# Crit-bit Trees

Adam Langley (agl@imperialviolet.org)
(Version 20080926)

**1.** Introduction

This code is taken from Dan Bernstein's `qhasm` and implements a binary crit-bit (alsa known as PATRICA) tree for `NUL` terminated strings. Crit-bit trees are underused and it's this author's hope that a good example will aid their adoption.

Internal nodes in a crit-bit store a position in the input and two children. The position is the next location in which two members differ (the *crit*ical *bit*). For a given set of elements there is a unique crit-bit tree representing that set, thus a crit-bit tree does not need complex balancing algorithms. The depth of a crit-bit tree is bounded by the length of the longest element, rather than the number of elements (as with an unbalanced tree). Thus, if the crit-bit tree is defined on a finite domain (say, the set of 32-bit integers) then the maximum depth is 32, since no two 32-bit integers can differ in the $33^{rd}$ bit.

Crit-bit trees also support the usual tree operations quickly: membership-testing, insertion, removal, predecessor, successor and easy iteration. For `NUL` terminated strings they are especially helpful since they don't require an expensive string comparison at each step.

This code, like Prof. Bernstein's original code, is released into the public domain. It can be found at `http://github.com/agl/critbit`.

**2.** Structures

We start with the structures used in the crit-bit tree. We'll cover the semantics of each of the members of these structures as need arises.

```
#define _POSIX_C_SOURCE 200112
#define uint8 uint8_t
#define uint32 uint32_t
  format critbit0_node int
  format critbit0_tree int
  format uint8 int
#include <stdint.h>
#include <string.h>
#include <stdlib.h>
#include <sys/types.h>
#include <errno.h>
  typedef struct {
    void *child[2];
    uint32 byte;
    uint8 otherbits;
  } critbit0_node;
  typedef struct {
    void *root;
  } critbit0_tree;
```

**3. Membership testing.**
The first function that we'll deal with will be membership testing. The following function takes a tree, $t$, and a `NUL` terminated string, $u$, and returns non-zero iff $u \in t$.

```
int critbit0_contains(critbit0_tree *t, const char *u)
{
  const uint8 *ubytes = (void *) u;
  const size_t ulen = strlen(u);
  uint8 *p = t->root;
  ⟨ Test for empty tree 4 ⟩
  ⟨ Walk tree for best member 5 ⟩
  ⟨ Check for successful membership 7 ⟩
}
```

**4.** An empty tree
An empty tree simply is simply one where the root pointer is Λ, (that's `NULL` for those who are new to `CWEB`).

```
⟨ Test for empty tree 4 ⟩ ≡
  if (¬p) return 0;
```

This code is used in section 3.

**5.** Searching the tree
Once we have established that the tree is not empty, it therefore has one or more members. Now we need to distinguish between internal and external nodes.

Internal nodes are **critbit0_node** structures. They record that the tree diverges at some point. External nodes are allocated strings. Thus, a tree with a single member is one where the root pointer points at an allocated string. However, we need to be able to test a given pointer to know if it points at an internal or external node. Several possibilities present themselves: a common structure as a prefix to both the internal and external nodes, tags associated with every pointer, *etc.* In this case, we include the tag in the pointer itself as the least-significant bit. We assume that both types of nodes are aligned to, at least, two bytes and thus the LSB is free to be used as a tag bit. Internal nodes are tagged with a 1 and external nodes are tagged with a 0.

When walking the tree we obviously want to break out when we reach an external node. Thus we use a **while** loop that tests that the current node pointer is always pointing at an internal node.

```
⟨ Walk tree for best member 5 ⟩ ≡
  while (1 & (intptr_t)p) {
    critbit0_node *q = (void *)(p - 1);
    ⟨ Calculate direction 6 ⟩
    p = q->child[direction];
  }
```

This code is used in sections 3 and 8.

**6.** Encoding a location

Recall that a crit-bit tree works by encoding the bit-number that differs at each branch in the tree. The obvious way to do this would either be with a single number (the number of bits from the beginning of the string), or with a (byte number, bit number $\in [0..7]$) pair.

