Tries

Computing Lab

http://www.isical.ac.in/~dfslab

Motivation

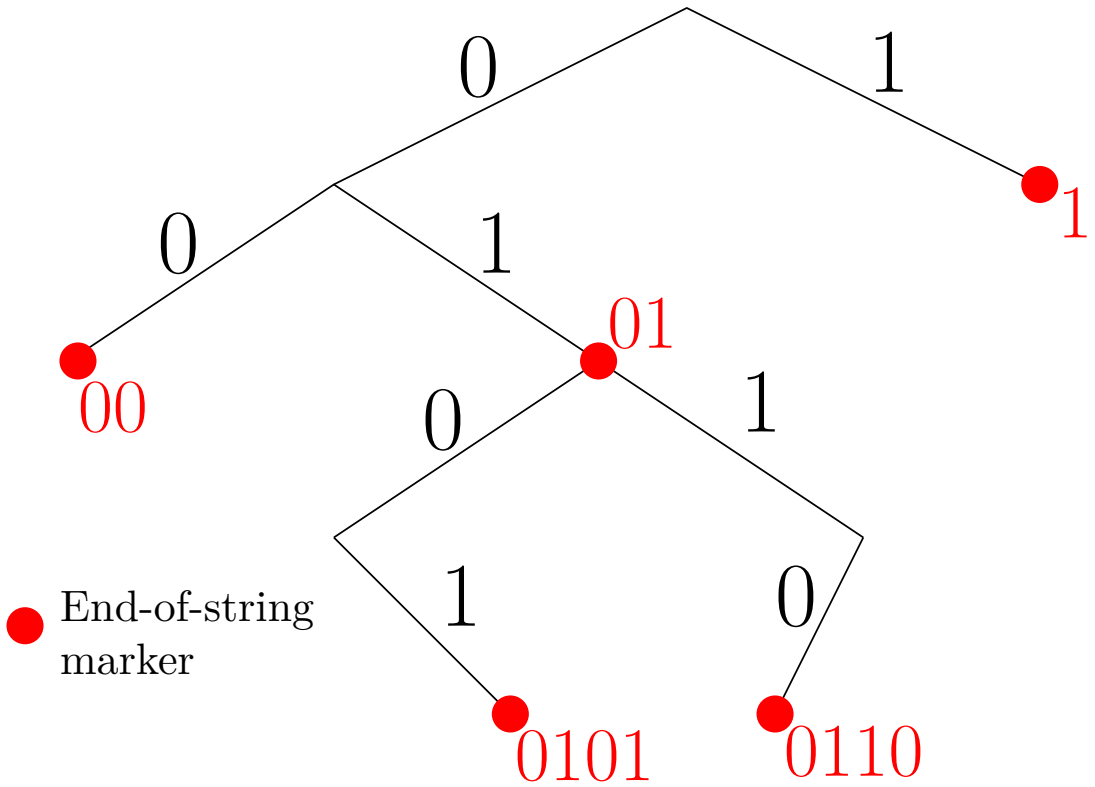
Problem

Determine whether there are any duplicates in a given list of N binary strings (i.e., strings consisting of 0s and 1s only). Note that the strings are too long to be stored as integers.

Motivation

Problem

Determine whether there are any duplicates in a given list of N binary strings (i.e., strings consisting of 0s and 1s only). Note that the strings are too long to be stored as integers.

