This is an implementation of the ternary search trie algorithm, intended for use with null terminated strings. A package which handles arbitrary sequences of unsigned bytes has been done, but I need to clean up the code and write some documentation before I will feel comfortable making it available. Note that keys are case sensitive, so you should force your keys to lower case if you want to deal with things in a case insensitive manner. All of this code is mine and not copied from anywhere, so all bugs and sloppy code are my doing. Please contact me if you have suggestions for changes or corrections.

Peter A. Friend
pafriend@octavian.org
http://www.octavian.org

1. Structures. This implementation uses a ternary search trie to store the characters of a C string. This type of tree works much like a binary tree, yet has three child nodes. The additional middle node is used when a character of a key string matches the character at the current node in the tree.

⟨ Node structure 1 ⟩ ≡
```
struct node {
  unsigned char value;
  struct node *left;
  struct node *middle;
  struct node *right;
};
```
This code is used in section 49.

2. All function calls in the TST package refer to a **struct** *tst*. This structure stores various values set during initialization as well as the node storage area.

The *node_line_width* member refers to how many nodes are allocated at once when no nodes are available from the free list.

The *node_lines* member is a pointer to a **struct** *node_lines*, which will be explained below. This member exists so that all memory allocated for node structures can be freed with a call to *tst_cleanup*( ).

The *free_list* member is a pointer to a **struct node**. This is actually a linked list of nodes, linked together by the *middle* pointers. When nodes are needed for the tree, they are removed from the head of this list, and during deletion of keys, the nodes are inserted.

The *head* member is an array of 256 pointers to **struct node**. All of these pointers are NULL initially, and are filled in as keys are added. Having a separate slot for each letter of the alphabet aids in balancing the top of the tree.

⟨ TST structure 2 ⟩ ≡
```
struct tst {
  int node_line_width;
  struct node_lines *node_lines;
  struct node *free_list;
  struct node *head[256];
};
```
This code is used in section 49.

3. Nodes are allocated in chunks of size **tst** → *node_line_width*. Each time a chunk is allocated, another line of nodes is added to the *node_lines* member of the **struct tst**.

The *node_line* member is allocated by a call to calloc(), so it is not a linked list of nodes, but nodes in contiguous memory. During each allocation, the nodes are added to the *free_list* member of **struct tst**, and the pointers updated.

The *next* pointer is just a pointer to the next line of nodes.

⟨ Node lines structure 3 ⟩ ≡
```
struct node_lines {
  struct node *node_line;
  struct node_lines *next;
};
```
This code is used in section 49.

4. Constants. Some functions return pointers while others return integer values. All functions that return a pointer return NULL on failure. All functions that return an integer return one of the constants below. There are also other constants for use in function calls.

⟨ TST constants 4 ⟩ ≡

```
enum tst_constants {
  TST_OK, TST_ERROR, TST_NULL_KEY, TST_DUPLICATE_KEY, TST_REPLACE, TST_SUBSTRING_MATCH
};
```

This code is used in section 49.

5. Functions. The definitions for all of the functions are below. Note that *tst_grow_node_free_list*( ) is an internal function used only by *tst_insert*( ).

6. This function allocates a **struct tst** and returns the pointer. The *node_line_width* argument controls how many nodes are allocated during initialization and by each call to *tst_grow_node_free_list*( ). This function returns a valid pointer if it succeeds, NULL otherwise.

⟨ Declaration for *tst_init*( ) 6 ⟩ ≡

struct tst ∗*tst_init*(**int** *node_line_width*);

This code is used in section 49.

7. This function inserts *key* into the the tree and associates *key* with a pointer to some *data*. The *data* argument must not be NULL, since NULL is the value returned when a search or delete fails. If *option* is set to `TST_REPLACE`, when an attempt is made to insert a key that is already in the tree, the new *data* replaces the old. Otherwise, `TST_DUPLICATE_KEY` is returned. If the key is successfully inserted, `TST_OK` is returned. If *key* is zero length, `TST_NULL_KEY` is returned. A return value of `TST_ERROR` indicates a memory allocation failure occurred while tring to grow the node free list.

5/05/1999 - Change made to *tst_insert*

When an insert has failed we return `TST_DUPLICATE_KEY`, and if we still want to do anything with the data for that key we have to make a separate call to *tst_search* to get the pointer which is wasteful. A new argument *exist_ptr* has been added to *tst_insert*. When `TST_DUPLICATE_KEY` is returned, *exist_ptr* will contain the data pointer for the existing key.

⟨ Declaration for *tst_insert*( ) 7 ⟩ ≡

int *tst_insert* (**const unsigned char** ∗*key*, **void** ∗*data*, **struct tst** ∗**tst** , **int** *option*, **void** ∗∗*exist_ptr*) ;

This code is used in section 49.

8. This function searches for *key* in the tree. If it succeeds, it returns the *data* pointer associated with the key, NULL otherwise. If a substring match is desired, then specify `TST_SUBSTRING_MATCH` as the option, otherwise set the option to 0. If *match_len* is not NULL, then the length of the match not counting the NULL terminator will be stored there. For example, if the trie contains the strings "test" and "testing" a search for "testi" with `TST_SUBSTRING_MATCH` will return "test" as a match.

⟨ Declaration for *tst_search*( ) 8 ⟩ ≡

void ∗ *tst_search* (**const unsigned char** ∗*key*, **struct tst** ∗**tst** , **int** *option*, **unsigned int** ∗*match_len*) ;

This code is used in section 49.