However, for reasons that should become clear later, here we encode it as a byte number and a single byte where all the bits *except* the critical bit are true. By performing a bitwise OR with the correct byte there are only two results: If the byte did not have the critical bit set, the result is the same as the mask. If it did, the result is all ones. The latter case is the unique 8-bit value where adding one and right-shifting 8 places results in a 1. We use this to obtain the direction.

Note also that our strings are treated as if they had an infinitely long suffix of `NUL` bytes following them. Thus, if the critical bit is beyound the end of our string, we treat it as if it had a zero bit there.

⟨ Calculate direction 6 ⟩ ≡

```
uint8 c = 0;
if (q→byte < ulen) c = ubytes[q→byte];
const int direction = (1 + (q→otherbits | c)) ≫ 8;
```

This code is used in section 5.

**7.** The final test

Once we have reached an external node we can only conclude that certain bits of the string are shared with a string in the tree. We still need to test the best match to make sure that it's correct. If the test fails, however, we can conclude that the string is not in the tree.

Note that the pointer cannot be Λ. We tested that the root pointer was not Λ at the start of the function and, if an internal node had a Λ pointer then the tree would be invalid - that internal node should be removed.

⟨ Check for successful membership 7 ⟩ ≡

```
return 0 ≡ strcmp(u, (const char *) p);
```

This code is used in section 3.

**8. Inserting into the tree.**

This is a more complex function. It takes a tree, $t$, and possibly mutates it such that a `NUL` terminated string, $u$, is a member on exit. It returns:

$$\begin{cases} 0 & \text{if out of memory} \\ 1 & \text{if } u \text{ was already a member} \\ 2 & \text{if } t \text{ was mutated successfully} \end{cases}$$

Note that the section for walking the tree is the same as before and is not covered again.

```
int critbit0_insert(critbit0_tree *t, const char *u)
{
  const uint8 *const ubytes = (void *) u;
  const size_t ulen = strlen(u);
  uint8 *p = t->root;
  ⟨ Deal with inserting into an empty tree 9 ⟩
  ⟨ Walk tree for best member 5 ⟩
  ⟨ Find the critical bit 10 ⟩
  ⟨ Insert new string 13 ⟩
  return 2;
}
```

**9.** Inserting into an empty tree

Recall that an empty tree has a $\Lambda$ root pointer. A singleton tree, the result of inserting into the empty tree, has the root pointing at an external node - the sole member of the tree.

We require the ability to malloc a buffer with alignment 2 and so use *posix_memalign* to allocate memory.

```
⟨ Deal with inserting into an empty tree 9 ⟩ ≡
  if (¬p) {
    char *x;
    int a = posix_memalign((void **) &x, sizeof(void *), ulen + 1);
    if (a) return 0;
    memcpy(x, u, ulen + 1);
    t->root = x;
    return 2;
  }
```

This code is used in section 8.

**10.** Finding the critical bit

```
⟨ Find the critical bit 10 ⟩ ≡
  ⟨ Find differing byte 11 ⟩
  ⟨ Find differing bit 12 ⟩
```

This code is used in section 8.

**11.** Finding the differing byte

Now that we have found the best match for the new element in the tree we need to check to see where the new element differs from that element. If it doesn't differ, of course, then the new element already exists in the tree and we can return 1. Remember that we treat strings as if they had an infinite number of `NUL`s following them and that the best match string might be longer than $u$.

While calculating the differing byte we also calculate *newotherbits*, the XOR of the differing byte. This will become clear in the next section.

```
⟨ Find differing byte 11 ⟩ ≡
  uint32 newbyte;
  uint32 newotherbits;
  for (newbyte = 0; newbyte < ulen; ++newbyte) {
    if (p[newbyte] ≠ ubytes[newbyte]) {
      newotherbits = p[newbyte] ⊕ ubytes[newbyte];
      goto different_byte_found;
    }
  }
  if (p[newbyte] ≠ 0) {
    newotherbits = p[newbyte];
    goto different_byte_found;
  }
  return 1; different_byte_found:
```

This code is used in section 10.

