UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

CS61B **P. N. Hilfinger**
Fall 1999

Balanced Searching: Tries, Treaps*

1 Balance

We've seen that binary search trees have a weakness: a tendency to be become *unbalanced,* so that they are ineffective in dividing the set of data they represent into two substantially smaller parts. Let's consider what we can do about this.

Of course, we could always rebalance an unbalanced tree by simply laying all the keys out in order and then re-inserting them in such a way as to keep the tree balanced. That operation, however, requires time linear in the number of keys in the tree, and it is difficult to see how to avoid having a $\Theta(N^2)$ factor creep in to the time required to insert N keys. By contrast, only $O(N \lg N)$ time is required to make N insertions if the data happen to be presented in an order that keeps the tree bushy. So let's look at operations to re-balance a tree without taking it apart and reconstructing it.

What we need is an operation that changes the balance of a BST—choosing a new root that moves keys from a deep side to a shallow side—while preserving the binary search tree property. The simplest such operations are the *rotations* of a tree. Figure 1 shows two BSTs holding identical sets of keys. Consider the rightRotation first (the left is a mirror image). First, the rotation preserves the binary search tree property. In the unrotated tree, the nodes in A are the exactly the ones less than B, as they are on the right; D is greater, as on the right; and subtree C is greater, as on the right. You can also assure yourself that the nodes under D in the rotated tree bear the proper relation to it.

Turning to height, let's use the notation H_A, H_C, H_E, H_B, and H_D to denote the heights of subtrees A, C, and E and of the subtrees whose roots are nodes B and D. Any of A, C, or E can be empty; we'll take their heights in that case to be -1. The height of the tree on the left is $1 + \max(H_E, 1 + H_A, 1 + H_C)$. The height of the tree on the right is $1 + \max(H_A, 1 + H_C, 1 + H_E)$. Therefore, as long as $H_A > \max(H_C + 1, H_E)$ (as would happen in a left-leaning tree, for example), the height of the right-hand tree will be less than that of the left-hand tree. One gets a similar situation in the other direction.

In fact, it is possible to convert any BST into any other that contains the same keys by means of rotations. This amounts to showing that by rotation, we can move any node of

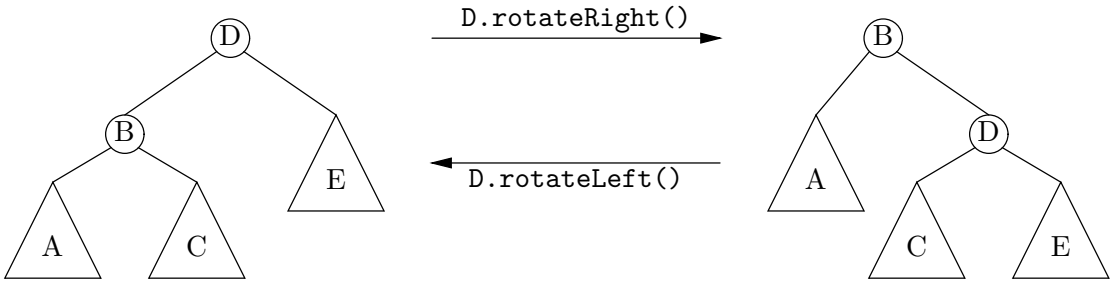

Figure 1: Rotations in a binary search tree. Triangles represent subtrees and circles represent individual nodes. The binary search tree relation is maintained by both operations, but the levels of various nodes are affected.

a BST to the root of the tree while preserving the binary search tree property [why is this sufficient?]. The argument is an induction on the structure of trees.

- It is clearly possible for empty or one-element trees.
- Suppose we want to show it for a larger tree, assuming (inductively) that all smaller trees can be rotated to bring any of their nodes their root. We proceed as follows:
  - If the node we want to make the root is already there, we're done.
  - If the node we want to make the root is in the left child, rotate the left child to make it the root of the left child (inductive hypothesis). The perform a right rotation on the whole tree.
  - Similarly if the node we want is in the right child.

Of course, knowing that it is possible to re-arrange a BST by means of rotation doesn't tell us which rotations to perform. There are various schemes that involve explicitly or implicitly keeping track of the heights of subtrees and performing rotations when they get too far out of line: AVL trees, red-black trees, 2-3 trees, B-trees are some traditional examples. They are all a bit complicated, and since they are almost never actually used (with the exception of B-trees, and those only in large database systems), I thought we might perhaps look as well look at something equally unused but perhaps a bit more interesting: a *randomized search tree,* one that is balanced with high probability. One isn't certain of balance, just pretty certain.

2 Treaps: Probablistic balancing

One example of such a structure is the *treap*, a combination (as the name implies) of a binary search tree and a heap[1] More specifically, each node of a treap contains a search key and a

[1]See "Randomized search trees," Cecilia R. Aragon and Raimund G. Seidel, *30th Annual Symposium on Foundations of Computer Science (FOCS)*, IEEE Computer Society Press, 1989, pp. 540–545. The same data structure (with a different use) was called a *Cartesian tree* by Jean Vuillemin ("A unifying look at data structures," *CACM* **23** (4) (April, 1980), pp. 229-239).

number called its *priority*. The data structure is maintained in such a way that it is always a binary search tree with respect to the keys, and it is always a binary heap with respect to the priorities.

