A Skip List in C#

Leslie Sanford

1 Sep 2003 MIT

Skip Lists, their Algorithms, and a SkipList class in C#.

- **Download demo project - 36.4 Kb**
- **Download source - 7.5 Kb**

SkipList Demo

IDictionary

Key

1

Value

4124

Add

Contains

Remove

Item

Execute

Clear

1

Keys

4124

Values

IDictionaryEnumerator

Get Enumerator

Move Next

Reset

ICollection

4124

Copy To

ICollectionEnumerator

Get Enumerator

Move Next

Reset

Count: 1

Skip List Stress Test

Insertion

Forward

Reverse

Random

Number of items

124124

Insert

Search

Key

Search

Type

SkipList

SortedList

Introduction

Skip lists are an alternative to balanced trees. Invented by William Pugh, they are a data structure which uses a probabilistic algorithm to keep the structure in balance. The algorithms for skip lists are very simple and elegant. Skip lists are also very fast. This combination makes them an attractive alternative to other data structures.

In this article, I will describe what skip lists are and present their algorithms in pseudo code. The pseudo code is straight from William Pugh's paper which can be found here.

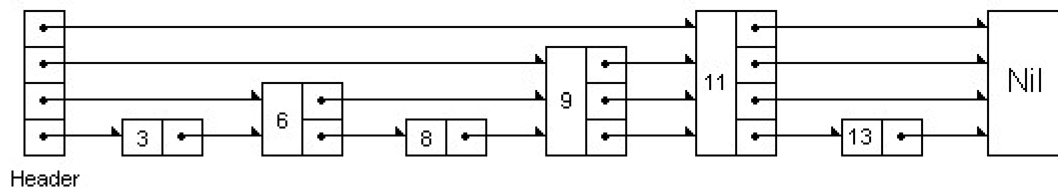
Skip List Structure

Skip lists are made up of a series of nodes connected one after the other. Each node contains a key/value pair as well as one or more references, or pointers, to nodes further along in the list. The number of references each node contains is determined randomly. This gives skip lists their probabilistic nature, and the number of references a node contains is called its node level.

Each node will have at least one node reference, and the first reference will always point to the next node in the list. In this way, skip lists are very much like linked lists. However, any additional node references can skip one or more intermediate nodes and point to nodes later in the list. This is where skip lists get their name.

Two nodes that are always present in a skip list are the header node and the NIL node. The header node marks the beginning of the list and the NIL node marks the end of the list. The NIL node is unique in that when a skip list is created, it is given a key greater than any legal key that will be inserted into the skip list. This is important to how skip lists algorithms work.

Skip lists have three more important properties: maximum level, current overall level, and probability. Maximum level is the highest level a node in a skip list may have. In other words, maximum level is the maximum number of references a skip list node may have to other nodes. The current overall level is the value of the node level with the highest level in the skip list. Probability is a value used in the algorithm for randomly determining the level for each node.



A skip list with several nodes.

Determining Node Level

The level for each node is determined using the following algorithm:

```
randomLevel()
    lvl := 1
    --random() that returns a random value in [0...1)
    while random() < p and lvl < MaxLevel do
        lvl := lvl + 1
    return lvl
```

Where `p` is the skip list's probability value and `MaxLevel` is the maximum level allowed for any node in the skip list.

The node level is initialized to a value of 1. Each time the `while` loop executes, the level value is incremented by 1. If `p` is set to a value of 0.5, then there is a 50% chance that the `while` loop will execute once, a 25% chance it will execute twice, and a 12.5% chance it will execute three times. This creates a structure in which there will be more nodes with a lower level than higher ones.

There is room for optimization here. Suppose the overall level of a skip list is 4 and a value of 7 is returned by the `randomLevel` algorithm for a new node. Since 7 is larger than 4, the new skip list level will be 7. However, 7 is 3 levels greater than 4. What this means is that when searching the skip list, there will be 2 additional levels that will have to be traversed unnecessarily (this will become more clear when we examine the search algorithm). What is needed is a way to limit the results of the `randomLevel` algorithm so that it never produces a level greater than one more than the present overall skip list level. Pugh makes a suggestion to "fix the dice." Here is the altered `randomLevel` algorithm:

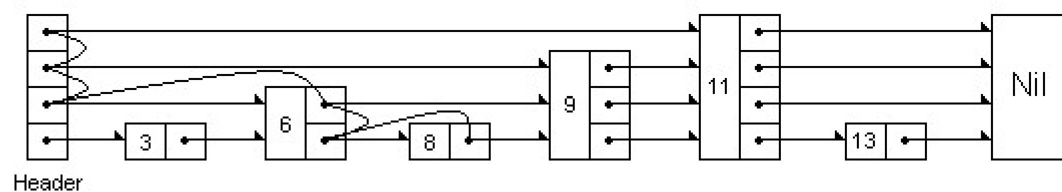
```
randomLevel(list)
    lvl := 1
    --random() that returns a random value in [0...1)
    while random() < p and lvl < MaxLevel and lvl <= list->level do
        lvl := lvl + 1
    return lvl
```

Searching

Searching for a key within a skip list begins with starting at header at the overall list level and moving forward in the list comparing node keys to the search key. If the node key is less than the search key, the search continues moving forward at the same level. If on the other hand, the node key is equal to or greater than the search key, the search drops down one level and continues forward. This process continues until the search key has been found if it is present in the skip list. If it is not, the search will either continue to the end of the list or until the first key with a value greater than the search key is found.

```
Search(list, searchKey)
    x := list->header
    --loop invariant: x->key < searchKey
    for i := list->level downto 1 do
        while x->forward[i]->key < searchKey do
            x := x->forward[i]
    --x->key < searchKey <= x->forward[1]->key
    x := x->forward[1]
    if x->key = search then return x->value
        else return failure
```

Where `forward` is the array of node references each node has to nodes further in the list.



