Lecture 16: Trie Cont

CSE 374: Intermediate Programming Concepts and Tools

Lecture Participation Poll #16

Log onto pollev.com/cse374
Or
Text CSE374 to 22333
Assignments
- HW4 turn in coming later today
Binary Trees

struct BinaryTreeNode
{
    int data;
    struct BinaryTreeNode* left;
    struct BinaryTreeNode* right;
}

struct BinaryTree
{
    struct BinaryTreeNode* root;
}
N-Ary Tree

struct TrinaryTreeNode
{
    char* data;
    struct TrinaryTreeNode* left;
    struct TrinaryTreeNode* middle;
    struct TrinaryTreeNode* right;
}

struct QuadTreeNode
{
    char* data;
    struct QuadTreeNode* children[4];
}

- Binary trees just one formal can have any “branching number”
- Trinary trees have branching number of three
- For arbitrarily large branching numbers, arrays can make more sense than lists of named pointers.
Prefix Tree (Trie)

Tries are a character-by-character set-of-Strings implementation
Nodes store *parts of keys* instead of *keys*

Compact data storage
Key of each node defined entirely by position
efficient worst case searching
strings often use 26-ary tree
  - predictive text
  - spell check

Trying to Understand Tries.
In this assignment, you will build programs to implement T9 predictive text, a text input mode developed originally for cell phones and still used for numeric keypads. Each number from 2-9 on the keypad represents three or four letters, the number 0 represents a space, and 1 represents a set of symbols such as {, , ! ?} etc. The numbers from 2-9 represent letters as follows:

- 2 => ABC
- 3 => DEF
- 4 => GHI
- 5 => JKL
- 6 => MNO
- 7 => PQRS
- 8 => TUV
- 9 => WXYZ

Classic trie data structures have edges labeled with letters to store prefixes of strings. But for this application, we use a compressed trie that has only 10 possible branches at each node instead of 26, since the digits 0-9 represent the 26 letters, space and symbols. Because of this, an extra layer of complexity is needed to figure out the string represented by a path.

If a word with the same numeric sequence already exists in the trie, add the new word to the trie as a link to a new node with an edge labeled '#' instead of one of the digits 2-9. (The words linked from a node by the '#' edges essentially form a "linked list" of words that have the same numeric code, but we use additional tree nodes to link the words together instead of defining a separate kind of linked-list node just for this situation.)
typedef struct TrieNode {
    char *word;
    struct TrieNode *children[NUM_CHILDREN];
} TrieNode;

typedef struct Trie {
    TrieNode *root;
} Trie;

TrieNode* makeNode() {
    TrieNode *t = (TrieNode*) malloc(sizeof(TrieNode));
    if (t == NULL) {
        return NULL;
    }
    for (int i = 0; i < NUM_NODES; i++) {
        t->next[i] = NULL;
    }
    t->word = NULL;
    return t;
}
/*
Recurisve follows or inserts nodes starting from previous_node until the location for word is found, at which
point it is inserted. Current_letter is the index of word where the recursive algorithm is currently at.
*/

int node_insert(TrieNode *previous_node, char word[], int current_letter) {
    if (word[current_letter] == '\0') {
        // word is empty
        return;
    }
    int digit = letter_to_digit(word[current_letter]);

    if (previous_node->children[digit] == NULL) {
        // node doesn't exist, create it
    } else {
        // node already exists
        current_node = // next unexamined child of previous node
    }
    if (word[current_letter + 1] == '\0') {
        // at the end of the word
        if (current_node->word == NULL) {
            // current node doesn’t have a word yet
            // save word here
        } else {
            // current node already has a word, add it as an additional completion
        }
    } else { // not at the end of the string, so continue to the next letter
        return node_insert(current_node, word, current_letter + 1);
    }
}
Memory Architecture
public int sum1(int n, int m, int[][] table) {
    int output = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            output += table[i][j];
        }
    }
    return output;
}

public int sum2(int n, int m, int[][] table) {
    int output = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            output += table[j][i];
        }
    }
    return output;
}

What do these two methods do?
What is the big-Θ
Θ(n^2m)
Thought Experiment Graphed

Running sum1 vs sum2 on tables of size n x 4096
Incorrect Assumptions

- Accessing memory is a quick and constant-time operation.
- Sometimes accessing memory is cheaper and easier than at other times.
- Sometimes accessing memory is very slow.
Memory Architecture

- **CPU Register**: The brain of the computer!
  - **Size**: 32 bits
  - **Time**: ≈ free

- **L1 Cache**: Extra memory to make accessing it faster
  - **Size**: 128KB
  - **Time**: 0.5 ns

- **L2 Cache**: Extra memory to make accessing it faster
  - **Size**: 2MB
  - **Time**: 7 ns

- **RAM**: Working memory, what your programs need
  - **Size**: 8GB
  - **Time**: 100 ns

- **Disk**: Large, longtime storage
  - **Size**: 1 TB
  - **Time**: 8,000,000 ns
RAM (Random-Access Memory)

- RAM is where data gets stored for the programs you run. Think of it as the main memory storage location for your programs.