**12.** Finding the differing bit

Once we have the XOR of first differing byte in *newotherbits* we need to find the most significant differing bit. We could do this with a simple for loop, testing bits 7..0, instead we use the following trick:

The only non-zero values for which the sets of true bits for $x$ and $x-1$ are disjoint, are powers of two. To see this consider the bit representation of the value in three pieces: a series of zeros (maybe empty), a one and a series of ones and zeros. Since we are only considering non-zero values this can be performed without loss of generality. If the third part contains any ones, this number is not a power of two and subtracting one will only alter the third part. Thus, in this case, $x$ and $x-1$ have at least one element in common: the leading one.

However, if the third part consists only of zeros then the number is a power of two. Also, subtracting one will result in clearing the bit in the second part and turning the third part to all ones. Thus the sets are disjoint and $x \,\&\, (x-1)$ is false.

So, we have a test for finding values with only a single bit set. Now consider that, if the test fails, $x \& (x-1)$ must preseve the most-significant one and must be less than $x$: since the bit pattern in the third part changes, at least one bit must be zeroed. Therefore, repeatedly applying the test and, if it fails, updating $x$ in this fashion, must result in a value with only the leading one set.

Once we have this value, we invert all the bits resulting in a value suitable for our *otherbits* member.

```
⟨ Find differing bit 12 ⟩ ≡
  while (newotherbits & (newotherbits − 1)) newotherbits &= newotherbits − 1;
  newotherbits ⊕= 255;
  uint8 c = p[newbyte];
  int newdirection = (1 + (newotherbits | c)) ≫ 8;
```

This code is used in section 10.

**13.** Inserting the new node

```
⟨ Insert new string 13 ⟩ ≡
  ⟨ Allocate new node structure 14 ⟩
  ⟨ Insert new node 15 ⟩
```

This code is used in section 8.

**14.** Allocating a new node

This is obviously fairly pedestrian code. Again, we use *posix_memalign* to make sure that our node structures have an alignment of at least two. We store the new copy of the string into the correct *child* pointer and save the other for when we have worked out where to insert the new node

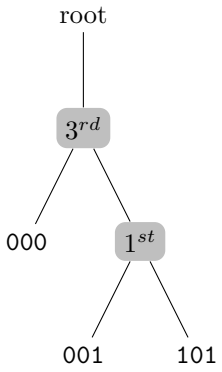
```
⟨ Allocate new node structure 14 ⟩ ≡
  critbit0_node *newnode;
  if (posix_memalign((void **) &newnode, sizeof(void *), sizeof(critbit0_node))) return 0;
  char *x;
  if (posix_memalign((void **) &x, sizeof(void *), ulen + 1)) {
    free(newnode);
    return 0;
  }
  memcpy(x, ubytes, ulen + 1);
  newnode→byte = newbyte;
  newnode→otherbits = newotherbits;
  newnode→child[1 − newdirection] = x;
```

This code is used in section 13.

**15.** Inserting a new node in the tree

Here we must recall that, for a given set of elements, there is a unique crit-bit tree representing them. This statement needs a little bit of qualification because it also requires that we define a total ordering of crit-bits.

Consider the set of bitstrings {000, 001, 101}, inserted into a crit-bit tree in that order. One could imagine the resulting tree looking like this:




(Where internal nodes are shaded light gray and contain the critical bit.)

That would be a valid tree for searching as far as our searching algorithm goes, but it does make a mess of predecessor and successor operations when you can't know that the least-significant-bit wasn't a fork at the top of the tree.

So, in short, we need the order of the crit-bits to match the lexocographical order that we expect the predecessor and successor operations to follow. Thus, inserting the new node in the tree involves walking the tree from the root to find the correct position to insert at.

We keep track of the pointer to be updated (to point to the new internal node) and, once the walk has finished, we can update that pointer.

```
⟨ Insert new node 15 ⟩ ≡
  void **wherep = &t→root;
  for ( ; ; ) {
    uint8 *p = *wherep;
    if (¬(1 & (intptr_t)p)) break;
    critbit0_node *q = (void *)(p − 1);
    if (q→byte > newbyte) break;
    if (q→byte ≡ newbyte ∧ q→otherbits > newotherbits) break;
    uint8 c = 0;
    if (q→byte < ulen) c = ubytes[q→byte];
    const int direction = (1 + (q→otherbits | c)) ≫ 8;
    wherep = q→child + direction;
  }
  newnode→child[newdirection] = *wherep;
  *wherep = (void *)(1 + (char *) newnode);
```

This code is used in section 13.