Searching in a treap is identical to searches in a binary search tree. The priorities are used to keep the tree balanced. Specifically, every time we add a new node to the tree, we assign it a priority by applying a hashing function to its key, and then re-arrange the tree to keep it both a binary search tree and a heap. (Now, of course, you see the *real* reason to introduce this data structure; it combines BSTs, heaps, and hashing all into one data structure.) Likewise, whenever we delete a node from the tree, we "re-heapify" the tree, maintaining its status as a binary search tree. The idea is that *with high probability*, the resulting tree will be reasonably balanced (its height will be within a fixed constant factor of $\ln N$), because the priorities effectively choose at random between the many possible binary search trees that can hold those keys.

Let's now turn to effects of rotation on the heap structure of the tree. Consider first the left tree in Figure 1, and assume that it satisfies the heap property, *except* that node B contains a priority that is larger than that of D. Then the result of the right rotation is easily seen to satisfy the heap property completely. Since C and E were under D on the left, they must contain priorities smaller than that of D, so it is valid to put them as children. D is clearly a valid child of B, and since A was originally under B, it remains a valid child of B. Furthermore, B is now higher than it previously was.

Alternatively, suppose that the left tree is a heap except that node D contains a value less than either of its children, and that B is the larger child. Then it is valid for all the subtrees, A, C, and E to be under B, and the right tree is again a heap, except that the priority of D may be smaller than those of its children, and D now has fewer descendents. Thus by continuing to rotate D down, we will eventually restore the heap property.

The rotations are mirror images of each other. Thus, when we have a tree configured like the right tree in Figure 1, and it is a valid heap except that D's priority is larger than B's, a *left* rotation fixes the problem.

These considerations indicate the necessary insertion and deletion routines. Here is our data structure in outline (as usual, using integers as labels):

```
class Treap {
   private int label;
   protected Treap left, right;
   protected int priority;

   /** A singleton Treap. */
   public Treap(int label)
   { this.label = label; priority = someHashFunction(L); }

   public static final Treap EMPTY = new EmptyTreap();

   public boolean isEmpty() { return false; }
   public Treap left() { return this.left; }
```

```
   public Treap right() { return this.right; }
   public int label() { return this.label; }

   /* A node in this Treap with label L, or null if none. */
   public Treap find(int L) { /* same as for BST */ }

   /* Tree rotations */
   protected Treap rotateLeft() {...}
   protected Treap rotateRight() {...}

   public Treap insert(int L) { ... }
   public Treap remove(int L) { ... }
}
```

Rotations are routine:

```
/** The result of performing a right rotation on the root of this
 *  treap, returning the root of the result. Assumes my left child.
 *  is not empty. */
protected Treap rotateRight() {
   Treap result = left;
   left = left.right;
   result.right = this;
   return result;
}
/* rotateLeft is the same, swapping left for right. */
```

For insertion, we first insert the new label in the appropriate child. We assume recursively that if the newly inserted node has a priority larger than that of parent, it will rise to the root of the child, where a single rotation will restore the heap property.

```
/** Insert label X into this treap. */
public Treap insert(int X)
{
   if (X < label) {
      left = left.insert(X);
      if (left.priority > this.priority)
         return rotateRight();
   } else {
      right = right.insert(X);
      if (right.priority > this.priority)
         return rotateLeft();
   }
   return this;
}
```

As usual, removal is harder. If one child is empty, we may simply replace the Treap with the other child (empty or not). Otherwise, the trick is to rotate the node to be removed down the tree, each time making it a child of its child with the larger root, until one of its children is empty.

```
/** Remove an instance of label X from this Treap. */
Treap remove(int X)
{
   if (X < label())
      left = left.remove(X);
   else if (X > label())
      right = right.remove(X);
   // Removing a label from a child never violates the heap
   // property with respect to this node.
   else
      return removeMe();
}

/** Remove this node from the Treap, returning the
 *  result. */
protected Treap removeMe()
{
   if (left.isEmpty())
      return right;
   else if (right.isEmpty())
      return left;
   else if (left.priority < right.priority) {
      Treap result = rotateLeft();  // My right child now root
      result.left = removeMe();
      return result;
   } else {
      Treap result = rotateRight();  // My left child now root
      result.right = removeMe();
      return result;
   }
}
```

Figure 2 illustrates a sequence of insertions of consecutive integers (which, as you may recall, causes trouble with an ordinary binary tree) starting from an empty treap. Figure 3 shows a sequence of deletions from a treap.

