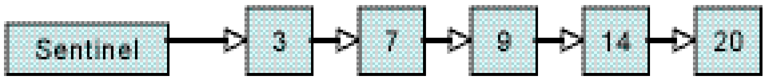
Worksheet 28: Skip Lists

In Preparation: Read Chapter 8 to learn more about the Bag data type. If you have not done so already, you should do Lessons 17 and 18 to learn more about the basic features of the linked list.

We have seen two implementation techniques for the Bag, using a dynamic array and a linked list. However, because the order of insertion is unimportant for a Bag, a container is free to reorganize the elements to make various operations run more quickly. We will see in this lesson and future lessons several different data structures that make use of this principle.

The SkipList is a little more complex data structure than we have seen up to this point, and so we will spend more time in development and walking you through the implementation. To motivate the need for the skip list, consider that ordinary linked lists and vectors have fast (O(1)) addition of new elements, but a slow time for search and removal. A sorted vector has a fast O(log n) search time, but is still slow in addition of new elements and in removal. A skip list is fast O(log n) in all three times.

We begin the development of the skip list by first considering an simple ordered list, with a sentinel on the left, and single links:



To add a new value to such a list you find the correct location, then set the value. Similarly to see if the list contains an element you find the correct location, and see if it is what you want. And to remove an element, you find the correct location, and remove it. Each of these three has in common the idea of finding the location at which the operation will take place. We can generalize this by writing a common routine, named **slideRight**, that will move to the right as long as the next element is smaller than the value being considered. The implementation of each of the three basic operations is then a simple matter:

```
struct skipLink * slideRight (struct skipLink * current, EleType d) {
   while ((current->next != 0) && LT(current->next->value, d))
      current = current->next;
   return current;
}

void add (struct skipList *sl, EleType d) {
   struct skipLink * current = slideRight(sl->sentinel, d);
   struct skipLink * newLink = (struct skipLink *)
                      malloc(sizeof(struct skipLink));
   assert(newLink != 0);
   newLink->next = current->next;
   current->next = newLink;
   st->size++;
}
```

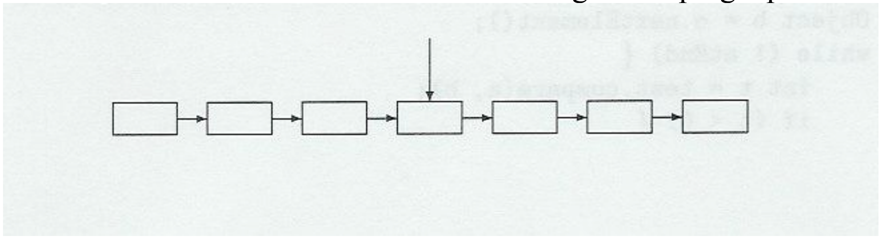
```
 int contains (struct skipList *sl, EleType d) {
   struct skipLink *current = slideRight(sl->sentinel, d);
   if ((current->next != null) && EQ(current->next->value, d))  return 1;
   return 0;
}

void remove (struct skipList *sl, EleType d) {
   struct skipLink *current = slideRight(sl->sentinel, d);
   if ((current->next != 0) && EQ(current->next->value, d)) {
      struct lnk = current->next;
      current->next = lnk->next;
      free(lnk);
      sl->size--;
   }
}
```

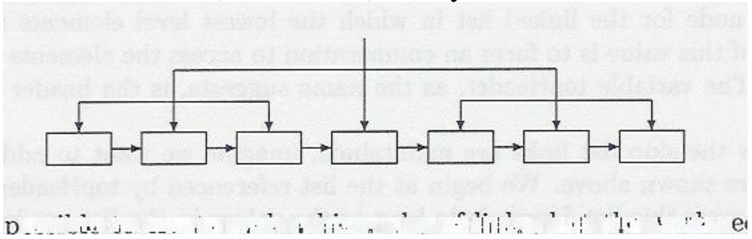
Try simulating some of the operations on this structure using the list shown until you understand what the function slideRight is doing. What happens when you insert the value 10? Search for 14? Insert 25? Remove the value 9?

By itself, this new data structure is not particularly useful or interesting. Each of the three basic operations still loop over the entire collection, and are therefore O(n), which is no better than an ordinary vector or linked list.

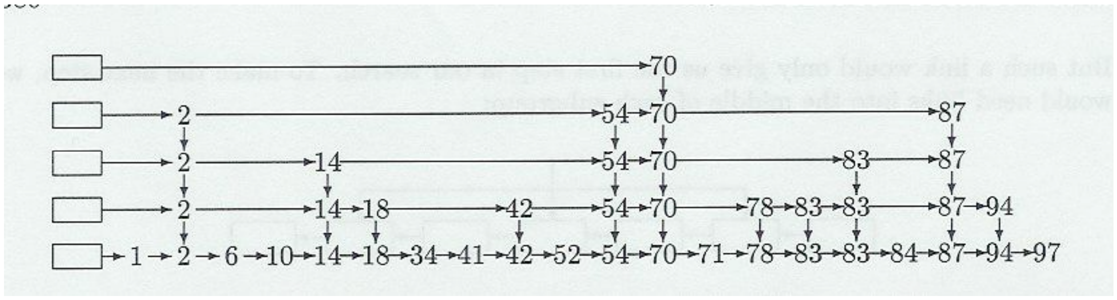
Why can't one do a binary search on a linked list? The answer is because you cannot easily reach the middle of a list. But one could imagine keeping a pointer into the middle of a list:



and then another, an another, until you have a tree of links



In theory this would work, but the effort and cost to maintain the pointers would almost certainly dwarf the gains in search time. But what if we didn't try to be precise, and instead maintained links *probablistically*? We could do this by maintaining a stack of ordered lists, and links that could point down to the lower link. We can arrange this so that each level has *approximately* half the links of the next lower. There would therefore be approximately log n levels for a list with n elements.



We are going to lead you through the implementation of the Skip List. The sentinel will be the topmost sentinel in the above picture. As you are developing the code you should simulate the execution of your routine on the picture shown.

The simplest operation is the contains test. The pseudo-code for this algorithm is as follows:

```
skipListContains (struct skipList *slst, EleType testValue) {
   current = topmost sentinel;
   while (current is not null) {
      current = slide right (current, testValue)
      if the next element after current is equal to testValue
            then return true
      current = current.down
   }
   return false;
}
```

Try tracing the execution of this routine as you search for the following values in the picture above: 70, 85, 1, 5, 6. Once you are convinced that the pseudo-code is correct, then rewrite it in C. The end of this worksheet as the skeleton of these functions.

The next operation is **remove**. This is similar to the contains test, with the additional problems that (a) a value can be on more than one list, and you need to remove them all, and (b) you need to reduce the element count only when removing from the bottommost level. In pseudo-code this is written as follows:

```
skipListRemove (struct skipList *slst, EleType testValue) {
   current = sentinel;
   while (current is not null) {
      current = slide right (current, testValue)
      if the next element after current is equal to testValue then {
         remove the next node
         if current.down is null reduce the element count
      }
      current = current.down
   }
}
```

Notice that finding the value on one level does not immediately stop the operation - you must proceed all the way down to the bottom level. Try simulating the execution of this algorithm on the earlier picture, removing the following elements: 97, 78, 41, 85, 70.

The addition operation is the most complex of the three basic operations. The **add** operation is the place where probability comes into play. This operation is most easily written using a recursive helper function that takes a link and a value and may (probabilistically) either return a newly installed link, or a null value. The link that is the first argument is the position at which a value should be inserted (e.g., the result of calling **slideRight**). Imagine, for example, that we are inserting the value 15. The result of calling slideRight on the structure shown earlier would be simply the sentinel. The following describes the add operation in pseudo-code:

```
add (Link current, EleType newValue) {
   if we are at the bottom (that is, current.down is null) then
      make a new link for the new value and insert it following current
      add 1 to the counter for the collection size
      return the new link
   else {  // we are not at the bottm level
      downlink= add(slideRight(current,down, testValue), testValue)
      if (downLink is not null and coin flip is heads then {
         make new link for test value and install it after current
         return new link
      }
   }
   return null
}
```

Flipping a coin can be simulated by using random numbers. The method rand() returns a random integer value. If this value is even, assume the coin is heads. If off, assume tails.The topmost add operation performs an initial slide before calling the helper routine. If the helper has returned a link the coin is flipped one more time, and if heads a new level of list is created. This involves creating a new sentinel and installing a new link as the one element in the list. This is written in pseudo-code as follows:

```
add (EleType newValue) {
   Link downLink = add(slideRight(sentinel, testValue), testValue);
   if (downLink is not null and coin flip is heads then {
      // make new level of list
      make new link for newValue, pointing down to downLink
      make new sentinel, pointing right to new link and down to existing
sentinel
   }
}
```

You should try simulating the addition of new values to the picture shown earlier.

The following is the beginning of an implementation of this data abstraction. Much of the structure has been provided for you. Your task is to fill in the remaining parts. The task of allocating and initializing a new link has been moved into a separate helper function named `createSkipLink(double, struct skipLink* nlnk, struct skipLink *dlnk).`

```
struct skipList {
        struct skipLink *topSentinel;
        int size;
};
```

```
struct skipLink {
        EleType value;
        struct skipLink *next;
        struct skipLink *down;
};

struct skipLink * skipLinkCreate (EleType d, struct skipLink *nlnk, struct
skipLink *dlnk) {
   /* create a new link and initialize it with the given values */






}


void skipListInit (struct skipList *slst) {
   /* initialize a skip list structure */





}

int skipFlip() { return rand() % 2; }

struct skipLink * skipLinkAdd (struct skipLink * current, EleType d) {
   /* inner helper routine, add value at current location in list */







}

void skipListAdd (struct skipList *slst, EleType d) {









}

int skipListContains (struct skipList *slst, EleType d) {
}

void skipListRemove (struct skipList *slst, EleType d) {





}

int skipListSize (struct skipList *slst) { return slst->size; }
```