CSE 374 Programming Concepts and Tools

Lecture 13 - C Data Structures

This Week

Today: C Data Structures

Next Week

- More on Variable Types and Storage
- Memory Architecture
- C++ Intro

Today

Makefile and structs review

Linked Lists

Trees and Tries

- T9!

HW5/6 overview

Review

Review: Makefile

`make` uses Makefiles to encode build processes

- Automate process for convenience and reliability
- Reduce unnecessary rebuilds
- Provide build options

```
target: source
    recipe
```

`make` relies on rules -- tuples of **[Target, Source, Recipe]**

`make` relies on timestamps and shell commands

- Language *in*dependent

Review: `struct`

Struct is a method of constructing new data types

- Store a collection of values together in memory, called fields
- Similar to a Java class, but *no methods*
- Individual values are referred to using the `.` operator
- Can use **`typedef`** to rename and turn struct tag into a "type"
  - Then you don't need keyword "`struct`"

```
typedef struct Dog {
  int age;
  char *name;
  char *breed;
} Dog;
```

```
Dog syrus;
syrus.age = 1;
syrus.name = "Syrus";
syrus.breed = "Alaskan Klee Kai";
```

Review: working with `struct`

Structs are passed as a copy by default, so it is more common to intentionally pass as *pointers*

- Avoids copying large objects
- Allows manipulation of original struct
- To access members you must `(*ptr).`
- Shortcut: **`ptr->`**

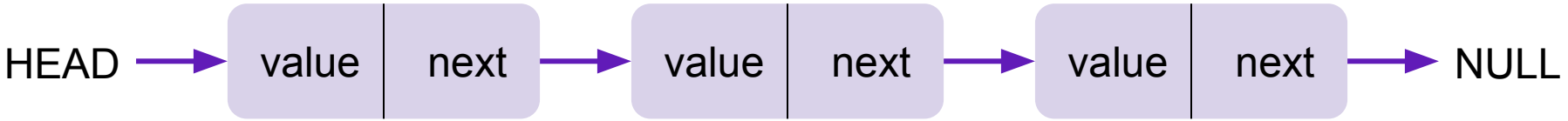
```
Dog* ptr = (Dog*)malloc(sizeof(Dog));
// not shown: initialize name…
ptr->age = 0;
ptr->age++;
char* name = ptr->name;
```

Data Structures

Linked List

Our good old friend (again)

Review: Linked List




```
typedef struct Node {
    int value;
    struct Node* next;
} Node;
```

Review: make_node

The `make_node` function creates and initializes a new node for a linked list.

- Dynamically allocates memory for a new `Node` structure on the heap
- Initializes the fields in the node
- Returns a pointer to the newly created and initialized node

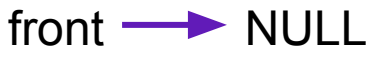
```
Node* make_node(int value, Node* next) {
    Node* node = (Node*)malloc(sizeof(Node));
    node->value = value;
    node->next = next;
    return node;
}
```

linkedlist.h

```c
// A single list node that stores an int as value
typedef struct Node {
 int value;
 struct Node* next;
} Node;
// Allocates a new node on the heap.
Node* make_node(int value, Node* next);
// Builds a heap-allocated linked list with the values in the array.
Node* from_array(int* array, int length);
// Frees all nodes in the linked list.
void free_list(Node* list);
// Prints the contents of the linked list.
void print_list(Node* list);
```

linkedlist.c: from_array

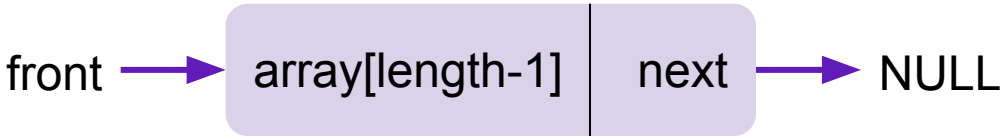
The `from_array` function builds a heap-allocated linked list with the values in the array.




```
Node* from_array(int* array, int length) {
    Node* front = NULL;
    for (int i = length - 1; i >= 0; i--) {
        front = make_node(array[i], front);
    }
    return front;
}
```

linkedlist.c: from_array (2)

