Computer Science 50
Fall 2008
Scribe Notes

Week 7 Wednesday: November 5, 2008
Andrew Sellergren, Anjuli Kannan

Contents

Computer Science 50
Fall 2008
Scribe Notes

Week 7 Wednesday: November 5, 2008
Andrew Sellergren, Anjuli Kannan

1 Announcements (0:00–2:00)

- *The Simpsons* is the greatest show on television.
- Lunch with David is back this week![1] E-mail `rsvp@cs50.net` if you're interested.
- Use the Bulletin Board! Your question might already be answered, or, if not, it *will* be answered by a member of the staff or perhaps even a fellow student.
- Only three announcements this week![2]

2 Hash Tables and Linked Lists (2:00–37:00)

- Last week we looked at linked lists. Recall what the asymptotic running time of searching a linked list? $O(n)$. What advantage did they have, though? We have a lot more control over insert and delete as well as how much memory our data structure takes up.
- Can we achieve constant time for search or insert? Hash tables get us pretty darn close, though they still have collisions. Remember the birthday problem.
- With coalesce chaining, we keep a small amount of metadata that allows us to remember where we hashed to when we inserted an element. Think of them as bread crumbs.
- In its typical implementation, a hash table is not much different from an array–it has a fixed size. Each of its elements, however, is a pointer to the beginning of a linked list.
- Linked lists, if you recall, are implemented as connected nodes, each of which holds some value as well as a pointer to the next node.
- Say we have a node which holds a `char *` and a `next` pointer. How big is this node? 8 bytes, 4 for each pointer!
- More accurately, though, this doesn't capture how much space our linked list takes up, because the names themselves have to be stored somewhere, as well. Instead, we could define a node as containing a `next` pointer and an array of `char`'s. To be safe, we'll declare an array of size 1024 just in case someone has a really long name.[3] Specifying this fixed size for name storage will cost us in memory, but will save us in speed, since we won't have to make multiple calls to `malloc`, as we would if we use a `char *`.

[1] ☺

[2] Those of you who thought to yourself "Actually, four," congratulations! Click here to claim your prize!

[3] Say, Apu Nahasapeemapetilon or Selma Bouvier-Terwilliger-Hutz-McClure-Simpson, to take two examples from real life.

- Once we start storing more than just a value in each of our nodes, we should begin to think about defining another structure to store our data so that it doesn't get mixed in with our metadata:

```
typedef struct
{
    int id;
    char *name;
    char *house;
}
student;

typedef struct node
{
    student *student;
    struct node *next;
}
node;
```

  Why did we write the word `node` before the second set of curly braces, but not the word `student` before the first set of curly braces? We need a temporary name so that in the second `struct` definition, we can tell the compiler what type of data structure the `next` pointer will be pointing to.

- Looking at the `traverse()` function from last week's `list2.c`:

```
void
traverse()
{
    // traverse list
    printf("\nLIST IS NOW: ");
    node *ptr = first;
    while (ptr != NULL)
    {
        printf("%s of %s (%d)  ",
               ptr->student->name, ptr->student->house, ptr->student->id);
        ptr = ptr->next;
    }

    // flush standard output since we haven't outputted any newlines yet
    fflush(stdout);

    // pause before continuing
    sleep(1);
    printf("\n\n");
}
```

Notice the use of arrow notation. What is `first`? Is it a local variable? No, in fact, it's a global, and it's our entire linked list! Or rather, it's all we need to keep track of in order to access the linked list. Why did we write `ptr->student->name` and not `ptr->student.name`? Quite simply because `student` is a pointer! We need to dereference it before we access its contents.

- This week, you'll be making design decisions in order to optimize the search time of a spell checker. If you want it to be as fast as possible, you might think about implementing a hash table.
- What exactly is a hash function? It's a mathematical method of determining an index into the array of linked lists which is our hash table.
- If we're hashing an `int`, an ID number, for example, then using the modulo operator seems like a pretty clear choice. What if we're hashing a string, however?
- We could take the first letter, cast it to an `int` and begin with something similar to this:

