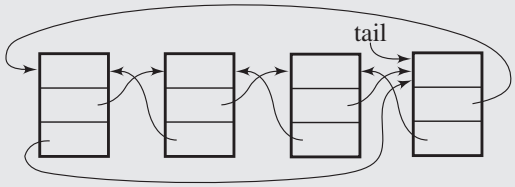
FIGURE 3.15 A circular doubly linked list.



3.4 Skip Lists

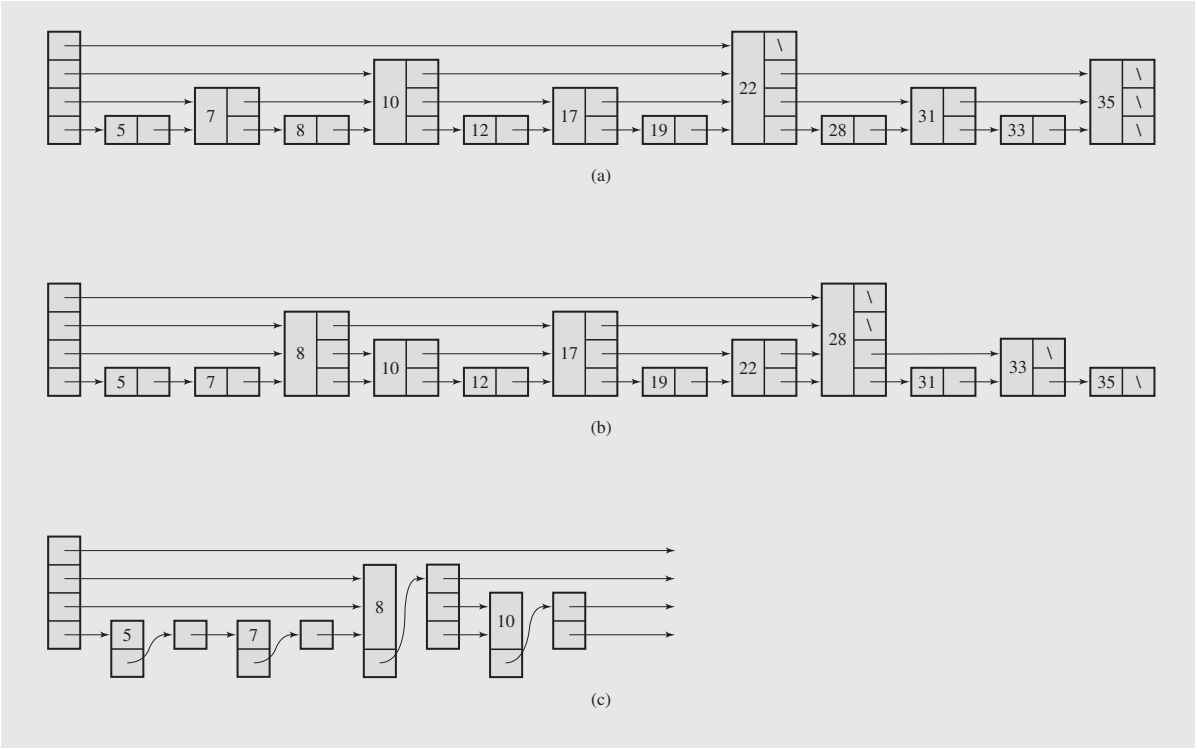
Linked lists have one serious drawback: they require sequential scanning to locate a searched-for element. The search starts from the beginning of the list and stops when either a searched-for element is found or the end of the list is reached without finding this element. Ordering elements on the list can speed up searching, but a sequential search is still required. Therefore, we may think about lists that allow for skipping certain nodes to avoid sequential processing. A *skip list* is an interesting variant of the ordered linked list that makes such a nonsequential search possible (Pugh 1990).

In a skip list of n nodes, for each k and i such that $1 \le k \le \lfloor \lg n \rfloor$ and $1 \le i \le \lfloor n/2^{k-1} \rfloor - 1$, the node in position $2^{k-1} \cdot i$ points to the node in position $2^{k-1} \cdot (i + 1)$. This means that every second node points to the node two positions ahead, every fourth node points to the node four positions ahead, and so on, as shown in Figure 3.16a. This is accomplished by having different numbers of pointers in nodes on the list: half of the nodes have just one pointer, one-fourth of the nodes have two pointers, one-eighth of the nodes have three pointers, and so on. The number of pointers indicates the *level* of each node, and the number of levels is $maxLevel = \lfloor \lg n \rfloor + 1$.