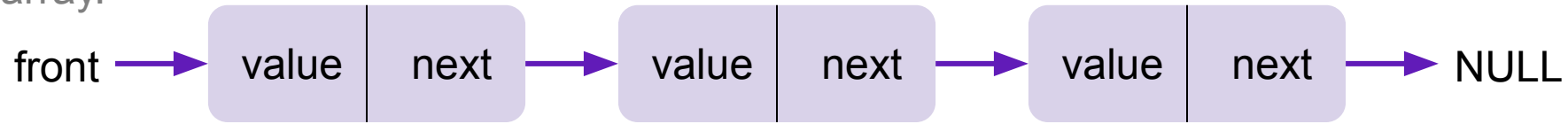
The `from_array` function builds a heap-allocated linked list with the values in the array.




```
Node* from_array(int* array, int length) {
    Node* front = NULL;
    for (int i = length - 1; i >= 0; i--) {
        front = make_node(array[i], front);
    }
    return front;
}
```

linkedlist.c: from_array (3)

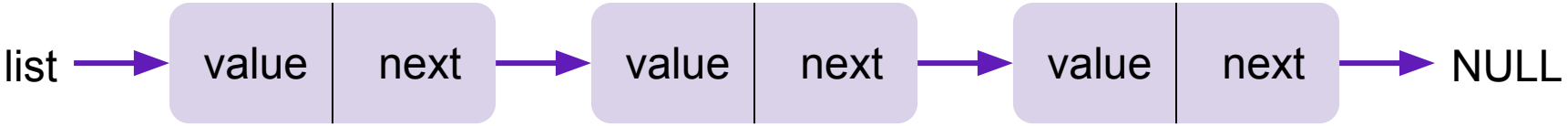
The `from_array` function builds a heap-allocated linked list with the values in the array.




```
Node* from_array(int* array, int length) {
    Node* front = NULL;
    for (int i = length - 1; i >= 0; i--) {
        front = make_node(array[i], front);
    }
    return front;
}
```

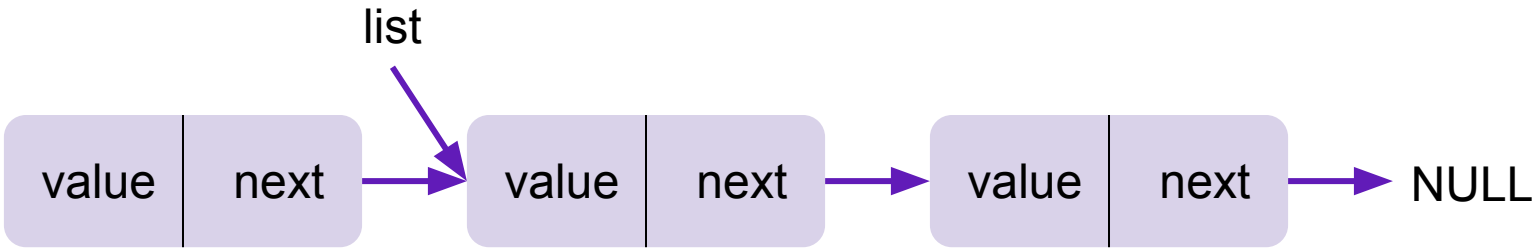
linkedlist.c: print_list

The `print_list` function prints the contents of the linked list.



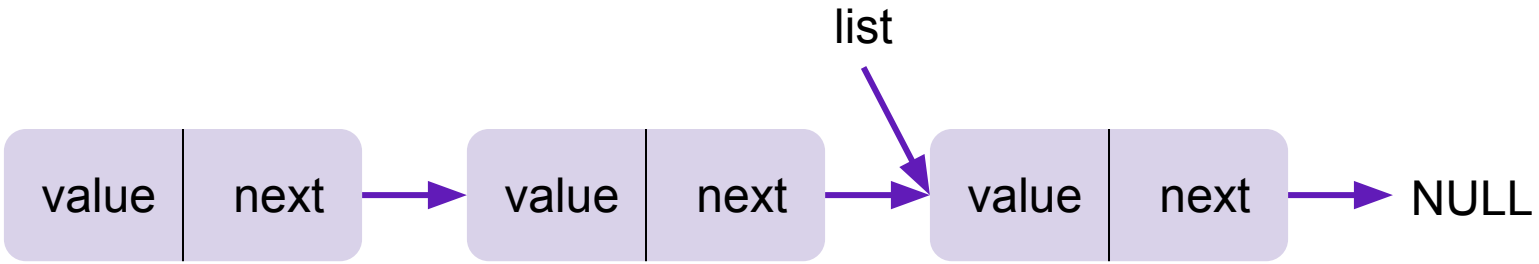

```
void print_list(Node* list) {
    printf("[");
    while (list != NULL) {
        printf(" %d", list->value);
        list = list->next;
    }
    printf(" ]\n");
}
```