9. This function deletes *key* from the tree and returns the *data* pointer associated with it. NULL is returned if the key is not in the tree.

⟨ Declaration for *tst_delete*( ) 9 ⟩ ≡

void ∗ *tst_delete* (**const unsigned char** ∗*key*, **struct tst** ∗**tst**) ;

This code is used in section 49.

10. This function is used to grow the free list in the **struct tst**. This must not be called by the user. It is only called by *tst_insert*( ) when inserting keys. It returns 1 on success, `TST_ERROR` otherwise.

⟨ Declaration for *tst_grow_node_free_list*( ) 10 ⟩ ≡

int *tst_grow_node_free_list* (**struct tst** ∗**tst**) ;

This code is used in section 21.

11. The function *tst_cleanup*( ) is used to free the lines of nodes allocated, as well as the **struct tst** itself.

⟨ Declaration for *tst_cleanup*( ) 11 ⟩ ≡

void *tst_cleanup* (**struct tst** ∗**tst**) ;

This code is used in section 49.

12. Initialization with *tst_init*( ).

⟨ tst_init.c 12 ⟩ ≡

```
#include "tst.h"
#include <stdio.h>
#include <stdlib.h>
  struct tst *tst_init(int width)
  {
    struct tst *
        tst;
        struct node *current_node;
        int i;
    ⟨ Allocate tst structure 13 ⟩⟨ Allocate node_lines member 14 ⟩⟨ Set node_line_width and allocate first
        chunk of nodes 15 ⟩⟨ Build free list from just allocated node_line 16 ⟩
  }
```

13. Allocate space for the **struct tst**. If this fails we return NULL;

⟨ Allocate tst structure 13 ⟩ ≡

```
  if ( (tst = (struct tst *) calloc(1, sizeof(struct tst))) ≡ Λ ) return Λ;
```

This code is used in section 12.

14. Allocate space for the **node_lines** member of **struct tst**. If this fails we have to free our **struct tst** and return NULL;

⟨ Allocate **node_lines** member 14 ⟩ ≡

```
  if ( ( tst → node_lines = (struct node_lines *) calloc(1, sizeof(struct node_lines)) ) ≡ Λ )
  {
    free(tst);
    return Λ;
  }
```

This code is used in section 12.

15. After we have our tst structure and the **node_lines** member allocated, we need to set the *node_line_width* member for this first chunk of nodes as well as further allocations. If we fail to allocate our chunk of nodes, we must free our **struct tst** as well as the **node_lines** member and return NULL.

⟨ Set *node_line_width* and allocate first chunk of nodes 15 ⟩ ≡

```
  tst→node_line_width = width; tst → node_lines→next = Λ; if ( ( tst → node_lines→node_line = (struct
       node *) calloc(width, sizeof(struct node)) ) ≡ Λ ) { free ( tst → node_lines ) ;
  free(tst);
  return Λ; }
```

This code is used in section 12.

16. Now we have to step through the just allocated *node_line* and link them together in a linked list fashion. Then we set **tst** → *free_list* to the first node. Finally, we return a pointer to the new **struct tst**.

⟨ Build free list from just allocated *node_line* 16 ⟩ ≡

```
  current_node = tst → node_lines→node_line;
  tst→free_list = current_node; for (i = 1; i < width; i++) { current_node→middle = & ( tst →
       node_lines→node_line[i] ) ;
  current_node = current_node→middle; } current_node→middle = Λ; return tst;
```

This code is used in section 12.

17. Growing the free list with *tst_grow_node_free_list*( ).

```
⟨ tst_grow_node_free_list.c 17 ⟩ ≡
#include "tst.h"
#include <stdio.h>
#include <stdlib.h>
  int tst_grow_node_free_list ( struct tst *tst )
  {
    struct node *current_node;
    struct node_lines *new_line;
    int i;
    ⟨ Allocate tst → node_lines → next 18 ⟩⟨ Allocate the node_line member of tst → node_lines →
        next 19 ⟩⟨ Add the nodes from node_line to tst → free_list 20 ⟩
  }
```

18. Allocate a struct **node_lines** to fill *new_line*. We do this so that we can insert the new structure at the beginning of the linked list. If the allocation fails we return TST_ERROR. We do not reset **tst** → **node_lines** until all of the other allocations have completed successfully.

```
⟨ Allocate tst → node_lines → next 18 ⟩ ≡
  if ((new_line = (struct node_lines *) malloc(sizeof(struct node_lines))) ≡ Λ) return TST_ERROR;
```

This code is used in section 17.

19. Now that we have a new **node_lines** placeholder, we allocate its *node_line* member with the number of nodes specified in **tst** → *node_line_width*. If this fails, we have to deallocate the **node_lines** structure we just allocated, and return TST_ERROR. If the allocation goes okay, we can then update **tst** → **node_lines**.

```
⟨ Allocate the node_line member of tst → node_lines → next 19 ⟩ ≡
  if ((new_line→node_line = (struct node *) calloc(tst→node_line_width, sizeof(struct node))) ≡ Λ) {
    free(new_line);
    return TST_ERROR;
  }
  else { new_line→next = tst → node_lines; tst → node_lines = new_line; }
```

This code is used in section 17.