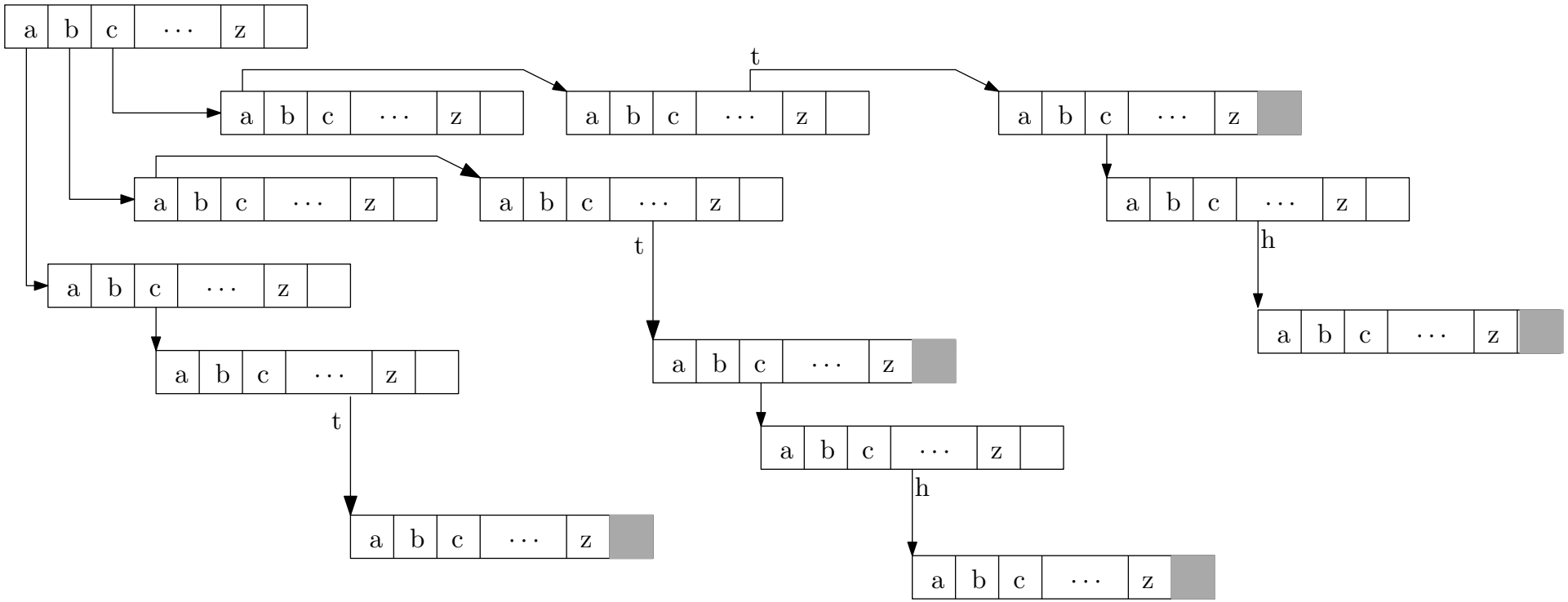


Motivation (contd.)

Problem

Repeat the above problem, but assume that the strings consist of 0s, 1s and 2s only.

Trie



Tries – implementation I

- Trie $\equiv$ array of trie nodes
- Each node is an array
  - indexed by symbols
  - array values correspond to indices of child nodes

	a	b	...	y	z	flag
0						
1						
⋮			⋮			

Implementation

```
#define NUM_SYMS 26

typedef unsigned int TRIE_NODE[NUM_SYMS + 1];


unsigned int max_nodes, num_nodes;
TRIE_NODE *trie;
```

The last field stores how many times this string has occurred as a complete word.

`trie`, `num_nodes`, `max_nodes` are global variables in the following code.

Implementation: `init_trie`

```
int init_trie()
{
    max_nodes = 10000;
    if (NULL == (trie = (TRIE_NODE *) calloc(max_nodes, sizeof(TRIE_NODE
    ))))
        ERR_MESG("init-trie: out of memory\n");
    num_nodes = 1;
    return 0;
}
```

Implementation: `insert_node`

```
int insert_node()
{
    if (num_nodes == max_nodes) {
        max_nodes *= 2;
        if (NULL == (trie = Realloc(trie, max_nodes, TRIE_NODE)))
            ERR_MESG("insert-node: out of memory\n");
        bzero((void *) (trie + num_nodes), num_nodes * sizeof(TRIE_NODE)
    );
    }
    num_nodes++;
    return num_nodes - 1;
}
```

Implementation: insert_string I

```
int insert_string(char *s)
{
    unsigned int index = 0;
    int c, new_index;

    while (*s) {
        c = *s;
        if (c >= 'A' && c <= 'Z')
            c = 'a' + c -'A';
        if (c >= 'a' && c <= 'z') {
            c = c - 'a';
            if (trie[index][c] != 0)
                /* just follow the pointer */
                index = trie[index][c];
            else {
                /* need new node */
                if (UNDEF == (new_index = insert_node()))
                    return UNDEF;
```

