Week 6

This is CS50. Harvard University. Fall 2014.

Cheng Gong

Table of Contents

Announcements

- We had some audio issues that turned out to be a bad cable, so we had some quick photo and question-and-answer opportunities with the handful of us who were there. (We get really casual on Fridays.)
    # Fun fact, we have data from last year that shows lecture attendance from lecture to lecture, with attendance increasing slightly every Monday, but declining by Wednesday and generally from week to week.
    # The one lecture on Friday saw a huge drop!
- Today we'll look more at data structures for Problem Set 5: Mispellings, where we give you a dictionary of words to load into memory.

Linked List Recap

- One way to do this is with a linked list. The linked list we introduced last time had at least one advantage over arrays.
    # Dynamic insertion is possible with a linked list, meaning we can place a new element anywhere inside the list without having to shuffle anything, since it takes a lot of

time to move those elements around. But with a linked list, it's just a `malloc` and updating a few pointers.

    # Linked lists can also be resized easily, meaning you're not committing some fixed size of memory like an array, so you can use exactly as much memory as you need. But with an array, you might accidentally allocate too little, and need to make a new array, or accidentally allocate too much and waste memory.

- Dynamism and flexibility are the advantages, but with that said:

    # There is no random access, so algorithms like binary search can't just find the middle element and jump to it. To find an element, we would have to start at the first one and linearly search through the list.

    # The pointers in each node take up additional memory, so a linked list for integers will use up twice as much memory with the addition of the pointer. This becomes less significant as your nodes start to include more fields, and the relative size of the pointer to the next node will diminish.

- Recall that insertion, search, and deletion on a linked list are #(*n*), or linear, as the element could be at the end. You might get lucky and find the element as the first in the list, for a lower bound of $\Omega(1)$.
- But we want to be able to have #(*1*), finding or adding or removing elements in constant time.

Hash Tables

- Let's look at some blue books, each labeled with some letter of the alphabet. If they were shuffled and we started to sort them, we might instinctively start putting them into piles based on the first letter, and later go sort each pile individually.

    # The idea is that we look at the input and make a decision based on that input (*A* goes in this pile over here, *Z* goes in that pile over there). This technique is known as **hashing**, or taking an input and computing a value, generally a number, that becomes the index into a storage container (like an array).

    # We might map *A* to 0, and *Z* to 25, and place them into numbered piles that way.

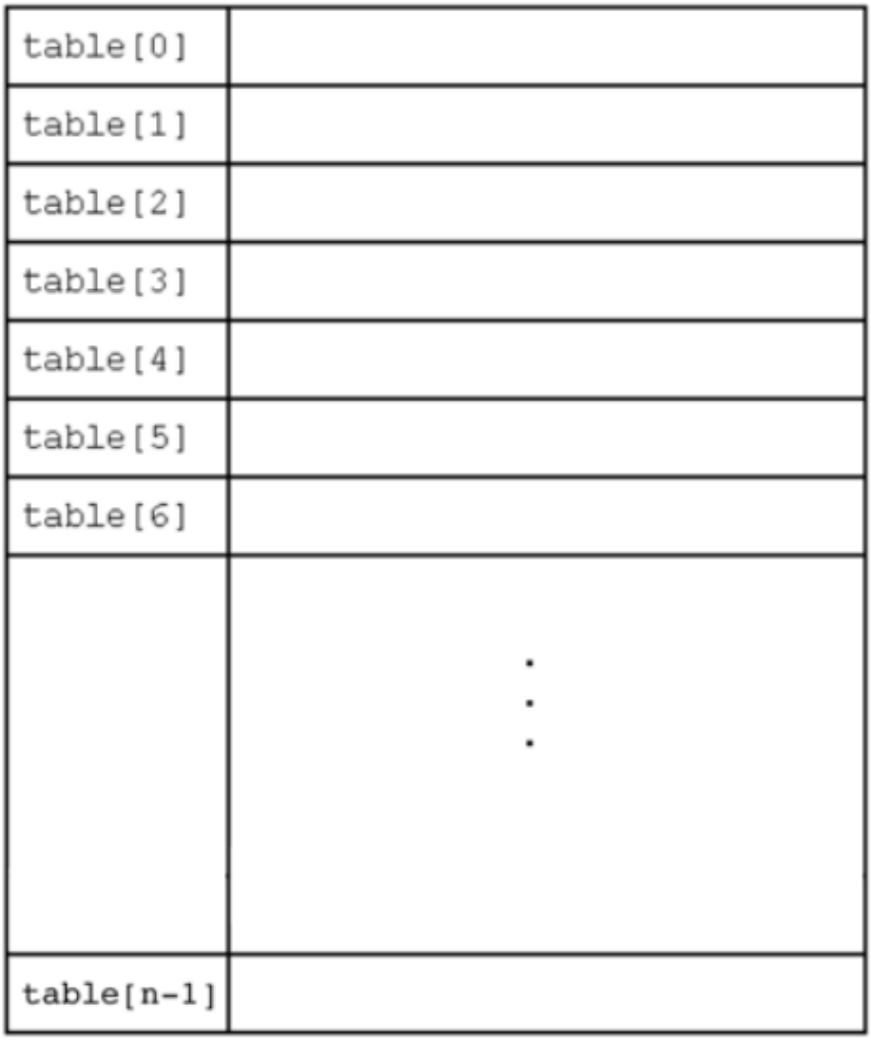
    # A C program could just use the ASCII value, but subtract 65 from a capital *A*, or 97 from a lowercase *a*, so we can place them in buckets starting from 0.

