

1. Trie Data Structure.

A program to create and manipulate trie data structures.

© 2008, Carlos Oliveira

2. A trie is used to store character strings in encoded form, such that each character is a node of a tree data structure, and a word is formed by concatenating the characters from the root of the trie to one of its terminals. One of the advantages of using tries is that insertion and search can be done in time linear on the size of the string.

The main restriction of this implementation is that it works only for ASCII characters the range [A-Z]. I am doing this mainly to test how well the algorithm performs. Real implementations should be able to handle most character sets, and therefore they need more space to store child nodes at each trie node.

```
<trie.cpp 2> ≡
  <trie includes 3>
  <trie functions 4>
```

3.

```
<trie includes 3> ≡
#include "trie.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#define New(t) ( t * ) malloc(sizeof (t))
#define NewV(t,s) ( t * ) malloc(sizeof (t) * (s))
```

This code is used in section 2.

4. A simple utility function that aborts the program with an error message to stderr.

```
<trie functions 4> ≡
void abort(const char *msg)
{
    fprintf(stderr, msg);
    exit(1);
}
```

See also sections 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 16, 17, 18, and 19.

This code is used in section 2.

5. A function that checks if a character can be inserted as a child of a trie node. All characters stored are in uppercase. The function returns an uppercase character, or 0 if the character is not between A and Z.

```
<trie functions 4> +≡
char check_char(char c)
{
    if ('a' ≤ c ∧ c ≤ 'z') c -= 32;
    if ('A' ≤ c ∧ c ≤ 'Z') return c;
    return 0;
}
```

6. The node initialization function just returns a new trie node, initialized to the character *c*. Everything else in the node is NULL.

```

⟨ trie functions 4 ⟩ +≡
  static TrieNode*trie_new_node(char c)
  {
    TrieNode *t = New(TrieNode);
    if (!t) return 0;
    t->data = c;
    t->children = Λ;
    t->n_child = 0;
    t->size = 0;
    t->is_terminal = 0;
    return t;
  }

```

7. A function that creates a new trie data structure.

```

⟨ trie functions 4 ⟩ +≡
  Trie * trie_new()
  {
    Trie *t = New(Trie);
    if (!t) return 0;
    t->n_words = 0;
    t->root = trie_new_node(0); /* alloc root element */
    if (!t->root) goto e1;
    return t;
e1: free(t);
    return 0;
  }

```

8. Allocates a new array of pointers to child trie node, and initialize it to NULLs.

```

⟨ trie functions 4 ⟩ +≡
  static TrieNode**new_child_array(int size){ TrieNode **nodes = NewV ( TrieNode *, size );
    if (!nodes) return 0;
    int i;
    for (i = 0; i < size; ++i) nodes[i] = Λ;
    return nodes; }

```

9. searches an array of children of the first type (with 8 characters). Returns a trie node if the character can be found, NULL otherwise.

```

⟨ trie functions 4 ⟩ +≡
  static TrieNode*in_children8(TrieNode *t, char c)
  {
    int i, n = t->n_child;
    for (i = 0; i < n; ++i)
      if (t->children[i]->data == c) return t->children[i];
    return 0;
  }

```

10. adds a new node to a list of children of the first type.

```

⟨ trie functions 4 ⟩ +=
  static TrieNode*add_children8 (TrieNode * t, char c)
  {
    assert(t->n_child < 8);
    return t->children[t->n_child++] = trie_new_node(c);
  }

```

11. implementation for search/add nodes to children list of second type

```

⟨ trie functions 4 ⟩ +=
  static TrieNode*search_add_children26 (TrieNode * t, char c, int insert)
  {
    assert(t->size == 26);
    int pos = c - 'A';
    if (t->children[pos]) return t->children[pos];
    if (!insert) return 0;
    TrieNode * node = t->children[pos] = trie_new_node(c);
    if (!node) abort("cannot add node in children26");
    t->n_child++;
    return node;
  }