**16. Deleting elements.**

This function takes a tree, $t$, and a `NUL` terminated string, $u$, and possibly mutates the tree such that $u \notin t$. It returns 1 if the tree was mutated, 0 otherwise.

```
int critbit0_delete(critbit0_tree *t, const char *u)
{
  const uint8 *ubytes = (void *) u;
  const size_t ulen = strlen(u);
  uint8 *p = t->root;
  void **wherep = &t->root;
  void **whereq = 0;
  critbit0_node *q = 0;
  int direction = 0;
  ⟨ Deal with deleting from an empty tree 17 ⟩
  ⟨ Walk the tree for the best match 18 ⟩
  ⟨ Check the best match 19 ⟩
  ⟨ Remove the element and/or node 20 ⟩
  return 1;
}
```

**17.** Deleting from the empty tree

Since no element is the member of the emtpy tree, this is a very easy case: we can just return 0.

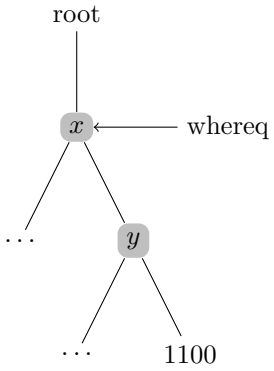
```
⟨ Deal with deleting from an empty tree 17 ⟩ ≡
  if (¬p) return 0;
```

This code is used in section 16.

**18.** Finding the best candidate to delete

Walking the tree to find the best match for a given element is almost the same as the two previous versions that we've seen. The only exception is that we keep track of the last jump to an internal node in *whereq*. Actually, we keep track of a pointer to the last pointer that got us to an internal node.

To see why, consider the typical case:




Here we wish to remove `1100`, however if we left its parent with a single child pointer, that would make the parent nothing more than a bump in the road - it should also be removed. Thus we need a pointer to the grandparent in order to remove both the string and the internal node that pointed to it.

⟨ Walk the tree for the best match 18 ⟩ ≡

```
while (1 & (intptr_t)p) {
  whereq = wherep;
  q = (void *)(p - 1);
  uint8 c = 0;
  if (q->byte < ulen) c = ubytes[q->byte];
  direction = (1 + (q->otherbits | c)) >> 8;
  wherep = q->child + direction;
  p = *wherep;
}
```

This code is used in section 16.

**19.** Checking that we have the right element

As usual, we have now found the best match, an external node, but we still need to compare the strings to check that we actually have a match. If we don't, then the element cannot be in the tree and we can return 0. Otherwise, the external node is no longer useful and can be freed.

⟨ Check the best match 19 ⟩ ≡

```
if (0 != strcmp(u, (const char *) p)) return 0;
free(p);
```

This code is used in section 16.

**20.** Removing the node

We now have to deal with two cases. The simple case is as outlined in the diagram above: we remove the parent node and point the grand parent to to other child of the parent.

We also have to keep in mind that there might not *be* a grandparent node. This is the case when the tree only has one element. In this case, we remove that element and set the root pointer to $\Lambda$.

```
⟨ Remove the element and/or node 20 ⟩ ≡
  if (¬whereq) {
    t→root = 0;
    return 1;
  }
  *whereq = q→child[1 − direction];
  free(q);
```

This code is used in section 16.

**21. Clearing a tree.**

Clearing a tree (freeing all members) brings us our first code for walking the whole tree rather than just tracing a path through it.