Searching for the key with a value of 8.

Inserting

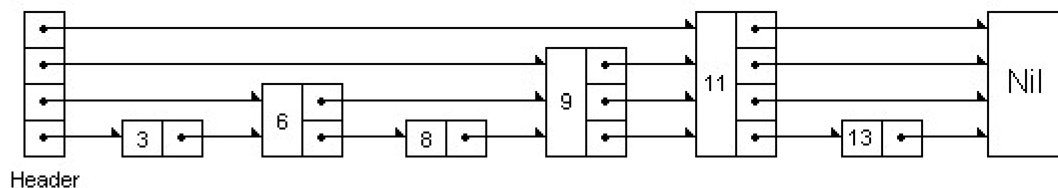
Insertion begins with a search for the place in the skip list to insert the new key/value pair. The search algorithm is used with one change: an array of nodes is added to keep track of the places in the skip list where the search dropped down one level. This is done because the pointers in those nodes will need to be rearranged when the new node is inserted into the skip list.

```
Insert(list, searchKey, newValue)
    local update[1..MaxLevel]
    x := list->header
    --loop invariant: x->key < searchKey
    for i := list->level downto 1 do
        while x->forward[i]->key < searchKey do
            x := x->forward[i]
        --x->key < searchKey <= x->forward[1]->key
        update[i] := x
    x := x->forward[1]
    if x->key = search then x->value := newValue
    else
        lvl := randomLevel()
        if lvl > list->level then
            for i := list->level + 1 to lvl do
                update[i] := list->header
            list->level := lvl
         x := makeNode(lvl, searchKey, newValue)
         for i := 1 to lvl do
             x->forward[i] := update[i]->forward[i]
             update[i]->forward[i] := x
```

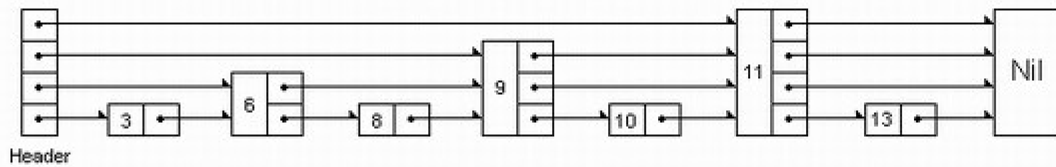
The first part of this algorithm should look familiar. It is the same as the search algorithm except that it uses the `update` array to hold references to the nodes where the search drops down one level. After the search has ended, a check is made to see if the key in the node where the search stopped matches that of the search key. If so, the value for that key is replaced with the new value. If on the other hand, the keys do not match, a new node is created and inserted into the skip list.

To insert a new node, a node level is retrieved from the `randomLevel` algorithm. If this value is greater than the current overall level of the skip list, the references in the `update` array from the overall skip list level up to the new level are assigned to point to the header. This is done because if the new node has a greater level than the current overall level of the skip list, the forward references in the header will need to point to this new node instead of the NIL node. This reassignment takes place during the next step of the algorithm.

Next, the new node is actually created and it is spliced into the skip list in the next `for` loop. What this loop does is work from the bottom of the skip list up to the new node's level reassigning the forward references along the way. It's much the same as rearranging the references in a linked list when a new node is inserted except that with a skip list there are an array of references that have to be reassigned rather than just one or two.



The skip list before inserting key 10.



The skip list after inserting key 10.

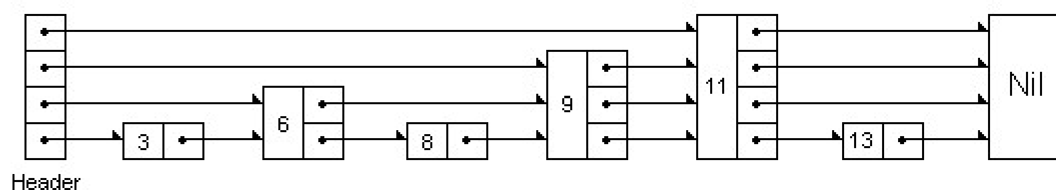
Deletion

Deletion uses the same search algorithm as insertion; it keeps track of each place in the list in which the search dropped down one level. If the key to be deleted is found, the node containing the key is removed.

