Skip Lists in C++

Skip lists are an efficient alternative to balanced trees, and rather easier to implement correctly.

Bill Whitney

Bill Whitney is an Application Consultant currently working for GTE Data Services in Tampa, Florida. He specializes in Object-Oriented design, Client/Server application architecture, and Telecommunications Management Networks.

Introduction

If you're like me, you're always looking for an alternative data structure that not only performs admirably, but is easy to implement and understand as well. The skip list, described by William Pugh in "Skip Lists: A Probabilistic Alternative to Balanced Trees" (*Communications of the ACM*, June 1990)[1] fits this description.

Skip lists are ordered key/object-based containers offering excellent performance with minimal processing overhead. Although skip lists resemble linked lists, they actually have more in common with balanced trees when it comes to performance. The most important difference is that skip lists require none of the periodic reorganization required with balanced trees — the searching, insertion, and removal algorithms are simpler to code, understand, and extend.

How Skip Lists Work

Figure 1 shows a skip list containing a product inventory. All data is contained in a node and identified by a key. Probably the most outstanding feature of the nodes is the variations in their heights. By filling a list with nodes of varying heights, the search algorithm can "see" past subsequent nodes when searching for a particular value. For example, given a node list in which 50% of the nodes are twice as high as the rest, search time is cut in half. The algorithm needs to look at only half of the values.

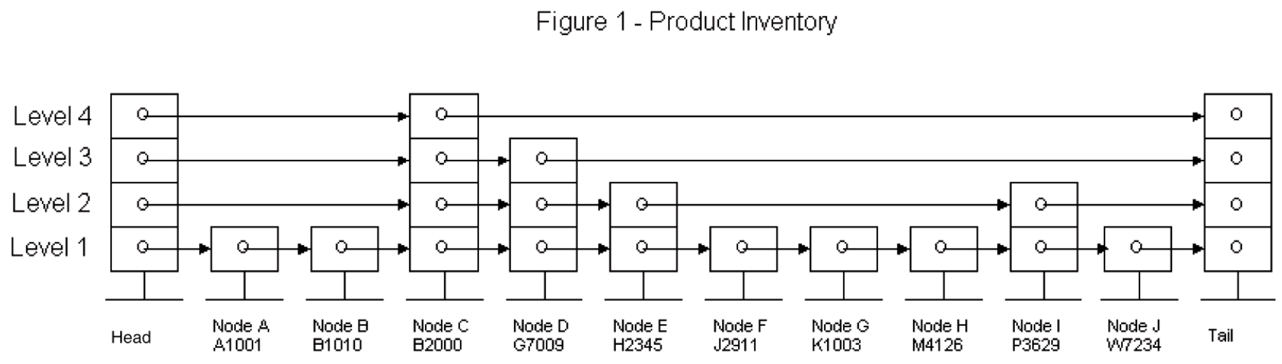

Figure 1: A sample skip list

As the height of the nodes increase, the visibility between nodes has the potential to be much greater, resulting in even shorter searches. In the ideal skip list, nodes of designated heights would be distributed throughout the list at pre-determined intervals. This would guarantee the

1 ftp://ftp.cs.umd.edu/pub/skipLists/skiplists.pdf

maximum benefit from the nodes' look-ahead capability. To sustain performance, however, skip lists employ a probabilistic approach to assigning node height. This approach also removes the requirement for constant reorganization to maintain pre-determined intervals. I'll show the algorithm for probabilistic node height a little later. Consult the reference at the end of this article for a detailed mathematical analysis.

Looking back at Figure 1, the head and tail are actual nodes in the skip list, but they serve only as markers for the beginning and end; they contain no useful data. The head points to the first visible node at each height (only A and C in Figure 1). Any node for which there exists no subsequent node at its height (C, D, I and J) points to the "tail" node. Suppose a search is made for product code H2345 in the product inventory. Each search starts at the head of the skip list, comparing the sought-after key with the first node pointed to by the root node at the list's current height.