    # We actually do this with quizzes, placing them into piles based on TF, so each TF can find their students' quizzes easily.

    # During the CS50 Hackathon last year, we also separated nametags into piles by alphabet, so greeters like Zamyla could find them more easily as people arrived.

        # One TF made a clever sign reading "Hash Yourself (by First Name)."

- Let’s formalize this idea. The following is an array with the index of each row on the left:

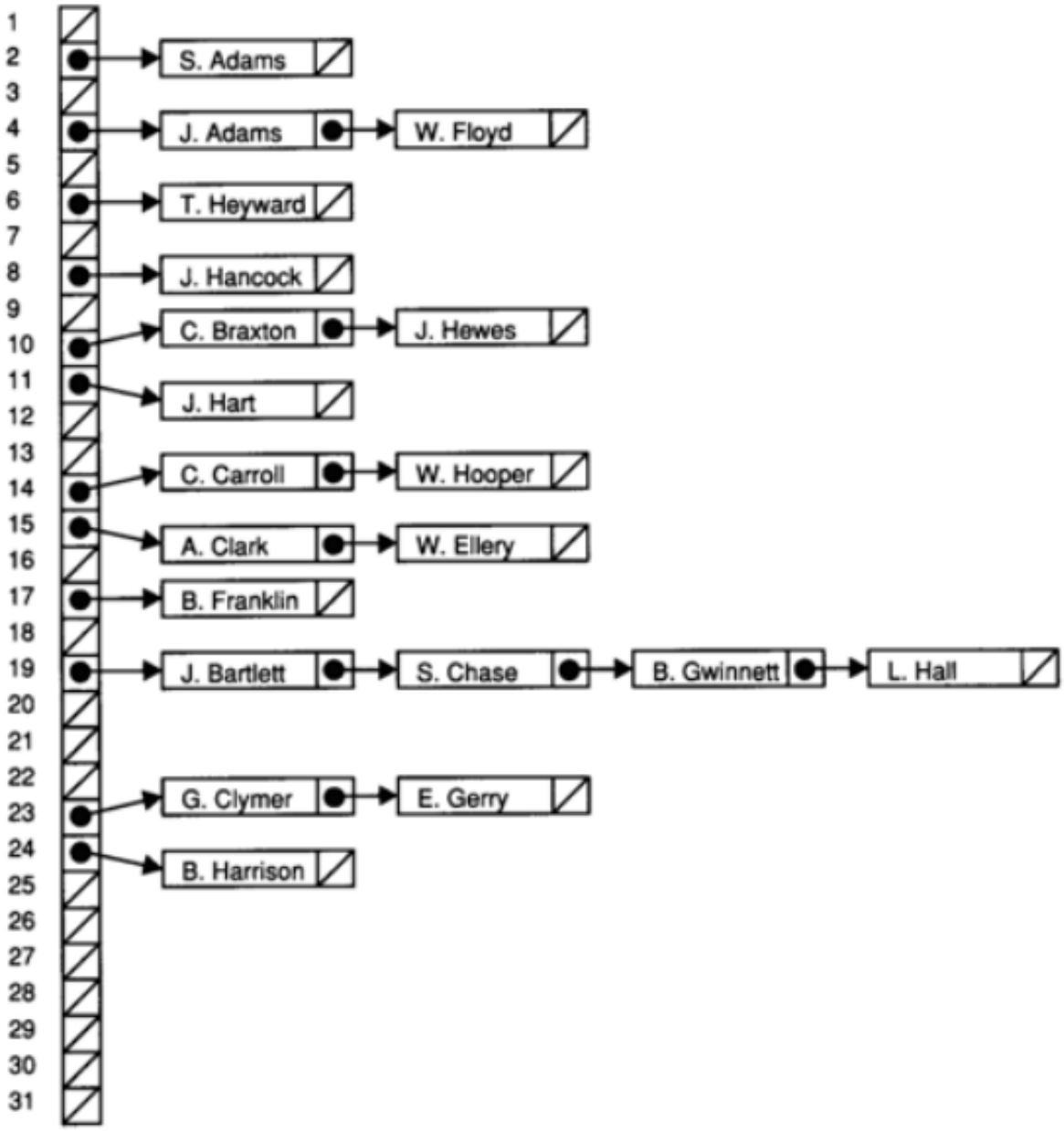


    # This `array` has 26 elements and is named `table`.

- This is one implementation of a **hash table**, a higher-level data structure, which could use an array or linked list or some other fundamental ingredient as a building block to making a more useful final result.
- We can simply declare a hash table with something like `char* table[CAPACITY];`, in which case it will be an array with size `CAPACITY`, but we might call a hash table an **abstract data type** that is layering on top of this array.
- Earlier we used a physical table to place piles of blue books by the letters they were labeled. But the table is actually a fixed size, and if we had two elements of the same number, we couldn't really place them into the same spot if the table was really an array.
- So maybe we could place it into the next spot, where a book with the letter *B* would go. But now the *B* will go lower, and could become problematic, but is a technique called **linear probing** where we look at each element in the array, looking for an available spot to put the next one.
- Insertion starts by being #(*1*) since the array is empty and you can jump to the correct location immediately, but once your data set becomes larger, insertion becomes #(*n*) since we have to move down the array looking for a spot. If your array is big enough and your data is sparse enough, you get the advantage of constant time, but the risk makes linear probing imperfect.

Separate Chaining