linkedlist.c: print_list (2)




```
void print_list(Node* list) {
    printf("[");
    while (list != NULL) {
        printf(" %d", list->value);
        list = list->next;
    }
    printf(" ]\n");
}
```

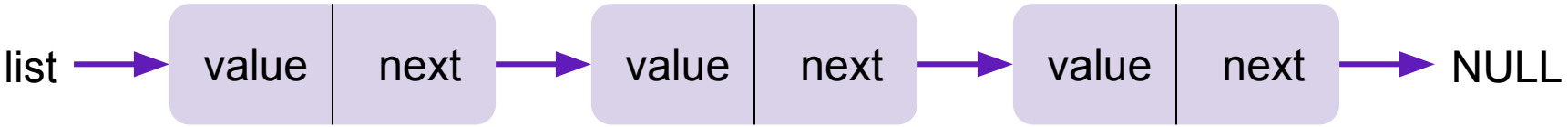
linkedlist.c: print_list (3)




```
void print_list(Node* list) {
    printf("[");
    while (list != NULL) {
        printf(" %d", list->value);
        list = list->next;
    }
    printf(" ]\n");
}
```

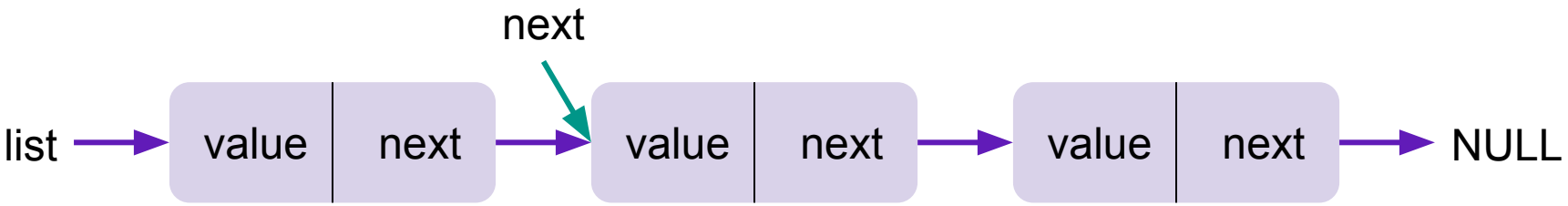
linkedlist.c: free_list

The `free_list` function frees all nodes in the linked list.



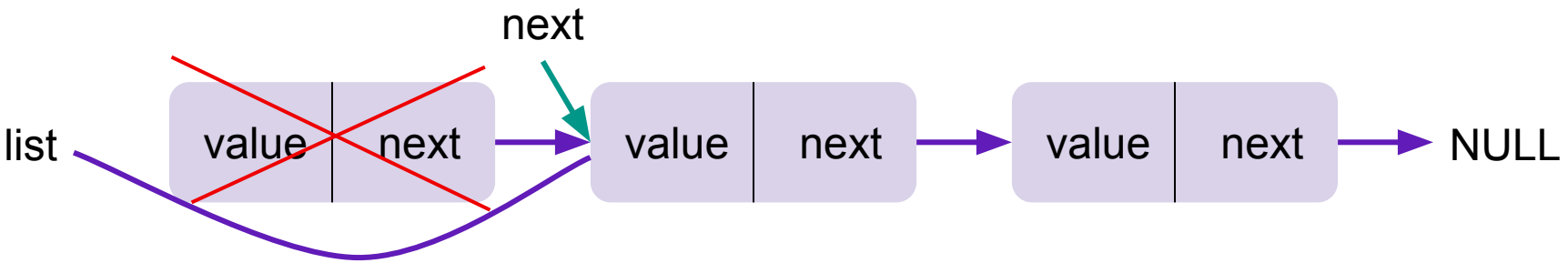

```
void free_list(Node* list) {
    while (list != NULL) {
        Node* next = list->next;
        free(list);
        list = next;
    }
}
```

linkedlist.c: free_list (2)