Before I go any further, I should explain the concept of *maximum height*. The maximum height of a skip list defines the number of vertical pointers the skip list can maintain. I use the term *current height* to refer to the tallest node that has been inserted into the skip list so far. Because node height is probabilistic, current height reaches maximum height as a skip list matures. This will become clearer when I present the code.

Referring to Figure 1, the product sought (H2345) is located in node E. This sample skip list has a current height of 4, a maximum height of 4, and contains ten nodes labeled A through J. All searches start at the root node, from the level indicated by current height (4). Root node level 4 points to node C at level 4. Comparing the sought key (H2345) with node C's key (B2000), shows that the search is still below the value needed. Next, the search algorithm follows node C's level 4 pointer, which brings it to the list's tail. That's obviously too far, so the search drops down to node C's level 3 pointer and looks ahead to node D. Node D's key value is G7009, which is too low. Node D's level 3 pointer leads to the list's tail again so the search drops down to level 2 and looks ahead. The next node is E, and checking its key produces a match. The search algorithm had to traverse three nodes to locate the key value H2345 in node E. If it were looking for the value P3629 in node I (9th node in the list), the algorithm would have had to look at only four nodes.

The C++ Implementation

I've implemented the skip list in two template classes, **SkipList** and **SkipNode**. I also use a helper class called **RandomHeight**, which is based on Pugh's algorithm for generating probabilistically random node heights. **RandomHeight** is called upon by the **SkipList insert** method when new nodes are generated. It appears in Listing 1 (**RandomHeight.h**) and Listing 2 (**RandomHeight.cpp**).

Listing One

```
#ifndef SKIP_LIST_RAND
#define SKIP_LIST_RAND
class RandomHeight
{
  public:
    RandomHeight(int maxLvl, float prob);
    ~RandomHeight() {}
    int newLevel(void);

  private:
    int maxLevel;
    float probability;
};
#endif //SKIP_LIST_RAND
```

```
/* End of File */
```

Listing Two

```
#include <stdlib.h>
#include "RandomHeight.h"

RandomHeight::RandomHeight
    (int maxLvl, float prob)
{
  randomize();
  maxLevel = maxLvl;
  probability = prob;
}

int RandomHeight::newLevel(void)
{
int tmpLvl = 1;
  // Develop a random number between 1 and
  // maxLvl (node height).
  while ((random(2) < probability) &&
         (tmpLvl < maxLevel))
    tmpLvl++;

  return tmpLvl;
}

//End of File
```

The **RandomHeight** constructor lays the ground rules for the scope of the random numbers to be generated. These rules consist of the maximum node height (**level**), and the percentage of nodes that should appear at level 1. When new nodes are created, a random node height is obtained from **RandomHeight**'s **newLevel** method. Given a height of 4 and a percentage of 0.5, for example, 50% of the generated values would produce level 1 nodes, 25% of the nodes would be level 2, 12.5% of the nodes would be level 3 nodes, and so on. The **random(2)** call produces a number between 0 and 1 using my compiler.

The SkipNode Class

Skip nodes are implemented in **SkipNode.h** (Listing 3). Each node represents a single element in a skip list, and contains pointers to the node's key, its object, and subsequent skip nodes at or below its height.

Listing Three

```
#include <stdlib.h>
#include "RandomHeight.h"

RandomHeight::RandomHeight
    (int maxLvl, float prob)
{
  randomize();
  maxLevel = maxLvl;
  probability = prob;
}

int RandomHeight::newLevel(void)
{
```

```
int tmpLvl = 1;
  // Develop a random number between 1 and
  // maxLvl (node height).
  while ((random(2) < probability) &&
         (tmpLvl < maxLevel))
    tmpLvl++;

  return tmpLvl;
}

//End of File
```

A skip node can be instantiated with one of two constructors. The first takes pointers to a key and an object, and an integer indicating the height of the new node. The node's height is kept in a private attribute called **nodeHeight**, and is reflected in the number of allocated forward node pointers kept in **fwdNodes**. To make the code easier to read, I've made the **fwdNodes** attribute public. This also avoids the overhead of a method call when checking whether any of the node's forward pointers actually point to anything.