- We can do better by using linked lists. Let's look at this picture of **separate chaining**.



This image, from some dusty textbook of David's era (as you can tell by the old-fashioned names), is an array of size 31, with strings hashed not by the letters they are made of, but the day of the month the person was born in, so the names might be spread out a bit more evenly.

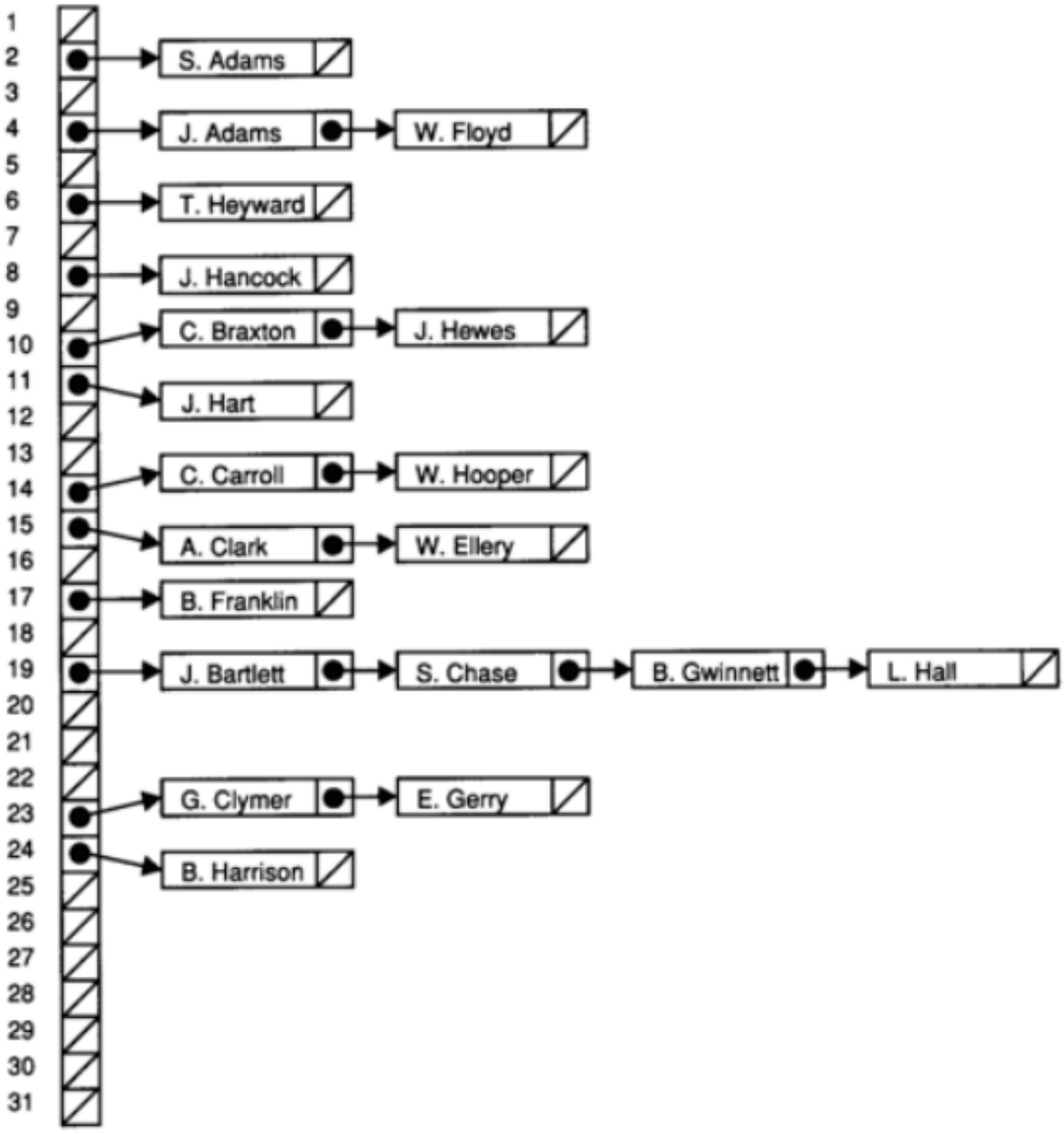
- But this is still troublesome since people certainly have the same day of the month they were born on, so it looks like we have an array on the left side (drawn vertically), that looks like it's an array of linked lists.
  # As we get more advanced, we can start combining fundamental ingredients into things like this, with basic blocks of arrays and linked lists and structs combined to create a more sophisticated data structure.
- The hash table above is an array with "chains" that can grow or shrink based on the elements you insert and remove.
- The running time now for insertion still becomes #(*n*), for example if we run into lots of people born on the 31st, but it's a little better than linear probing. In the worst case, say everyone was born on the 31st, the chain will be *n* people long, but in general if birthdays are close to being uniform, the chains will eventually be about the same length and we can claim we have #(*n*/31). Though technically we have to look at the biggest term and simplify that to #(*n*) because *n*/31 asympototically approaches *n* as *n* gets bigger and bigger, in reality (as you'll see in Problem Set 5), this algorithm will be about 31 times faster.
- Realize that theoretical analysis has value, but in the real world small differences can make things much faster.
- We could even combine names and birthdays somehow to create a larger hash table that's even more uniform and have smaller chains, and be even faster.
- Let's borrow code from before:

```
typedef struct node
{
    char* word;
    struct node* next;
}
node;
```