- RAM goes by a ton of different names: memory, main memory, RAM are all names for this same thing.
RAM can be represented as a huge array

RAM:
- addresses, storing stuff at specific locations
- random access

Arrays
- indices, storing stuff at specific locations
- random access

If you’re interested in deeper than this: https://www.youtube.com/watch?v=fpnE6UAfftU or take some EE classes?
Processor – Memory Gap

“Moore’s Law”

- μProc: 55%/year (2X/1.5yr)
- Processor-Memory Performance Gap (grows 50%/year)

1989 first Intel CPU with cache on chip
1998 Pentium III has two cache levels on chip
Processor-Memory Bottleneck

Processor performance doubled about every 18 months.

Bus latency / bandwidth evolved much slower.

Core 2 Duo:
- Can process at least 256 Bytes/cycle.

Core 2 Duo:
- Bandwidth: 2 Bytes/cycle
- Latency: 100-200 cycles (30-60ns)

Problem: lots of waiting on memory

cycle: single machine step (fixed-time)
Problem: Processor-Memory Bottleneck

Processor performance doubled about every 18 months

CPU  Reg  

Cache  

Bus latency / bandwidth evolved much slower

Main Memory

Core 2 Duo:
Can process at least 256 Bytes/cycle

Core 2 Duo:
Bandwidth 2 Bytes/cycle
Latency 100-200 cycles (30-60ns)

Solution: caches

cycle: single machine step (fixed-time)
Example Memory Hierarchy

- **Registers**: <1 ns
- **On-chip L1 cache (SRAM)**: 1-2 min
- **Off-chip L2 cache (SRAM)**: 5-10 s
- **Main memory (DRAM)**: 15-30 min
- **SSD**: 31 days
- **Local secondary storage (local disks)**: 66 months = 5.5 years
- **Remote secondary storage (distributed file systems, web servers)**: 1-15 years

Smaller, faster, costlier per byte

Larger, slower, cheaper per byte

- **Disk**: 10,000,000 ns (10 ms)
- **Remote secondary storage**: 1-150 ms

Upward movement indicates faster access but higher cost per byte.
Example Memory Hierarchy

- **registers**: CPU registers hold words retrieved from L1 cache
- **on-chip L1 cache (SRAM)**: L1 cache holds cache lines retrieved from L2 cache
- **off-chip L2 cache (SRAM)**: L2 cache holds cache lines retrieved from main memory
- **main memory (DRAM)**: Main memory holds disk blocks retrieved from local disks
- **local secondary storage (local disks)**: Local disks hold files retrieved from disks on remote network servers
- **remote secondary storage (distributed file systems, web servers)**:
Example Memory Hierarchy

- Registers: explicitly program-controlled (e.g. refer to exactly %rax, %rbx)
- On-chip L1 cache (SRAM)
- Off-chip L2 cache (SRAM)
- Main memory (DRAM)
- Local secondary storage (local disks)
- Remote secondary storage (distributed file systems, web servers)

- Smaller, faster, costlier per byte
- Larger, slower, cheaper per byte

Program sees “memory”; hardware manages caching transparently.
Review: Binary, Bits and Bytes

- **binary**
  - A base-2 system of representing numbers using only 1s and 0s
  - vs decimal, base 10, which has 9 symbols

- **bit**
  - The smallest unit of computer memory represented as a single binary value either 0 or 1

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Decimal Break Down</th>
<th>Binary</th>
<th>Binary Break Down</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>1100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>127</td>
<td>01111111</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Memory Architecture

Takeaways:
- the more memory a layer can store, the slower it is (generally)
- accessing the disk is very slow

Computer Design Decisions
- Physics
  - Speed of light
  - Physical closeness to CPU
- Cost
  - “good enough” to achieve speed
  - Balance between speed and space
Appendix