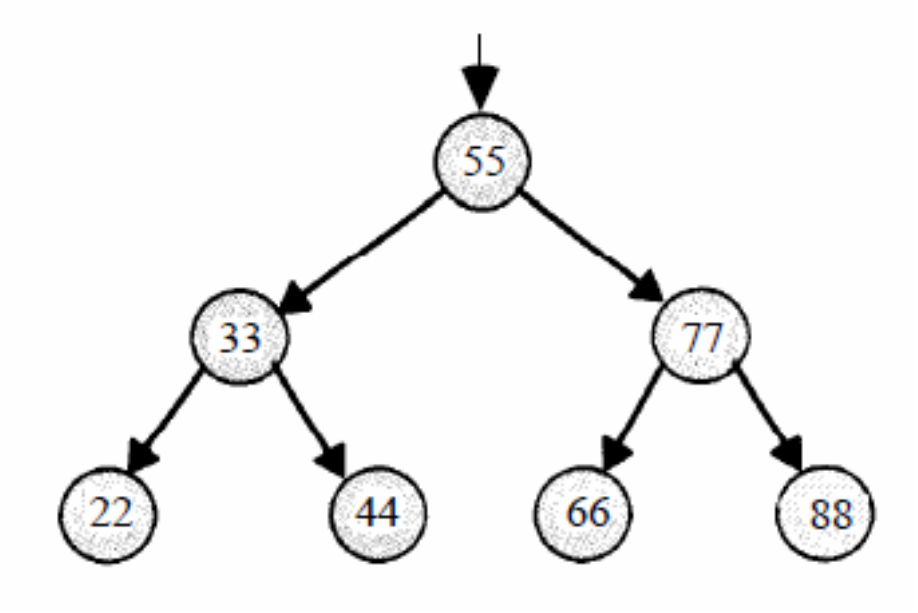
```
int
hash(char *s)
{
    char c = s[0];
    return ((int) c % 6);
}
```

Can this code crash? Most certainly yes! We should think about checking for `NULL`, although you should realize that it's not very easy to come up with an integer value which will obviously signify an error and yet *not* be a valid index into our array!

- A minor matter: it might be a good idea to invoke `toupper()` so as to standardize inputs and make sure that uppercase and lowercase are treated similar. (A name lazily entered in lowercase is still the same name.)
- What are the advantages of this function? It's simple, easy-to-code, deterministic. And the disadvantages? There probably aren't as many people whose names start with Q as there are people whose names start with A or B. So we're going to get clustering around certain indices.
- How do we fix this? We could try the second letter of the name, but it's probably distributed as unevenly as the first letter. Why not hash the entire name? We could walk through and sum through the letters of the name to come up with a relatively unique number for each name.

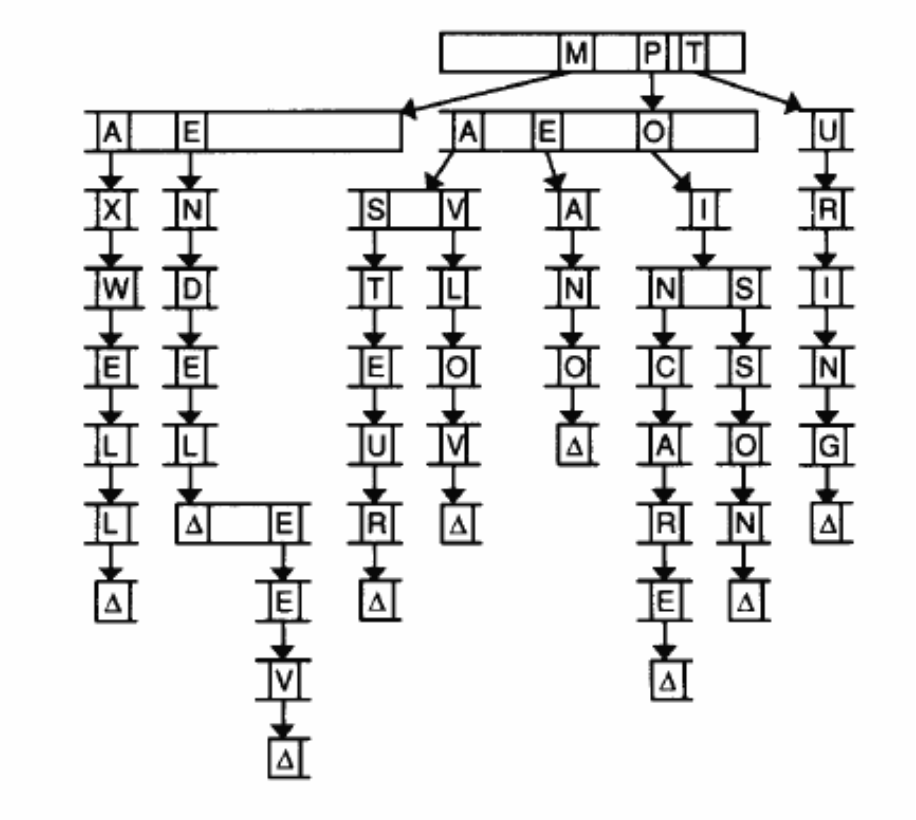
- What happens after we hash? Well, we want to follow the pointer at that index in the array (assuming it's non-`NULL`). If we're searching for a name, we'll have to walk through all the nodes in that particular linked list. If we're inserting, we probably want to insert at the beginning of the linked list because this can be done in constant time.
- Can we get even faster?[4] Yes! But first, a two-minute break.