```

12. wrappers for function above

```

⟨ trie functions 4 ⟩ +=
  static TrieNode*in_children26 (TrieNode * t, char c)
  {
    return search_add_children26 (t, c, 0);
  }
  static TrieNode*add_to_children26 (TrieNode * t, char c)
  {
    return search_add_children26 (t, c, 1);
  }

```

13. add given nodes (usually from a list of the first type) into a list of the second type

```

⟨ trie functions 4 ⟩ +=
  void add_existing_nodes26 (TrieNode * t, TrieNode **nodes, int n)
  {
    assert(t->size == 26);
    int i;
    for (i = 0; i < n; ++i) {
      int pos = nodes[i]-data - 'A';
      assert(!t->children[pos]); /* this position must be NULL */
      t->children[pos] = nodes[i];
    }
  }

```

14. A function that checks if a character c exists as a child of the given trie node. If $insert$ is true, the value should be inserted as a child if it is not present. $modified$ is an output value, and will be true if there was any insertion on the child list.

⟨ trie functions 4 ⟩ +≡

```
static TrieNode*get_or_insert_child_for_char(TrieNode * t, char c, int insert, int *modified, int
    is_terminal){ assert('A' ≤ c ∧ c ≤ 'Z');
    TrieNode * node; ⟨ check cases of child list 15 ⟩
```

15. To reduce memory consumption, each node can have a child list of one of two types: 8-position or 26-position arrays. The first type stores at most 8 characters. They are unordered, so any search needs to look at all characters currently stored (which on average is 4). The second type of list has an array for all 26 possible characters. The list is in alphabetical order, so it is very easy to find and insert elements

⟨ check cases of child list 15 ⟩ ≡

```
if (t->size ≡ 0) {
    if (¬insert) return 0; /* add memory, return position */
    t->children = new_child_array(8);
    if (¬t->children) abort("cannot_add_child_array");
    t->size = 8;
    node = add_children8(t, c);
}
else if (t->size ≡ 8) { /* check if it is there */
    node = in_children8(t, c);
    if (node) return node; /* if not, add */
    if (¬insert) return 0;
    if (t->n_child < 8) node = add_children8(t, c);
    else { /* we need more space */
        TrieNode **current = t->children;
        t->children = new_child_array(26);
        if (¬t->children) abort("cannot_add_child_array");
        t->size = 26;
        add_existing_nodes26(t, current, 8);
        node = add_to_children26(t, c);
    }
}
else if (t->size ≡ 26) {
    node = in_children26(t, c);
    if (node) return node;
    if (¬insert) return 0;
    node = add_to_children26(t, c);
}
else {
    abort("wrong_size_of_child_list");
} /* we come here only in case the child list was modified */
if (¬node) abort("cannot_add_node");
if (is_terminal) node->is_terminal = 1;
*modified = 1;
return node; }
```

This code is used in section 14.

16. we visit the trie recursively, because word lengths are usually small

⟨trie functions 4⟩ +≡

```

static int trie_visit_rec(TrieNode *t, char *word, int len, int insert, int *modified)
{
    if (len ≡ 0) { /* end of recursion */
        if (*modified) return 2;
        return 1;
    }
    char c = check_char(word[0]);
    TrieNode *child = get_or_insert_child_for_char(t, c, insert, modified, len ≡ 1);
    if (¬child) {
        if (insert) abort("cannot insert word");
        else return 0;
    }
    return trie_visit_rec(child, word + 1, len - 1, insert, modified);
}

```

17. searches or inserts a word into a trie. If insert is true, the word will be inserted if it is not already in the tree. Returns true if the operation is successful.