20. Finally, we need to step through **tst** → **node_lines** → *node_line* and insert the nodes into **tst** → *free_list*. We use the local variable *current_node* to move the pointers from *node_line* to **tst** → *free_list*. Note the essential assumption that *free_list* is empty. Therefore, allocation of nodes with this function must only be done when the free list is empty. When done, we set the last pointer to NULL so we know when the list is empty later, and return 1 to indicate true.

```
⟨ Add the nodes from node_line to tst → free_list 20 ⟩ ≡
  current_node = tst → node_lines→node_line;
  tst→free_list = current_node; for (i = 1; i < tst→node_line_width; i++ ) { current_node→middle = & ( tst
      → node_lines→node_line[i] ) ;
  current_node = current_node→middle; } current_node→middle = Λ;
  return 1;
```

This code is used in section 17.

21. Inserting keys with *tst_insert*( ). This function inserts a key into the symbol table. The main idea is to follow the nodes of the tree until we hit a NULL node. Once we do, we can skip to *found_null_branch* and allocate nodes freely since we know that we will not collide with nodes for previously entered keys. If we end up going through the entire tree without hitting a NULL node, then the key is either a proper prefix of a previously entered key, or we have a duplicate key. For the proper prefix, all we have to do is tack on a terminating node. For the duplicate, if *option* is set to `TST_REPLACE` we replace the overwite the old data with *data*, otherwise, we return `TST_DUPLICATE_KEY`. A return value of `TST_ERROR` indicates a memory allocation failure while trying to grow the node free list.

5/05/1999 - Change made to *tst_insert*

When an insert has failed we return `TST_DUPLICATE_KEY`, and if we still want to do anything with the data for that key we have to make a separate call to *tst_search* to get the pointer which is wasteful. A new argument *exist_ptr* has been added to *tst_insert*. When `TST_DUPLICATE_KEY` is returned, *exist_ptr* will contain the data pointer for the existing key.

11/03/1999 - Change made to *tst_insert*

If a Λ is passed as the *exist_ptr* argument bad things could happen. Before setting this pointer with an existing item, it must be checked to see if it is Λ. In addition, previously a call to *tst_insert*( ) with the `TST_REPLACE` argument specified would NOT return the existing data for the key. Now, the existing data pointer is placed in *exist_ptr* before it is overwritten. The check for a Λ *exist_ptr* is done there as well.

```
⟨ tst_insert.c 21 ⟩ ≡
#include "tst.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
  ⟨ Declaration for tst_grow_node_free_list( ) 10 ⟩int tst_insert (const unsigned char *key, void *data,
                  struct tst *tst , int option, void **exist_ptr )
              {
                struct node *current_node;
                struct node *new_node_tree_begin = Λ;
                struct node *new_node;
                int key_index;
                int perform_loop = 1;
                ⟨ Check for NULL key 22 ⟩⟨ Check head entry to see if it is NULL 23 ⟩⟨ Traverse tree when
                    head entry is not NULL 24 ⟩⟨ Found null branch so insert rest of key 29 ⟩
              }
```

22. The first thing we need to do is check for a NULL, or zero length key, which is an error.

```
⟨ Check for NULL key 22 ⟩ ≡
  if (key ≡ Λ) return TST_NULL_KEY;
  if (key[0] ≡ 0) return TST_NULL_KEY;
```

This code is used in section 21.

23. Here, we look at the first character of *key*, and use it to index into **tst** → *head*. If the indexed node is NULL, then we know that this key is not in the tree. The entries in **tst** → *head* represent all of the possible starting points for keys. The actual node in the *head* array store the value of the *second* character of *key*, because the first character is indicated implicitly by *head*[*key*[0]] not being NULL. This is why we set the *value* member of the very first node to *key*[1].

If we the head entry is NULL, then there are several things we must perform. First, we have to allocate a node then set the *value* member to *key*[1]. Then we have to check the length of the key. If the length is 1, then we set the *middle* pointer to *data* and return `TST_OK`. Otherwise, we set *perform_loop* to 0 to disable the loop and insert the rest of the key.

⟨ Check head entry to see if it is NULL 23 ⟩ ≡

if (**tst**→*head*[(**int**) *key*[0]] ≡ Λ) { ⟨ Check **tst** → *free_list* and grow if necessary 30 ⟩**tst** → *head*[(**int**) *key*[0]] = **tst**→*free_list*; ⟨ Update free list after taking a node 31 ⟩*current_node* = **tst**→*head*[(**int**) *key*[0]];

current_node→*value* = *key*[1];
if (*key*[1] ≡ 0) {
current_node→*middle* = *data*;
return `TST_OK`;
}
else *perform_loop* = 0;
}

This code is used in section 21.

24. All we do here is traverse the tree based on characters in *key*. We handle cases where we have to take the left, middle or right branch, and the code for each is explained in their own section. The odd looking test for the left and right branches is there so we can avoid one way branching at terminating nodes. If we are at a terminating node, then we take a branch by comparing the character in *key* with 64, which is basically the 127 valid ASCII characters divided by 2. If the node is not a terminating node, then we can just compare the character in *key* with *current_node*→*value*. 11/03/1999 - Change made to *tst_insert*

⟨ Traverse tree when head entry is not NULL 24 ⟩ ≡

current_node = **tst**→*head*[(**int**) *key*[0]];

key_index = 1;
while (*perform_loop* ≡ 1) {
if (*key*[*key_index*] ≡ *current_node*→*value*) {⟨ Key is equal to node value 26 ⟩}
if (*key*[*key_index*] ≡ 0) {⟨ Key is a proper prefix of an existing entry 25 ⟩}
if (((*current_node*→*value* ≡ 0) ∧ (*key*[*key_index*] < 64)) ∨ ((*current_node*→*value* ≠ 0) ∧ (*key*[*key_index*] < *current_node*→*value*))) {⟨ Key is less than node value 27 ⟩}
else {⟨ Key is greater than node value 28 ⟩}
}

This code is used in section 21.