Implementation: `insert_string` II

```
                index = trie[index][c] = new_index;
            }
        }
        else
            fprintf(stderr, "Unexpected character %d\n", c);
        s++;
    }
    trie[index][NUM_SYMS]++;

    return 0;
}
```

Other trie operations

- Searching: similar to insertion
- Deletion: find the leaf node corresponding to the string, and set value (e.g., frequency) to NULL / 0
- Enumeration: similar to pre-order traversal

Trie performance

- Search hit: linear in length of string
- Search miss: usually sub-linear
- Space: depends on whether many strings share a common prefix

Persistent tries

- For tries that don't change (e.g., dictionaries)

```
if (NULL == (fp = fopen("dict.h", "w")))
    ERR_MESG("make-dict: error opening output file\n");
fprintf(fp, "#include \"trie.h\"\n\nTRIE_NODE dict[] = {\n");
for (i = 0; i < num_nodes; i++) {
    fprintf(fp, "    { ");
    for (j = 0; j < NUM_SYMS + 1; j++)
        fprintf(fp, "%u, ", trie[i][j]);
    fprintf(fp, "},\n");
}
fprintf(fp,"};\n");
fclose(fp);
```

Persistent tries

```
#include "trie.h"

TRIE_NODE dict[] = {
    { 1, 2695, 5429, 8565, 10135, 11370, 12397, 14016, 15784, 16439,
    17334, 18692, 20363, 23617, 24742, 25516, 27617, 27774, 29282,
    32524, 34362, 34671, 35389, 36448, 36548, 36892, 0, },
    { 3, 5, 9, 221, 336, 387, 426, 496, 14, 549, 553, 582, 19, 1261, 21,
     1666, 1764, 1795, 2, 30, 2423, 2573, 35, 2624, 2627, 39, 2, },
    { 2153, 43985, 24, 0, 44065, 0, 2162, 2166, 2218, 0, 44114, 28,
    2230, 0, 2234, 2237, 2251, 0, 2256, 2276, 2309, 0, 2315, 0, 44635,
    0, 3, },
    { 0, 0, 43, 0, 0, 0, 0, 0, 0, 0, 0, 47, 0, 0, 0, 0, 0, 53, 4, 0, 0,
    0, 0, 0, 0, 0, 0, 0, },
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 1, },
    ...
```

What if the alphabet is large?

```
typedef struct {
    AVL_TREE alphabet;
    int count;
} TRIE_NODE;

typedef struct {
    unsigned long max_nodes, num_nodes;
    TRIE_NODE *trie;
} TRIEPP;
```

Tries – C-like implementation I

- As in C
  - trie is an array of trie nodes
  - each node is an array
    - indexed by symbols
    - array values correspond to indices of child nodes

	a	b	. . .	y	z	flag
0						
1						
⋮			⋮			

Tries – C-like implementation II

```
if __name__ == '__main__': # can use code in this file via import (?)
    if len(sys.argv) != 2 : # == argc
        # Note the printf-like syntax below
        sys.exit('Usage: %s <dict file name>' % sys.argv[0])

    # CAUTION:
    # trie = [ [ 0 ] * (NUM_SYMBOLS+1) ] * INIT_SIZE WILL NOT WORK
    # The above creates copies of the SAME list, so when you modify
    # one of them they all change.
    # See https://stackoverflow.com/questions/6667201/how-to-define-a-
    two-dimensional-array-in-python
    trie = [ [ 0 ] * (NUM_SYMBOLS+1) for i in range(INIT_SIZE) ]

    with open(sys.argv[1]) as f : # with => don't need a f.close()
        for line in f: # ASSUMPTION: one word per line
            insert_string(trie, line.strip())
```

Tries – C-like implementation III

```
def insert_string(trie, s) :
    index = 0
    for c in s.lower() :
        i = ord(c) - ord('a')    # get ASCII value
        if i < 0 or i >= NUM_SYMBOLS :
            # print("Unexpected character %c in string %s" % (c, s))
            continue
        if trie[index][i] == 0 : # need new node
            new_index = get_new_node(trie)
            trie[index][i] = new_index
            index = new_index
        else :
            index = trie[index][i]
    trie[index][NUM_SYMBOLS] += 1
```

