Lecture Participation Poll #15

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

Lecture 15: Intro to Trie

CSE 374: Intermediate Programming Concepts and Tools

Administrivia

Assignments

- HW4 Due Thursday Nov 11

Make Files

- **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

tab not spaces! →

```
target: source
    recipe
```

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

- 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

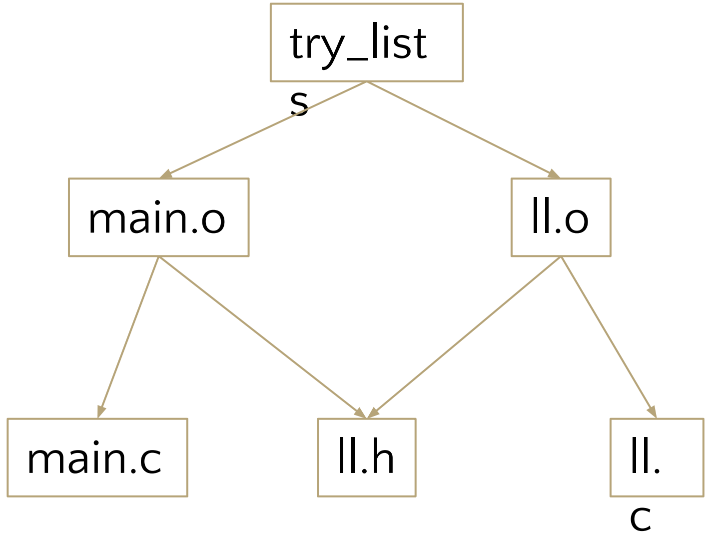
https://www.gnu.org/software/make/manual/make.html#Introduction

Makefile Example: Linked List

```
#include "ll.h"

int main() {
    Node *n1 = make_node(4, NULL);

    // rest of main…
```
main.c

```
#ifndef LL_H
#define LL_H

typedef struct Node {

    // rest of Node def…
```
ll.h

```
#include <stdlib.h>
#include <stdio.h>

#include "ll.h"

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




```
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
  - $ˆ for all sources
  - $< for left-most source
  - \ enables multiline recipies
  - * functions as wildcard (use carefully)
  - % enables implicit rule definition by using % as a make specific wildcard

```
CC = gcc
CGLAGS = -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)\
	    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
```

Makefile

http://cslibrary.stanford.edu/107/UnixProgrammingTools.pdf

Phony Targets

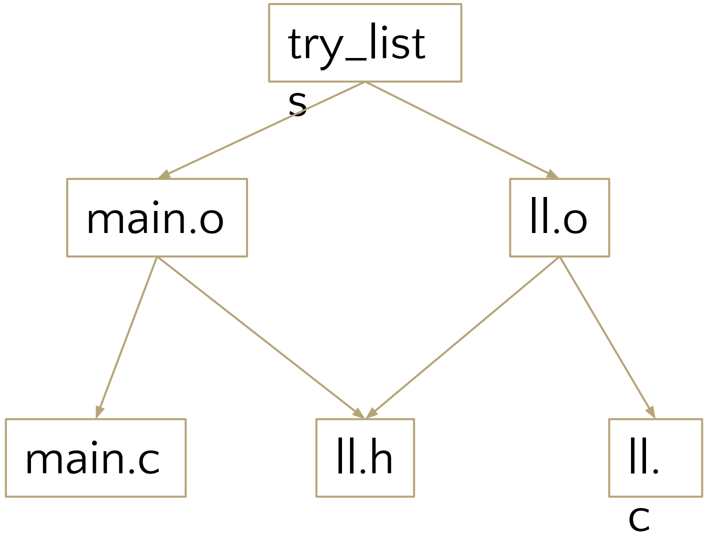
- A target that doesn't create the listed output
- A way to force run commands regardless of dependency tree
- Common uses:
  - all – used to list all top notes across multiple dependency trees
  - clean – cleans up files after usage
  - test – specifies test functionality
  - printing messages or info

```
all: try_lists test_suite
clean:
    rm objectfiles
test: test_suite
    ./test_suite
```

```
CC = gcc
CGLAGS = -Wall

all: my_program your_program

my_program: foo.o bar.o
    $(CC) $(CFLAGS) -o my_program foo.o bar.o

your_program: bar.o baz.o
    $(CC) $(CFLAGS) -o your_program foo.o baz.o

#not shown: foo.o, bar.o, baz.o targets

clean:
    rm *.o my_program your_program
```

Makefile

Example Makefile



variable definitions

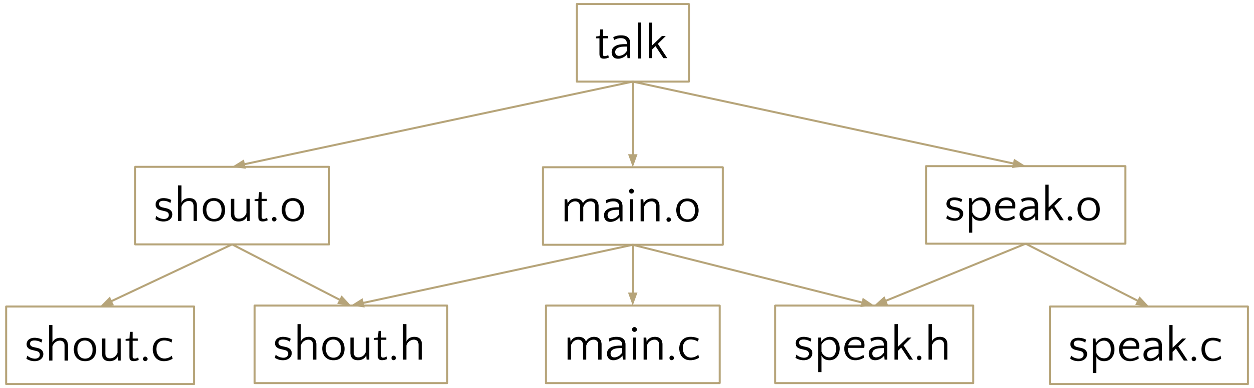
must include rules for each file

rules define dependency hierarchy

```
CC = gcc
CGLAGS = -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
```

Makefile

Example




```c
#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);
}
```

shout.c

```c
#ifndef SPEAK_H
#define SPEAK_H
/* Write message m to stdout */
void speak(char m[]);
#endif /* ifndef SPEAK_H */
```

speak.h

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

speak.c

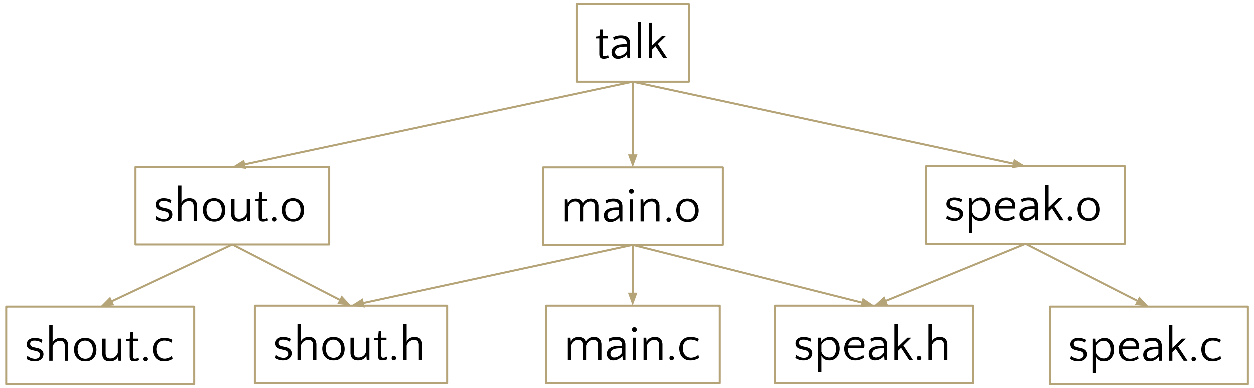
```c
#include "speak.h"
#include "shout.h"
/* Say HELLO and goodbye */
int main(int argc, char* argv[])
{
   shout("hello");
   speak("goodbye");
   return 0;
}
```