⟨trie functions 4⟩ +≡

```

int trie_visit(Trie *t, char *word, int insert)
{
    int modified = 0;
    int res = trie_visit_rec(t→root, word, strlen(word), insert, &modified);
    if (insert ∧ res ≡ 2) t→n_words++;
    return res;
}

```

18. Print all words contained in a trie. the algorithm is recursive. If, at some iteration, the *is_terminal* flag is set, then print the current word. For each of the children of the current node, add the character they represent to the word, and call the function recursively.

```

⟨ trie functions 4 ⟩ +≡
static int trie_print_rec(TrieNode * t, char *word, int pos, int max_size)
{
    if (pos == max_size) { /* check buffer overrun */
        word[pos] = '\0';
        printf("word truncated: %s\n", word);
        return 0;
    }
    if (t->is_terminal) { /* print current word */
        word[pos] = '\0';
        printf("%s\n", word);
    }
    int i;
    for (i = 0; i < t->size; ++i) { /* print all children */
        TrieNode * node = t->children[i];
        if (node) {
            word[pos] = node->data;
            trie_print_rec(node, word, pos + 1, max_size);
        }
    }
    return 1;
}
#define MAX_SIZE 64

```

19. Function that calls the recursive implementation.

```

⟨ trie functions 4 ⟩ +≡
int trie_print(Trie * t)
{
    printf("printing words\n");
    char word[MAX_SIZE + 1];
    trie_print_rec(t->root, word, 0, MAX_SIZE);
    return 1;
}

```

20.

```

⟨ main.cpp 20 ⟩ ≡
⟨ main headers 21 ⟩
⟨ main functions 23 ⟩

```

21.

```

⟨ main headers 21 ⟩ ≡
#include "trie.h"
#include <stdio.h>
#include <string.h>

```

This code is used in section 20.

22. String Operations.

We need a few functions to process a file and return all words found in that file. We want to strip characters that are not handled by our trie implementation.

23. returns true if the string *w* is a word composed only of the characters from A-Z

```

⟨main functions 23⟩ ≡
int is_word(char *w, int len)
{
    int i;
    for (i = 0; i < len; ++i)
        if (¬check_char(w[i])) return 0;
    return i;
}

```

See also sections 24, 25, 26, 27, 28, and 29.

This code is used in section 20.

24. true if the character is a punctuation character

```

⟨main functions 23⟩ +≡
int is_punctuation(char c)
{
    switch (c) {
        case '.,': case ',': case ';': case ':': return 1;
    }
    return 0;
}

```

25. true if the character is a quote character

```

⟨main functions 23⟩ +≡
int is_quote(char c)
{
    switch (c) {
        case '\': case '"': case '\': return 1;
    }
    return 0;
}

```

26.

```

⟨main functions 23⟩ +≡
#define MAX_WSIZE 64

```

27. adjust word to remove quotes and/or punctuation marks

```

⟨main functions 23⟩ +≡
char *adjust_word(char *w)
{
    /* try to remove at most two quotes */
    if (is_quote(*w)) w++;
    if (is_quote(*w)) w++; /* try to remove at most three quotes/punctuation marks */

    int len = strlen(w);
    int i;

    for (i = 0; i < 3 ^ i < len; ++i) {
        int last = len - i - 1;
        if (is_quote(w[last]) ^ is_punctuation(w[last])) w[last] = 0;
    }
    return w;
}

```

28. reads all words in the given file name, and inserts them on the trie.

```

⟨main functions 23⟩ +≡
int proc_file(Trie *t, char *file)
{
    FILE *f = fopen(file, "r");
    if (!f) abort("error opening file");
    char w[MAX_WSIZE + 1]; /* write format string */
    char format[6];
    sprintf(format, "%%%ds", MAX_WSIZE); /* insert each word found in file */
    for ( ; ; ) {
        if (EOF == fscanf(f, format, w)) break;
        char *word = adjust_word(w);
        if (is_word(word, strlen(word))) trie_visit(t, word, 1);
    }
    return 1;
}

```

29.

```

⟨main functions 23⟩ +≡
int main(int argc, char **argv)
{
    Trie *t = trie_new();
    if (!t) return 1;
    if (argc < 2 ^ argc == 3 ^ strcmp(argv[2], "-p")) {
        printf("usage: \n%s inputFile [-p] \n", argv[0]);
        return 1;
    }
    proc_file(t, argv[1]);
    printf("number of words: %d\n", t->n_words);
    if (argc == 3) trie_print(t);
    return 0;
}