The second **SkipNode** constructor takes an integer identifying the node height. When the node is constructed with only a height, the forward pointers are allocated, and the key and object pointers are set to **NULL**. This constructor is used only for the list's head node. Both constructors set all allocated forward pointers to **NULL**. These forward pointers will be used heavily by the **SkipList** class when inserting, searching for, and deleting **SkipNode** objects.

SkipNode also contains methods you can call to retrieve a key (**getKey**), an object (**getObj**), and the node's height **(getHgt**). Here's a short example in which I instantiate a product node with a key, object, and height, followed by a query for the key and height:

```
String* key =
    new String("Bill Whitney");
productData* obj  = new productData;
SkipNode* newNode =
    new SkipNode<Key,Obj>(key, obj,
            rndGen->newLevel());

    String* x = newNode->getKey();
    int hgt = newNode->getHgt();
```

SkipNode's destructor deletes memory allocated for the key, object, and forward pointers.

Creating a SkipList Object

While **SkipNode** objects represent the individual nodes in a skip list, the **SkipList** class manages their insertion, deletion, and retrieval (see Listing 4, **SkipList.h**). **SkipList** has one constructor, which takes three values: the probability and maximum node height (which are used to instantiate **RandomHeight**), and a parameter representing the maximum key value. The maximum key value is stored in the tail of the skip list, preventing the search algorithm from attempting to look beyond that point. This sentinel operation could be implemented differently; for example, by comparing a **SkipNode**'s forward pointer to the address of the tail node.