So, the *critbit0_clear* function takes a tree, $t$, and frees every member of it, mutating the tree such that it is empty on exit.

```
static void traverse(void *top)
{
  ⟨ Recursivly free current node 22 ⟩
}
void critbit0_clear(critbit0_tree *t)
{
  if (t->root) traverse(t->root);
  t->root = Λ;
}
```

**22.** Recursivly clearing the tree

Each pointer in the tree has to be tested to see if it's a pointer to an internal node (a **critbit0_node**) or to a malloced string. If it's a node, we need to recursivly free its children.

```
⟨ Recursivly free current node 22 ⟩ ≡
  uint8 *p = top;
  if (1 & (intptr_t)p) {
    critbit0_node *q = (void *)(p - 1);
    traverse(q->child[0]);
    traverse(q->child[1]);
    free(q);
  }
  else {
    free(p);
  }
```

This code is used in section 21.

**23. Fetching elements with a given prefix.**

One of the operations which crit-bit trees can perform efficiently that hash tables cannot is the extraction of the subset of elements with a given prefix.

The following function takes a tree, $t$, and a `NUL` terminated string, *prefix*. Let $S \subseteq t$ where $x \in S$ iff *prefix* is a prefix of $x$, then $\forall x : S$. *handle* is called with arguments $x$ and *arg*. It returns:

$$\begin{cases} 0 & \text{if } handle \text{ returned } 0 \\ 1 & \text{if successful} \\ 2 & \text{if } handle \text{ returned a value } \notin [0,1] \end{cases}$$

(Note that, if *handle* returns 0, the iteration is aborted)

```
static int allprefixed_traverse(uint8 *top, int(*handle)(const char *, void *), void *arg)
{
  ⟨ Deal with an internal node 26 ⟩
  ⟨ Deal with an external node 27 ⟩
}
int critbit0_allprefixed(critbit0_tree *t, const char *prefix, int(*handle)(const char *, void *), void
        *arg)
{
  const uint8 *ubytes = (void *) prefix;
  const size_t ulen = strlen(prefix);
  uint8 *p = t→root;
  uint8 *top = p;
  if (¬p) return 1;      /* S = ∅ */
  ⟨ Walk tree, maintaining top pointer 24 ⟩
  ⟨ Check prefix 25 ⟩
  return allprefixed_traverse(top, handle, arg);
}
```

**24.** Maintaining the *top* pointer

The *top* pointer points to the internal node at the top of the subtree which contains exactly the subset of elements matching the given prefix. Since our critbit values are sorted as we decend the tree, this subtree exists (if the subset is non-empty) and can be detected by checking for the critbit advancing beyound the length of the prefix.

⟨ Walk tree, maintaining top pointer 24 ⟩ ≡

```
while (1 & (intptr_t)p) {
  critbit0_node *q = (void *)(p - 1);
  uint8 c = 0;
  if (q→byte < ulen) c = ubytes[q→byte];
  const int direction = (1 + (q→otherbits | c)) ≫ 8;
  p = q→child[direction];
  if (q→byte < ulen) top = p;
}
```

This code is used in section 23.

**25.** Checking that the prefix exists

As with our other functions, it's possible that the given prefix doesn't actually exist in the tree at this point. We need to check the actual contents of the external node that we have arrived at.

```
⟨ Check prefix 25 ⟩ ≡
  for (size_t i = 0; i < ulen; ++i) {
    if (p[i] ≠ ubytes[i]) return 1;
  }
```

This code is used in section 23.

**26.** Dealing with an internal node while recursing

The *allprefixed_traverse* function is called with the root of a subtree as the *top* argument. We need to test the LSB of this pointer to see if it's an internal node. If so, we recursivly walk down the subtree and return. Otherwise we fall through into the code from the section below for handling an external node.

```
⟨ Deal with an internal node 26 ⟩ ≡
  if (1 & (intptr_t)top) {
    critbit0_node *q = (void *)(top - 1);
    for (int direction = 0; direction < 2; ++direction)
      switch (allprefixed_traverse(q->child[direction], handle, arg)) {
      case 1: break;
      case 0: return 0;
      default: return -1;
      }
    return 1;
  }
```

This code is used in section 23.

**27.** Dealing with an external node while recursing

An external node is a malloced string that matches the given prefix. Thus we call the callback and we're done.

```
⟨ Deal with an external node 27 ⟩ ≡
  return handle((const char *) top, arg);
```

This code is used in section 23.

# CRITBIT