Lecture 15: Intro to Trie

Lecture Participation Poll #15

Log onto pollev.com/cse374
Or
Text CSE374 to 22333
Administrivia

Assignments
- HW4 Due Thursday Nov 11
Make is a program which automates building **dependency trees**
- List of rules written in a Makefile declares the commands which build each intermediate part
- Helps you avoid manually typing gcc commands, easier and less prone to typos
- Automates build process

Makefiles are a list of with **Make rules** which include:
- **Target** – An output file to be generated, dependent on one or more sources
- **Source** – Input source code to be built
- **Recipe** – command to generate target

Makefile logic
- Make builds based on structural organization of how code depends on other code as defined by includes
- Recursive – if a source is also a target for other sources, must also evaluate its dependencies and remake as required
- Make can check when you’ve last edited each file, and only build what is needed!
  - Files have "last modification date". make can check whether the sources are more recent than the target
- Make isn’t language specific: recipe can be any valid shell command

**run make command from within same folder**
- `$make [ -f makefile] [ options ] ... [ targets ] ..` /
- Starts with first rule in file then follows dependency tree
- -f specifies makefile name, if non provided will default to “Makefile”
- if no target is specified will default to first listed in file

https://www.gnu.org/software/make/manual/make.html#Introduction
Makefile Example: Linked List

```c
#include "ll.h"

int main() {
    Node *n1 = make_node(4, NULL);
    // rest of main...
}

#include <stdlib.h>
#include <stdio.h>
#include "ll.h"

Node *make_node(int value, Node *next) {
    // rest of linked list code...
}
```

```
try_lists: ll.o main.o
gcc -o try_lists ll.o main.o

ll.o: ll.c ll.h
gcc -c ll.c -o ll.o

main.o: main.c ll.h
gcc -c main.c -o main.o
```

```
Makefile
```
More Make Tools

- make variables help reduce repetitive typing and make alterations easier
  - can change variables from command line
  - enables us to reuse Makefiles on new projects
  - can use conditionals to choose variable settings

- ifdef checks if a given variable is defined for conditional execution
  - ifndef checks if a given variable is NOT defined

- Special characters:
  - $@ for target
  - $^ for all sources
  - $< for left-most source
  - \ enables multiline recipes
  - * functions as wildcard (use carefully)
  - % enables implicit rule definition by using % as a make specific wildcard

---

CC = gcc
CFLAGS = -Wall

foo.o: foo.c foo.h bar.h
 $(CC) $(CFLAGS) -c foo.c -o foo.o

make CFLAGS=-g

EXE=
ifdef WINDIR #defined on Windows
  EXE=.exe
endif

widget$(EXE): foo.o bar.o
 $(CC) $(CFLAGS) -o widget$(EXE)\widget$(EXE)
foo.o bar.o

OBJFILES = foo.o bar.o baz.o

widget: $(OBJFILES)
gcc -o widget $(OBJFILES)

%.o: %.c
  $(CC) -c $(CFLAGS) $< -o $@

clean:
  rm *o widget

---

Phony Targets

▪ A target that doesn’t create the listed output
▪ A way to force run commands regardless of dependency tree
▪ Common uses:
  - all – used to list all top notes across multiple dependency trees
  - clean – cleans up files after usage
  - test – specifies test functionality
  - printing messages or info

```
all: try_lists test_suite
clean:
  rm objectfiles
test: test_suite
  ./test_suite
CC = gcc
CCLAGS = -Wall
all: my_program your_program
  $(CC) $(CFLAGS) -o my_program foo.o bar.o
my_program: foo.o bar.o
  $(CC) $(CFLAGS) -o your_program foo.o baz.o
your_program: bar.o baz.o
  #not shown: foo.o, bar.o, baz.o targets
clean:
  rm *.o my_program your_program
```