25. Wow, the first change since 1999. This particular change is needed to support longest match lookups. Basically, if we are going to be adding a string that is a proper prefix of an existing entry (like test in testing) then we need to store the trailing NULL of test at the point where it mismatches with the i in testing. In older versions when test is added after testing the trailing NULL of test ends up in a left branch where the longest match search doesn't see it. To make this happen, we create a single new node. We then copy the data from the mismatched node *current_node* to this new node. Then set *current_node*→*value* to 0, and calculate which branch the new node should go on.

⟨ Key is a proper prefix of an existing entry 25 ⟩ ≡

```
⟨ Check tst → free_list and grow if necessary 30 ⟩new_node = tst→free_list;
⟨ Update free list after taking a node 31 ⟩memcpy((void *) new_node, (void *) current_node, sizeof(struct
      node));
current_node→value = 0;
if (new_node→value < 64) {
   current_node→left = new_node;
   current_node→right = '\0';
}
else {
   current_node→left = '\0';
   current_node→right = new_node;
}
current_node→middle = data;
return TST_OK;
```

This code is used in section 24.

26. When the value of *key*[*key_index*] is equal to *current_node*→*value*, we must check first to see if we are looking at the NULL terminator for the string. It so, we have a duplicate key, and return `TST_DUPLICATE_KEY` unlees the *option* is set to `TST_REPLACE`, and in that case we replace overwrite the old data with *data*.

If we are not looking at the NULL terminator, then we have to check the *middle* pointer of the current node to see if it is NULL. If it is, we allocate a node, set the pointers, and break out of the loop. If the *middle* pointer is not NULL, then we increment *key_index* and set *current_node* to *current_node*→*middle*.

⟨ Key is equal to node value 26 ⟩ ≡
```
if (key[key_index] ≡ 0) {
   if (option ≡ TST_REPLACE) {
      if (exist_ptr ≠ Λ) *exist_ptr = current_node→middle;
      current_node→middle = data;
      return TST_OK;
   }
   else {
      if (exist_ptr ≠ Λ) *exist_ptr = current_node→middle;
      return TST_DUPLICATE_KEY;
   }
}
else { if (current_node→middle ≡ Λ) { ⟨ Check tst→free_list and grow if necessary 30 ⟩current_node→middle
      = tst→free_list;
⟨ Update free list after taking a node 31 ⟩new_node_tree_begin = current_node;
current_node = current_node→middle;
current_node→value = key[key_index];
break; }
else {
   current_node = current_node→middle;
   key_index++;
   continue;
}
}
```
This code is used in section 24.

27. Here we handle the case when the character *key*[*key_index*] is less than *current_node*→*value*. This means that we need to take the left branch of the tree. Before we can take this branch, we must check to see if the left branch is NULL. If it is, allocate a new node, set the values and break out of the loop. Otherwise, take the branch, and note that we do not increment *key_index* becuase we are still moving through the tree, looking for the current character.

If we do happen to allocate a new node for the left branch, we also have to check if we are at the end of *key*. If so, we set the *middle* pointer to *data* and return `TST_OK`.

⟨ Key is less than node value 27 ⟩ ≡

```
if (current_node→left ≡ Λ) { ⟨ Check tst → free_list and grow if necessary 30 ⟩current_node→left =
      tst→free_list;
  ⟨ Update free list after taking a node 31 ⟩new_node_tree_begin = current_node;
  current_node = current_node→left;
  current_node→value = key[key_index];
  if (key[key_index] ≡ 0) {
    current_node→middle = data;
    return TST_OK;
  }
  else break;
  }
  else {
    current_node = current_node→left;
    continue;
  }
```

This code is used in section 24.

28. Here we handle the case where *key*[*key_index*] is greater than *current_node*→*value*. This means that we need to take the right branch of the tree. Before we can take this branch, we must check to see if the right branch is NULL. If it is, allocate a new node, set the values and break out of the loop. Otherwise, take the branch, and note that we do not increment *key_index* becuase we are still moving through the tree, looking for the current character.

Note that in this case we are not checking to see if *key*[*key_index*] is 0, meaning that we have reached the end of *key*. This is because 0 will always be less than *current_node*→*value*, and the equality case in handled in another module.

⟨ Key is greater than node value 28 ⟩ ≡

```
if (current_node→right ≡ Λ) { ⟨ Check tst → free_list and grow if necessary 30 ⟩current_node→right =
      tst→free_list;
  ⟨ Update free list after taking a node 31 ⟩new_node_tree_begin = current_node;
  current_node = current_node→right;
  current_node→value = key[key_index];
  break; }
  else {
    current_node = current_node→right;
    continue;
  }
```

This code is used in section 24.

29. When this code is reached, we have broken out of the while loop, so we must have reached a NULL branch. We insert new nodes into the tree until the end of the key is reached, then we store *data* and return success.