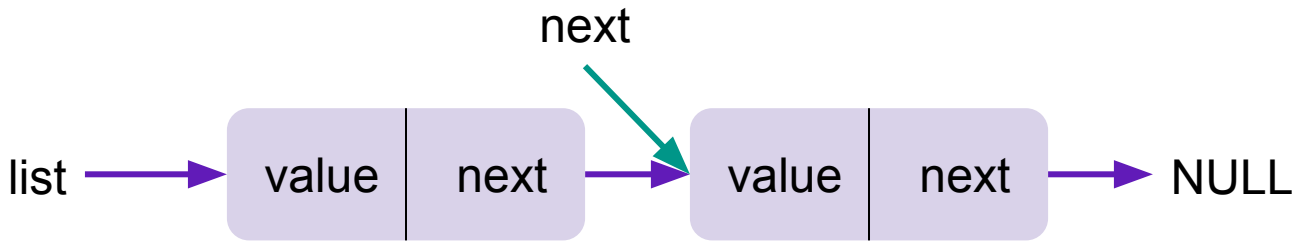

```
void free_list(Node* list) {
    while (list != NULL) {
        Node* next = list->next;
        free(list);
        list = next;
    }
}
```

linkedlist.c: free_list (3)



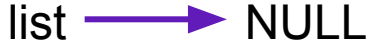

```
void free_list(Node* list) {
    while (list != NULL) {
        Node* next = list->next;
        free(list);
        list = next;
    }
}
```

linkedlist.c: free_list (4)




```
void free_list(Node* list) {
    while (list != NULL) {
        Node* next = list->next;
        free(list);
        list = next;
    }
}
```

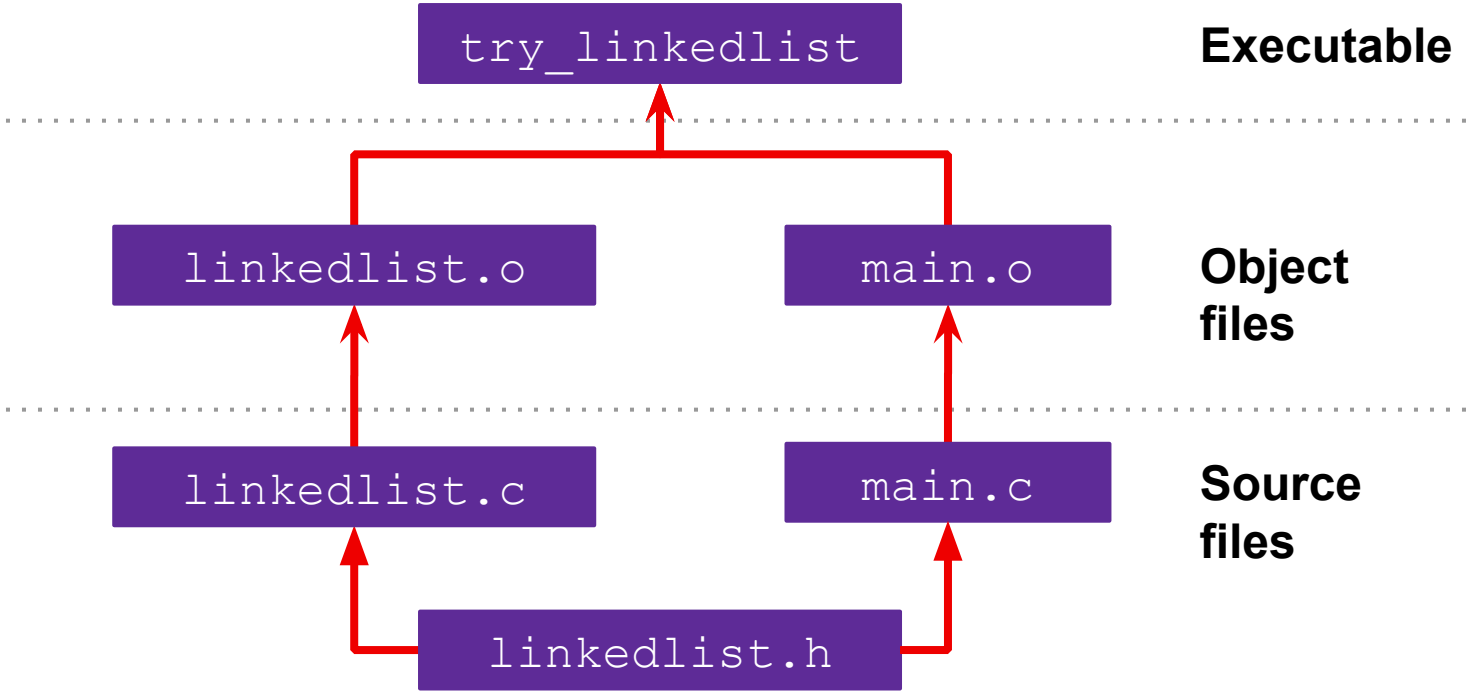
linkedlist.c: free_list (5)