  # So we start by creating `node`s for storing strings (remember that we've taken off the training wheels of the CS50 Library and revealed them to be `char*`s). In this case, we'll plan on storing a person's name in the string `word`. And these `node`s will represent a node in the chains of a linked list.
- Now we have to declare the whole structure of the hash table:

```
node* table[CAPACITY];
```

    # We'll have an array with each element being a pointer that points to the first element of the linked list in that spot. Let's take a look at the picture again:

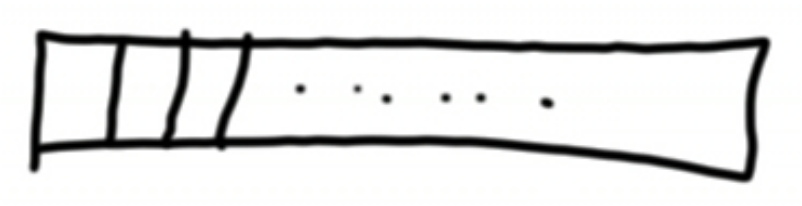


    # The slots with a slash through them are pointers to `NULL`, and the ones that have a linked list will only store the pointer to the first element of that list.

- Let's try to get a data structure with constant time. The hash table with separate chaining is not quite there yet.

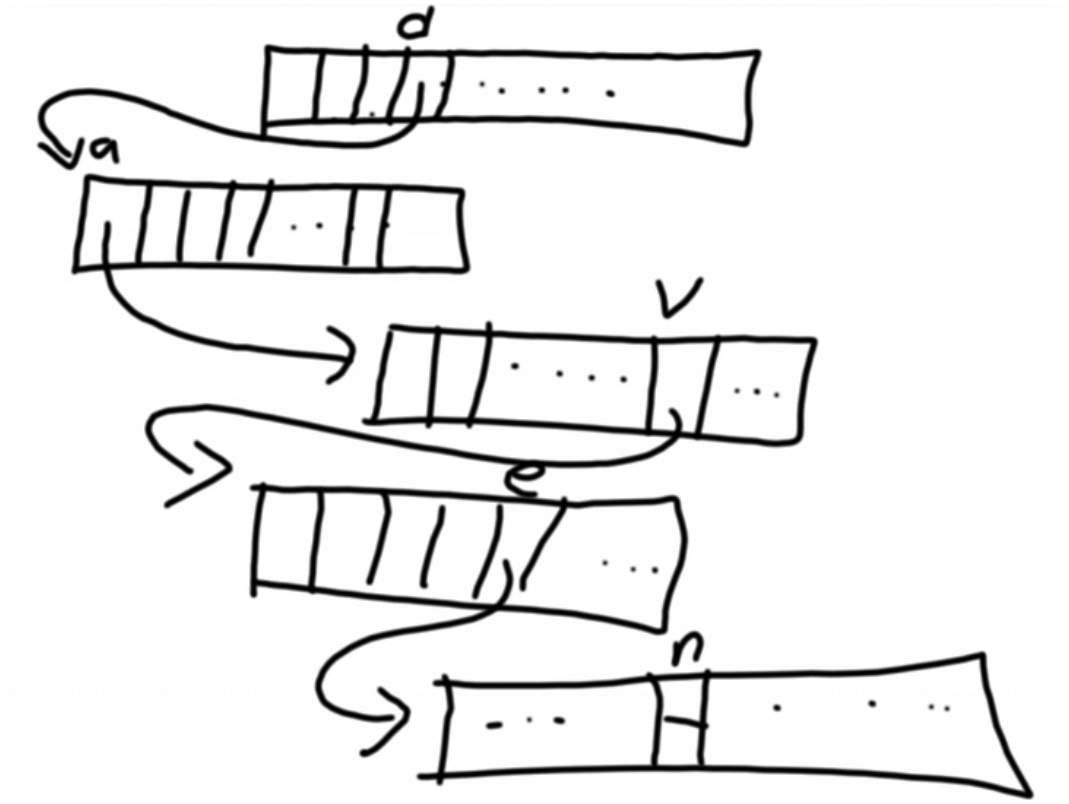
Trees and Tries

- Instead of a hash table, we can use a tree, like a family tree. If we started with an empty tree, we can create a node that looks like this:

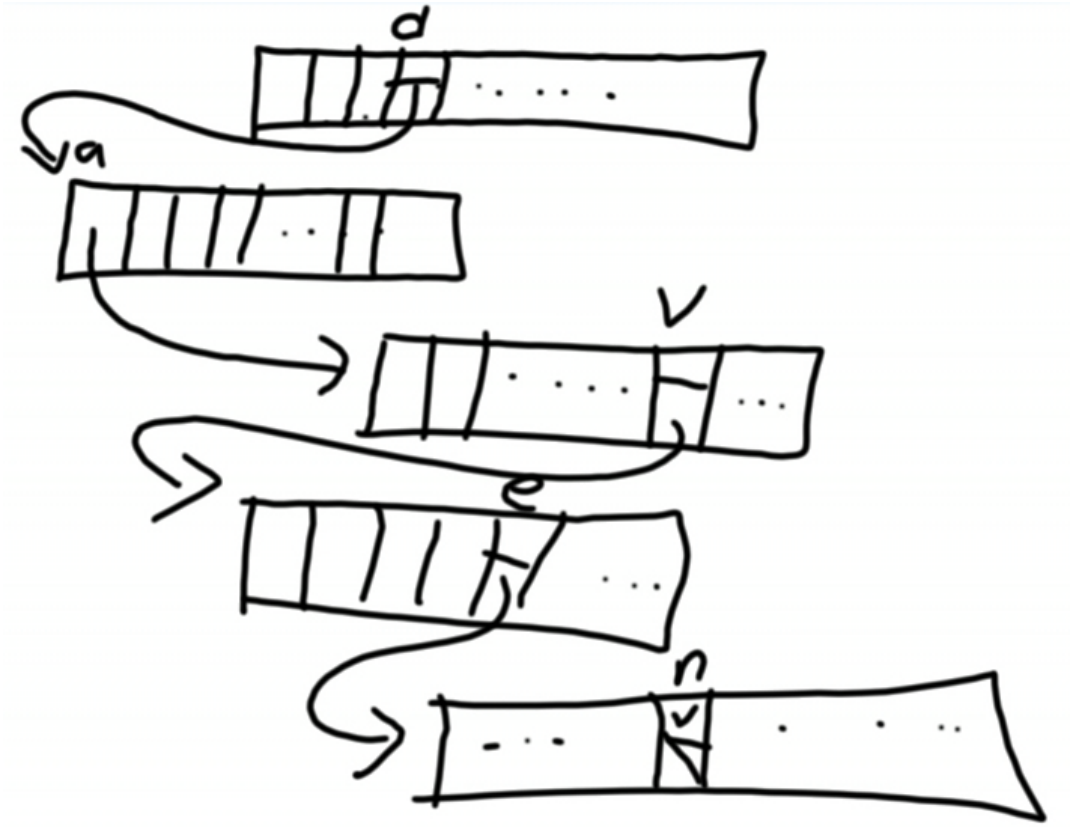


  # And each box within this rectangle (which has a total of 26 elements in it) will be a letter of the alphabet. Specifically, the first box will be the letter A, the second B, and so on.

- If we wanted to insert the name of "Daven", we'd first need to hash his name and realize it starts with D. But we can't fit Daven's entire name into the rectangle, so we can build a tree by allocating another node (rectangle) for each letter, and setting the pointers of each letter to the next node like this:

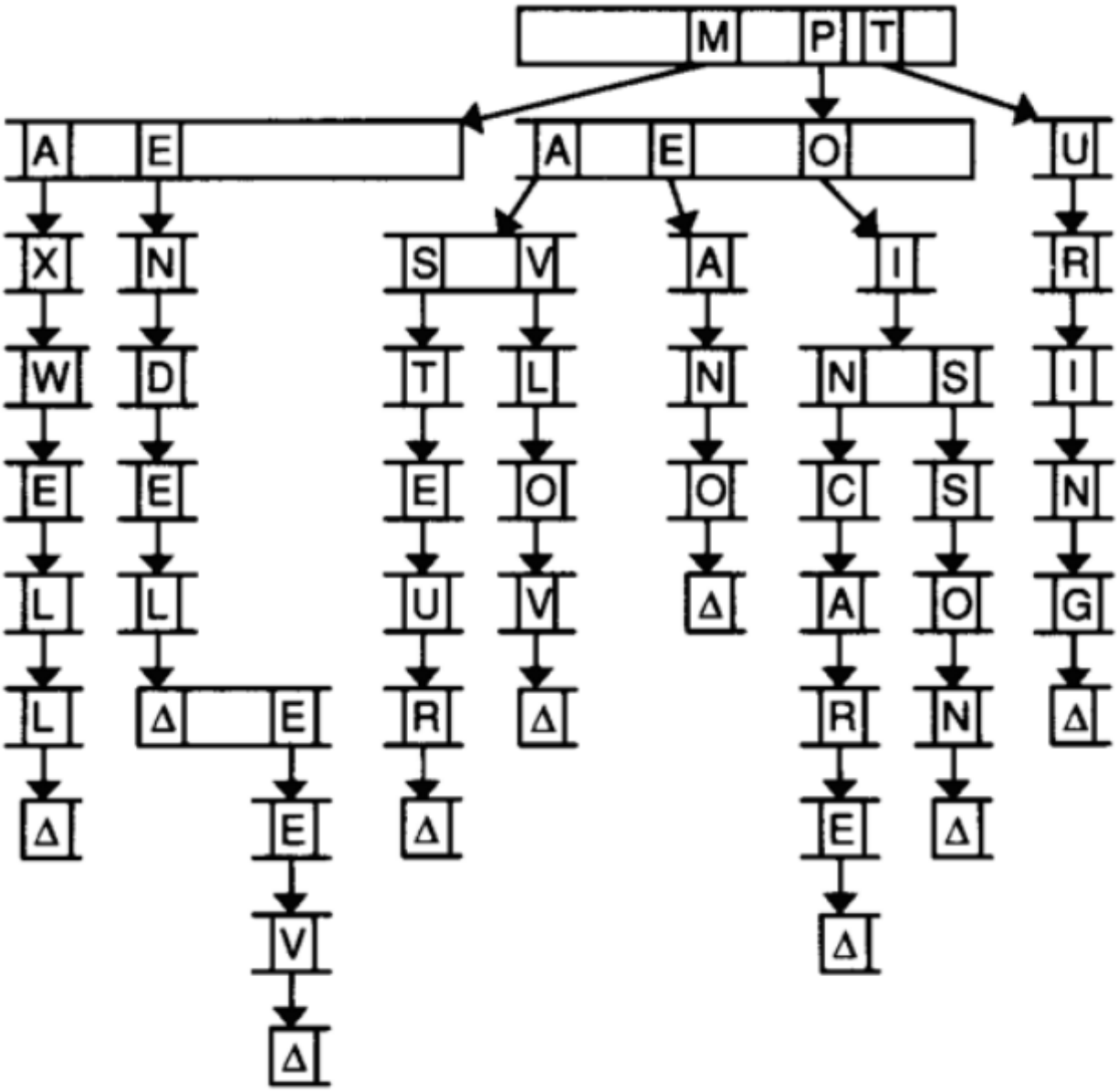


- For the final letter in Daven's name, we don't want to put just `NULL` since `Daven` is actually short for `Davenport` and we want to accept both strings, so we should actually split each box in all these rectangles into two. Here, we've only drawn the last box, labeled *n*, as containing two fields. We can use the top one as a signal for the end of a string, and the bottom one as the pointer:



- This looks complicated, and already messy because of David's handwriting, so consider a tree that looks like this:[1]

[1] Figure from Lewis and Denenberg's Data Structures & Their Algorithms.



Each rectangle has 27 indices, because we'll allow for a word to have an apostrophe.

- We'll start to call this data structure a **trie**, a clever name for a tree optimized for retrieval, but that's spelled with "ie", so it ends up being pronounced like "try." Go figure.
- These tries use a lot of memory since each node allocates space for so many indices, and most of that space will be empty, at least initially when we have few strings.