03/23/200 There is potentially nasty problem with how this section of code was implemented. If we get a memory error somewhere in the middle of adding the new nodes, we return return an error but leave the nodes we were able to allocate hanging off in space, which can cause lots of problems. We solve this by saving the node where we hit a Λ link and are going to start adding the rest of the nodes for the key one after the other. If there is a failure, we put the nodes back on the free list and reset the middle pointer of the saved node to Λ.

⟨ Found null branch so insert rest of key 29 ⟩ ≡

```
do { key_index++; if (tst→free_list ≡ Λ) { if (tst_grow_node_free_list(tst) ≠ 1) {
      current_node = new_node_tree_begin→middle;
  while (current_node→middle ≠ Λ) current_node = current_node→middle;
  current_node→middle = tst→free_list;
  tst→free_list = new_node_tree_begin→middle;
  new_node_tree_begin→middle = Λ;
  return TST_ERROR; } } ⟨ Check tst → free_list and grow if necessary 30 ⟩ current_node→middle =
      tst→free_list;
  ⟨ Update free list after taking a node 31 ⟩ current_node = current_node→middle;
  current_node→value = key[key_index]; }
  while (key[key_index] ≠ 0) ;
  current_node→middle = data;
  return TST_OK;
```

This code is used in section 21.

30. This is code that is used throughout this function that checks to see if **tst** → *free_list* is empty. If it is, then we call *tst_grow_node_free_list*( ).

⟨ Check **tst** → *free_list* and grow if necessary 30 ⟩ ≡

```
if (tst→free_list ≡ Λ)
{
  if (tst_grow_node_free_list(tst) ≠ 1) return TST_ERROR;
}
```

This code is used in sections 23, 25, 26, 27, 28, and 29.

31. This is only one line of code, but it is included here as a module to make it stand out more, so hopefully it will not be forgotten. This code updates **tst** → *free_list* to the next node in the free list. This *must* be called after a node is taken off of the free list.

⟨ Update free list after taking a node 31 ⟩ ≡

```
tst → free_list = tst→free_list→middle;
```

This code is used in sections 23, 25, 26, 27, 28, and 29.

32. Searching for keys with *tst_search*( ).

⟨ `tst_search.c` 32 ⟩ ≡

```
#include "tst.h"
#include <stdio.h>
#include <stdlib.h>
  void * tst_search (const unsigned char *key, struct tst *tst , int option, unsigned int *match_len
          ) { struct node *current_node;
        struct node *longest_match = Λ;
        unsigned int longest_match_len = 0;
        int key_index; ⟨ Fail if key is NULL 33 ⟩⟨ Return NULL if head is NULL 34 ⟩
        if (match_len) *match_len = 0;
        ⟨ Initialize current_node, key_index, start search loop and return NULL on failure 35 ⟩}
```

33. Here we check for the NULL key, which is not allowed.

⟨ Fail if *key* is NULL 33 ⟩ ≡

```
  if (key[0] ≡ 0) return Λ;
```

This code is used in section 32.

34. Here we simply check the head node to see if it is NULL. If it is, then we know that the key cannot exist in the tree so we return NULL to indicate failure.

⟨ Return NULL if head is NULL 34 ⟩ ≡

```
  if (tst→head[(int) key[0]] ≡ Λ) return Λ ;
```

This code is used in section 32.

35. Here we set *current_node* node to the head node and set our index to 1. The loop runs until we hit a NULL node, in which case we return NULL to indicate failure, otherwise, we return the data stored in the terminating node.

⟨ Initialize *current_node*, *key_index*, start search loop and return NULL on failure 35 ⟩ ≡

```
current_node = tst→head[(int) key[0]];
key_index = 1;
while (current_node ≠ Λ) {
  if (key[key_index] ≡ current_node→value) {
    if (current_node→value ≡ 0) {
      if (match_len) *match_len = key_index;
      return current_node→middle;
    }
    else {
      current_node = current_node→middle;
      key_index++;
      continue;
    }
  }
  else {
    if (current_node→value ≡ 0) {
      if (option & TST_SUBSTRING_MATCH) {
        longest_match = current_node→middle;
        longest_match_len = key_index;
      }
      if (key[key_index] < 64) {
        current_node = current_node→left;
        continue;
      }
      else {
        current_node = current_node→right;
        continue;
      }
    }
    else {
      if (key[key_index] < current_node→value) {
        current_node = current_node→left;
        continue;
      }
      else {
        current_node = current_node→right;
        continue;
      }
    }
  }
}
if (match_len) *match_len = longest_match_len;
return longest_match;
```

This code is used in section 32.

36. Deleting keys with *tst_delete*( ). This is the most complex function of the package. If the *key* is found, the *data* associated with the key is returned, otherwise the return value is NULL. The basic task of this function is to find something I call the *last_branch*. This node is the last node in the path for a key which has non-NULL children, or is a node branched off of another. We also have to store the parent of this node, because we have to NULL the branch that leads to *last_branch*.

t   t is implied by the existence of **tst** → *head*[*key*[0]]

e   this is the actual node stored in **tst** → *head*[*key*[0]]

s

t

0   *last_branch* when deleting "test"

    i   *last_branch* when deleting "testing"

    n

    g

    0

```
⟨ tst_delete.c 36 ⟩ ≡
#include "tst.h"
#include <stdio.h>
#include <stdlib.h>
  void * tst_delete (const unsigned char *key, struct tst *tst ) { struct node *current_node;
      struct node *current_node_parent;
      struct node *last_branch;
      struct node *last_branch_parent;
      struct node *next_node;
      struct node *last_branch_replacement;
      struct node *last_branch_dangling_child;
      int key_index; ⟨ NULL keys and head nodes return failure 37 ⟩⟨ Find last branch 38 ⟩
      if (current_node ≡ Λ) return Λ;
      ⟨ Handle key deletion 40 ⟩}
```

37. Here we check for the NULL key, which is not allowed.

We also check the head node to see if it is NULL. If it is, then we know that the key cannot exist in the tree so we return NULL to indicate failure.

```
⟨ NULL keys and head nodes return failure 37 ⟩ ≡
  if (key[0] ≡ 0) return Λ;
  if (tst→head[(int) key[0]] ≡ Λ) return Λ ;
```

This code is used in section 36.