In Figure 2, it might appear at first that we get the same kind of $\Theta(N)$ behavior that caused problems with ordinary binary search trees, especially when looking at the long chain that starts with the left child of the root in the last treap. However, notice that by the time we get to that configuration, the priority of the root node (93) is quite high. If we continue

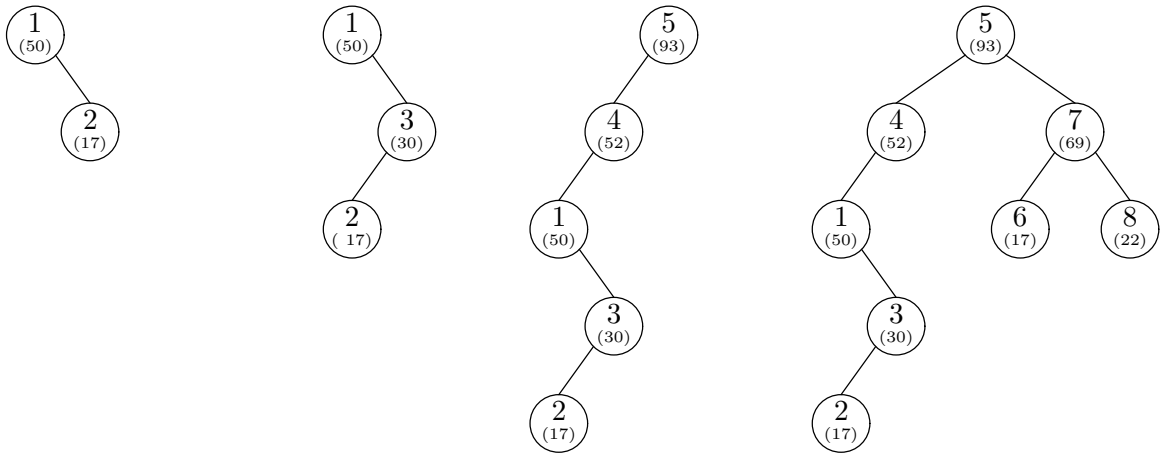

Figure 2: Insertions into a treap. The small numbers in parentheses are the randomly-assigned priorities (in the range 0–99). The numbers 1–8 are added in sequence to an initially-empty treap: first 1 and 2 (no rotations needed), then 3 (one rotation), then 4 and 5 (three rotations), and finally 6–8 (one rotation).

adding numbers consecutively, the left child of the root is not likely to be expanded much more (little else is likely to rotate with the root). At the same time, the right child of the root maintains a reasonable height. It can be shown that these trees tend to be balanced on average. Intuitively, the idea is that if the size of the priority is statistically independent of the size of the key, then the node with largest priority (which will be the root of the tree) is unlikely to coincide with the largest or smallest few keys. That is, the size of the set keys left of it will tend to be within some constant factor of the size of the keys on the right, and that is enough to insure that the amount of data remaining to be searched as `find` searches down the tree will diminish by some multiplicative factor at each step. In a perfect binary search tree, that multiplicative factor is 1/2, but any constant strictly less than 1 will do.

Interestingly enough, the average number of rotations required for each insertion is bounded by a constant (in fact, a constant less than 2), regardless of the size of the tree. Intuitively, this is because the probability of requiring r rotations is the probability that the newly-inserted node is larger than its r most recent ancestors and therefore, by the heap property, greater than the priorities of *all* the descendents of its great^{r-2} grandparent. This probability decreases exponentially, and the expected value of r is constant.

Of course, this is all probabilistic. For any given hashing function, it is possible, as the complexity theorists like to say, that an "adversary" will hand us a string of keys that cause our trees to keep getting longer and thinner. If you want to do this right, therefore, you choose `someHashFunction` *at random.* For integer keys, for example, we can make the hash function something like this:

$$\texttt{someHashFunction}(x) = c_5x^5 + c_4x^4 + c_3x^3 + c_2x^2 + c_1x + c_0$$

(computed modulo 2^32) where the c_i are chosen at random at the beginning of the program.