Listing Four

```
#ifndef SKIP_LIST
#define SKIP_LIST
#include <iostream.h>
#include <fstream.h>
#include "SkipNode.h"
```

```
#include "RandomHeight.h"

template <class Key, class Obj>
  class SkipList
  {
  public:
    SkipList(float,int,Key*);
    ~SkipList();

    bool insert(Key*, Obj*);
    bool remove(Key*);
    Obj* retrieve(Key*);
    void dump(ofstream&);

  private:
    SkipNode<Key,Obj>* head;
    SkipNode<Key,Obj>* tail;
    float probability;
    int maxHeight;
    int curHeight;
    RandomHeight* randGen;
  };

template <class Key, class Obj>
  SkipList<Key,Obj>::SkipList(float p, int m, Key* k)
  {
    curHeight = 1;
    maxHeight = m;
    probability = p;
    randGen = new RandomHeight(m,p);

    // Create head and tail and attach them
    head = new SkipNode<Key,Obj>(maxHeight);
    tail = new SkipNode<Key,Obj>(k, (Obj*) NULL, maxHeight);
    for ( int x = 1; x <= maxHeight; x++ )
        head->fwdNodes[x] = tail;
  }

template <class Key, class Obj>
  SkipList<Key,Obj>::~SkipList()
  {
    // Walk 0 level nodes and delete all
    SkipNode<Key,Obj>* tmp;
    SkipNode<Key,Obj>* nxt;
    tmp = head;
    while ( tmp )
    {
      nxt = tmp->fwdNodes[1];
      delete tmp;
      tmp = nxt;
    }
  }

template <class Key, class Obj>
  bool SkipList<Key,Obj>::insert(Key* k, Obj* o)
  {
    int lvl = 0, h = 0;
    SkipNode<Key,Obj>** updateVec =
      new SkipNode<Key,Obj>* [maxHeight+1];
    SkipNode<Key,Obj>* tmp = head;
    Key* cmpKey;

    // Figure out where new node goes
    for ( h = curHeight; h >= 1; h-- )
    {
      cmpKey = tmp->fwdNodes[h]->getKey();
      while ( *cmpKey < *k )
      {
        tmp = tmp->fwdNodes[h];
        cmpKey = tmp->fwdNodes[h]->getKey();
      }
      updateVec[h] = tmp;
    }
    tmp = tmp->fwdNodes[1];
    cmpKey = tmp->getKey();

    // If dup, return false
    if ( *cmpKey == *k )
    {
      return false;
    }
    else
    {
      // Perform an insert
      lvl = randGen->newLevel();
      if ( lvl > curHeight )
      {
        for ( int i = curHeight + 1; i <= lvl; i++ )
          updateVec[i] = head;
        curHeight = lvl;
      }
      // Insert new element
      tmp = new SkipNode<Key,Obj>(k, o, lvl);
      for ( int i = 1; i <= lvl; i++ )
      {
        tmp->fwdNodes[i] = updateVec[i]->fwdNodes[i];
        updateVec[i]->fwdNodes[i] = tmp;
      }
    }
    return true;
  }


template <class Key, class Obj>
  bool SkipList<Key,Obj>::remove(Key* k)
  {
    SkipNode<Key,Obj>** updateVec =
      new SkipNode<Key,Obj>* [maxHeight+1];
    SkipNode<Key,Obj>* tmp = head;
    Key* cmpKey;

     // Find the node we need to delete
    for ( int h = curHeight; h > 0; h-- )
    {
      cmpKey = tmp->fwdNodes[h]->getKey();
      while ( *cmpKey < *k )
      {
        tmp = tmp->fwdNodes[h];
        cmpKey = tmp->fwdNodes[h]->getKey();
      }
      updateVec[h] = tmp;
    }
    tmp = tmp->fwdNodes[1];
    cmpKey = tmp->getKey();

    if ( *cmpKey == *k )
    {
      for ( int i = 1; i <= curHeight; i++ )
      {
        if ( updateVec[i]->fwdNodes[i] != tmp )
          break;
        updateVec[i]->fwdNodes[i] = tmp->fwdNodes[i];
      }
      delete tmp;
      while ( ( curHeight > 1 ) &&
            ( ( head->fwdNodes[curHeight]->getKey()
                == tail->getKey() ) ) )
        curHeight--;
      return true;
    }
    else
    {
      return false;
    }
  }

template <class Key, class Obj>
  Obj* SkipList<Key,Obj>::retrieve(Key* k)
  {
    int h = 0;
    SkipNode<Key,Obj>** updateVec =
      new SkipNode<Key,Obj>* [maxHeight+1];
    SkipNode<Key,Obj>* tmp = head;
    Key* cmpKey;

    // Find the key and return the node
    for ( h = curHeight; h >= 1; h-- )
    {
      cmpKey = tmp->fwdNodes[h]->getKey();
      while ( *cmpKey < *k )
      {
        tmp = tmp->fwdNodes[h];
        cmpKey = tmp->fwdNodes[h]->getKey();
      }
      updateVec[h] = tmp;
    }
    tmp = tmp->fwdNodes[1];
    cmpKey = tmp->getKey();
    if ( *cmpKey == *k )
      return tmp->getObj();
    else
      return (SkipNode<Key,Obj>*) NULL;
  }

template <class Key, class Obj>
  void SkipList<Key,Obj>::dump(ofstream& of)
  {
    SkipNode<Key,Obj>* tmp;

    tmp = head;
    while ( tmp != tail )
    {
      if ( tmp == head )
        of << "There's the head node!" << endl << flush;
      else
        // Your key class must support "<<"
        of << "Next node holds key: " << tmp->getKey() << endl
           << flush;
      tmp = tmp->fwdNodes[1];
    }
    of << "There's the tail node!" << endl << flush;
  }

#endif //SKIP_LIST

/* End of File */
```

When a **SkipList** is instantiated, it creates an instance of **RandomHeight**, and then creates the skip list's head and tail nodes. The constructor sets the pointers for **key** and **object** in the head node to **NULL**, and sets all forward pointers in the head node to the address of the tail node. The tail node's forward pointers and object pointer are set to **NULL**. The key pointer will point to the maximum key value provided in the **SkipList** constructor. After construction the skip list appears as shown in Figure 2, with a maximum height of 4 and a current height of 0.