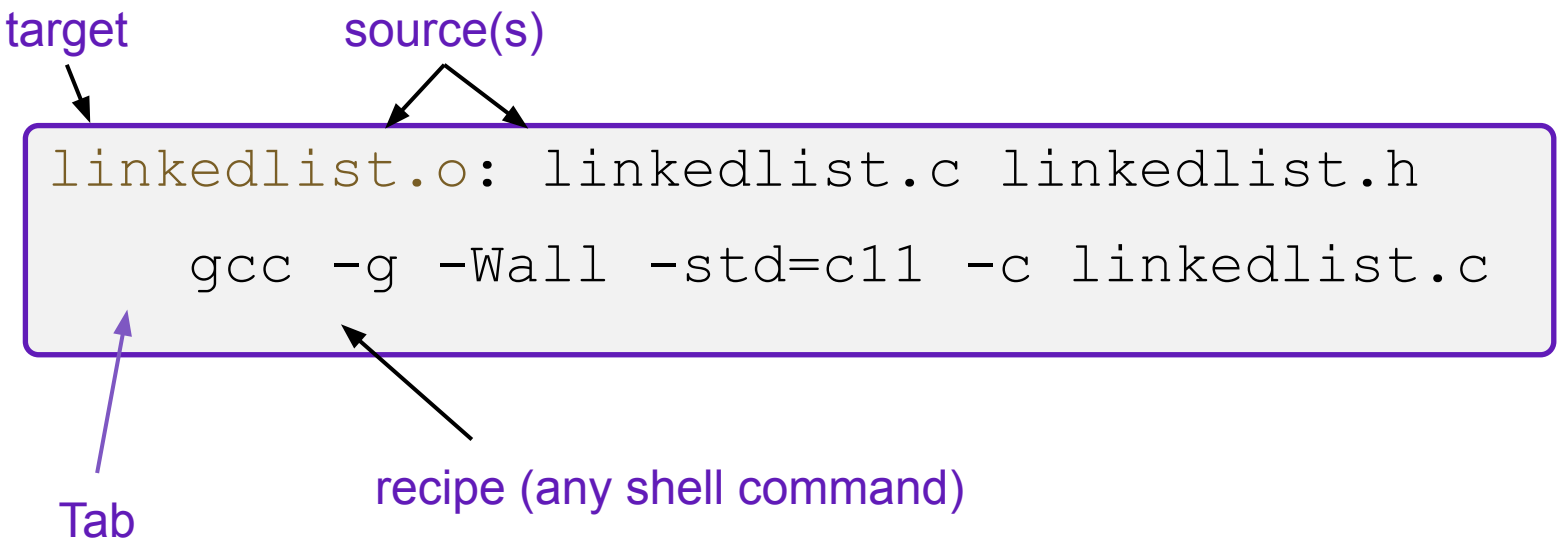

```
void free_list(Node* list) {
    while (list != NULL) {
        Node* next = list->next;
        free(list);
        list = next;
    }
}
```

LinkedList + Make Demo (if time)