main.c

```c
#ifndef SHOUT_H
#define SHOUT_H
/* Write message m in uppercase to stdout */
void shout(char m[]);
#endif /* ifndef SHOUT_H */
```

shout.h

Example



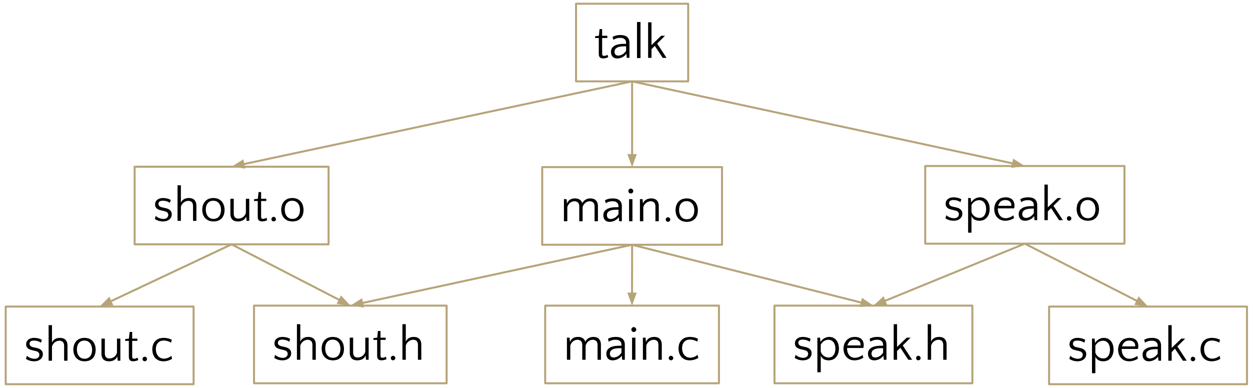

```
all: talk
# The executable
talk: main.o speak.o shout.o
   gcc -Wall -std=c11 -g -o talk main.o speak.o shout.o

# 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

Example

Makefile

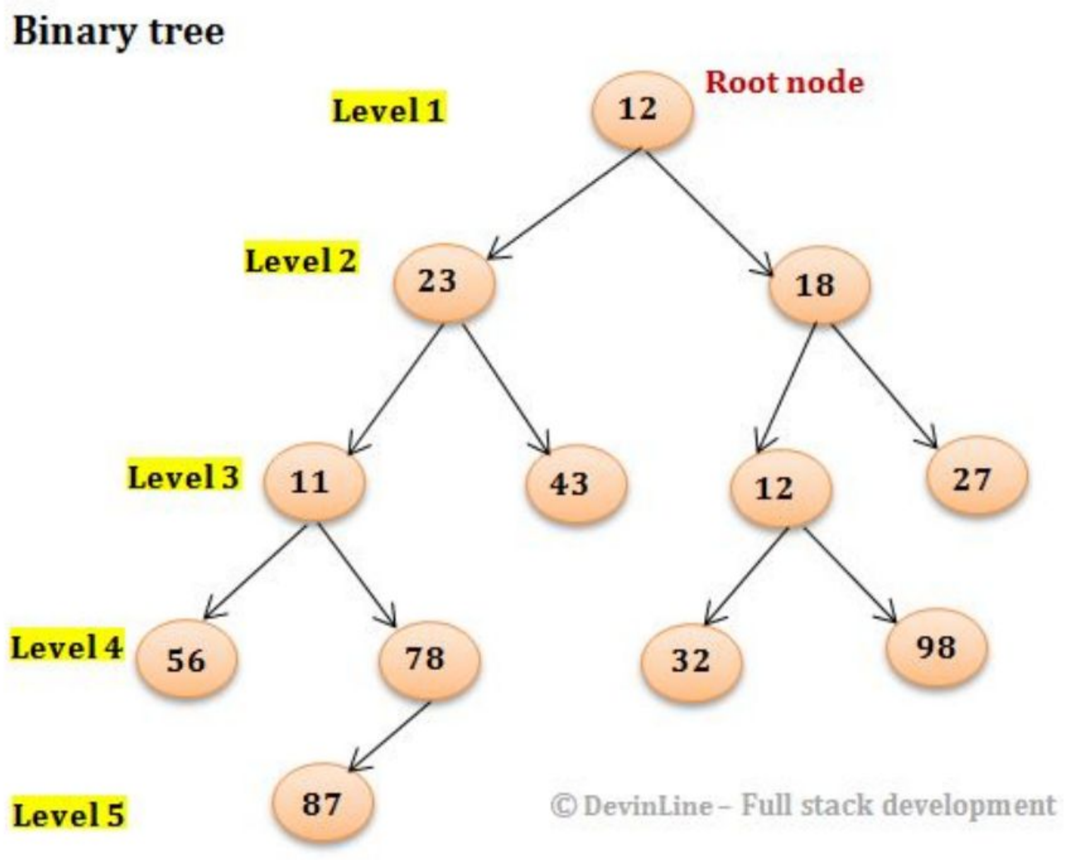
```
CC = gcc
# Compiler flags: -Wall for debugger warnings
# -std=c11 for updated standards
CFLAGS = -Wall -std=c11

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.c

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

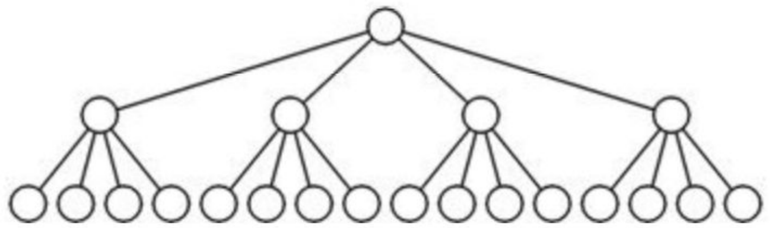
Binary Trees

```
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];
}
```

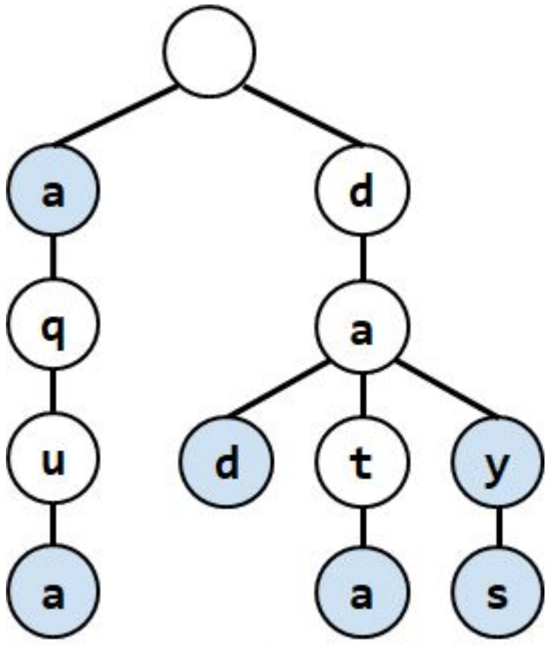


- Binary trees just one formal can have any “branching number”
- Trinary tres have branching number of three
- For arbitrarily large branching numbers, arrays can make more sense than lists of named pointers.