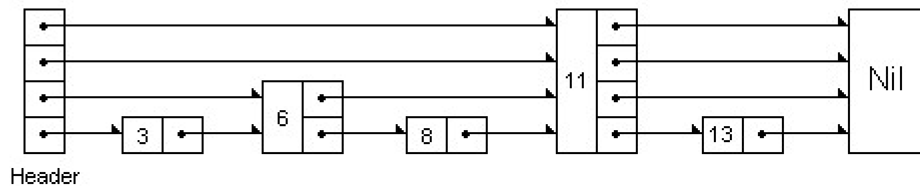
```
Delete(list, searchKey)
    local update[1..MaxLevel]
    x := list->header
    --loop invariant: x->key < searchKey
    for i := list->level downto 1 do
        while x->forward[i]->key < searchKey do
            x := x->forward[i]
        --x->key < searchKey <= x->forward[1]->key
        update[i] := x
    x := x->forward[1]
    if x->key = searchKey then
        for i := 1 to list->level do
            if update[i]->forward[i] != x then break
            update[i]->forward[i] := x->forward[i]
        free(x)
        while list->level > 1 and
              list->header->forward[list->level] = NIL do
            list->level := list->level - 1
```

After the key is found, the `for` loop begins from the bottom of the skip list to the top reassigning the nodes with references to the soon to be deleted node to the nodes that come after it. Again, very much like a linked list except that here there are an array of links to nodes further along in the list that must be managed.

Once this has been done, the node is deleted. The only thing left to do is to update the overall current list level if necessary. This is done in the final `while` loop.



The skip list before removing key 9.



The skip list after removing key 9.

C# Implementation

The C# implementation of William Pugh's skip list implements the `IDictionary` interface. Any where an `IDictionary` can be used, this version of the skip list can be used.

There are a couple of places where I deviated from the algorithms described above. First, as you have probably noticed, the algorithms used base 1 arrays rather than base 0. This had to be changed for the C# implementation, which wasn't hard to do at all. Second, and more importantly, I removed the requirement that a skip list must be initialized with the highest possible key. I did this by removing the NIL node and making the header node the beginning and the end of the skip list. This complicated the algorithms slightly, and as a result, made them slower. However, I think it is a trade off worth making because, in my opinion, it is an undue burden on clients to have to know beforehand what is the highest possible key that will be used.

Demonstration Apps

I've created two demonstration applications. One simply lets you perform `IDictionary` and `ICollection` operations on the skip list. This application lets you perform the operations raw. By that I mean that it does not stop you from doing something that will cause an exception. This is intentional. I wanted to show that the `SkipList` class performs to the `IDictionary` and `ICollection` specifications including throwing exceptions when certain events occur, such as trying to use an existing enumerator after the `SkipList` has been modified. The application, of course, catches these exceptions and displays the error messages.

The second application stress tests the `SkipList` class by allowing you to insert up to a million items in forward, reverse, or random order. Timing results are returned. It also lets you search for any item. For comparison, I've included the .Net Framework's `SortedList` class. You can perform the same operation on either class and compare the results. The reason I chose the `SortedList` class is that it seems the closest in purpose to the `SkipList` class.

A note here: the `SkipList` class beats the socks off the `SortedList` class in most situations. I've included code so that the `SortedList` object's `Capacity` property is modified to match the number of items for insertion before actually inserting the objects. My thinking is that this would make things more fair and probably even give the advantage to the `SortedList` class, but that wasn't the case.

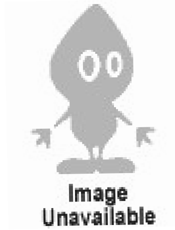
The demo zip file contains a Visual C# solution with three projects. One project is the SkipList itself, another project is the SkipList demo, and the final project is the `SkipList` stress test. In order to reduce the file size, I deleted the *bin* and *obj* folders from each of the projects. What this means is that you will need to build the solution from scratch and then select either the SkipListDemo project or the SkipListStressTest project to be the startup project. You can do this by right-clicking on the project in the Solution Explorer and choosing Set as StartUp Project.

I hope you find this class useful. All comments are welcomed. Thanks!

License

This article, along with any associated source code and files, is licensed under The MIT License

About the Author



Leslie Sanford

Aside from dabbling in BASIC on his old Atari 1040ST years ago, Leslie's programming experience didn't really begin until he discovered the Internet in the late 90s. There he found a treasure trove of information about two of his favorite interests: MIDI and sound synthesis.

After spending a good deal of time calculating formulas he found on the Internet for creating new sounds by hand, he decided that an easier way would be to program the computer to do the work for him. This led him to learn C. He discovered that beyond using programming as a tool for synthesizing sound, he loved programming in and of itself.

Eventually he taught himself C++ and C#, and along the way he immersed himself in the ideas of object oriented programming. Like many of us, he gotten bitten by the design patterns bug and a copy of GOF is never far from his hands.

Now his primary interest is in creating a complete MIDI toolkit using the C# language. He hopes to create something that will become an indispensable tool for those wanting to write MIDI applications for the .NET framework.

Besides programming, his other interests are photography and playing his Les Paul guitars.

Comments and Discussions

Article Copyright 2003 by Leslie Sanford
Everything else Copyright © CodeProject, 1999-2019

Web01 2.8.190627.1