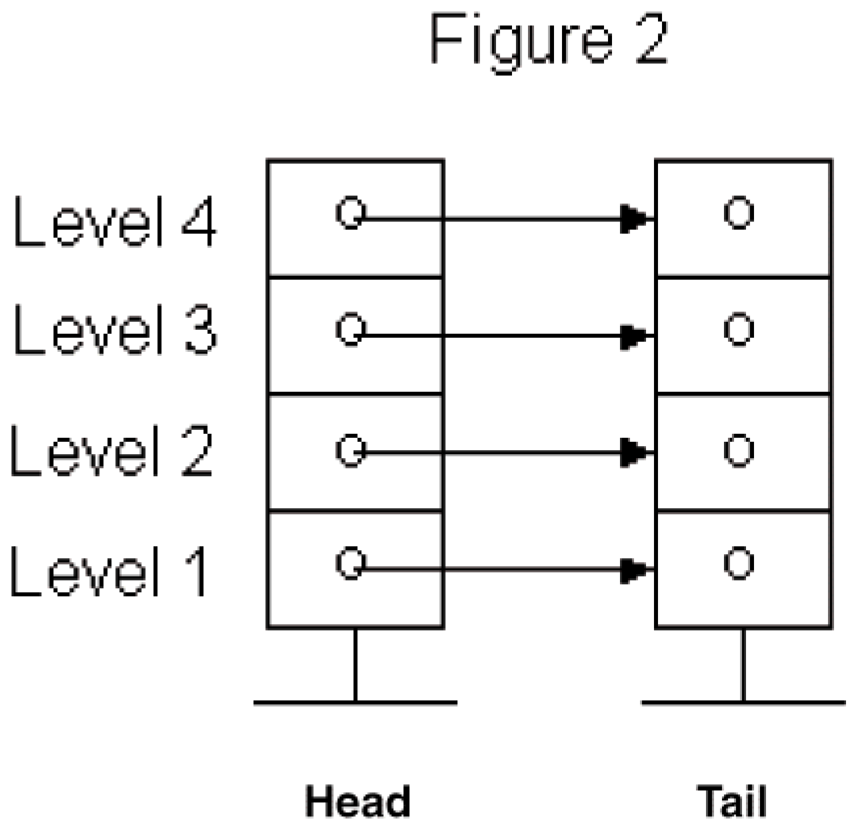

Figure 2: A newly constructed skip list

The following snippet shows how to instantiate the **SkipList** object in your code. The maximum key value shown here is based on the example in Figure 1, and reflects the highest possible product identifier. In reality, this value will depend on your application.

```
String* maxKey = new String("Z9999");
```

```
SkipList<String, productData>
    *aSkipList =
        new SkipList<String,
                productData>
            ((float).5, 4, maxKey);
```

Searching the Skip List

All operations on a skip list rely on the search algorithm, so I discuss this first.

Earlier I provided a high-level explanation of this process, showing how a search for product H2345 would take place. The following explains how the code itself works. Here's the search algorithm broken out of the **insert** method:

```
SkipNode<Key,Obj>* tmp = head;
Key* cmpKey;

    // Figure out where new node goes
    for ( h = curHeight; h >= 1; h- )
    {
        cmpKey =
   tmp->fwdNodes[h]->getKey();
        while ( *cmpKey < *k )
        {
          tmp = tmp->fwdNodes[h];
          cmpKey =
  tmp->fwdNodes[h]->getKey();
        }
        updateVec[h] = tmp;
    }
    tmp = tmp->fwdNodes[1];
```

The search starts at the topmost forward pointer in the head node, which is indicated by **curHeight,** and controlled by the outermost **for** loop. The head has no key, so the code always retrieves the value of the key in the next node at this level. It does so via the **fwdNodes** attribute. If the key (**k**) in the node being compared is less than the key (**cmpKey**), then the next node becomes the current node, and the comparison key pointer is reset to the value of the key in the next forward node (handled in the **while** loop). This process continues until the code finds a key that is less than the search key. (This explains why the tail was set to a sentinel value of Z9999). At this point, the search drops down a level in the current node, and the **cmpKey** pointer is set to the key value of the next forward node.