Prefix Tree (Trie)

Tries are a character-by-character set-of-Strings implementation

Nodes store *parts of keys* instead of *keys*



Trie

Compact data storage

Key of each node defined entirely by position

efficient worst case searching

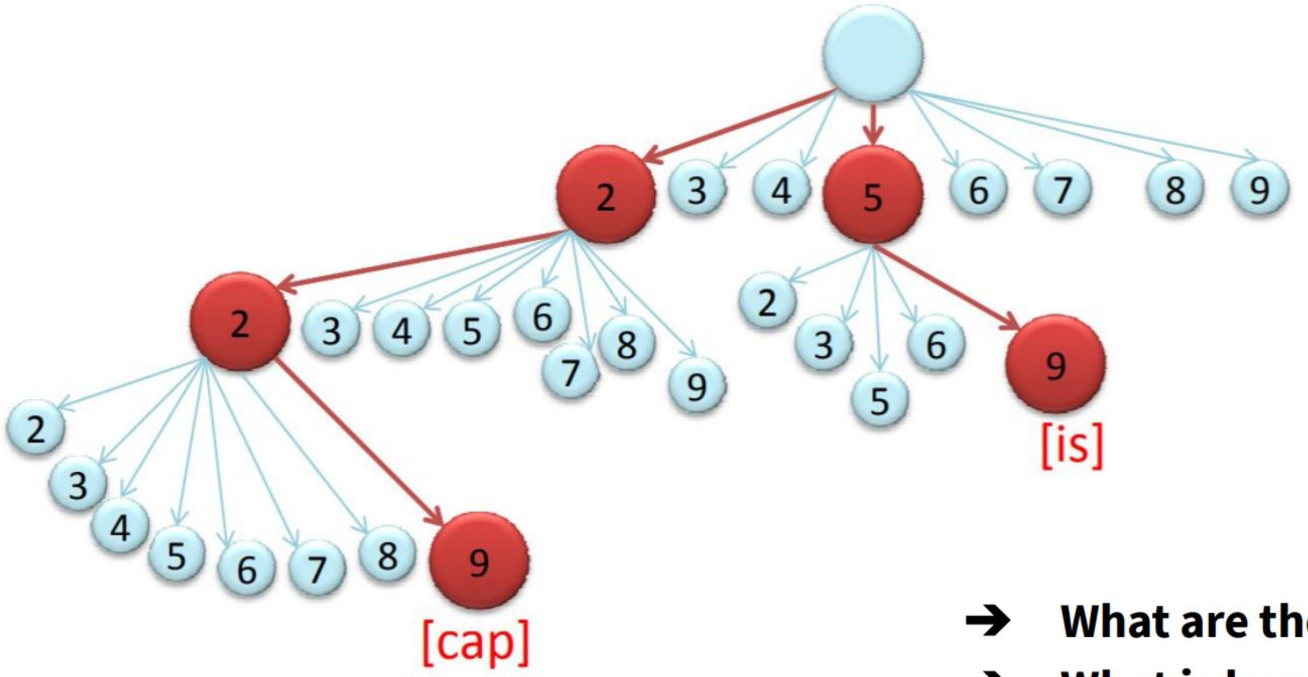
strings often use 26-ary tree

- predictive text
- spell check

Trying to Understand Tries.

T9 Trie

T9 Trie



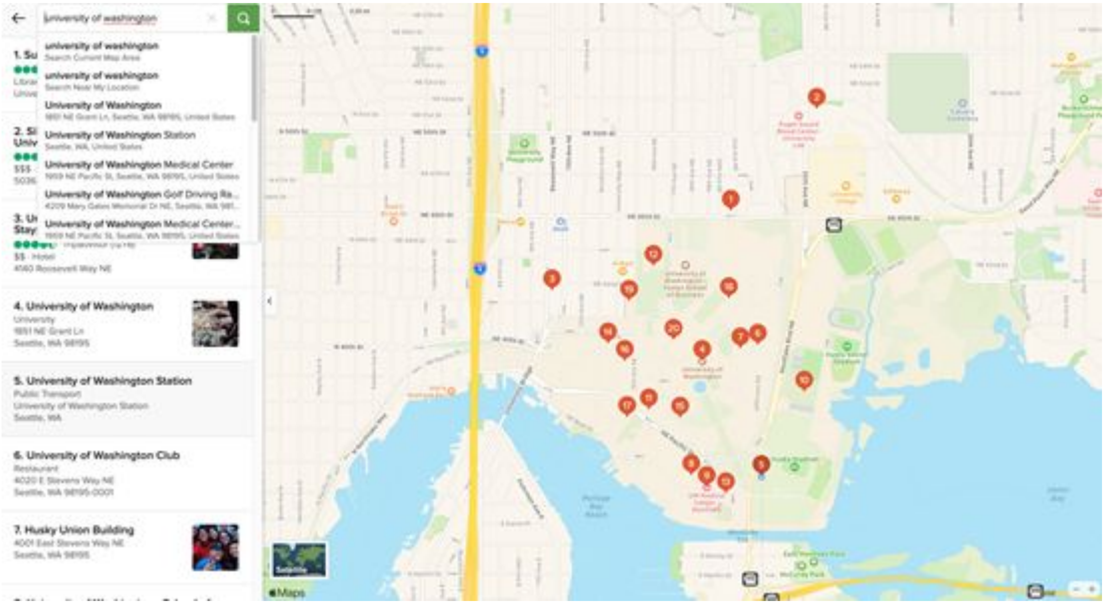
- ➔ **What are the branches labeled?**
- ➔ **What is branching factor?**
- ➔ **What data is stored in each node?**

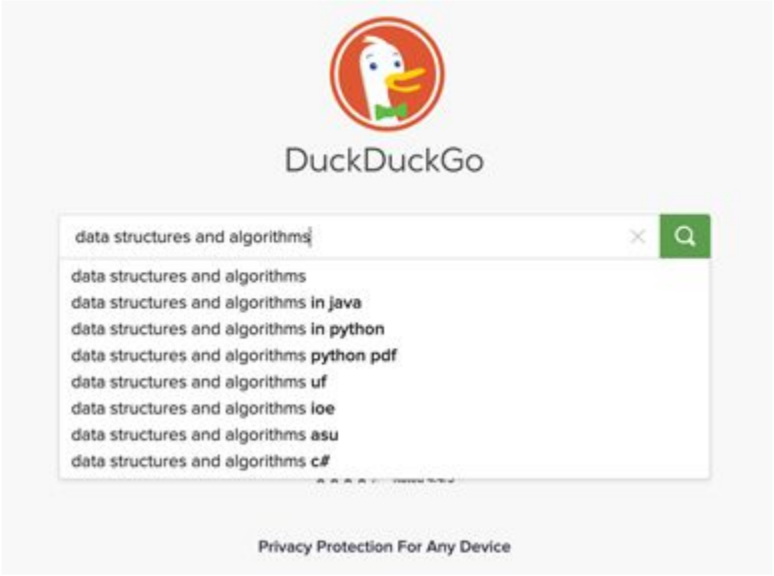
Autocomplete

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.