---

Makefile
Example Makefile

```makefile
CC = gcc
CFLAGS = -g -Wall -std=c11

try_lists: main.o ll.o
    $(CC) $(CFLAGS) -o try_lists main.o ll.o

main.o: main.c ll.h
    $(CC) $(CFLAGS) -c main.c

ll.o: ll.c ll.h
    $(CC) $(CFILES) -c ll.c

clean:
    rm *.o
```
```c
#include <stdio.h>
#include "speak.h"
#include "shout.h"
/* Write message m to stdout */
void speak(char m[])
{
    printf("%s\n", m);
}

#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "speak.h"
#include "shout.h"
/* Write message m in uppercase to stdout */
void shout(char m[])
{
    int len; /* message length */
    char *mcopy; /* copy of original message */
    int i;
    len = strlen(m);
    mcopy = (char *)malloc(len*sizeof(char)+1);
    strcpy(mcopy,m);
    for (i = 0; i < len; i++)
        mcopy[i] = toupper(mcopy[i]);
    speak(mcopy); free(mcopy);
}

#include <stdio.h>
#include "speak.h"
/* Write message m to stdout */
void speak(char m[])
{
    printf("%s\n", m);
}

#include "speak.h"
#include "shout.h"
/* Say HELLO and goodbye */
int main(int argc, char* argv[])
{
    shout("hello");
    speak("goodbye");
    return 0;
}
```

```
#include <stdio.h>
#include "speak.h"
/* Write message m to stdout */
void speak(char m[])
{
    printf("%s\n", m);
}
```
Example

```
all: talk
# The executable
talk: main.o speak.o shout.o
  gcc -Wall -std=c11 -g -o talk main.o speak.o shout.o

# Individual source files
speak.o: speak.c speak.h
  gcc -Wall -std=c11 -g -c speak.c
shout.o: shout.c shout.h speak.h
  gcc -Wall -std=c11 -g -c shout.c
main.o: main.c speak.h shout.h
  gcc -Wall -std=c11 -g -c main.c

# A "phony" target to remove built files and backups
clean: rm -f *.o talk *~
```

Makefile
Makefile

CC = gcc
# Compiler flags: -Wall for debugger warnings
# -std=c11 for updated standards
CFLAGS = -Wall -std=c11

ifdef DEBUG
CFLAGS += -g
endif

# The name of the program that we are producing.
TARGET = talk

# This is a "phony" target that tells
# make what other targets to build.
all: $(TARGET)

# All the .o files we need for our executable.
OBJS = main.o speak.o shout.o

# The executable
$(TARGET): $(OBJS)
	$(CC) $(CFLAGS) -o talk $(OBJS)

# Individual source files
speak.o: speak.c speak.h
	$(CC) $(CFLAGS) -c speak.c

shout.o: shout.c shout.h speak.h
	$(CC) $(CFLAGS) -c shout.c

main.o: main.c speak.h shout.h
	$(CC) $(CFLAGS) -c main.o

# A "phony" target to remove built files and backups
clean: rm -f *.o talk *~
Binary Trees

```c
struct BinaryTreeNode
{
    int data;
    struct BinaryTreeNode* left;
    struct BinaryTreeNode* right;
}
struct BinaryTree
{
    struct BinaryTreeNode* root;
}
```
N-Ary Tree

```c
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
T9 Trie

What are the branches labeled?
What is branching factor?
What data is stored in each node?
Autocomplete

Search Engines support autocomplete.

How do you efficiently implement autocomplete with the ADTs we know so far?

Formal Problem: Given a “prefix” of a string, find all strings in a set of possible strings that have the given prefix.
Abstract Trie

- Each level represents an index
  - Children represent next possible characters at that index
- This Trie stores the following set of Strings:
  - a, aqua, dad,
  - data, day, days
- How do we deal with a and aqua?
  - Mark complete Strings with a boolean (shown in blue)
  - Complete string: a String that belongs in our set
Searching in Tries

**Search hit:** the final node is a key (colored blue)

**Search miss:** caused in one of two ways
1. The final node is not a key (not colored blue)
2. We “fall” off the Trie

```plaintext
contains("data")   // hit,  l = 4
contains("da")    // miss, l = 2
contains("a")     // hit,  l = 1
contains("dubs")  // miss, l = 4
```

`contains` runtime given key of length `l` with `n` keys in Trie: $\Theta(l)$
Prefix Operations with Tries

- The main appeal of Tries is its efficient prefix matching!

- **Prefix**: find set of keys associated with given prefix
  
  ```java
  keysWithPrefix("day") returns ["day", "days"]
  ```

- **Longest Prefix From Trie**: given a String, retrieve longest prefix of that String that exists in the Trie
  
  ```java
  longestPrefixOf("aquarium") returns "aqua"
  longestPrefixOf("aqueous") returns "aqu"
  longestPrefixOf("dawgs") returns "da"
  ```
Collecting Trie Keys

• **Collect**: return set of all keys in the Trie (like `keySet()`)
  
  \[
  \text{collect}(\text{trie}) = ["a", "aqua", "dad", "data", "day", "days"]
  \]