Tries – C-like implementation IV

```
def get_new_node(trie) :
    global num_nodes, max_nodes
    if num_nodes == max_nodes :
        trie.extend([ [ 0 ] * (NUM_SYMBOLS+1) for i in range(INIT_SIZE)
    ])
        max_nodes += INIT_SIZE
    num_nodes += 1
    return num_nodes - 1
```

Tries++

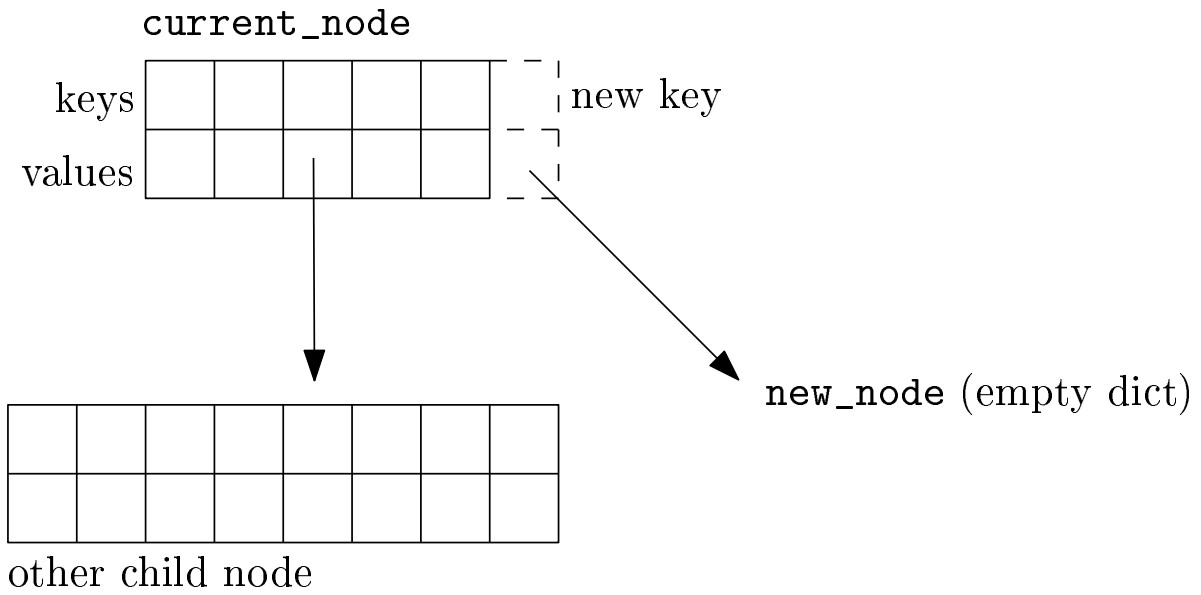
- Each trie node is a dictionary (list of key, value pairs)
- Keys in each dictionary: symbols in your alphabet
- Value for any key: reference (pointer) to child node (another dictionary)

Initialisation

```
trie = { } # Just the first node
```

Tries++ – Insertion

```
def insert_string(trie, s) :
    current_node = trie
    for c in s.lower() :
        if c not in current_node : # c is not a key in this dictionary
            new_node = {}
            current_node[c] = new_node
            current_node = new_node
        else :
            current_node = current_node[c]
```



Tries++ – Searching

```
def search(trie, s) :
    current_node = trie
    for c in s.lower() :
        if c not in current_node :
            return False
        else :
            current_node = current_node[c]
```

```
    while True:
        s = input("Enter word to search for: ")
        if s == "quit" : # Why can you not use "is" ?
            break
        if search(unpickled_trie, s) :
            print("Word found")
        else :
            print("Word not found")
```

Exercise

1. Given a sequence of **non-whitespace** characters $a_1 a_2 \ldots a_M$, a character *n*-gram is defined as any sequence $a_i a_{i+1} a_{i+2} \ldots a_{i+n-1}$ where $n > 0$ and $1 \leq i \leq M - n + 1$. Write a program to find the frequency of the most frequent *n*-gram in a given text that consists *only of lower case letters*. The value of *n* and the text will be given to you as inputs. You may assume that the input consists of lower case letters, blanks and newlines only.