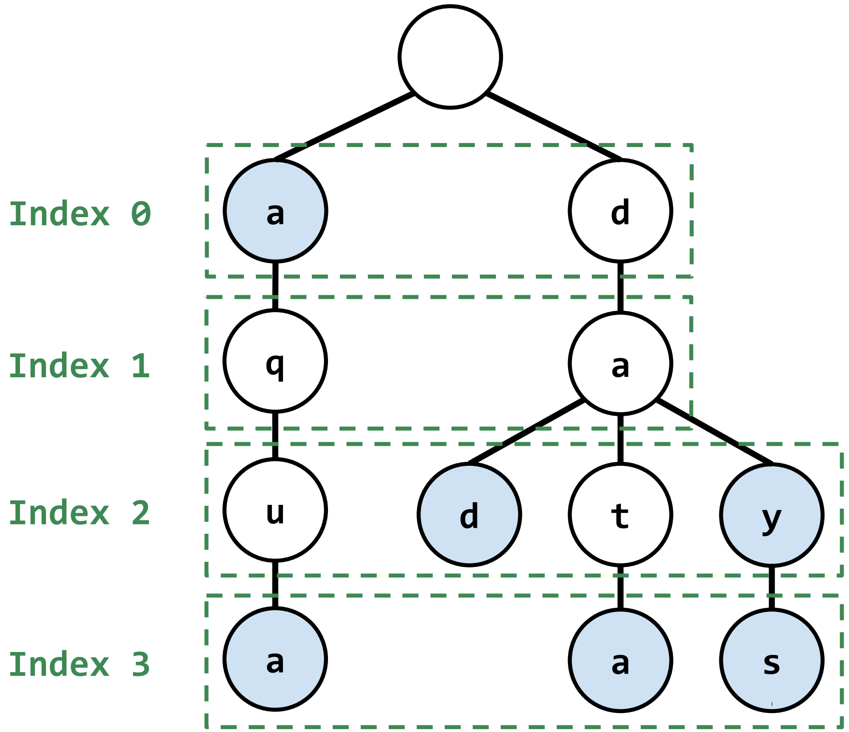


Abstract Trie

- Each level represents an index
  - Children represent next possible characters at that index
- This Trie stores the following set of Strings:

```
   0   0123  012
a, aqua, dad,

  0123  012   0123
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



Abstract Trie

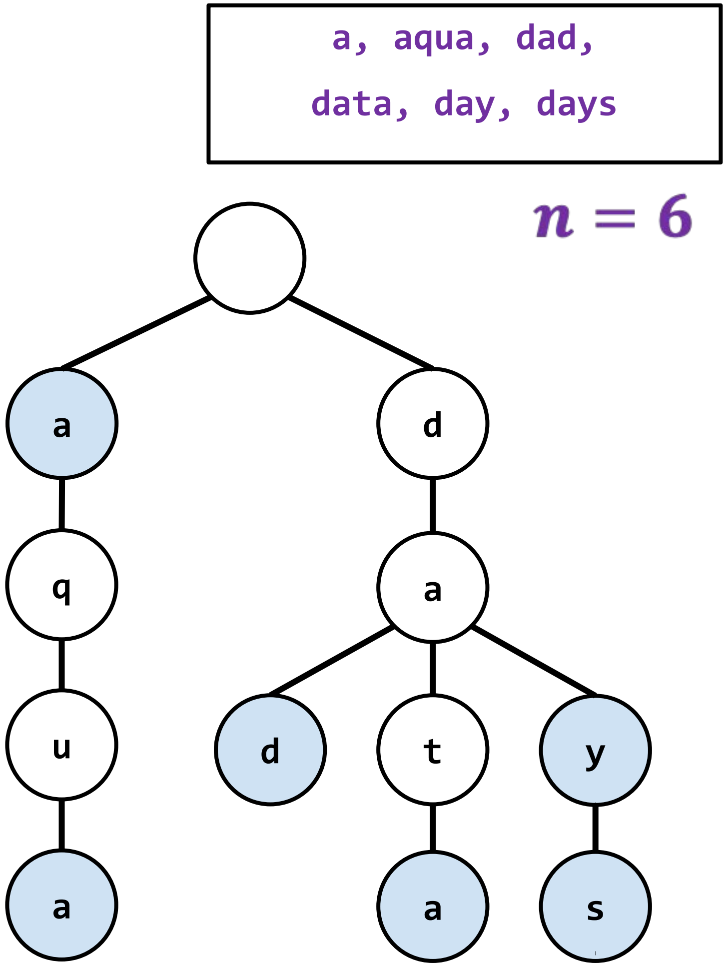
Searching in Tries

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

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



Abstract Trie

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

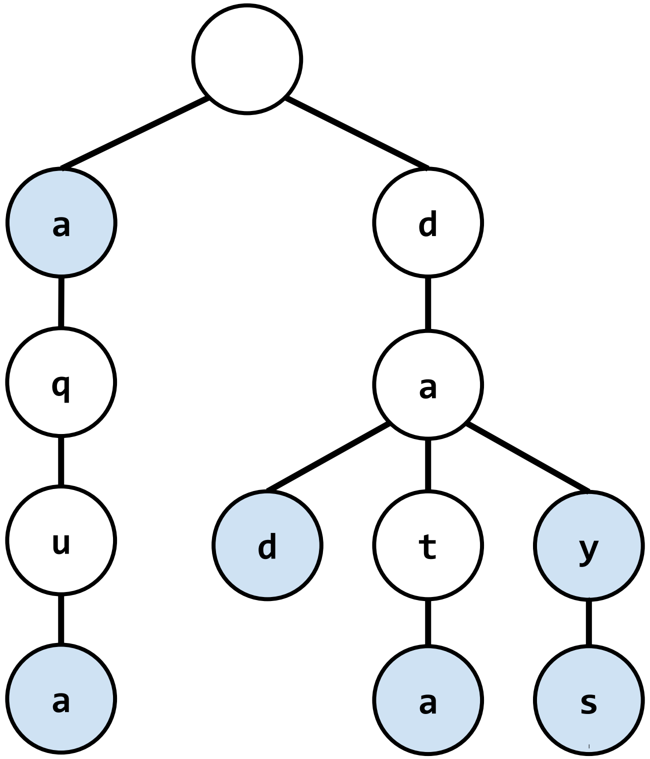
Prefix Operations with Tries

a, aqua, dad,
data, day, days

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

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

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



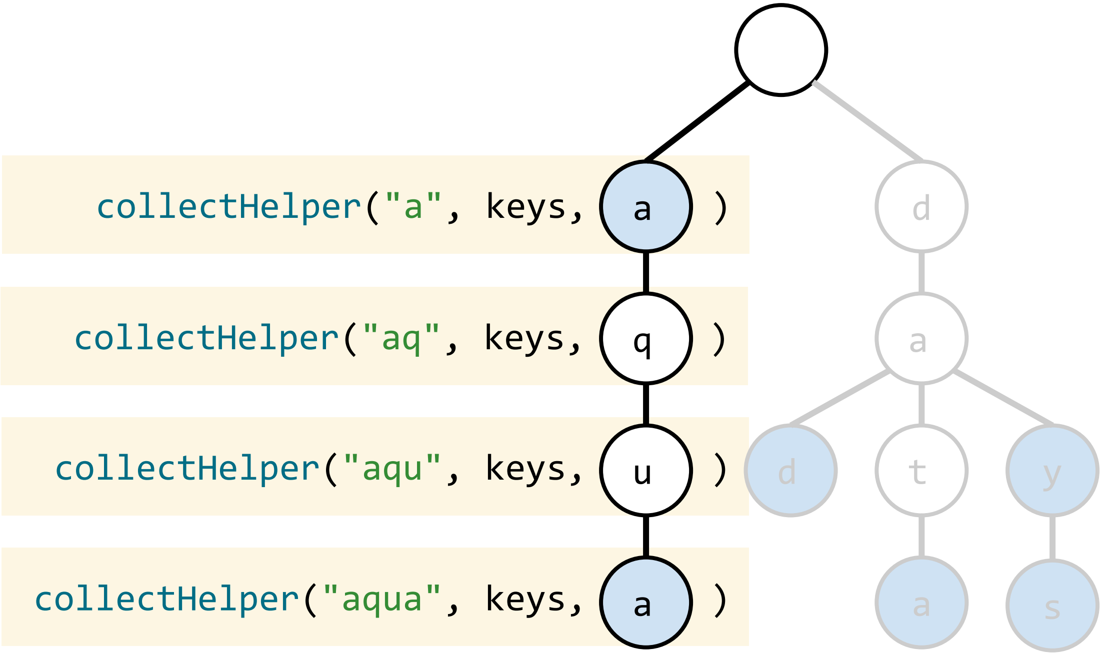
Abstract Trie

Collecting Trie Keys

- **Collect:** return set of all keys in the Trie (like `keySet()`)

  `collect(trie) = ["a", "aqua", "dad", "data", "day", "days"]`

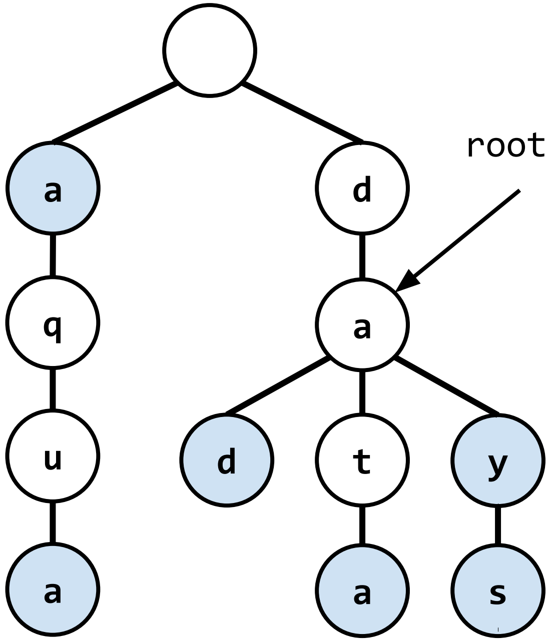
```
List collect() {
    List keys;
    for (Node c : root.children) {
        collectHelper(n.char, keys, c);
    }
    return keys;
}