Eventually, the search ends up at level 1, in the node just prior to the node less than or equal to the key being sought (because the search code is always looking at the key value for the next node). This is the reason for the final forward shift in **tmp** following the **for** loop. It is at this point where all add, delete, and insert operations take place.

Another important point about the search routine is that following the **while** loop (or more precisely, just prior to dropping down a level in the current node), the search routine sets **updateVec[h]** to the current node. This vector of **SkipNode** pointers is used as a placeholder for pointers that must be moved when splicing the skip list back together after **insert** or **delete** operations. The **updateVec** pointers always point to the most previous node prior to the insertion/deletion point at each level.

To retrieve an object, **SkipList** contains a method called **retrieve**, which takes the key as an argument, and returns the object (or **NULL** if the object doesn't exist). Here's an example:

```
String *srch = new String("H2345");
productData* prod =
    sList->retrieve(srch);
```

Inserting Objects into the SkipList

Adding a node to the skip list is as simple as calling the **insert** method with the new key and object. In my **SkipList** class, new memory for the key will be allocated in the **SkipNode** object, but **productData**'s memory will not be copied (so you should not delete that memory after calling **insert**). Here's a sample insertion:

```
String* aKey = new String("Y8792");
productData* prod  = new productData;

aList->insert(aKey, prod);
```

The first thing **SkipList**'s **insert** method does is use the search algorithm to select a location for a new node. After it locates the perfect spot, the **insert** algorithm checks if the height of the new node is greater than any previously created node. If so, then **insert** resets **SkipList**'s **curHeight** to match the new level, and sets the **updateVec** pointers for the previously unused levels to the head node. The search algorithm will use this information later to enable finding the new node from the head node. (Remember, the head points to the tail at all unused levels).

Once the new node is instantiated, it gets spliced into the list by re-pointing the previous node pointers to the new node. All pointers up to and including the new node's height are retargeted. The new node then acquires from **updateVec** the forward pointers from its previous nodes for the levels it occupies.

A successful insert will return **true**. If an insertion fails (most likely due to a duplicate key) then **insert** will return **false**. You can easily change the behavior of **insert** to accept duplicate keys.

Removing Skip List Items

You can remove any item from the skip list by calling the **remove** method. It takes an argument equal to the key you want to delete. As with **insert**, the first thing to take place is a search for the existing key. The search routine also builds the **updateVec** array containing pointers to the most previous nodes at each preceding level. When the node you want to remove is located, its forward pointers are copied into the forward pointers of the previous nodes. This effectively slices the node out of the list. The **remove** method then performs a **delete** on the **SkipNode** object.

After the object is deleted, you should check whether the node was the tallest in the list. If it was, the head will once again have forward pointers to the tail. The code then uses this information to adjust **curHeight** to the correct value. If the key value you passed to **remove** doesn't exist, the routine returns **false**. Otherwise **remove** returns **true**.

Dumping the List

You can still perform linear operations on the list by traversing the nodes at level 1 (as all nodes are linked at that level). I demonstrate this feature in the **dump** method used to print out the entire skip list. This method accepts an **ofstream** pointer and prints the value of the key for each node.

Skip List Optimization

Pugh suggests that for any given probability (*p*), you can compute the optimal node height if you know the approximate number of nodes (*n*) you'll need. The formula is: log1/p n. If we arbitrarily chose 50 nodes and a probability of 0.5, the formula would yield an optimal height of 6. (5.64 is the actual result.)

Conclusion

With any data structure, the application should dictate the most appropriate choice. Skip lists are easy to code and understand, and therefore easy to implement and extend. As an added bonus, they offer average O(log n) performance, and the convenience of both binary and linear operations.

Reference

William Pugh's web site can be found at: www.cs.umd.edu/~pugh and contains a link to "Skip Lists: A Probabilistic Alternative to Balanced Trees," *Communications of the ACM*, June 1990.