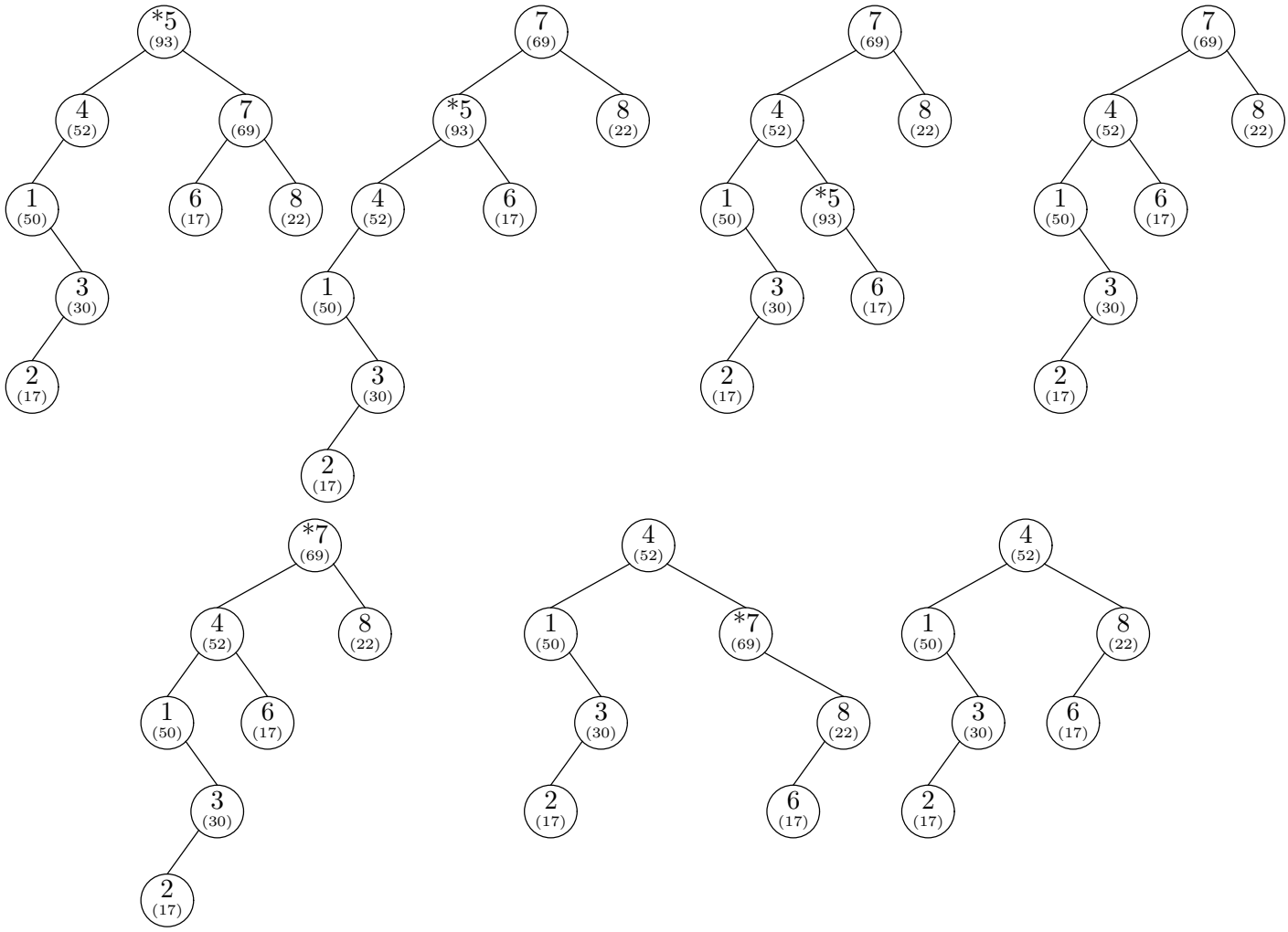

Figure 3: Deletions of 5 (top row) and then 7 from the last treap in Figure 2. The starred nodes indicate the ones doomed to be deleted by `removeMe`.

3 Tries

Loosely speaking, balanced (maximally bushy) binary search trees containing N keys require $\Theta(\lg N)$ time to find a key. This is not entirely accurate, of course, because it neglects the possibility that the time required to *compare* against a key depends on the key. For example, the time required to compare two strings depends on the length of the shorter string. Therefore, in all the places I've said "$\Theta(\lg N)$" before, I *really* meant "$\Theta(L \lg N)$" for L a bound on the number of bytes in the key. In most applications, this doesn't much matter, since L tends to increase very slowly, if at all, as N increases. Nevertheless, this leads to an interesting question: we evidently can't get rid of the factor of L too easily (after all, you have to look at the key you're searching for), but can we get rid of the factor of $\lg N$?

3.1 Tries: basic properties and algorithms

It turns out that we can, using a data structure known as a *trie*[2]. A pure trie is a kind of tree that represents a set of strings from some alphabet of fixed size, say $A = \{a_0, \ldots, a_{M-1}\}$. One of the characters is a special delimiter that appears only at the ends of words, '□'. For example, A might be the set of printable ASCII characters, with □ represented by an unprintable character, such as `'\000'` (NUL). A trie, T, may be abstractly defined by the following recursive definition[3]: A trie, T, is either

- empty, or
- a leaf node containing a string, or
- an internal node containing M children that are also tries. The edges leading to these children are labeled by the characters of the alphabet, a_i, like this: $C_{a_0}, C_{a_1}, \ldots C_{a_{M-1}}$.

We can think of a trie as a tree whose leaf nodes are strings. We impose one other condition:

- If by starting at the root of a trie and following edges labeled $s_0, s_1, \ldots, s_{h-1}$, we reach a string, then that string begins $s_0 s_1 \cdots s_{h-1}$.

Therefore, you can think of every internal node of a trie as standing for some *prefix* of all the strings in the leaves below it: specifically, an internal node at level k stands for the first k characters of each string below it.