38. Here is where we look for *last_branch*.

⟨ Find last branch 38 ⟩ ≡

```
last_branch = Λ;
last_branch_parent = Λ; current_node = tst→head[(int) key[0]];
current_node_parent = Λ;
key_index = 1; while (current_node ≠ Λ) { if (key[key_index] ≡ current_node→value) { ⟨ Check node for
      branches 39 ⟩
  if (key[key_index] ≡ 0) break;
  else {
    current_node_parent = current_node;
    current_node = current_node→middle;
    key_index++;
    continue;
  }
}
else
  if (((current_node→value ≡ 0) ∧ (key[key_index] < 64)) ∨ ((current_node→value ≠ 0) ∧ (key[key_index] <
        current_node→value))) {
    last_branch_parent = current_node;
    current_node_parent = current_node;
    current_node = current_node→left;
    last_branch = current_node;
    continue;
  }
  else {
    last_branch_parent = current_node;
    current_node_parent = current_node;
    current_node = current_node→right;
    last_branch = current_node;
    continue;
  }
}
```

This code is used in section 36.

39. Here we check whether one or both of the children of *current_node* are not NULL, which means that *key* up to this point is a proper prefix of another key in the tree, so we can delete this node, but we have to balance the tree first. We therefore set *last_branch* to *current_node* and *last_branch_parent* to *current_node_parent*.

⟨ Check node for branches 39 ⟩ ≡

```
if ((current_node→left ≠ Λ) ∨ (current_node→right ≠ Λ)) {
  last_branch = current_node;
  last_branch_parent = current_node_parent;
}
```

This code is used in section 38.

40.

⟨ Handle key deletion 40 ⟩ ≡
  if (*last_branch* ≡ Λ)
  {⟨ *last_branch* is NULL so we can remove the whole key and set the head to NULL 41 ⟩}
  else if ((*last_branch*→*left* ≡ Λ) ∧ (*last_branch*→*right* ≡ Λ))
  {⟨ Both children are NULL so we can delete from *last_branch* 42 ⟩}
  else {⟨ Determine values for *last_branch_replacement* and *last_branch_dangling_child* 43 ⟩⟨ Deal with case where *last_branch_parent* is NULL 44 ⟩⟨ Move *last_branch_dangling_child* to new slot in left subtree of *last_branch_replacement* 45 ⟩}
  ⟨ Free nodes from *next_node* onward and return data 46 ⟩

This code is used in section 36.

41. When *last_branch* is NULL, we set *next_node* to the head node, NULL the head, then fall through the statements so we can remove the entire key.

⟨ *last_branch* is NULL so we can remove the whole key and set the head to NULL 41 ⟩ ≡
  next_node = **tst**→*head*[(**int**) *key*[0]];
  tst→*head*[(**int**) *key*[0]] = Λ;

This code is used in section 40.

42. When both children of *last_branch* are NULL, we can safely remove all nodes from that point on without having to balance any other nodes. All we have to do is set the path out of *last_branch_parent* to NULL.

⟨ Both children are NULL so we can delete from *last_branch* 42 ⟩ ≡
  if (*last_branch_parent*→*left* ≡ *last_branch*) *last_branch_parent*→*left* = Λ;
  else *last_branch_parent*→*right* = Λ;
  next_node = *last_branch*;

This code is used in section 40.

43. At this point we know that *last_branch* has one or more children, so we have to move nodes around before we can start deleting them. Since the node at *last_branch* is going to be removed, we have the variable *last_branch_replacement*. When both children are valid, we arbitrarily set this to the right child, otherwise, we set it to the child that is not NULL. Also in the case where both children are valid, we use the variable *last_branch_dangling_child* to store the extra child.

⟨ Determine values for *last_branch_replacement* and *last_branch_dangling_child* 43 ⟩ ≡
  if ((*last_branch*→*left* ≠ Λ) ∧ (*last_branch*→*right* ≠ Λ)) {
    last_branch_replacement = *last_branch*→*right*;
    last_branch_dangling_child = *last_branch*→*left*;
  }
  else if (*last_branch*→*right* ≠ Λ) {
    last_branch_replacement = *last_branch*→*right*;
    last_branch_dangling_child = Λ;
  }
  else {
    last_branch_replacement = *last_branch*→*left*;
    last_branch_dangling_child = Λ;
  }

This code is used in section 40.

44. If *last_branch_parent* is NULL, then wee have a situation where *last_branch* is actually equal to **tst**→*head*[*key*[0]], or in other words, it is the head node and we need to handle this in a special way. We do this by setting the head node to *last_branch_replacement*. On the other hand, if *last_branch_parent* is not NULL, then we need to find which path was taken out of *last_branch_parent* to *last_branch*. We set this path, or rather pointer, to *last_branch_replacement*.