```java
List collect() {
    List keys;
    for (Node c : root.children) {
        collectHelper(n.char, keys, c);
    }
    return keys;
}

void collectHelper(String str, List keys, Node n) {
    if (n.isKey()) {
        keys.add(s);
    }
    for (Node c : n.children) {
        collectHelper(str + c.char, keys, c);
    }
}
```
**keysWithPrefix** Implementation

- **keysWithPrefix(String prefix)**
  - Find all the keys that corresponds to the given prefix

```java
List keysWithPrefix(String prefix) {
    Node root;  // Node corresponding to given prefix
    List keys;  // Empty list to store keys
    for (Node n : root.children) {
        collectHelper(prefix + n.char, keys, c);
    }
}

void collectHelper(String str, List keys, Node n) {
    if (n.isKey()) {
        keys.add(s);
    }
    for (Node c : n.children) {
        collectHelper(str + c.char, keys, c);
    }
}
```
Autocomplete with Tries

• Autocomplete should return the **most relevant results**

• One method: a Trie-based `Map<String, Relevance>`
  - When a user types in a string "hello", call `keysWithPrefix("hello")`
  - Return the 10 Strings with the highest relevance
**Trie Implementation Idea:** *Encoding*

### ASCII Table

<table>
<thead>
<tr>
<th>Dec</th>
<th>Hex</th>
<th>Oct</th>
<th>Char</th>
<th>Dec</th>
<th>Hex</th>
<th>Oct</th>
<th>Char</th>
<th>Dec</th>
<th>Hex</th>
<th>Oct</th>
<th>Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>00</td>
<td>[space]</td>
<td>32</td>
<td>20</td>
<td>40</td>
<td></td>
<td>64</td>
<td>40</td>
<td>100</td>
<td>@</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>01</td>
<td>!</td>
<td>33</td>
<td>21</td>
<td>41</td>
<td>!</td>
<td>65</td>
<td>41</td>
<td>101</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>02</td>
<td>&quot;</td>
<td>34</td>
<td>22</td>
<td>42</td>
<td>&quot;</td>
<td>66</td>
<td>42</td>
<td>102</td>
<td>B</td>
</tr>
<tr>
<td>3</td>
<td>03</td>
<td>03</td>
<td>#</td>
<td>35</td>
<td>23</td>
<td>43</td>
<td>#</td>
<td>67</td>
<td>43</td>
<td>103</td>
<td>C</td>
</tr>
<tr>
<td>4</td>
<td>04</td>
<td>04</td>
<td>$</td>
<td>36</td>
<td>24</td>
<td>44</td>
<td>$</td>
<td>68</td>
<td>44</td>
<td>104</td>
<td>D</td>
</tr>
<tr>
<td>5</td>
<td>05</td>
<td>05</td>
<td>%</td>
<td>37</td>
<td>25</td>
<td>45</td>
<td>%</td>
<td>69</td>
<td>45</td>
<td>105</td>
<td>E</td>
</tr>
<tr>
<td>6</td>
<td>06</td>
<td>06</td>
<td>&amp;</td>
<td>38</td>
<td>26</td>
<td>46</td>
<td>&amp;</td>
<td>70</td>
<td>46</td>
<td>106</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>07</td>
<td>07</td>
<td>'</td>
<td>39</td>
<td>27</td>
<td>47</td>
<td>'</td>
<td>71</td>
<td>47</td>
<td>107</td>
<td>G</td>
</tr>
<tr>
<td>8</td>
<td>08</td>
<td>10</td>
<td>(</td>
<td>40</td>
<td>28</td>
<td>50</td>
<td>(</td>
<td>72</td>
<td>48</td>
<td>110</td>
<td>H</td>
</tr>
<tr>
<td>9</td>
<td>09</td>
<td>11</td>
<td>)</td>
<td>41</td>
<td>29</td>
<td>51</td>
<td>)</td>
<td>73</td>
<td>49</td>
<td>111</td>
<td>I</td>
</tr>
<tr>
<td>10</td>
<td>A</td>
<td>12</td>
<td>*</td>
<td>42</td>
<td>2A</td>
<td>52</td>
<td>*</td>
<td>74</td>
<td>4A</td>
<td>112</td>
<td>J</td>
</tr>
<tr>
<td>11</td>
<td>B</td>
<td>13</td>
<td>+</td>
<td>43</td>
<td>2B</td>
<td>53</td>
<td>+</td>
<td>75</td>