Listing 3: Template class SkipNode

```
#ifndef SKIP_LIST_NODE
#define SKIP_LIST_NODE

struct Product
{
  float cost;
  int   quantity;
  int   location;
};

typedef Product productData;

template <class Key, class Obj>
  class SkipList;

template <class Key, class Obj>
  class SkipNode
  {
  public:
    SkipNode(Key*, Obj*, int);
    SkipNode(int);
    ~SkipNode();

    Key* getKey(void);
    Obj* getObj(void);
    int   getHgt(void);
    SkipNode** fwdNodes;

  private:
    int nodeHeight;
    Key* key;
    Obj* obj;
  };

template <class Key, class Obj>
  SkipNode<Key,Obj>::~SkipNode()
  {
    delete key;
    delete obj;
    delete [] fwdNodes;
  }

template <class Key, class Obj>
  SkipNode<Key,Obj>::SkipNode(Key* k,
    Obj* o, int h)
  {
    nodeHeight = h;
    key = k;
    obj = o;
    fwdNodes =
  new SkipNode<Key,Obj>* [h+1];
    for ( int x = 1; x <= nodeHeight; x++ )
      fwdNodes[x] =
  (SkipNode<Key,Obj>*) NULL;
  }

template <class Key, class Obj>
  SkipNode<Key,Obj>::SkipNode(int h)
  {
    nodeHeight = h;
    key = (Key*) NULL;
    obj = (Obj*) NULL;
    fwdNodes =
  new SkipNode<Key,Obj>* [h+1];
    for ( int x = 1; x <= nodeHeight; x++ )
      fwdNodes[x] =
  (SkipNode<Key,Obj>*) NULL;
  }


template <class Key, class Obj>
  Key* SkipNode<Key,Obj>::getKey(void)
  {
    return key;
  }

template <class Key, class Obj>
  Obj* SkipNode<Key,Obj>::getObj(void)
  {
    return obj;
  }

template <class Key, class Obj>
  int SkipNode<Key,Obj>::getHgt(void)
  {
    return nodeHeight;
  }

#endif //SKIP_LIST_NODE

/* End of File */
```

