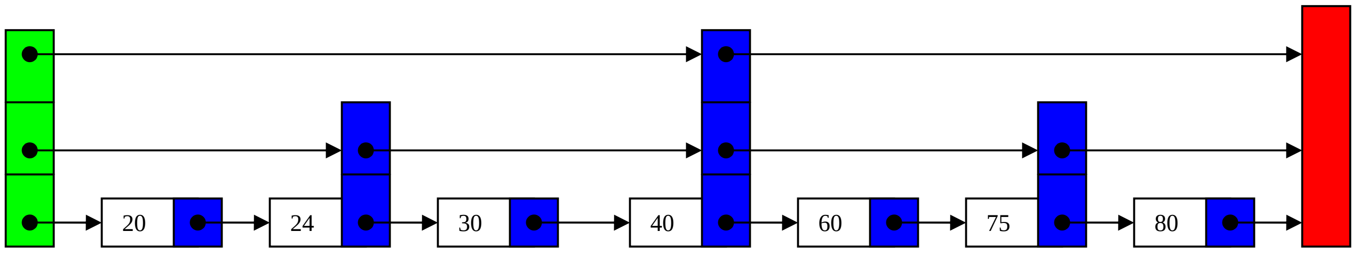


(Figure 2) A simple skip list with a pointer to the middle of the list

The number of nodes in the list in the two figures above is seven. If the search in the original ordered list were for the value 80, exactly seven (*n*) comparisons with values in the list would be performed before the search element was located in the list. Using the simple skip list the number of comparisons for this search decreases to exactly four (*n/2+1*).

Skip Lists

Skip lists, in general, extend the concept illustrated above with the simple skip lists to various levels of extreme. The level to which the skip list is extended depends upon the desired level of trade-off between the increasing level of complexity of the structure and the benefit in terms of reducing the number of comparisons required per search. Consider the skip list illustrated in Figure 3:



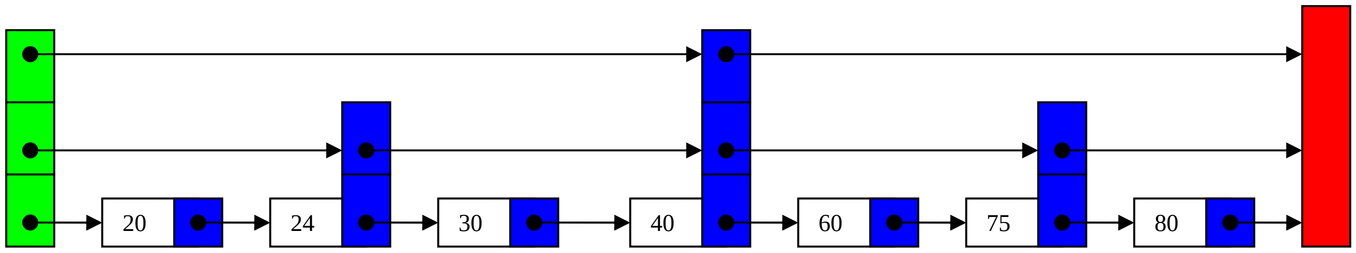
(Figure 3) Skip list with pointers to every second node

In the skip list shown in Figure 3 above, there are essentially three separate linked lists that are maintained within this structure. Working up from the bottom of this structure, the *level 0* list is essentially the list which was shown in Figure 1. The level 0 list maintains a pointer between each consecutive node in the list. Moving up one level, the *level 1* list maintains a pointer to every second node in the list. Moving up to the final level, the *level 2* list maintains a pointer to every fourth node in the list.