Searching for an element `el` consists of following the pointers on the highest level until an element is found that finishes the search successfully. In the case of reaching the end of the list or encountering an element `key` that is greater than `el`, the search is restarted from the node preceding the one containing `key`, but this time starting from a pointer on a lower level than before. The search continues until `el` is found, or the first-level pointers are followed to reach the end of the list or to find an element greater than `el`. Here is a pseudocode for this algorithm:

```
search(element el)
   p = the nonnull list on the highest level i;
   while el not found and i ≥ 0
       if p->key > el
            p = a sublist that begins in the predecessor of p on level --i;
       else if p->key < el
            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;
```

FIGURE 3.16 A skip list with (a) evenly and (b) unevenly spaced nodes of different levels; (c) the skip list with pointer nodes clearly shown.



For example, if we look for number 16 in the list in Figure 3.16b, then level four is tried first, which is unsuccessful because the first node on this level has 28. Next, we try the third-level sublist starting from the root: it first leads to 8, and then to 17. Hence, we try the second-level sublist that originates in the node holding 8: it leads to 10 and then again to 17. The last try is by starting the first-level sublist, which begins in node 10; this sublist's first node has 12, the next number is 17, and because there is no lower level, the search is pronounced unsuccessful. Code for the searching procedure is given in Figure 3.17.

Searching appears to be efficient; however, the design of skip lists can lead to very inefficient insertion and deletion procedures. To insert a new element, all nodes following the node just inserted have to be restructured; the number of pointers and the values of pointers have to be changed. In order to retain some of the advantages that skip lists offer with respect to searching and avoid problems with restructuring the lists when inserting and deleting nodes, the requirement on the positions of nodes of different levels is now abandoned and only the requirement on the number of nodes of different levels is kept. For example, the list in Figure 3.16a becomes the list in Figure 3.16b: both lists have six nodes with only one pointer, three nodes with two pointers, two nodes with three pointers, and one node with four pointers. The new list is searched exactly the same way as the

FIGURE 3.17 An implementation of a skip list.

```
//************************  genSkipL.h  ************************
//                      generic skip list class

const int maxLevel = 4;

template<class T>
class SkipListNode {
public:
    SkipListNode() {
    }
    T key;
    SkipListNode **next;
};

template<class T>
class SkipList {
public:
    SkipList();
    bool isEmpty() const;
    void choosePowers();
    int  chooseLevel();
    T* skipListSearch(const T&);
    void skipListInsert(const T&);
private:
    typedef SkipListNode<T> *nodePtr;
    nodePtr root[maxLevel];
    int powers[maxLevel];
};

template<class T>
SkipList<T>::SkipList() {
    for (int i = 0; i < maxLevel; i++)
        root[i] = 0;
}
template<class T>
bool SkipList<T>::isEmpty() const {
       return root[0] == 0;
}
template<class T>
void SkipList<T>::choosePowers() {
    powers[maxLevel-1] = (2 << (maxLevel-1)) - 1;  // 2^maxLevel - 1
    for (int i = maxLevel - 2, j = 0; i > = 0; i--, j++)
```

FIGURE 3.17 *(continued)*

```
        powers[i] = powers[i+1] - (2 << j);        // 2^(j+1)
}
template<class T>
int SkipList<T>::chooseLevel() {
    int i, r = rand() % powers[maxLevel-1] + 1;
    for (i = 1; i < maxLevel; i++)
        if (r < powers[i])
            return i-1; // return a level < the highest level;
    return i-1;         // return the highest level;
}
template<class T>
T* SkipList<T>::skipListSearch(const T& key) {
    if (isEmpty()) return 0;
    nodePtr prev, curr;
    int lvl;                              // find the highest non-null
    for (lvl = maxLevel-1; lvl >= 0 && !root[lvl]; lvl--);  // level;
    prev = curr = root[lvl];
    while (true) {
        if (key == curr->key)                    // success if equal;
            return &curr->key;
        else if (key < curr->key) {              // if smaller, go down
            if (lvl == 0)                        // if possible,
                return 0;
            else if (curr == root[lvl])          // by one level
                curr = root[--lvl];              // starting from the
            else curr = *(prev->next + --lvl); // predecessor which
        }                                        // can be the root;
        else {                                   // if greater,
            prev = curr;                         // go to the next
            if (*(curr->next + lvl) != 0)        // non-null node
                curr = *(curr->next + lvl);      // on the same level
            else {                               // or to a list on a
                                                 // lower level;
                for (lvl--; lvl >= 0 && *(curr->next + lvl)==0; lvl--);
                if (lvl >= 0)
                    curr = *(curr->next + lvl);
                else return 0;
            }
        }
    }
}
```