A string $S = s_0 s_1 \cdots s_{m-1}$ is in T if by starting at the root of T and following 0 or more edges with labeled $s_0 \cdots s_j$, we arrive at the string S. We will pretend that all strings in T end in □, which appears only as the last character of a string.

Figure 4 shows a trie that represents a small set of strings. To see if a string is in the set, we start at the root of the trie and follow the edges (links to children) marked with the successive characters in the string we are looking for (including the imaginary □ at the end). If we succeed in finding a string somewhere along this path and it equals the string we are searching for, then the string we are searching for is in the trie. If we don't, it is not in the trie. For each word, we need internal nodes only as far down as there are multiple words stored that start with the characters traversed to that point. The convention of ending everything with a special character allows us to distinguish between a situation in which the trie contains two words, one of which is a prefix of the other (like "a" and "abate"), from the situation where the trie contains only one long word.

[2]How is it pronounced? I have no idea. The word was suggested by E. Fredkin in 1960, who derived it from the word "re*trie*val. Despite this etymology, I usually pronounce it like "try" to avoid verbal confusion with "tree."

[3]This version of the trie data structure is described in D. E. Knuth, *The Art of Programming*, vol. 3, which is *the* standard reference on sorting and searching. The original data structure, proposed in 1959 by de la Briandais, was slightly different.

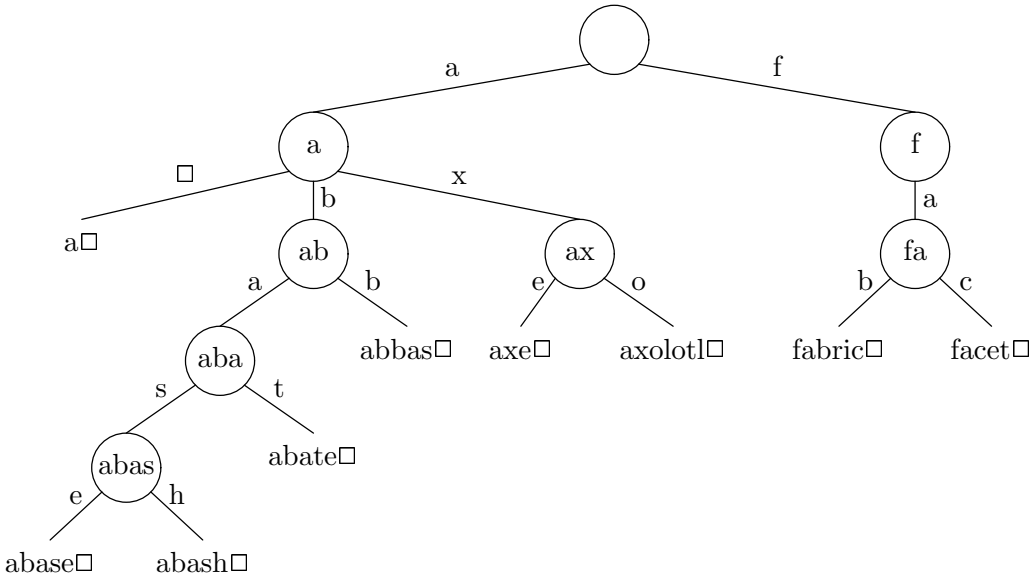

Figure 4: A trie containing the set of strings {a, abase, abash, abate, abbas, axolotl, axe, fabric, facet}. The internal nodes are labeled to show the string prefixes to which they correspond.

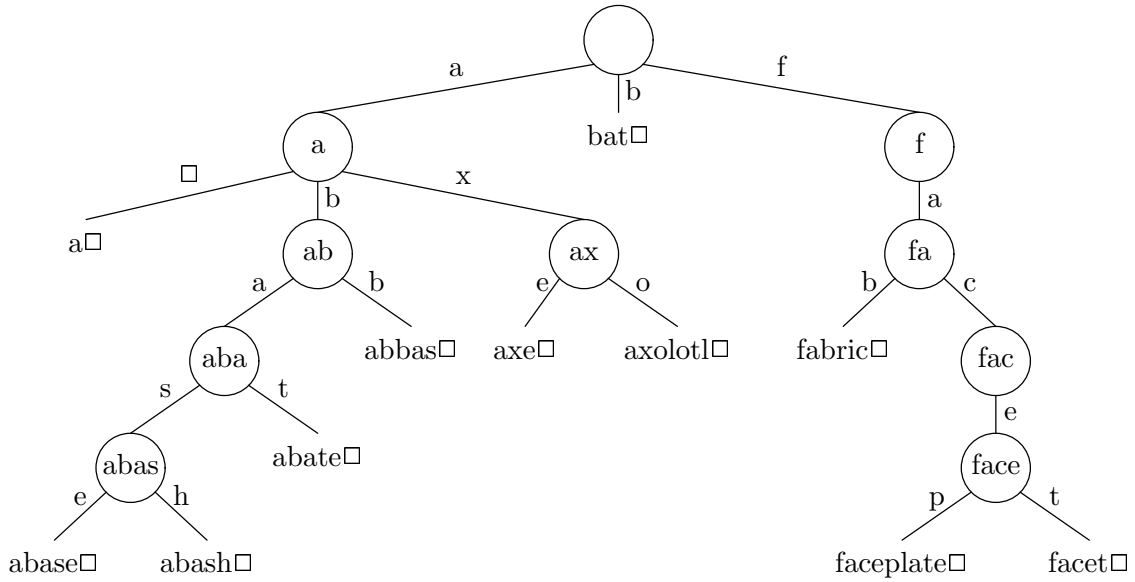

Figure 5: Result of inserting the strings “bat” and “faceplate” into the trie in Figure 4.