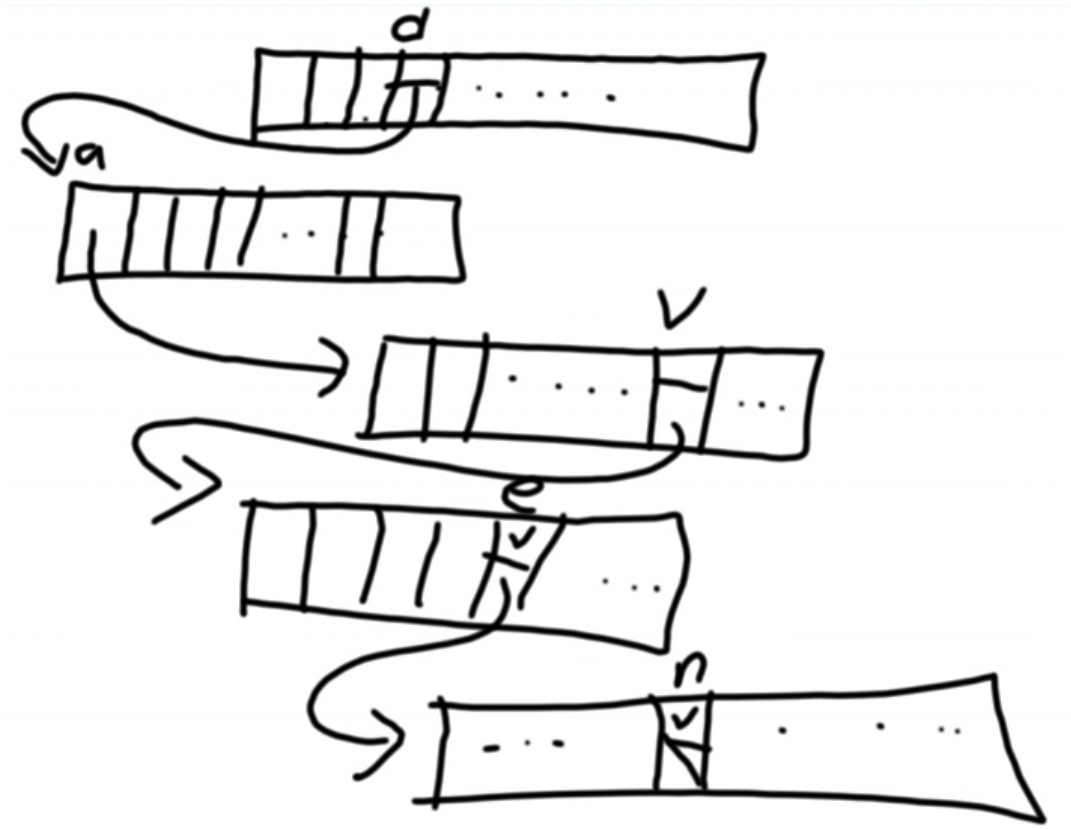
- But we gain speed and spend less time, since insertion time will be #(1), since no one has an infinitely long name. The longest word in the dictionary for the problem set will be 40-something letters long, but it will be constant and not depend on how many other names are in this data structure. The running time will now depend on the length of the string, which is asymptotically #(1).
- We might implement this with something like this:

```
typedef struct node
{
    bool word;
    struct node* children[27];
}
node;

node* trie;
```

  # Each `node` will have a `bool` indicating whether it could be the end of a word, and an `array` with `27` `node` pointers.

- To declare our trie, all we need is a single `node*` that points to the **root** element, just like a linked list.
- Looking back to our chart, if we wanted to insert a name like "Dave," all we need to do is follow the pointers and change the box to mark that we are at the end of a word:



Stacks and Queues

- Remember that trays from dining halls can be stacked, and there exists a data structure called a **stack** where you can perform two operations, push (adding a tray to the stack) and pop (taking a tray from the top).
- We have this property where the last item in is the first item out (LIFO) with interesting applications that we'll look at another time.
- A similar idea is a **queue** where the first item in is the first item out, (FIFO), enqueue (adding) and dequeue (subtracting).

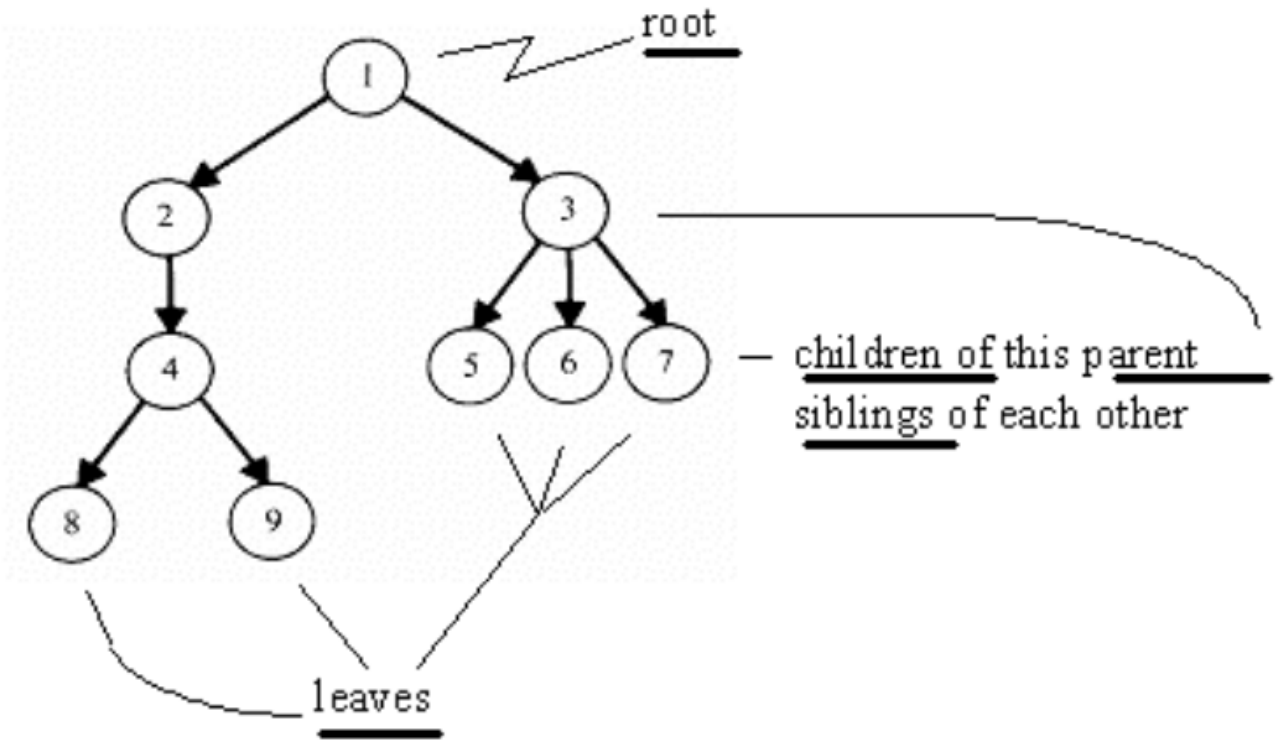
  # Imagine a line where the person waiting first gets to go first — that's a queue.