Review: Dependency Graph for linked list



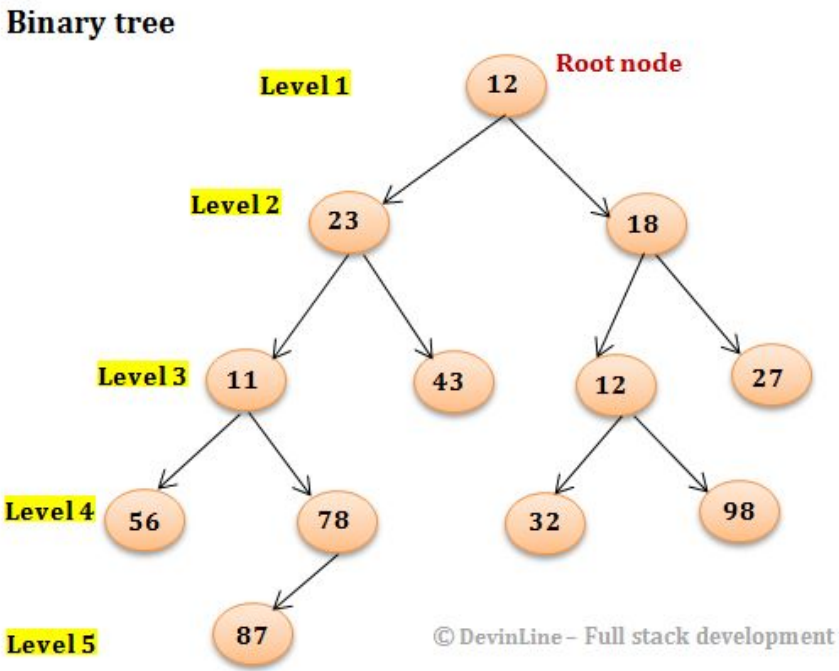
Rule Syntax


target
source(s)
linkedlist.o: linkedlist.c linkedlist.h
gcc -g -Wall -std=c11 -c linkedlist.c
Tab
recipe (any shell command)

Tree, Trie

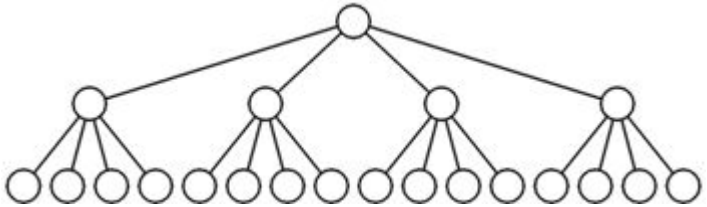
Linked Lists, but ✨next level✨

Binary Trees




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

struct BinaryTree {
  struct BinaryTreeNode* root;
};
```

N-Ary Tree



Binary trees just one form. You can have any "branching factor".

Trinary trees have branching number of 3.

For arbitrarily large branching numbers, arrays can make more sense than lists of named pointers.

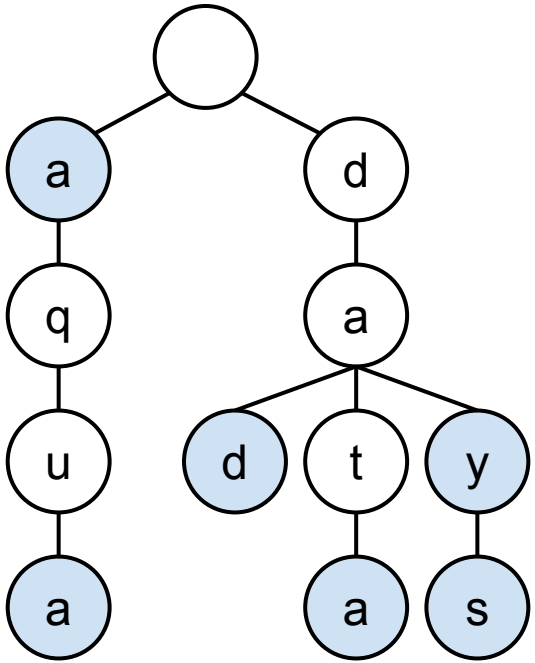
```
struct TrinaryTreeNode {
 char* data;
 struct TrinaryTreeNode* left;
 struct TrinaryTreeNode* middle;
 struct TrinaryTreeNode* right;
};

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

Prefix tree (Trie)