From a trie user's point of view, it looks like a kind of tree with String labels:

```
public abstract class Trie {
   /** The empty Trie. */
   public static final Trie EMPTY = new EmptyTrie();

   /** The label at this node.  Defined only on leaves. */
   abstract public String label();

   /** True is X is in this Trie. */
   public boolean isIn(String x) ...

   /** The result of inserting X into this Trie, if it is not
    *  already there, and returning this.  This trie is
    *  unchanged if X is in it already. */
   public Trie insert(String x) ...

   /** The result of removing X from this Trie, if it is present.
    *  The trie is unchanged if X is not present. */
   public Trie remove(String x) ...

   /** True if this Trie is a leaf (containing a single String). */
   abstract public boolean isLeaf();

   /** True if this Trie is empty */
   abstract public boolean isEmpty();

   /** The child numbered with character K. Requires that this node
    *  not be empty.  Child 0 corresponds to □. */
   abstract public Trie child(int k);

   /** Set the child numbered with character K to C.  Requires that
    *  this node not be empty.  (Intended only for internal use. */
   abstract protected void child(int k, Trie C);
}
```

The following algorithm describes a search through a trie.

```
/** True is X is in this Trie. */
public boolean isIn(String x) ...
{
   Trie P = longestPrefix(x, 0);
   return P.isLeaf() && x.equals(P.label());
}

/** The node representing the longest prefix of X.substring(K) that
 *  matches a String in this trie. */
private Trie longestPrefix(String x, int k)
{
   if (isEmpty() || isLeaf())
      return this;
   int c = nth(x, k);
   if (child(c).isEmpty())
      return this;
   else
      return child(c).longestPrefix(x, k+1);
}

/** Character K of X, or □ if K is off the end of X. */
static char nth(String x, int k)
{
   if (k >= x.length())
      return (char) 0;
   else
      return x.charAt(k);
}
```

I have chosen (here and in later methods) to make limited use of fancy object-orientation. For example, I have all those explicit tests like `isEmpty()` and `isLeaf()` rather than having a different `longestPrefix` method for each different kind of trie node (empty, leaf, internal). In this case, I think it makes things a little clearer to have the entire algorithm in one place.

It should be clear from following this procedure that the time required to find a key is proportional to the length of the key. In fact, the number of levels of the trie that need to be traversed can be considerably less than the length of the key, especially when there are few keys stored. However, if a string is in the trie, you will have to look at all its characters, so `isIn` has a worst-case time of $\Theta(x.\texttt{length})$.

To insert a key X in a trie, we again find the longest prefix of X in the trie, which corresponds to some node P. Then, if P is a leaf node, we insert enough internal nodes to distinguish X from $P.\texttt{label()}$. Otherwise, we can insert a leaf for X in the appropriate child

of P. Figure 5 illustrates the results of adding "bat" and "faceplate" to the trie in Figure 4. Adding "bat" simply requires adding a leaf to an existing node. Adding "faceplate" requires inserting two new nodes first.

The method `insert` below performs the trie insertion.

```
/** The result of inserting X into this Trie, if it is not
 *  already there, and returning this.  This trie is
 *  unchanged if X is in it already. */
public Trie insert(String X)
{
    return insert(X, 0);
}

/** Assumes this is a level L node in some Trie.  Returns the  */
 *  result of inserting X into this Trie.  Has no effect (returns
 *  this) if X is already in this Trie. */
private Trie insert(String X, int L)
{
   if (isEmpty())
      return new LeafTrie(X);

   int c = nth(X, L);
   if (isLeaf()) {
      if (X.equals(label()))
         return this;
      else if (c == label().charAt(L))
         return new InnerTrie(c, insert(X, L+1));
      else {
        Trie newNode = new InnerTrie(c, new LeafTrie(X));
        newNode.child(label().charAt(L), this);
        return newNode;
      }
   } else {
      child(c, child(c).insert(X, L+1));
      return this;
   }
}
```

Here, the constructor for `InnerTrie`(c, T), described later, gives us a Trie for which `child`(c) is T and all other children are empty.