⟨ Deal with case where *last_branch_parent* is NULL 44 ⟩ ≡

```
if (last_branch_parent ≡ Λ) tst→head[(int) key[0]] = last_branch_replacement;
  else {
    if (last_branch_parent→left ≡ last_branch) last_branch_parent→left = last_branch_replacement;
    else if (last_branch_parent→right ≡ last_branch) last_branch_parent→right = last_branch_replacement;
    else last_branch_parent→middle = last_branch_replacement;
  }
```

This code is used in section 40.

45. At this point we have replaced *last_branch* with *last_branch_replacement* in the tree, and now we have to handle the case where both children of *last_branch* were valid. If *last_branch_dangling_child* is NULL, then we have nothing to do. Otherwise, we need to find an open slot in the left subtree of *last_branch_replacement* to put *last_branch_dangling_child*.

⟨ Move *last_branch_dangling_child* to new slot in left subtree of *last_branch_replacement* 45 ⟩ ≡

```
if (last_branch_dangling_child ≠ Λ) {
  current_node = last_branch_replacement;
  while (current_node→left ≠ Λ) current_node = current_node→left;
  current_node→left = last_branch_dangling_child;
}
next_node = last_branch;
```

This code is used in section 40.

46. This puts the nodes back on the free list and returns the data associated with a key. To use, set *next_node* to the value of *last_branch* or whichever node the deletion needs to start from.

⟨ Free nodes from *next_node* onward and return data 46 ⟩ ≡

```
do {
  current_node = next_node;
  next_node = current_node→middle;
  ⟨ Return node to free list 47 ⟩
} while (current_node→value ≠ 0);
return next_node;
```

This code is used in section 40.

47. This code returns a node to the free list and makes sure that the child pointers are set to NULL.

⟨ Return node to free list 47 ⟩ ≡

```
current_node→left = Λ;
current_node→right = Λ; current_node→middle = tst→free_list;
tst→free_list = current_node;
```

This code is used in section 46.

48. Freeing all node space with *tst_cleanup*( ).

```
⟨ tst_cleanup.c 48 ⟩ ≡
#include "tst.h"
#include <stdio.h>
#include <stdlib.h>
  void tst_cleanup ( struct tst *tst ) { struct node_lines *current_line;
      struct node_lines *next_line; next_line = tst→node_lines;
      do {
        current_line = next_line;
        next_line = current_line→next;
        free(current_line→node_line);
        free(current_line);
      } while (next_line ≠ Λ);
      free(tst); }
```

49. Header file.

⟨ tst.h 49 ⟩ ≡
⟨ Node structure 1 ⟩⟨ TST structure 2 ⟩⟨ Node lines structure 3 ⟩⟨ TST constants 4 ⟩⟨ Declaration for *tst_init*( ) 6 ⟩⟨ Declaration for *tst_insert*( ) 7 ⟩⟨ Declaration for *tst_search*( ) 8 ⟩⟨ Declaration for *tst_delete*( ) 9 ⟩⟨ Declaration for *tst_cleanup*( ) 11 ⟩

calloc: 13, 14, 15, 19.
current_line: 48.
current_node: 12, 16, 17, 20, 21, 23, 24, 25, 26, 27, 28, 29, 32, 35, 36, 38, 39, 45, 46, 47.
current_node_parent: 36, 38, 39.
data: 7, 8, 9, 21, 23, 25, 26, 27, 29, 36.
exist_ptr: 7, 21, 26.
found_null_branch: 21.
free: 14, 15, 19, 48.
free_list: 2, 3, 16, 20, 23, 25, 26, 27, 28, 29, 30, 31, 47.
head: 2, 23, 24, 34, 35, 36, 37, 38, 41, 44.
i: 12, 17.
key: 7, 8, 9, 21, 22, 23, 24, 26, 27, 28, 29, 32, 33, 34, 35, 36, 37, 38, 39, 41, 44.
key_index: 21, 24, 26, 27, 28, 29, 32, 35, 36, 38.
last_branch: 36, 38, 39, 40, 41, 42, 43, 44, 45, 46.
last_branch_dangling_child: 36, 43, 45.
last_branch_parent: 36, 38, 39, 42, 44.
last_branch_replacement: 36, 43, 44, 45.
left: 1, 25, 27, 35, 38, 39, 40, 42, 43, 44, 45, 47.
longest_match: 32, 35.
longest_match_len: 32, 35.
malloc: 18.
match_len: 8, 32, 35.
memcpy: 25.
middle: 1, 2, 16, 20, 23, 25, 26, 27, 29, 31, 35, 38, 44, 46, 47.
new_line: 17, 18, 19.
new_node: 21, 25.
new_node_tree_begin: 21, 26, 27, 28, 29.
next: 3, 15, 19, 48.
next_line: 48.
next_node: 36, 41, 42, 45, 46.
node: 1, 2, 3, 12, 15, 17, 19, 21, 25, 32, 36.
node_line: 3, 15, 16, 19, 20, 48.
node_line_width: 2, 3, 6, 15, 19, 20.
node_lines: 2, 3, 14, 15, 16, 17, 18, 19, 20, 48.
option: 7, 8, 21, 26, 32, 35.
perform_loop: 21, 23, 24.
right: 1, 25, 28, 35, 38, 39, 40, 42, 43, 44, 47.
tst: 2, 3, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 34, 35, 36, 37, 38, 41, 44, 47, 48.
tst_cleanup: 2, 11, 48.
tst_constants: 4.
tst_delete: 9, 36.
`TST_DUPLICATE_KEY`: 4, 7, 21, 26.
`TST_ERROR`: 4, 7, 10, 18, 19, 21, 29, 30.
tst_grow_node_free_list: 5, 6, 10, 17, 29, 30.
tst_init: 6, 12.
tst_insert: 5, 7, 10, 21, 24.
`TST_NULL_KEY`: 4, 7, 22.
`TST_OK`: 4, 7, 23, 25, 26, 27, 29.
`TST_REPLACE`: 4, 7, 21, 26.
tst_search: 7, 8, 21, 32.
`TST_SUBSTRING_MATCH`: 4, 8, 35.
value: 1, 23, 24, 25, 26, 27, 28, 29, 35, 38, 46.
width: 12, 15, 16.