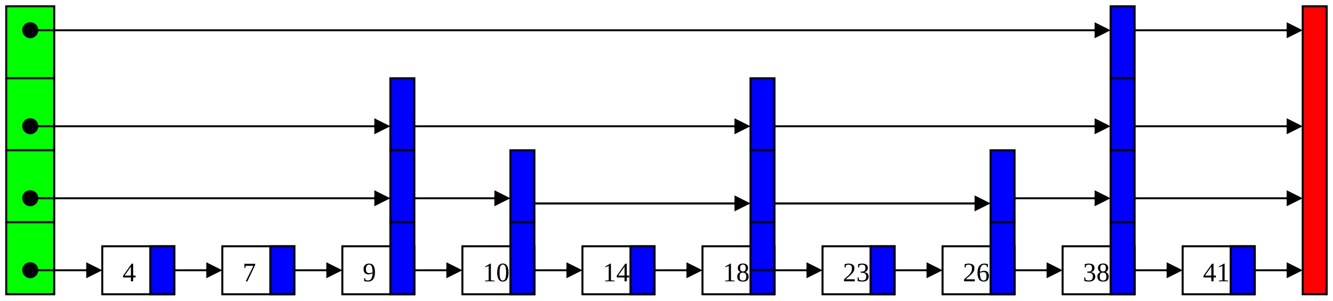
In general, an element in a skip list is called a ***level i element*** iff it is in the list for levels 0 through *i* and is not in the level *i+1* list (if this level list exists). In the example above, element 40 is the only level 2 element since it appears in levels

0, 1 and 2 (there is no level 3 in this case). Elements 24 and 75 are the level 1 elements (they appear in levels 0 and 1, but not 2). Elements 20, 30, 60, and 80 are the level 0 elements as they appear in the level 0 list but not in the level 1 list. In general, the *level i* list includes every 2^i th element that appears in the level 0 list.

A **skip list** is a structure that consists of a hierarchy of ordered linear lists. The level 0 list is a sorted list of all *n* elements. The level 1 list is an ordered linear list that consists of some subset of the elements in the level 0 list. In general, the level *i* list consists of a subset of the elements in the level *i-1* list. There are two general variations of skip lists: **regular skip lists** are those in which selected subsets generate evenly spaced "skips" and **irregular skip** lists, those in which the selected subsets generate unevenly spaced "skips". Figures 4 and 5 shown below illustrate the differences in the two variations of skip lists.



(Figure 4) Regular Skip List (3 levels in this example – numbered 0-2 bottom to top)



(Figure 5) Irregular Skip List (4 levels in this example – numbered 0-3 bottom to top)

From an analysis point of view, the performance of an irregular skip list for a random search or a sequence of random searches is much harder to determine than that of the regular skip list. Typically, irregular skip lists are developed using either randomized "skips" or those which are in some fashion determined based upon probability. We will concentrate on the regular skip list for the remainder of this set of notes and ignore the more complex irregular skip lists.

Regular Skip Lists

For a regular skip list of *n* nodes we have:

> For each *k* and *i* such that $1 \leq k \leq \lfloor \lg n \rfloor$ and $1 \leq i \leq \lfloor n/2^{k-1} \rfloor - 1$, the node in position $2^{k-1} \times i$ points to the node in position $2^{k-1} \times (i+1)$

This means (as we saw earlier) that the every second node points to the node two positions ahead (its second successor node), every fourth node points to the node four positions ahead (its fourth successor node), and so on. The figures illustrating the skip lists made apparent the fact that the various nodes in the skip list contain different numbers of reference fields. Half of the nodes contain just one reference field, one-fourth of the nodes will contain two reference fields, one-eighth of the nodes will contain three reference fields, and so on. The number of reference fields in a given node represents the *level* of the that node, and the number of levels in a regular skip list is given by:

$$\text{maximum\_level} = \lfloor \lg n \rfloor + 1.$$

Searching a Regular Skip List

Searching for an element consists of following references on the highest level of the structure until an element is found which successfully terminates the search. Successful termination of the search will occur whether the search element is in the skip list or not. Successful termination of the search will occur at some point within the structure if the search element exists in the skip list, otherwise successful termination will occur when the end of the level 0 list is reached and the search element was never encountered. At any point during the search if the end of a list (one of the level lists in the skip list) is reached or an element is encountered with a value greater than the search element, the search is restarted from the node preceding the one too large element just encountered , but this time, on the next level down.