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

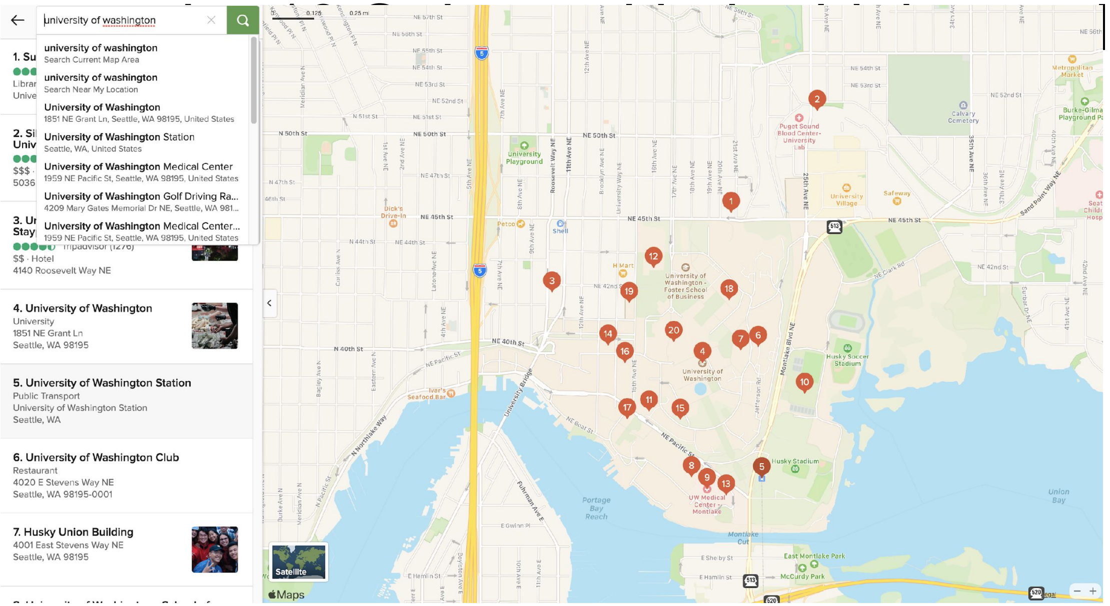
```
List keysWithPrefix(String prefix) {
    Node root;   // Node corresponding to given prefix
    List keys;   // Empty list to store keys

    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);
    }
}
```