Deleting from a trie just reverses this process. Whenever a trie node is reduced to containing a single leaf, it may be replaced by that leaf. The following program indicates the process.

```
public Trie remove(String x)
{
   return remove(x, 0);
}

/** Remove x from this Trie, which is assumed to be level L, and
 *  return the result. */
private Trie remove(String x, int L)
{
   if (isEmpty())
      return this;
   if (isLeaf(T)) {
      if (x.equals(label()))
         return EMPTY;
      else
         return this;
   }
   int c = nth(x, L);
   child(c, child(c).remove(x, L+1));
   int d = onlyMember();
   if (d >= 0)
       return child(d);
   return this;
}

/** If this Trie contains a single string, which is in
 *  child(K), return K.  Otherwise returns -1.
private int onlyMember() {  /* Left to the reader. */ }
```

3.2 Tries: Representation

We are left with the question of how to represent these tries. The main problem of course is that the nodes contain a variable number of children. If the number of children in each node is small, a linked tree representation like those described in Lecture Notes #18 will work. However, for fast access, it is traditional to use an array to hold the children of a node, indexed by the characters that label the edges.

This leads to something like the following:

```
class EmptyTrie extends Trie {
   public boolean isEmpty() { return true; }
   public boolean isLeaf() { return false; }
   public String label() { throw new Error(...); }
   public Trie child(int c) { throw new Error(...); }
   protected void child(int c, Trie T) { throw new Error(...); }
}
```

```
class LeafTrie extends Trie {
   private String L;

   /** A Trie containing just the string S. */
   LeafTrie(String s) { L = s; }

   public boolean isEmpty() { return false; }
   public boolean isLeaf() { return true; }
   public String label() { return L; }
   public Trie child(int c) { return EMPTY; }
   protected void child(int c, Trie T) { throw new Error(...); }
}

class InnerTrie extends Trie {
   // ALPHABETSIZE has to be defined somewhere */
   private Trie[] kids = new kids[ALPHABETSIZE];

   /** A Trie with child(K) == T and all other children empty. */
   InnerTrie(int k, Trie T) {
      for (int i = 0; i < kids.length; i += 1)
         kids[i] = EMPTY;
      child(k, T);
   }

   public boolean isEmpty() { return false; }
   public boolean isLeaf() { return false; }
   public String label() { throw new Error(...); }
   public Trie child(int c) { return kids[c]; }
   protected void child(int c, Trie T) { kids[c] = T; }
}
```

3.3 Table compression

Actually, our alphabet is likely to have "holes" in it—stretches of encodings that don't correspond to any character that will appear in the Strings we insert. We could cut down on the size of the inner nodes (the `kids` arrays) by performing a preliminary mapping of `chars` into a compressed encoding. For example, if the only characters in our strings are the digits 0–9, then we could re-do `InnerTrie` as follows:

```
class InnerTrie extends Trie {
   private static char[] charMap = new char['9'+1];

   static {
      charMap[0] = 0;
      charMap['0'] = 1; charMap['1'] = 1; ...
   }

   public Trie child(int c) { return kids[charMap[c]]; }
   protected void child(int c, Trie T) { kids[charMap[c]] = T; }
}
```

This helps, but even so, arrays that may be indexed by all characters valid in a key are likely to be relatively large (for a tree node)—say on the order of $M = 60$ bytes even for nodes that can contain only digits (assuming 4 bytes per pointer, 4 bytes overhead for every object, 4 bytes for a length field in the array). If there is a total of N characters in all keys, then the space needed is bounded by about $NM/2$. The bound is reached only in a highly pathological case (where the trie contains only two very long strings that are identical except in their last characters). Nevertheless, the arrays that arise in tries can be quite *sparse*.

One approach to solving this is to *compress* the tables. This is especially applicable when there are few insertions once some initial set of strings is accommodated. By the way, the techniques described below are generally applicable to any such sparse array, not just tries.

The basic idea is that sparse arrays (i.e., those that mostly contain empty or "null" entries) can be *overlaid* on top of each other by making sure that the non-null entries in one fall on top of null entries in the others. We allocate all the arrays in a single large one, and store extra information with each entry so that we can tell which of the overlaid arrays that entry belongs to. Here is an appropriate alternative data structure:

```
abstract class Trie {
   ...
   static protected Trie[] allKids;
   static protected char[] edgeLabels;
   static final char NOEDGE = /* Some char that isn't used. */
   static {
      allKids = new Trie[INITIAL_SPACE];
      edgeLabels = new char[INITIAL_SPACE];
      for (int i = 0; i < INITIAL_SPACE; i += 1) {
         allKids[i] = EMPTY; edgeLabels[i] = NOEDGE;
      }
   }
   ...
}
```

```
class InnerTrie extends Trie {
   /* Position of my child 0 in allKids.  My kth child, if
    * non-empty, is at allKids[me + k].  If my kth child is
    * not empty, then edgeLabels[me+k] == k.  edgeLabels[me]
    * is always 0 (□). */
   private int me;


   /** A Trie with child(K) == T and all other children empty. */
   InnerTrie(int k, Trie T) {
      // Set me such that edgeLabels[me + k].isEmpty(). */
      child(0, EMPTY);
      child(k, T);
   }

   public Trie child(int c) {
      if (edgeLabels[me + c] == c)
         return allKids[me + c];
      else
         return EMPTY;
   }

   protected void child(int c, Trie T) {
      if (edgeLabels[me + c] != NOEDGE &&
          edgeLabels[me + c] != c) {
        // Move my kids to a new location, and point me at it.
      }
      allKids[me + c] = T;
      edgeLabels[me + c] = c;
   }
}
```