```

30. Header File. This is the header file for the trie module.

```
<trie.h 30> ≡
#ifndef _TRIE_H_
#define _TRIE_H_
    struct TrieNode_ {
        char data;
        struct TrieNode_ **children;
        int n_child;
        int size;
        int is_terminal;
    };
    typedef struct TrieNode_ TrieNode;
    struct Trie_ {
        int n_words;
        TrieNode *root;
    };
    typedef struct Trie_ Trie;
    Trie *trie_new();
    int trie_visit(Trie *t, char *word, int insert);
    int trie_print(Trie *t);
    char check_char(char c);
    void abort(const char *msg);
#endif
```

31. Index.

_TRIE_H_: 30.
 abort: 4, 11, 15, 16, 28, 30.
 add_children8: 10, 15.
 add_existing_nodes26: 13, 15.
 add_to_children26: 12, 15.
 adjust_word: 27, 28.
 argc: 29.
 argv: 29.
 assert: 10, 11, 13, 14.
 c: 5, 6, 9, 10, 11, 12, 14, 16, 24, 25, 30.
 check_char: 5, 16, 23, 30.
 child: 16.
 children: 6, 9, 10, 11, 13, 15, 18, 30.
 current: 15.
 data: 6, 9, 13, 18, 30.
 EOF: 28.
 exit: 4.
 e1: 7.
 f: 28.
 file: 28.
 fopen: 28.
 format: 28.
 fprintf: 4.
 free: 7.
 fscanf: 28.
 get_or_insert_child_for_char: 14, 16.
 i: 8, 9, 13, 18, 23, 27.
 in_children26: 12, 15.
 in_children8: 9, 15.
 insert: 11, 14, 15, 16, 17, 30.
 is_punctuation: 24, 27.
 is_quote: 25, 27.
 is_terminal: 6, 14, 15, 18, 30.
 is_word: 23, 28.
 last: 27.
 len: 16, 23, 27.
 main: 29.
 malloc: 3.
 MAX_SIZE: 18, 19.
 max_size: 18.
 MAX_WSIZE: 26, 28.
 modified: 14, 15, 16, 17.
 msg: 4, 30.
 n: 9, 13.
 n_child: 6, 9, 10, 11, 15, 30.
 n_words: 7, 17, 29, 30.
 New: 3, 6, 7.
 new_child_array: 8, 15.
 NewV: 3, 8.
 node: 11, 14, 15, 18.
 nodes: 8, 13.
 pos: 11, 13, 18.
 printf: 18, 19, 29.
 proc_file: 28, 29.
 res: 17.
 root: 7, 17, 19, 30.
 search_add_children26: 11, 12.
 size: 6, 8, 11, 13, 15, 18, 30.
 sprintf: 28.
 stderr: 4.
 strcmp: 29.
 strlen: 17, 27, 28.
 t: 30.
Trie: 7, 17, 19, 28, 29, 30.
Trie_: 30.
 trie_new: 7, 29, 30.
 trie_new_node: 6, 7, 10, 11.
 trie_print: 19, 29, 30.
 trie_print_rec: 18, 19.
 trie_visit: 17, 28, 30.
 trie_visit_rec: 16, 17.
TrieNode: 6, 8, 9, 10, 11, 12, 13, 14, 15, 16, 18, 30.
TrieNode_: 30.
 w: 23, 27, 28.
 word: 16, 17, 18, 19, 28, 30.

⟨ check cases of child list 15 ⟩ Used in section 14.
⟨ main functions 23, 24, 25, 26, 27, 28, 29 ⟩ Used in section 20.
⟨ main headers 21 ⟩ Used in section 20.
⟨ main.cpp 20 ⟩
⟨ trie functions 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 16, 17, 18, 19 ⟩ Used in section 2.
⟨ trie includes 3 ⟩ Used in section 2.
⟨ trie.cpp 2 ⟩
⟨ trie.h 30 ⟩

TRIECODE

	Section	Page
Trie Data Structure	1	1
String Operations	22	7
Header File	30	9
Index	31	10