⟨ Add the nodes from *node_line* to **tst** → *free_list* 20 ⟩ Used in section 17.
⟨ Allocate the *node_line* member of **tst** → **node_lines** → *next* 19 ⟩ Used in section 17.
⟨ Allocate tst structure 13 ⟩ Used in section 12.
⟨ Allocate **node_lines** member 14 ⟩ Used in section 12.
⟨ Allocate **tst** → **node_lines** → *next* 18 ⟩ Used in section 17.
⟨ Both children are NULL so we can delete from *last_branch* 42 ⟩ Used in section 40.
⟨ Build free list from just allocated *node_line* 16 ⟩ Used in section 12.
⟨ Check for NULL key 22 ⟩ Used in section 21.
⟨ Check head entry to see if it is NULL 23 ⟩ Used in section 21.
⟨ Check node for branches 39 ⟩ Used in section 38.
⟨ Check **tst** → *free_list* and grow if necessary 30 ⟩ Used in sections 23, 25, 26, 27, 28, and 29.
⟨ Deal with case where *last_branch_parent* is NULL 44 ⟩ Used in section 40.
⟨ Declaration for *tst_cleanup*( ) 11 ⟩ Used in section 49.
⟨ Declaration for *tst_delete*( ) 9 ⟩ Used in section 49.
⟨ Declaration for *tst_grow_node_free_list*( ) 10 ⟩ Used in section 21.
⟨ Declaration for *tst_init*( ) 6 ⟩ Used in section 49.
⟨ Declaration for *tst_insert*( ) 7 ⟩ Used in section 49.
⟨ Declaration for *tst_search*( ) 8 ⟩ Used in section 49.
⟨ Determine values for *last_branch_replacement* and *last_branch_dangling_child* 43 ⟩ Used in section 40.
⟨ Fail if *key* is NULL 33 ⟩ Used in section 32.
⟨ Find last branch 38 ⟩ Used in section 36.
⟨ Found null branch so insert rest of key 29 ⟩ Used in section 21.
⟨ Free nodes from *next_node* onward and return data 46 ⟩ Used in section 40.
⟨ Handle key deletion 40 ⟩ Used in section 36.
⟨ Initialize *current_node*, *key_index*, start search loop and return NULL on failure 35 ⟩ Used in section 32.
⟨ Key is a proper prefix of an existing entry 25 ⟩ Used in section 24.
⟨ Key is equal to node value 26 ⟩ Used in section 24.
⟨ Key is greater than node value 28 ⟩ Used in section 24.
⟨ Key is less than node value 27 ⟩ Used in section 24.
⟨ Move *last_branch_dangling_child* to new slot in left subtree of *last_branch_replacement* 45 ⟩ Used in section 40.
⟨ NULL keys and head nodes return failure 37 ⟩ Used in section 36.
⟨ Node lines structure 3 ⟩ Used in section 49.
⟨ Node structure 1 ⟩ Used in section 49.
⟨ Return NULL if head is NULL 34 ⟩ Used in section 32.
⟨ Return node to free list 47 ⟩ Used in section 46.
⟨ Set *node_line_width* and allocate first chunk of nodes 15 ⟩ Used in section 12.
⟨ TST constants 4 ⟩ Used in section 49.
⟨ TST structure 2 ⟩ Used in section 49.
⟨ Traverse tree when head entry is not NULL 24 ⟩ Used in section 21.
⟨ Update free list after taking a node 31 ⟩ Used in sections 23, 25, 26, 27, 28, and 29.
⟨ `tst.h` 49 ⟩
⟨ `tst_cleanup.c` 48 ⟩
⟨ `tst_delete.c` 36 ⟩
⟨ `tst_grow_node_free_list.c` 17 ⟩
⟨ `tst_init.c` 12 ⟩
⟨ `tst_insert.c` 21 ⟩
⟨ `tst_search.c` 32 ⟩
⟨ *last_branch* is NULL so we can remove the whole key and set the head to NULL 41 ⟩ Used in section 40.

TST

	Section	Page
Structures	1	2
Constants	4	3
Functions	5	4
Initialization with *tst_init*( )	12	5
Growing the free list with *tst_grow_node_free_list*( )	17	6
Inserting keys with *tst_insert*( )	21	7
Searching for keys with *tst_search*( )	32	13
Deleting keys with *tst_delete*( )	36	15
Freeing all node space with *tst_cleanup*( )	48	19
Header file	49	20