<td>4B</td>
<td>113</td>
<td>K</td>
</tr>
<tr>
<td>12</td>
<td>C</td>
<td>14</td>
<td>,</td>
<td>44</td>
<td>2C</td>
<td>54</td>
<td>,</td>
<td>76</td>
<td>4C</td>
<td>114</td>
<td>L</td>
</tr>
<tr>
<td>13</td>
<td>D</td>
<td>15</td>
<td>-</td>
<td>45</td>
<td>2D</td>
<td>55</td>
<td>-</td>
<td>77</td>
<td>4D</td>
<td>115</td>
<td>M</td>
</tr>
<tr>
<td>14</td>
<td>E</td>
<td>16</td>
<td>.</td>
<td>46</td>
<td>2E</td>
<td>56</td>
<td>.</td>
<td>78</td>
<td>4E</td>
<td>116</td>
<td>N</td>
</tr>
<tr>
<td>15</td>
<td>F</td>
<td>17</td>
<td>/</td>
<td>47</td>
<td>2F</td>
<td>57</td>
<td>/</td>
<td>79</td>
<td>4F</td>
<td>117</td>
<td>O</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>20</td>
<td>0</td>
<td>48</td>
<td>30</td>
<td>60</td>
<td>0</td>
<td>80</td>
<td>50</td>
<td>120</td>
<td>P</td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>21</td>
<td>1</td>
<td>49</td>
<td>31</td>
<td>61</td>
<td>1</td>
<td>81</td>
<td>51</td>
<td>121</td>
<td>Q</td>
</tr>
<tr>
<td>18</td>
<td>12</td>
<td>22</td>
<td>2</td>
<td>50</td>
<td>32</td>
<td>62</td>
<td>2</td>
<td>82</td>
<td>52</td>
<td>122</td>
<td>R</td>
</tr>
<tr>
<td>19</td>
<td>13</td>
<td>23</td>
<td>3</td>
<td>51</td>
<td>33</td>
<td>63</td>
<td>3</td>
<td>83</td>
<td>53</td>
<td>123</td>
<td>S</td>
</tr>
<tr>
<td>20</td>
<td>14</td>
<td>24</td>
<td>4</td>
<td>52</td>
<td>34</td>
<td>64</td>
<td>4</td>
<td>84</td>
<td>54</td>
<td>124</td>
<td>T</td>
</tr>
<tr>
<td>21</td>
<td>15</td>
<td>25</td>
<td>5</td>
<td>53</td>
<td>35</td>
<td>65</td>
<td>5</td>
<td>85</td>
<td>55</td>
<td>125</td>
<td>U</td>
</tr>
<tr>
<td>22</td>
<td>16</td>
<td>26</td>
<td>6</td>
<td>54</td>
<td>36</td>
<td>66</td>
<td>6</td>
<td>86</td>
<td>56</td>
<td>126</td>
<td>V</td>
</tr>
<tr>
<td>23</td>
<td>17</td>
<td>27</td>
<td>7</td>
<td>55</td>
<td>37</td>
<td>67</td>
<td>7</td>
<td>87</td>
<td>57</td>
<td>127</td>
<td>W</td>
</tr>
<tr>
<td>24</td>
<td>18</td>
<td>30</td>
<td>8</td>
<td>56</td>
<td>38</td>
<td>70</td>
<td>8</td>
<td>88</td>
<td>58</td>
<td>130</td>
<td>X</td>
</tr>
<tr>
<td>25</td>
<td>19</td>
<td>31</td>
<td>9</td>
<td>57</td>
<td>39</td>
<td>71</td>
<td>9</td>
<td>89</td>
<td>59</td>
<td>131</td>
<td>Y</td>
</tr>
<tr>
<td>26</td>
<td>1A</td>
<td>32</td>
<td>z</td>
<td>58</td>
<td>3A</td>
<td>72</td>
<td>z</td>
<td>90</td>
<td>5A</td>
<td>132</td>
<td>Z</td>
</tr>
<tr>
<td>27</td>
<td>1B</td>
<td>33</td>
<td>i</td>
<td>59</td>
<td>3B</td>
<td>73</td>
<td>i</td>
<td>91</td>
<td>5B</td>
<td>133</td>
<td>I</td>
</tr>
<tr>
<td>28</td>
<td>1C</td>
<td>34</td>
<td>^</td>
<td>60</td>
<td>3C</td>
<td>74</td>
<td>^</td>
<td>92</td>
<td>5C</td>
<td>134</td>
<td>\</td>
</tr>
<tr>
<td>29</td>
<td>1D</td>
<td>35</td>
<td>7</td>
<td>61</td>
<td>3D</td>
<td>75</td>
<td>7</td>
<td>93</td>
<td>5D</td>
<td>135</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>1E</td>
<td>36</td>
<td>6</td>
<td>62</td>
<td>3E</td>
<td>76</td>
<td>6</td>
<td>94</td>
<td>5E</td>
<td>136</td>
<td>^</td>
</tr>
<tr>
<td>31</td>
<td>1F</td>
<td>37</td>
<td>7</td>
<td>63</td>
<td>3F</td>
<td>77</td>
<td>7</td>
<td>95</td>
<td>5F</td>
<td>137</td>
<td>_</td>
</tr>
</tbody>
</table>
class TrieSet {
    final int R = 128; // # of ASCII encodings
    Node overallRoot;