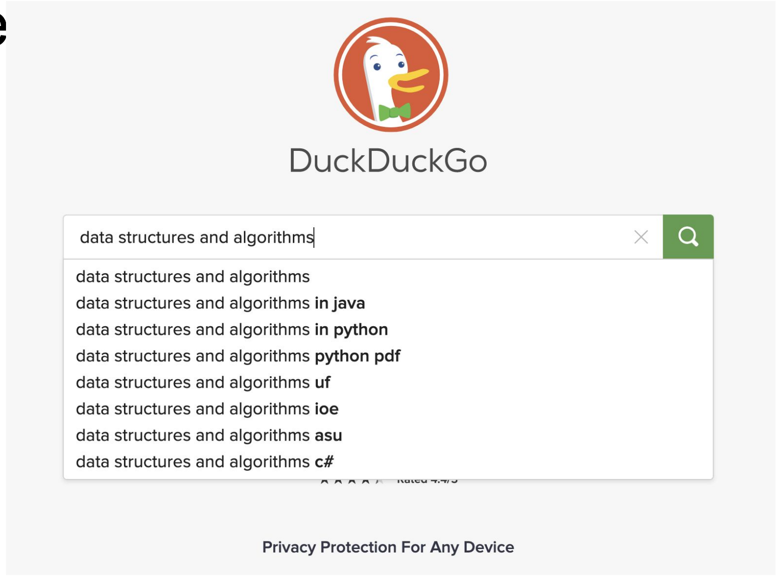


Autocomplete with Tries

- 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")`
  - R … levance





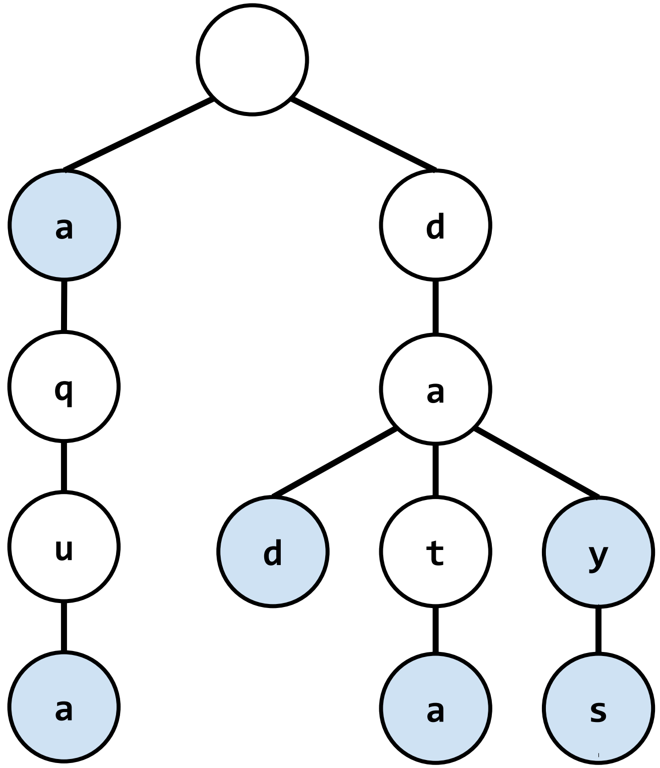
Trie Implementation Idea: *Encoding*

ASCII Table

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@	96	60	140	`
1	1	1		33	21	41	!	65	41	101	A	97	61	141	a
2	2	2		34	22	42	"	66	42	102	B	98	62	142	b
3	3	3		35	23	43	#	67	43	103	C	99	63	143	c
4	4	4		36	24	44	$	68	44	104	D	100	64	144	d
5	5	5		37	25	45	%	69	45	105	E	101	65	145	e
6	6	6		38	26	46	&	70	46	106	F	102	66	146	f
7	7	7		39	27	47	'	71	47	107	G	103	67	147	g
8	8	10		40	28	50	(	72	48	110	H	104	68	150	h
9	9	11		41	29	51	)	73	49	111	I	105	69	151	i
10	A	12		42	2A	52	*	74	4A	112	J	106	6A	152	j
11	B	13		43	2B	53	+	75	4B	113	K	107	6B	153	k
12	C	14		44	2C	54	,	76	4C	114	L	108	6C	154	l
13	D	15		45	2D	55	-	77	4D	115	M	109	6D	155	m
14	E	16		46	2E	56	.	78	4E	116	N	110	6E	156	n
15	F	17		47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10	20		48	30	60	0	80	50	120	P	112	70	160	p
17	11	21		49	31	61	1	81	51	121	Q	113	71	161	q
18	12	22		50	32	62	2	82	52	122	R	114	72	162	r
19	13	23		51	33	63	3	83	53	123	S	115	73	163	s
20	14	24		52	34	64	4	84	54	124	T	116	74	164	t
21	15	25		53	35	65	5	85	55	125	U	117	75	165	u
22	16	26		54	36	66	6	86	56	126	V	118	76	166	v
23	17	27		55	37	67	7	87	57	127	W	119	77	167	w
24	18	30		56	38	70	8	88	58	130	X	120	78	170	x
25	19	31		57	39	71	9	89	59	131	Y	121	79	171	y
26	1A	32		58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B	33		59	3B	73	;	91	5B	133	[	123	7B	173	{
28	1C	34		60	3C	74	<	92	5C	134	\	124	7C	174	|
29	1D	35		61	3D	75	=	93	5D	135	]	125	7D	175	}
30	1E	36		62	3E	76	>	94	5E	136	^	126	7E	176	~
31	1F	37		63	3F	77	?	95	5F	137	_	127	7F	177	

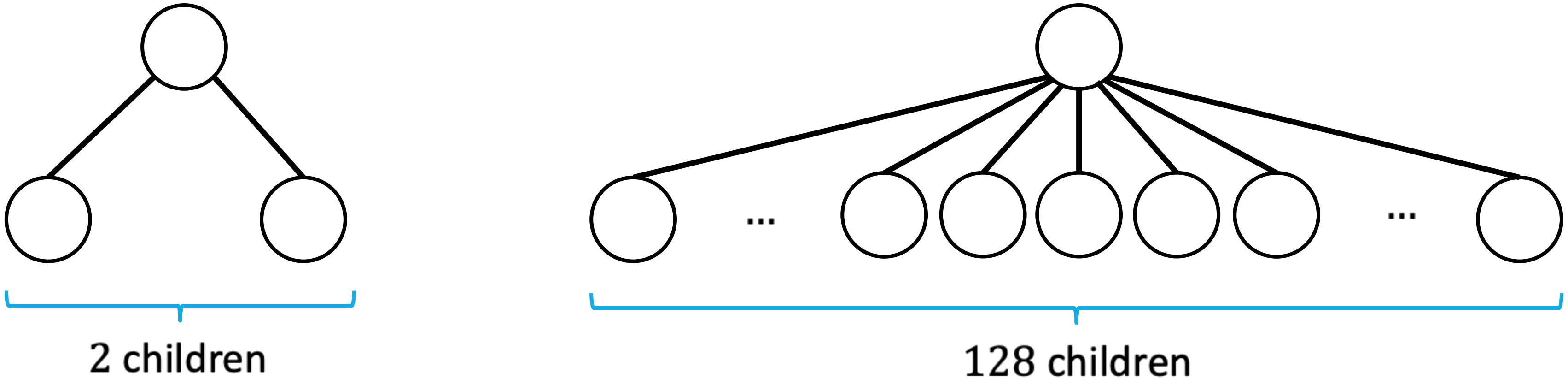
DataIndexedCharMap Pseudocode

```
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);
        }
    }
}
```



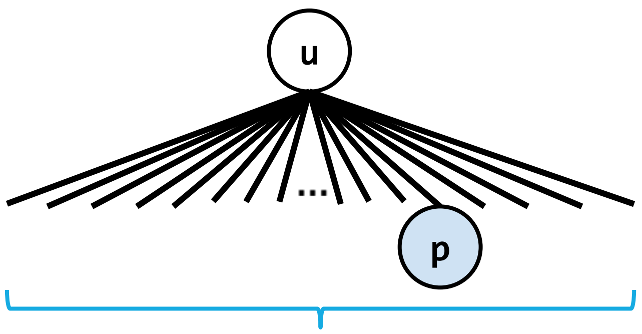
Data Structure for Trie Implementation

- 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?