Consider a search for the value 16 in the irregular skip list shown in Figure 5. The search begins at the head (root) node in the highest level list (level 3 in this case), so the first node which is compared contains the value 38. Since this is larger than the search value, the search returns to the predecessor node of 38, which in this case is the head node of the list, and drops down to the next level list (level 2) and begins again. This time the search first encounters the node containing element 9 followed by the node containing element 18. Since 18 is again larger than the search value, the search returns to the predecessor of 18, which is 9 and then drops into the next lower level list (level 1) and begins to search again, starting from the node containing element 9 using the level 1 list.

This time, the search encounters the nodes containing 10 and then 18 again. At this point, since 18 is larger than the search element, the search reverts to the predecessor of the node containing 18, which is 10 and drops into the next level list, which is level 0 and begins to search again. This time, the search encounters the nodes containing 14 followed again by 18. Since the search is now in the lowest level list, when the node containing 18 is encountered, the search successfully terminates and returns the fact that a node containing element 16 does not exist in this list.

For practice, use the regular skip list of Figure 4 to determine the nodes "visited" when searching for a node containing the element:

(a) 65
(b) 75
(c) 80
(d) 40

Answer part (a): header, 40, tail, 40, 75, 40, 60, 75 – search terminates, element not found

Answer part (b): header, 40, tail, 40, 75 – search terminates, element found

Answer part (c): header, 40, tail, 40, 75, tail, 75, 80 – search terminates, element found

Answer part (d): header, 40 – search terminates, element found

An algorithm to search a regular skip list is shown below:

```
search_regular_skip_list (element searchval)
   p = the nonnull list on the highest level i;
     while (searchval notfound and i > 0)
    {    if p.key < searchval
               p = a sublist that begins in the predecessor of p on level  --i;
         else  if p.key > searchval
                     if p is the last node on level i
                         p = a nonnull sublist that begins in p on the highest level  < i;
                          i = the number of the new level;
                     else p = p.next;
     }
```

The expected time to search for an element within a regular skip list is O($\log_2$ n). Therefore, searching a regular skip list has the same expected time bound as a binary search.

Insertion and Deletion in a Regular Skip List

While searching a regular skip list is expected to be an efficient operation, the design of skip lists can lead to very inefficient insertion and deletion techniques. To insert a new element into the skip list, all the nodes which are successors of the node just inserted will need to be restructured; the number of reference fields and the value of those references will need to be changed. This can be such a large problem that it is quite common to allow the regular skip list to degenerate into an irregular skip list rather than attempt to maintain the regularity of the original skip list as the restructuring is simply too costly. However, care must be taken when a new node is to be inserted that the relative number of nodes on the different levels is maintained, otherwise, the expected value for the search time will tend to drift away from the expected time and begin to approach linear time, particularly, if insertions occur only in the level 0 list and not at the appropriate higher levels in the structure.

Using the technique of allowing a regular skip list to become an irregular skip list upon insertion of a node does not require the restructuring of the list, only the new node requires links to its successor nodes. However, nodes must be generated in such a way that the distribution of nodes on different levels is kept adequate to maintain the logarithmic time bound for searching. In other words, we want the irregular skip list to be as close as possible to a regular skip list and not otherwise. The question is how to do this. Consider the following technique:

Assume that we have a skip list with a maximum_level = 4 (see Figure 5). Suppose that the list under consideration currently contains fewer than15 elements (this means that there is room for expansion of this list without increasing the number of levels in the list). The required number of nodes on level 1 would be 8 (remember that level 1 contains only half the number of nodes in level 0). The required number of nodes on level 2 would be 4, level 3 would contain 2 and finally, level 4 would contain a single node. AS insertions occur to the list we want to maintain this distribution of nodes between the various levels while relaxing the restriction the equally spaced skips of the original regular skip list. Every time a new node is inserted into the list, a random number *rand_num* between 1 and 15 is generated, and if *rand_num* < 9, then a node at level 1 is inserted. If *rand_num* < 13, a node on level 2 is inserted, and if *rand_num* < 15, a level 3 node is inserted, and finally if *rand_num* = 15, then a node at level 4 is generated and inserted.

rand_num	Level of node to be inserted (-1)	possible occurrences
7	3	1
5-6	2	2
1-4	1	4

Table 1 – Illustration of random numbers used to select level of new node (up to 7 nodes)

rand_num	Level of node to be inserted (-1)	possible occurrences
15	4	1
13-14	3	2
9-12	2	4
1-8	1	8

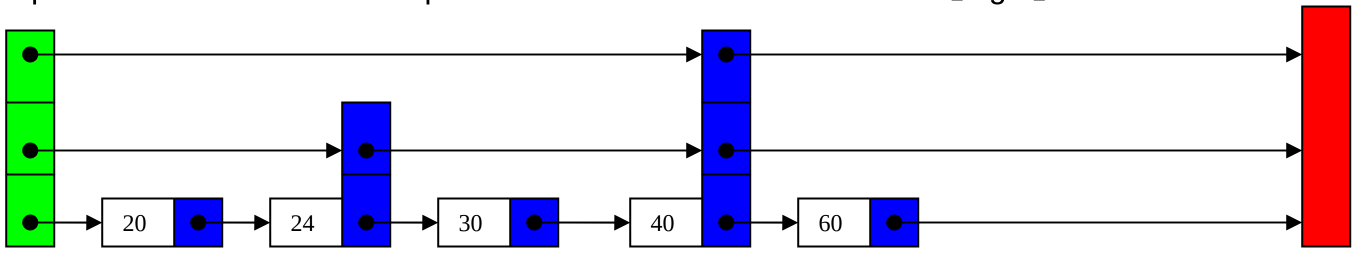
Table 2 – Illustration of random numbers used to select level of new node (up to 15 nodes)

rand_num	Level of node to be inserted (-1)	possible occurrences
31	5	1
29-30	4	2
25-28	3	4
17-24	2	8
1-16	1	16

Table 3 – Illustration of random numbers used to selected level of new node (up to 31 nodes)

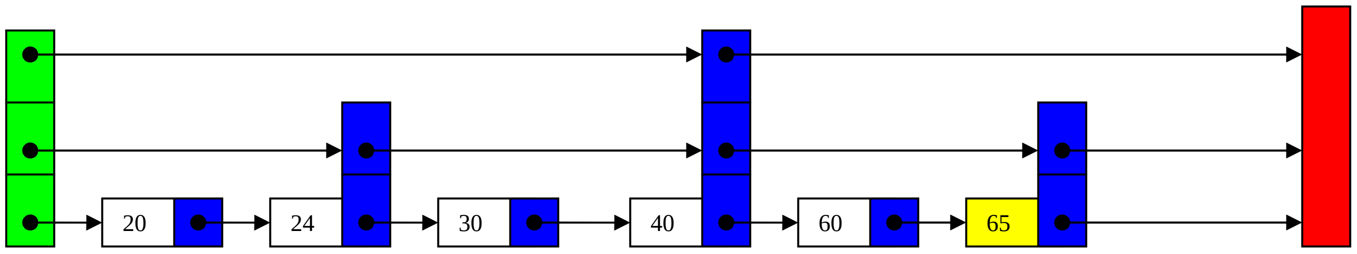
Consider the following example:

Assume the skip list currently contains 5 elements. At this point the skip list represented is a regular skip list and contains less than the maximum number of nodes for a regular skip list represented in three levels. Thus, there is room to insert into this list without increasing the depth of the structure. The depth will not need to increase until $3 < \lfloor \log n \rfloor + 1$.