    // Private internal class
    class Node {
        // Field declarations
        char ch;
        boolean isKey;
        DataIndexedCharMap<Node> next; // array encoding

        // Constructor
        Node(char c, boolean b, int R) {
            ch = c;
            isKey = b;
            next = new DataIndexedCharMap<Node>(R);
        }
    }
}
Data Structure for Trie Implementation

• Think of a Binary Tree
  - Instead of two children, we have 128 possible children
  - Each child represents a possible next character of our Trie

• How could we store these 128 children?
Data-Indexed Array Visualization

// Private internal class
class Node {
    // Field declarations
    char ch;
    boolean isKey;
    DataIndexedCharMap<Node> next;
}
Removing Redundancy

class TrieSet {
    final int R = 128;
    Node overallRoot;

    // Private internal class
    class Node {
        // Field declarations
        char ch;
        boolean isKey;
        DataIndexedCharMap<Node> next;

        // Constructor
        Node(char c, boolean b, int R) {
            ch = c;
            isKey = b;
            next = new DataIndexedCharMap<Node>(R);
        }
    }
}
Does the structure of a Trie depend on the order of insertion?
   a) Yes
   b) No
   c) I’m not sure…
Runtime Comparison

- Typical runtime when treating length \( l \) of keys as a constant:

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Key Type</th>
<th>( \text{contains} )</th>
<th>( \text{add} )</th>
<th>( \text{keysWithPrefix} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Balanced BST</td>
<td>Comparable</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hash Map</td>
<td>Hashable</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trie (Data-Indexed Array)</td>
<td>String (Character)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Where** \( p \) is the number of strings with the given prefix. Usually \( p \ll n \).

Takeaways:

- When keys are Strings, Tries give us a better \( \text{add} \) and \( \text{contains} \) runtime.
- **DataIndexedCharMap** takes up a lot of space by storing \( R \) links per node.
Trie Takeaways

• Tries can be used for storing Strings (or any sequential data)
• Real-world performance often better than Hash Table or Search Tree
• Many different implementations: `DataIndexedCharMap`, Hash Tables, BSTs (and more possible data structures within nodes), and TSTs
• Tries enable efficient prefix operations like `keysWithPrefix`
Appendix
File IO – working with strings

- **FILE *fopen(const char *path, const char *mode);**
  - opens the file whose name is the string pointed to by path and associates a stream with it.

- **char *fgets(char *s, int size, FILE *stream);**
  - reads in at most one less than size characters from stream and stores them into the buffer pointed to by s. Reading stops after an EOF or a newline. If a newline is read, it is stored into the buffer. A terminating null byte (‘\0’) is stored after the last character in the buffer.

- **int fprintf(FILE *stream, const char *format, ...);**
  - It’s printf, but to a file.

- **int fputc(int c, FILE *stream);** // print a single character

- **int fputs(const char *s, FILE *stream);** // print a string