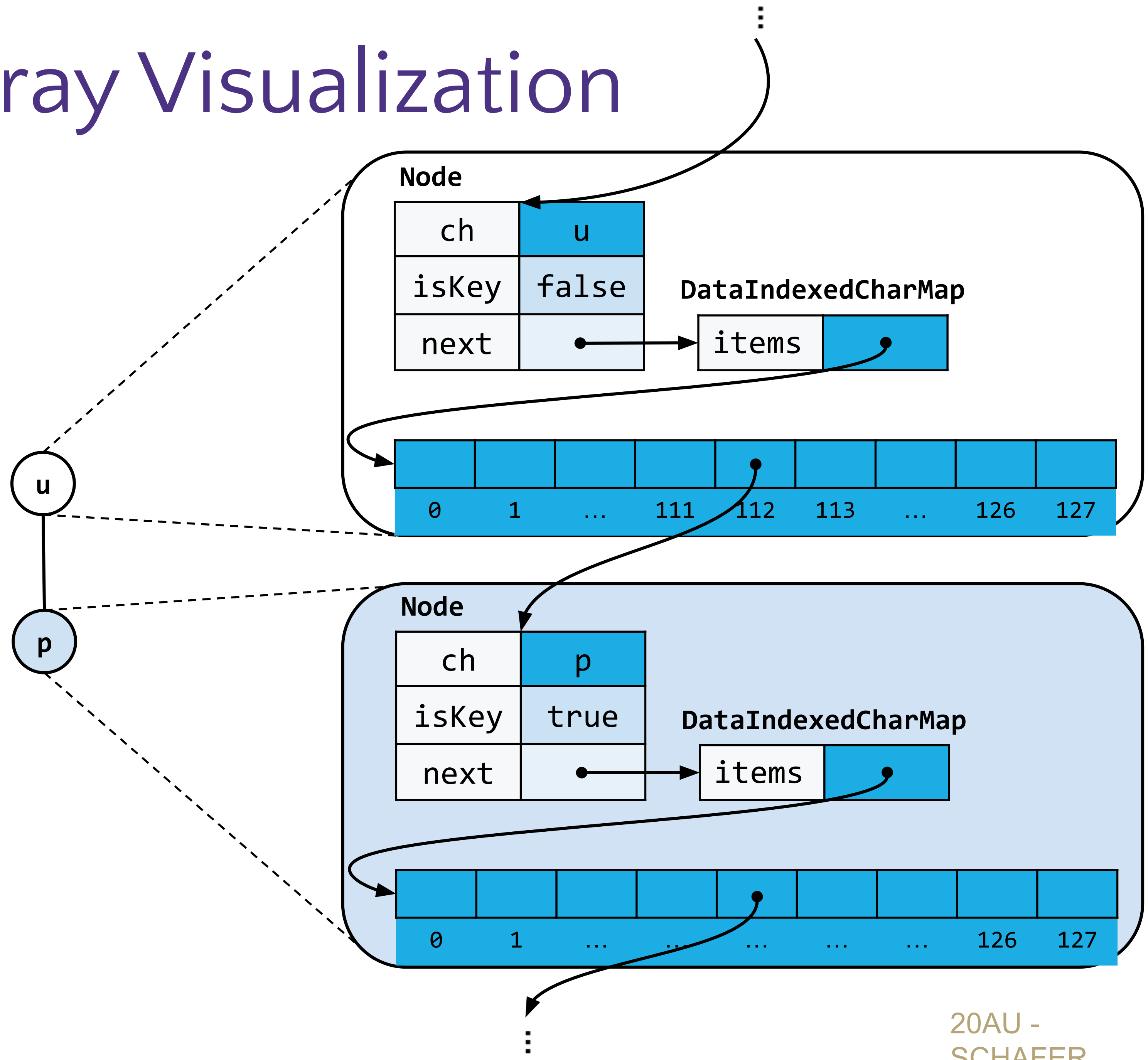


Data-Indexed Array Visualization

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

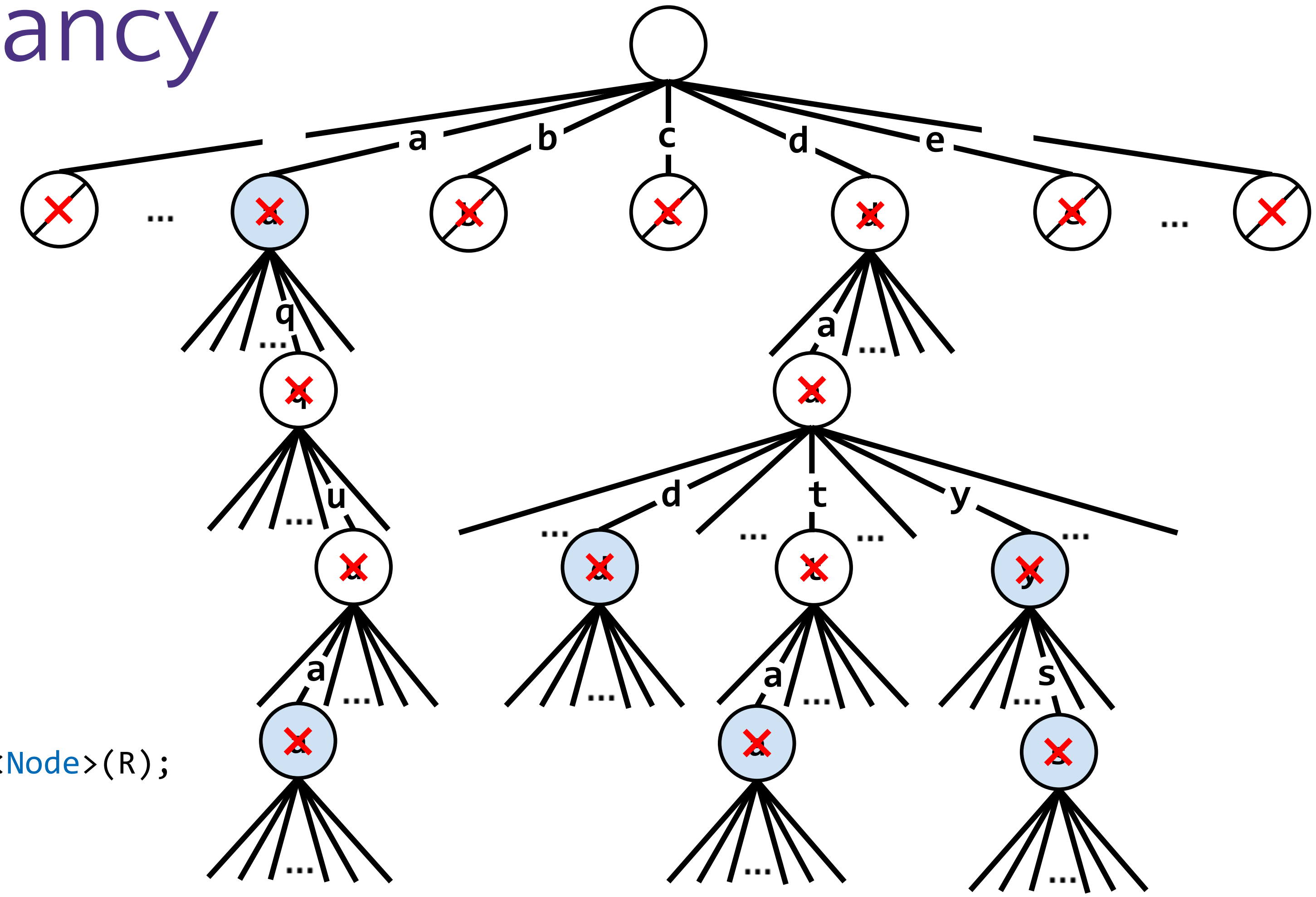


R = 128 links, 127 null



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);
        }
    }
}
```



Itempool

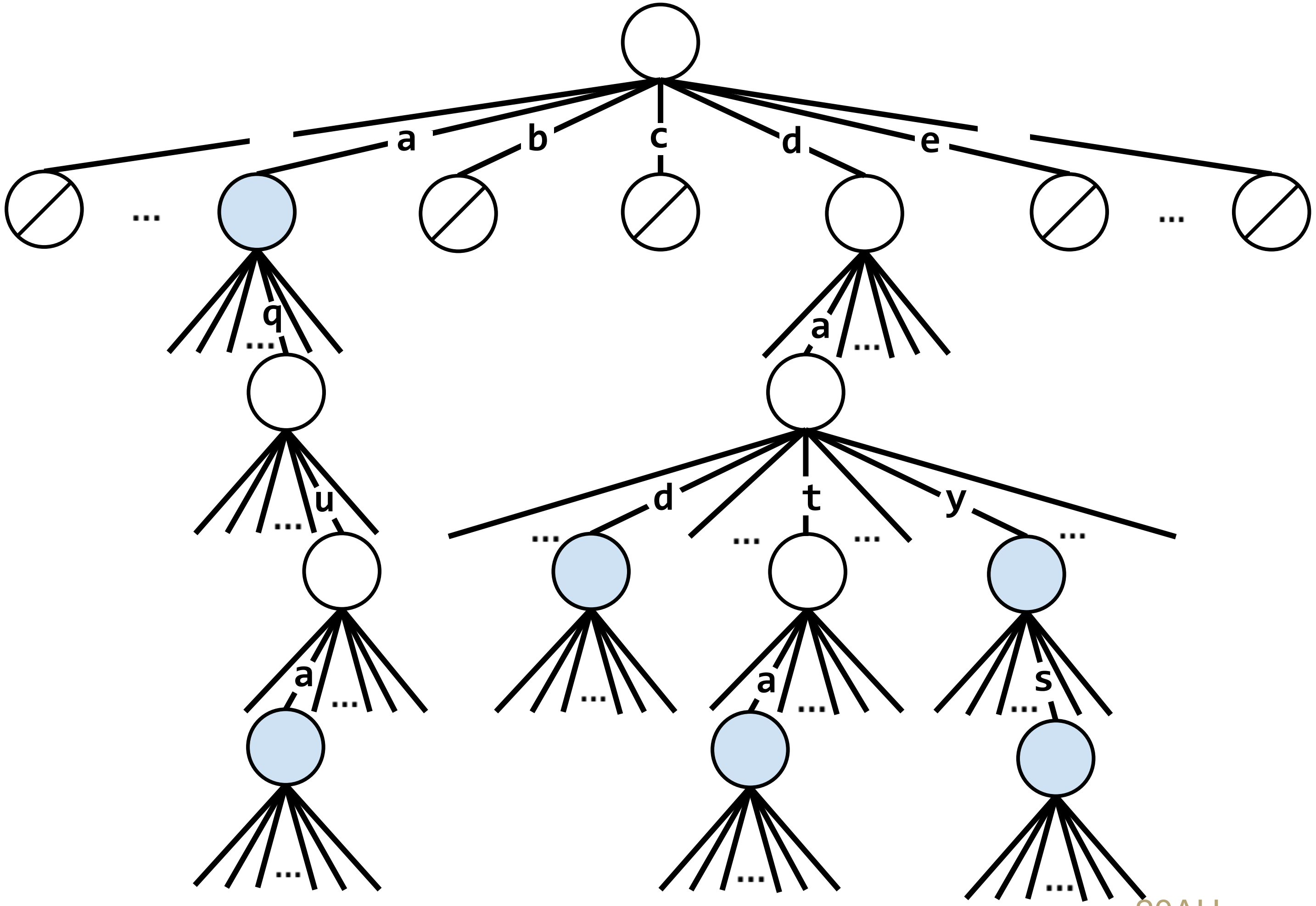
Pause Video when Prompted

Does the structure of a Trie depend on the order of insertion?

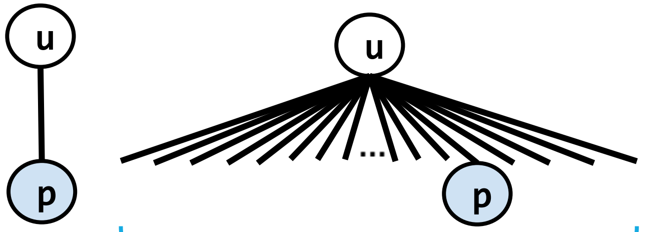
a) Yes

b) No

c) I'm not sure…



Runtime Comparison



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

Data Structure	Key Type	`contains`	`add`	`keysWithPrefix`
Balanced BST	Comparable			
Hash Map	Hashable			
Trie (Data-Indexed Array)	String (Character)			

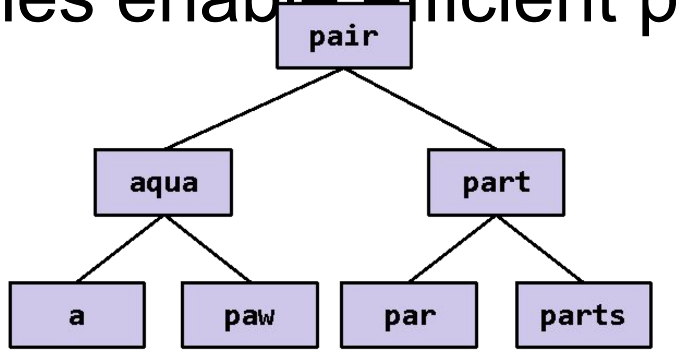
* In-practice runtime