Continues

FIGURE 3.17 *(continued)*

```
template<class T>
void SkipList<T>::skipListInsert(const T& key) {
    nodePtr curr[maxLevel], prev[maxLevel], newNode;
    int lvl, i;
    curr[maxLevel-1] = root[maxLevel-1];
    prev[maxLevel-1] = 0;
    for (lvl = maxLevel - 1; lvl >= 0; lvl--) {
        while (curr[lvl] && curr[lvl]->key < key) { // go to the next
            prev[lvl] = curr[lvl];           // if smaller;
            curr[lvl] = *(curr[lvl]->next + lvl);
        }
        if (curr[lvl] && curr[lvl]->key == key)      // don't include
            return;                          // duplicates;
        if (lvl > 0)                         // go one level down
            if (prev[lvl] == 0) {            // if not the lowest
                  curr[lvl-1] = root[lvl-1]; // level, using a link
                  prev[lvl-1] = 0;           // either from the root
             }
             else {                          // or from the predecessor;
                  curr[lvl-1] = *(prev[lvl]->next + lvl-1);
                  prev[lvl-1] = prev[lvl];
             }
    }
    lvl = chooseLevel();        // generate randomly level for newNode;
    newNode = new SkipListNode<T>;
    newNode->next = new nodePtr[sizeof(nodePtr) * (lvl+1)];
    newNode->key  = key;
    for (i = 0; i <= lvl; i++) {         // initialize next fields of
        *(newNode->next + i) = curr[i];  // newNode and reset to newNode
        if (prev[i] == 0)                // either fields of the root
             root[i] = newNode;          // or next fields of newNode's
        else *(prev[i]->next + i) = newNode;    // predecessors;
    }
}
```

original list. Inserting does not require list restructuring, and nodes are generated so that the distribution of the nodes on different levels is kept adequate. How can this be accomplished?

Assume that *maxLevel* = 4. For 15 elements, the required number of one-pointer nodes is eight, two-pointer nodes is four, three-pointer nodes is two, and four-pointer nodes is one. Each time a node is inserted, a random number *r* between 1 and 15 is

generated, and if $r < 9$, then a node of level one is inserted. If $r < 13$, a second-level node is inserted, if $r < 15$, it is a third-level node, and if $r = 15$, the node of level four is generated and inserted. If *maxLevel* = 5, then for 31 elements the correspondence between the value of *r* and the level of node is as follows:

r	**Level of Node to Be Inserted**
31	5
29–30	4
25–28	3
17–24	2
1–16	1

To determine such a correspondence between *r* and the level of node for any *maxLevel,* the function `choosePowers()` initializes the array `powers[]` by putting lower bounds on each range. For example, for *maxLevel* = 4, the array is [1 9 13 15]; for *maxLevel* = 5, it is [1 17 25 29 31]. `chooseLevel()` uses `powers[]` to determine the level of the node about to be inserted. Figure 3.17 contains the code for `choosePowers()` and `chooseLevel()`. Note that the levels range between 0 and *maxLevel*–1 (and not between 1 and *maxLevel*) so that the array indexes can be used as levels. For example, the first level is level zero.

But we also have to address the question of implementing a node. The easiest way is to make each node have *maxLevel* pointers, but this is wasteful. We need only as many pointers per node as the level of the node requires. To accomplish this, the `next` member of each node is not a pointer to the next node, but to an array of pointer(s) to the next node(s). The size of this array is determined by the level of the node. The `SkipListNode` and `SkipList` classes are declared, as in Figure 3.17. In this way, the list in Figure 3.16b is really a list whose first four nodes are shown in Figure 3.16c. Only now can an inserting procedure be implemented, as in Figure 3.17.

How efficient are skip lists? In the ideal situation, which is exemplified by the list in Figure 3.16a, the search time is $O(\lg n)$. In the worst situation, when all lists are on the same level, the skip list turns into a regular singly linked list, and the search time is $O(n)$. However, the latter situation is unlikely to occur; in the random skip list, the search time is of the same order as the best case; that is, $O(\lg n)$. This is an improvement over the efficiency of search in regular linked lists. It also turns out that skip lists fare extremely well in comparison with more sophisticated data structures, such as self-adjusting trees or AVL trees (see Sections 6.7.2 and 6.8), and therefore they are a viable alternative to these data structures (see also the table in Figure 3.20).

3.5 Self-Organizing Lists

The introduction of skip lists was motivated by the need to speed up the searching process. Although singly and doubly linked lists require sequential search to locate an element or to see that it is not in the list, we can improve the efficiency of the search by dynamically organizing the list in a certain manner. This organization