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

Nodes store parts of string instead of the full string

- Compact data storage
- Efficient worst-case searching
- Strings often use **26-ary tree**

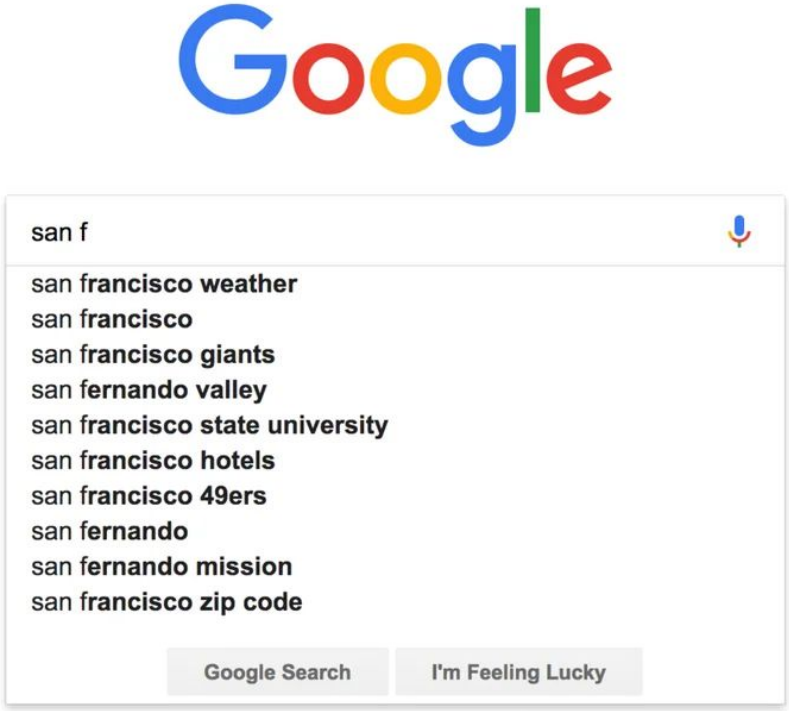


Use Case: Autocomplete

Search Engines support autocomplete.

How do you efficiently implement autocomplete with what 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.



Trie Index

Each level represents an index

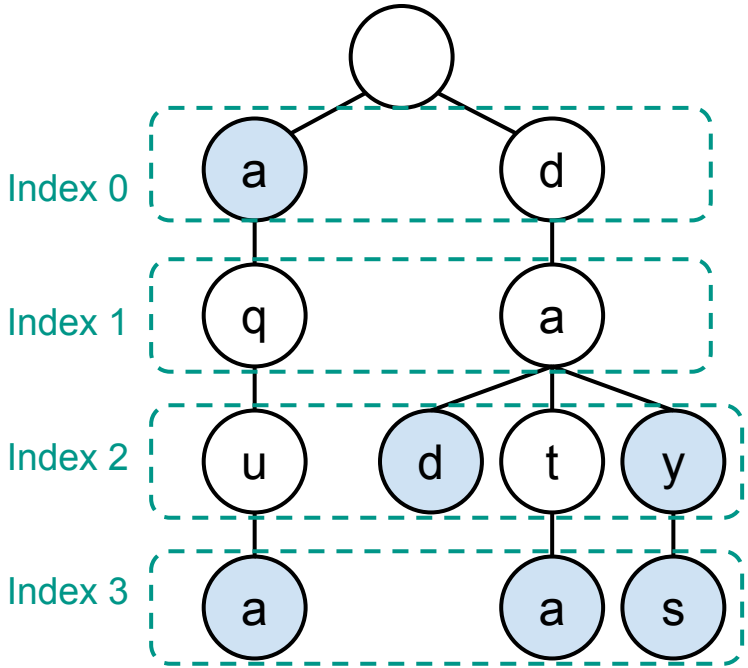
- Children represent next possible characters at that index