** Where p is the number of strings with the given prefix. Usually p << n.

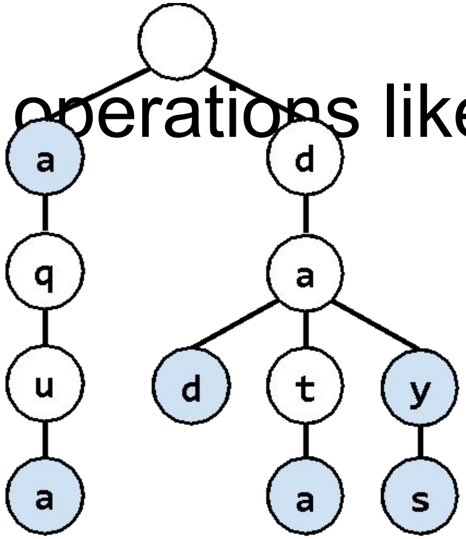
- Takeaways:
  + When keys are Strings, Tries give us a better `add` and `contains` runtime
  – `DataIndexedCharMap` takes up a lot of space by storing R links per node

Trie Takeaways

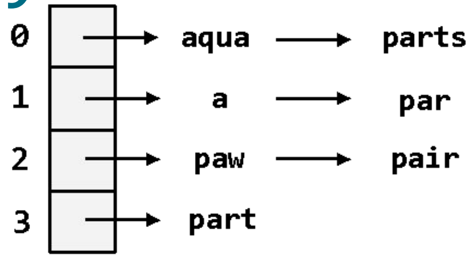
- Tries can be used for storing Strings (or any sequential data)
- Real-world performance often better than Hash Table or Search Tree
- Many different implementations: `DataIndexedCharMap`, Hash Tables, BSTs (and more possible data structures within nodes), and TSTs
- Tries enable efficient prefix operations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



Binary Search Tree



Trie



Hash Table

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*