3 Trees[5] and Tries (37:00–53:00)



- Knowing nothing about these data structures known as trees, what can you tell from the diagram above?
- Notice that each node branches into exactly two nodes (and the left node holds a smaller value than the right). These two nodes are called the *children* of that node. Because there are exactly two, this is a special case of a tree known as a *binary search tree*. Don't let the name fool you, though, because a naively implemented BST can have search time in $O(n)$.
- A trie is similar to a tree, except that each of its nodes is actually an array. In the context of dictionaries (hint hint), each of these arrays might be of size 26, as in the diagram below:

[4] No. Linked list : data structures :: Caesar's cipher : cryptography.
[5] Heh.

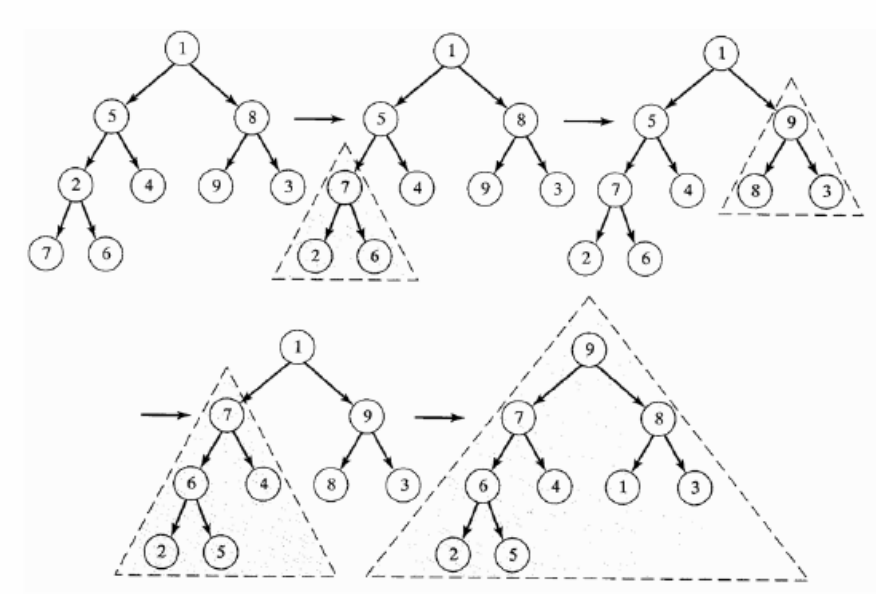


We walk through a trie much the same way we walk through a hash table. Each letter in the word we're looking up is also its index into the next level of the trie. So if we're looking up the name Maxwell, we first hash to M, then to A, then to X, etc. What happens when we get to the end of a word? We need some sort of flag (represented as a triangle in the diagram above) that marks the end of a word. That way, if two words share a prefix (e.g. Max and Maxwell), we will know that both of them are in our trie if this end-of-word flag is set at both the X and the last L.

- What's the lookup time of a trie? Constant time! ($O(k)$, we'll call it). This is an improvement, at least theoretically, on the lookup time of a hash table, which we might write as $O(n/m)$, where m is the size of our hash table.

4 Heaps (53:00–72:00)

- Let's talk about heaps.[6] Probably best to present this definition in full. A heap is a binary tree that:
  - is **complete** (i.e., every level of the tree is completely filled with nodes except for, perhaps, the bottommost level, whose nodes are in the leftmost locations)
  - satisfies the **heap-order property** (i.e., each node's value is greater than or equal to that of each of its children, if any)
- Heapifying an almost-heap[7] involves first manipulating the bottom, leftmost cluster so that it satisfies the two properties above, then working upward and to the right, taking larger and larger clusters until the entire tree is heapified.



The general process, as you can hopefully follow in this diagram, is to take a node and ask: is it bigger than its children? If not, swap it with the *bigger* of the two children. And again, proceed upward and to the right. The good news is that since the height of a binary tree is $\log n$, heapifying a binary tree is in $O(n \log n)$.

- How can we leverage this? We can sort an array *in place* using heapsort, thus chipping away at quadratic running times without using extra memory, as mergesort required.

[6] Man, I'm bad at transitions, huh?
[7] Srsly, what is this, Dr. Seuss?

- Heapsort works by finding the biggest value in the tree and putting it at the end of the list by swapping it with the smallest value. Then, we eliminate this biggest value from consideration and repeat the process. Worst-case scenario, the smallest value, when swapped to the root, will have to bubble down through $\log n$ steps in order to be put back in place. Thus, to take an arbitrary array of numbers, create a tree, and heapify that tree, all in $O(2n \log n)$, which, in fact, is the same as $O(n \log n)$. The really neat trick is that we can store this heapsorted list of numbers in a single array! Turns out that the index location of a left child node is `2i + 1` and the index location of a right child node is `2i + 2`. That's all there is to it!
- Why can't I ever find a good way to end these notes!?