- We won't really go into the code but if we think abstractly about adding people to a line one at a time, a linked list would work since we can add spots and people as needed, but an array would also work if we knew we had a limited number of spots.

- So we're reaching the point of having higher-level problems that can be implemented in a number of ways, giving us the flexibility of choosing between tradeoffs like time and space and code complexity.
- A stack can be an array (in Mather the trays in the dining hall are in an opening in a cabinet of a fixed height, so it's almost as though they're using an array) but can be a linked list too.

Trees Again

- Let's look at one more data structure, a simpler tree like this:[2]



- Each node is just a number, with other words like **leaves** and **children** and **parents** and **siblings** as illustrated.
- One way to implement a tree in code might look something like this:

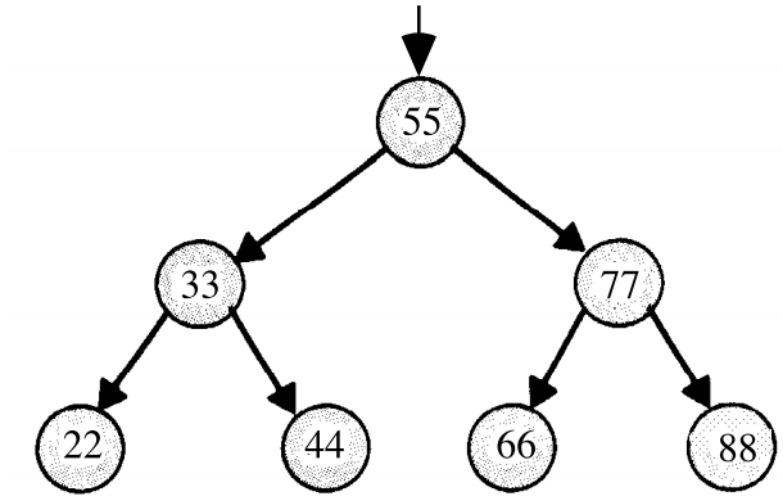
[2] Figure by Larry Nyhoff.

```
typedef struct node
{
    int n;
    struct node* left;
    struct node* right;
}
node;
```

Notice that we have set aside memory in each node to store two pointers, allowing us to build something like a two-dimensional linked list that follows a pattern.

This structure is still a tree, as it has no cycles, with a grandparent at the very top, and then parents and children and grandchilddren and so forth.

- Thinking back to recursion, if we put the integers in a sorted way like this, we'll have a **binary search tree**:[3]



Smaller numbers are on the left and bigger ones are on the right, so a parent is greater than its left child, but less than its right child, and this will only reach a base case when we are at the leaves, or the nodes at the very bottom of the tree, which have no more children.

[3] Figure from http://cs.calvin.edu/books/c++/ds/1e/.

    # To find the number 44 in this tree, we would go left from 55 since 55 > 44, then see that the child is 33, and go right, since 33 < 44.

- So we can implement the binary search algorithm a bit more elegantly than an array would allow, with code like this:

```
bool search(int n, node* tree)
{
    if(tree == NULL)
    {
        return false;
    }
    else if (n < tree->n)
    {
        return search(n, tree->left);
    }
    else if (n > tree->n)
    {
        return search(n, tree->right);
    }
    else
    {
        return true;
    }
}
```

    # We are looking for `n`, given `tree`, which is a `node*` to the root of the tree. We implement the logic fairly straightforwardly.

    # If `tree` is `NULL`, then there are no nodes so we return false.

    # The middle two conditions compare `n`, the value we are given, to `tree->n`, which dereferences `tree` and accesses the `n` field stored within. If the `n` we are looking for is less, we search again with the left child of the tree by calling `search(n, tree->left)`, which makes our problem smaller. Likewise, if `n` is greater, we search with the right child of the tree.

    # Finally, if the `tree` is not `NULL` and `n` is neither greater than or less than `tree->n`, then we found the value and can `return true`.

- So we introduce these data structures for you to evaluate and use in Problem Set 5, but let's conclude on a teaser[4] of how the real world implements web applications, as we transition to using higher-level languages.

[4] http://youtu.be/Cb8b1RMX6XY