Listing 4: Template class SkipList

```
#ifndef SKIP_LIST
#define SKIP_LIST
#include <iostream.h>
#include <fstream.h>
#include "SkipNode.h"
#include "RandomHeight.h"

template <class Key, class Obj>
  class SkipList
  {
  public:
    SkipList(float,int,Key*);
    ~SkipList();

    bool insert(Key*, Obj*);
    bool remove(Key*);
    Obj* retrieve(Key*);
    void dump(ofstream&);

  private:
    SkipNode<Key,Obj>* head;
    SkipNode<Key,Obj>* tail;
    float probability;
    int maxHeight;
    int curHeight;
    RandomHeight* randGen;
  };

template <class Key, class Obj>
  SkipList<Key,Obj>::SkipList(float p, int m, Key* k)
  {
    curHeight = 1;
    maxHeight = m;
    probability = p;
    randGen = new RandomHeight(m,p);

    // Create head and tail and attach them
    head = new SkipNode<Key,Obj>(maxHeight);
    tail = new SkipNode<Key,Obj>(k, (Obj*) NULL, maxHeight);
    for ( int x = 1; x <= maxHeight; x++ )
        head->fwdNodes[x] = tail;
  }

template <class Key, class Obj>
  SkipList<Key,Obj>::~SkipList()
  {
    // Walk 0 level nodes and delete all
    SkipNode<Key,Obj>* tmp;
    SkipNode<Key,Obj>* nxt;
    tmp = head;
    while ( tmp )
    {
      nxt = tmp->fwdNodes[1];
      delete tmp;
      tmp = nxt;
    }
  }

template <class Key, class Obj>
  bool SkipList<Key,Obj>::insert(Key* k, Obj* o)
  {
    int lvl = 0, h = 0;
    SkipNode<Key,Obj>** updateVec =
      new SkipNode<Key,Obj>* [maxHeight+1];
    SkipNode<Key,Obj>* tmp = head;
    Key* cmpKey;

    // Figure out where new node goes
    for ( h = curHeight; h >= 1; h-- )
    {
      cmpKey = tmp->fwdNodes[h]->getKey();
      while ( *cmpKey < *k )
      {
        tmp = tmp->fwdNodes[h];
        cmpKey = tmp->fwdNodes[h]->getKey();
      }
      updateVec[h] = tmp;
    }
    tmp = tmp->fwdNodes[1];
    cmpKey = tmp->getKey();

    // If dup, return false
    if ( *cmpKey == *k )
    {
      return false;
    }
    else
    {
      // Perform an insert
      lvl = randGen->newLevel();
      if ( lvl > curHeight )
      {
        for ( int i = curHeight + 1; i <= lvl; i++ )
          updateVec[i] = head;
        curHeight = lvl;
      }
      // Insert new element
      tmp = new SkipNode<Key,Obj>(k, o, lvl);
      for ( int i = 1; i <= lvl; i++ )
      {
        tmp->fwdNodes[i] = updateVec[i]->fwdNodes[i];
        updateVec[i]->fwdNodes[i] = tmp;
      }
    }
    return true;
  }


template <class Key, class Obj>
  bool SkipList<Key,Obj>::remove(Key* k)
  {
    SkipNode<Key,Obj>** updateVec =
      new SkipNode<Key,Obj>* [maxHeight+1];
    SkipNode<Key,Obj>* tmp = head;
    Key* cmpKey;

     // Find the node we need to delete
    for ( int h = curHeight; h > 0; h-- )
    {
      cmpKey = tmp->fwdNodes[h]->getKey();
      while ( *cmpKey < *k )
      {
        tmp = tmp->fwdNodes[h];
        cmpKey = tmp->fwdNodes[h]->getKey();
      }
      updateVec[h] = tmp;
    }
    tmp = tmp->fwdNodes[1];
    cmpKey = tmp->getKey();

    if ( *cmpKey == *k )
    {
      for ( int i = 1; i <= curHeight; i++ )
      {
        if ( updateVec[i]->fwdNodes[i] != tmp )
          break;
        updateVec[i]->fwdNodes[i] = tmp->fwdNodes[i];
      }
      delete tmp;
      while ( ( curHeight > 1 ) &&
            ( ( head->fwdNodes[curHeight]->getKey()
                == tail->getKey() ) ) )
        curHeight--;
      return true;
    }
    else
    {
      return false;
    }
  }

template <class Key, class Obj>
  Obj* SkipList<Key,Obj>::retrieve(Key* k)
  {
    int h = 0;
    SkipNode<Key,Obj>** updateVec =
      new SkipNode<Key,Obj>* [maxHeight+1];
    SkipNode<Key,Obj>* tmp = head;
    Key* cmpKey;

    // Find the key and return the node
    for ( h = curHeight; h >= 1; h-- )
    {
      cmpKey = tmp->fwdNodes[h]->getKey();
      while ( *cmpKey < *k )
      {
        tmp = tmp->fwdNodes[h];
        cmpKey = tmp->fwdNodes[h]->getKey();
      }
      updateVec[h] = tmp;
    }
    tmp = tmp->fwdNodes[1];
    cmpKey = tmp->getKey();
    if ( *cmpKey == *k )
      return tmp->getObj();
    else
      return (SkipNode<Key,Obj>*) NULL;
  }

template <class Key, class Obj>
  void SkipList<Key,Obj>::dump(ofstream& of)
  {
    SkipNode<Key,Obj>* tmp;

    tmp = head;
    while ( tmp != tail )
    {
      if ( tmp == head )
        of << "There's the head node!" << endl << flush;
      else
```

```
        // Your key class must support "<<"
        of << "Next node holds key: " << tmp->getKey() << endl
           << flush;
      tmp = tmp->fwdNodes[1];
    }
    of << "There's the tail node!" << endl << flush;
  }

#endif //SKIP_LIST

/* End of File */
```