The idea is that when we store everybody's array of kids in one place, and store an edge label that tells us what character is supposed to correspond to each kid. That allows us to distinguish between a slot that contains somebody else's child (which means that I have no child for that character), and a slot that contains one of my children. We arrange that the `me` field for every node is unique by making sure that the 0th child (corresponding to □ is always full.

As an example, Figure 6 shows the ten internal nodes of the trie in Figure 5 overlaid on top of each other. As the figure illustrates, this representation can be very compact. The number of extra empty entries that are needed on the right (so that `SEL` never indexes off the end of the array) is limited to $M - 1$, so that it becomes negligible when the array is large enough. (Aside: When dealing with a set of arrays that one wishes to compress in this way, it is best to allocate the fullest (least sparse) first.)

Such close packing comes at a price: insertions are expensive. When one adds a new child to an existing node, the necessary slot may already be used by some other array, making it necessary to move the node to a new location by (in effect) first erasing its non-null entries from the packed storage area, finding another spot for it and moving its entries there, and finally updating the pointer to the node being moved in its parent. There are ways to mitigate this, but we won't go into them here.

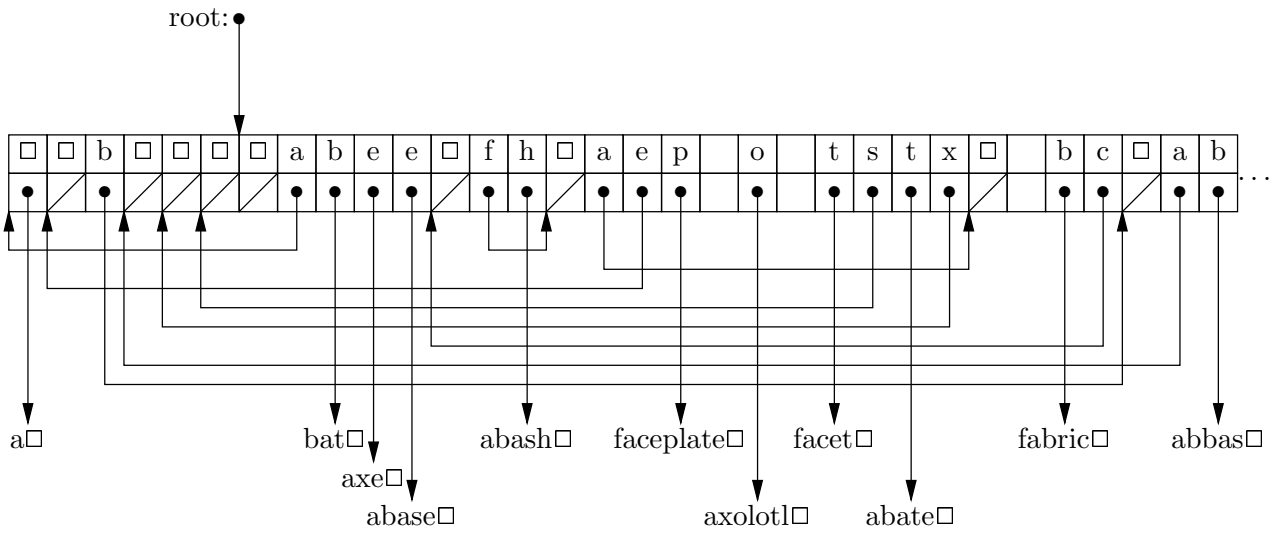

Figure 6: A packed version of the trie from Figure 5. Each of the trie nodes from that figure is represented as an array of children indexed by character, the character that is index of a child is stored in the upper row (which corresponds to the array `edgeLabels`). The pointer to the child itself is in the lower row (which corresponds to the `allKids` array). Empty boxes on top indicate unused locations (the `NOEDGE` value). To compress the diagram, I've changed the character set encoding so that □ is 0, 'a' is 1, 'b' is 2, etc. The crossed boxes in the lower row indicate empty nodes. There must also be an additional 24 empty entries on the right (not shown) to account for the c–z entries of the rightmost trie node stored. The search algorithm uses `edgeLabels` to determine when an entry actually belongs to the node it is currently examining. For example, the root node is supposed to contain entries for 'a', 'b', and 'f'. And indeed, if you count 1, 2, and 6 over from the "root" box above, you'll find entries whose edge labels are 'a', 'b', and 'f'. If, on the other hand, you count over 3 from the root box, looking for the non-existent 'c' edge, you find instead an edge label of 'e', telling you that the root node has no 'c' edge.