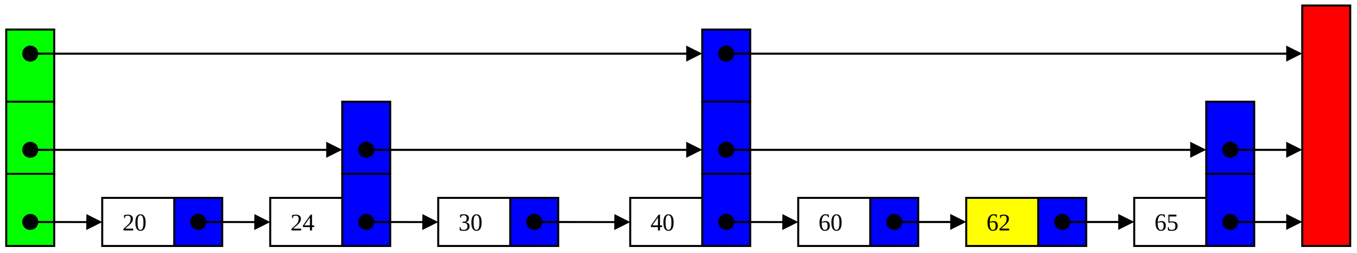
Initial skip list (regular)

Now assume a new node containing element value 65 is to be inserted in the list In order to determine the level of the node that will contain element 65, we generate a random number between 1 and 7.   Using Table 1 above, we can determine, given the random number that is generated which level node is to be constructed to hold the new element value 65. Let's suppose that the random number we generated was a 5. This implies that a level 1 node will be constructed to house the element 65. This is reflected in the following diagram:



Skip list after insertion of element 65 at level 1

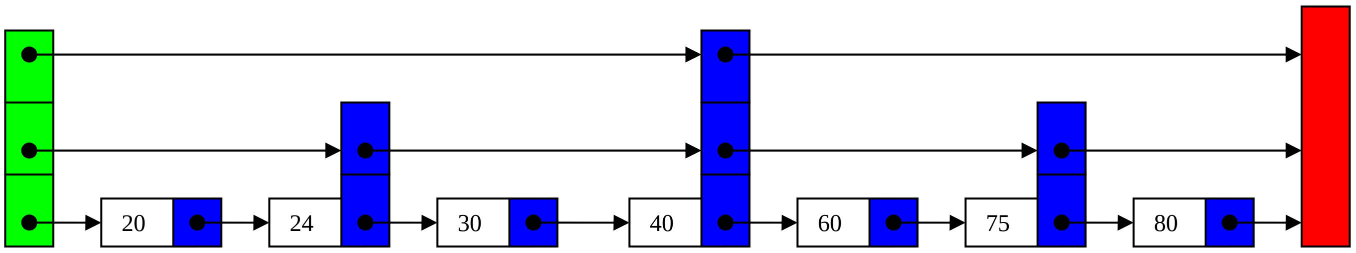
Notice that because of the value of the node which was just inserted that the skip list is still regular at this point. To illustrate when the list becomes irregular consider the following insertion of the element value 62. The way the list is currently structured, there is already a level 2 node (40), therefore another level 2 node cannot exist, similarly, there are now two level 1 nodes and this is also the maximum number of level 1 nodes that can exist in this structure. Therefore, only a level 0 node can be generated to hold the new element value 62. This is illustrated in the diagram below:



Skip list after insertion of element 62 at level 0

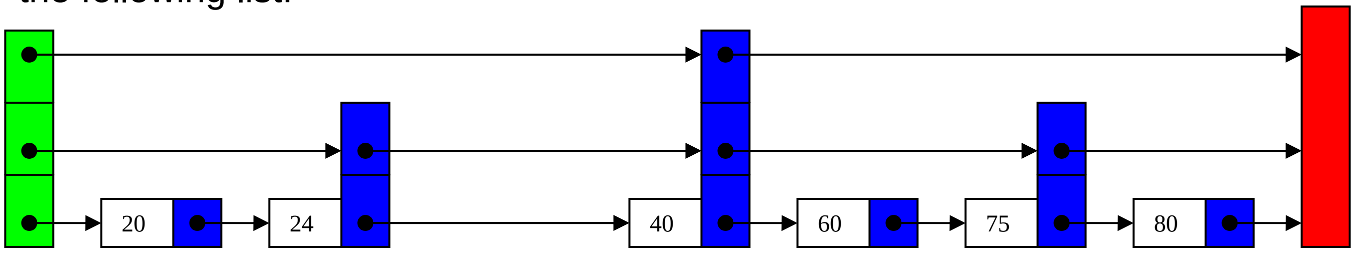
Notice that after this last insertion the skip list has become irregular.

Deletion of a node from a regular skip list also has the potential to render the resulting skip list irregular. As was the case with insertion, it is too costly to restructure the resulting skip list so that it remains regular. Therefore, as with insertion, deletion will commonly result in a regular skip list becoming irregular. To illustrate this, consider the regular skip list shown in Figure 4 (repeated below):

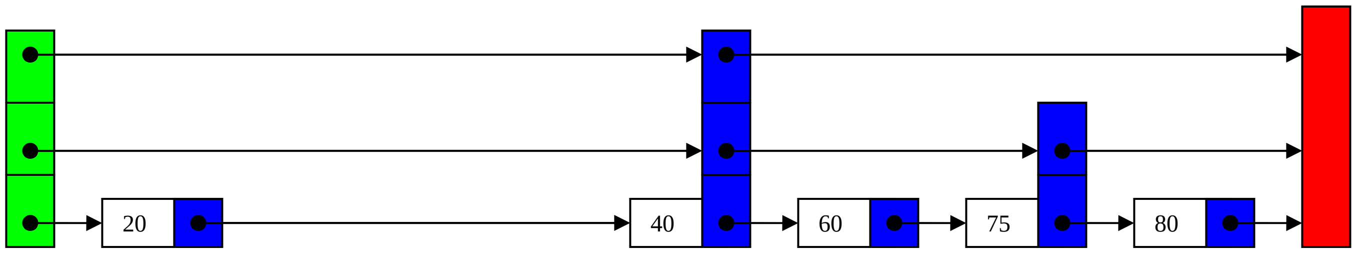


(Figure 4 repeated) – Regular skip list of 7 nodes

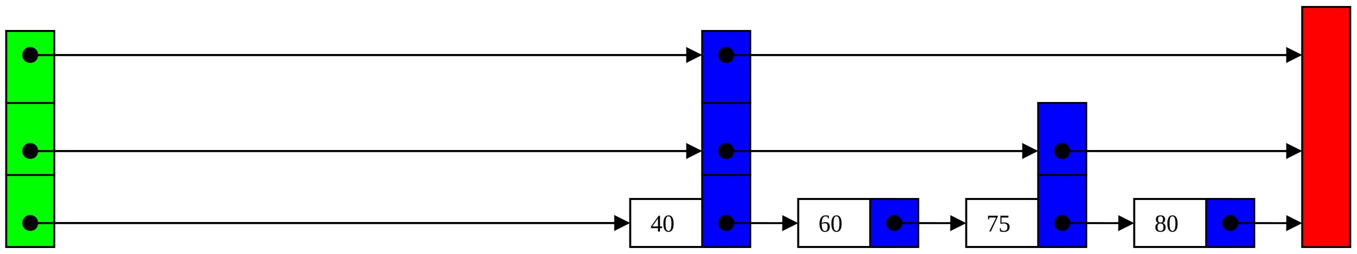
Suppose that the node containing element 30 is deleted from this list to produce the following list:



Notice that the list shown above is no longer regular since the level 1 node containing element 24 has as its successor node in level 1 a node which is only one node removed from node 24 and it should have a successor node which is two nodes away when referring to level 0 nodes (notice that it is not possible for the level 0 list to ever become irregular). Subsequent deletion of the node containing element 24 causes the list to become even more irregular at both level 1 and level 2. This is illustrated in the next diagram:

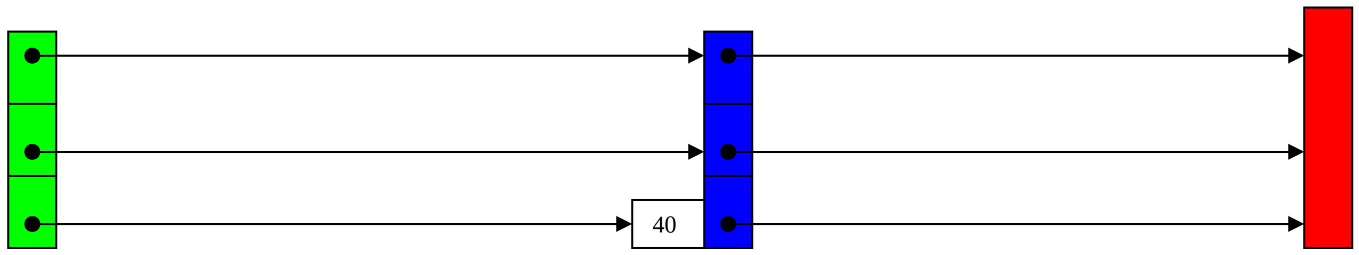


Next, let's assume that the node containing 20 is deleted to produce the following irregular skip list:



At this point, notice that all searches in this list for elements whose value is less than 40 will terminate without finding the value, all in the same time, regardless of the search value.

To carry our example to the extreme, let's assume that the nodes containing elements 60, 75, and 89 are all subsequently deleted, which will produce the following skip list.



When carried to this extreme, all searches for any value other than 40, will execute in exactly the same time, which should be obvious since the list contains only a single element. However, notice that since the technique we have employed for deletion has not restructured the list, the node containing element 40 has not "collapsed" into a level 0 node, but rather is still represented as a level 2 node. This is clearly costly both in terms of the storage space required for the remaining node, as well as for the cost of a search which still operates as before, beginning on the highest level and moving downward.

The above discussion on the efficiency of insertion and deletion in a skip list brings us to the following point. The entire reason for implementing a skip list is to reduce the cost of a search from the linear time bound of a sequential search of the list to the expected logarithmic time bound that we illustrated should exist with the skip list structure. The implication of this is that our application, whatever that might be, will predominantly search the list and make relatively few if any insertions or deletions from the list. In other words, the contents of the list are expected to remain fairly static. Given this scenario, we can conclude that the skip list structure is a reasonable structure to support such efficient searching activities. It has been shown by various researchers that the expected performance of a skip list is comparable with more sophisticated data structures such as self-adjusting or AVL trees and as such represent a viable alternative to such structures.

An Efficient Implementation of a Node in a Skip List

The simplest way to construct a node for a skip list is to have each node contain a number of reference fields which is equal to the maximum level of the skip list. For example, in Figure 4, the list has a maximum level of 3 (levels 0,1,2, and 3) and therefore each node in the structure will have four reference fields. This technique is too wasteful of space, since each of the *n* nodes on level 0 require only a single reference field, further each of the node on level 1, which contains half the number of nodes that appear on level 0, require only 2 reference fields, and so on.

In reality, each node requires only the number of reference fields which corresponds to the level of the node. To accomplish this, the *next* field of each node is not a reference to the next node, but rather to an array of reference(s) to the next node(s). The size of this array is determined by the level of the node. Thus, all level 0 nodes refer to an array of size 1, all level 1 nodes refer to an array of size 2, all level 2 nodes refer to an array of size 3, and so on.