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 is in our set



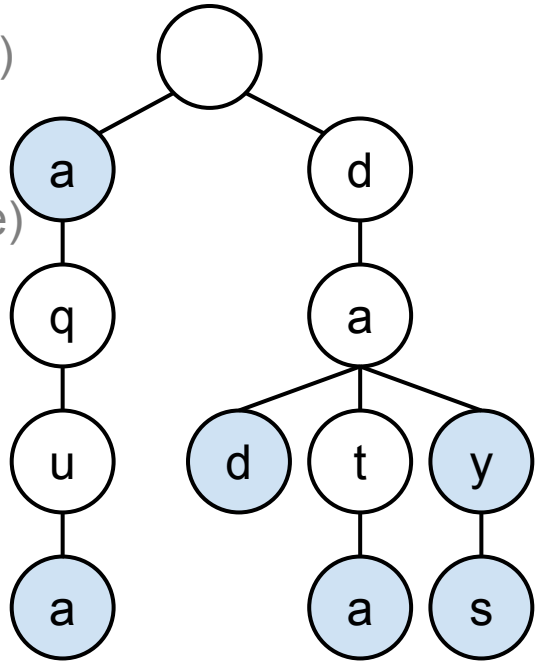
Searching in Tries

Search hit: the final node is a complete string (colored blue)

Search miss: caused in one of two ways

1. The final node is not a complete string (not colored blue)
2. We "fall" off the Trie (when branch not defined)

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

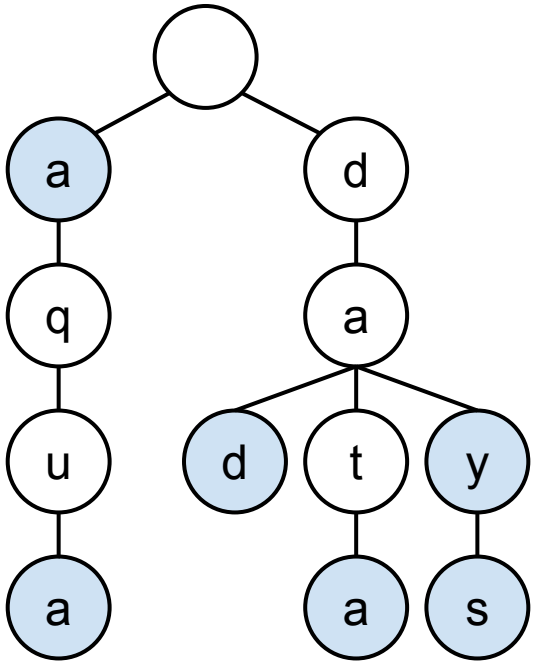


Prefix Operations with Tries

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

Prefix: find set of keys associated with given prefix

```
stringsWithPrefix("day") // "day", "days"
stringsWithPrefix("a")   // "a", "aqua"
```



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 "seattle", call `stringsWithPrefix("seattle")`
- Return the 10 strings with the highest relevance

"How do we determine these predictions? We look at the real searches that happen on Google and show common and trending ones relevant to the characters that are entered and also related to your location and previous searches."

Danny Sullivan, Public Liaison for Google Search (source)

Polling Time!

Please answer in the LP13 assignment on Gradescope!

What is the main *advantage* of a **trie** data structure?

- Efficient search, insert, and delete operations for sorted arrays.
- Fast look-up for unordered lists.
- Reduced space complexity.
- Efficient prefix-based operations and autocomplete functionality.

Assignment Reminders

EX11 due Friday 7/25 **@10:49am**

HW4: Debugging is due this **Friday 7/25 @11:59pm**

- Basic premise: find the memory errors in the given program, and fix them!
- You'll get practice hunting down these bugs using `valgrind` and `gdb`

Mid-Course Survey due on **Sunday 7/27 @11:59pm**

- Let us know how this course is going so far!

Make sure to submit LP13 before the next class as well.

All of these assignments can be found on our course's Gradescope page.

Testing

Testing Framework

Many programming languages have a *testing framework*

- This is a library/tool that provides utility functions for testing, as well as ways of organizing tests
- Examples: JUnit, PyUnit, etc.

We will give you a testing framework: `safe_assert.h`

In order to use it, you have to #include it: `#include "safe_assert.h"`

- You'll be using this in HW5!

safe_assert Testing Syntax

Groups of tests are organized into `suite(...) { … }` blocks

Individual tests are kept within `test(...) { … }` blocks

You can make assertions that a boolean must be true using `safe_assert(...)`

```
suite("My test suite") {
  test("Test case 1") {
    int res = pow(10, 2); // 10^2 = 100
    safe_assert(res == 100);
  }
}
```