Last Time

• Quick sort:
  – Partition an array into two halves around a fixed pivot
  – Try to partition each half
    • Partition an array into two halves around a fixed pivot
    • Try to partition each half
      – Partition an array into two halves around a fixed pivot
      – Try to partition each half
        » Partition an array into two halves around a fixed pivot
        » Try to partition each half
String Searches

• How can we search for a string?
  – Store them in a data structure that we’ve already talked about? Array, list, tree, BST, Hash table?

• What if we had a data structure designed with complexity $W$, where $W$ is the length of the string?
  – Search hits take time proportional to the length of the search key.
  – Search misses take a few characters, until we know that the search key cannot be in the data structure.
Trie

• Composed of nodes, just like our other tree data structures.
  – One root node.
  – Each node has 1 value and $R$ child links, corresponding to the $R$ possible next letters.
Trie – Visual

1. Characters are implicitly defined by the link index.
2. Each node has an array of links and a value.
Trie

• Composed of nodes, just like our other tree data structures.
  – One root node.
  – Each node has 1 value and $R$ child links, corresponding to the $R$ possible next letters.

• Search: Follow the child nodes. If you get to the end of the string and there’s a value, return the value. If not, return null. If you ever hit a null link, return null.

• Insertion: Follow the search process. If you get to the end of the string, add the value. If you ever hit a null node, add new nodes.
Trie

Node put(Node x, String key, Value val, int d) {
    if (x == null) {
        x = new Node();
    } //if
    if (d == key.length()) {
        x.val = val;
        return x;
    } //if
    char c = key.charAt(d);
    x.next[c] = put(x.next[c], key, val, d+1);
    return x;
} //put
Trie

Value get(String key) {
    Node x = get(root, key, 0);
    if (x == null) {
        return null;
    } //if
    return (Value) x.val;
} //get

Node get(Node x, String key, int d) {
    if (x == null) {
        return null;
    } //if
    if (d == key.length()) {
        return x;
    } //if
    char c = key.charAt(d);
    return get(x.next[c], key, d+1);
} //get
Trie – Simpler Visual

- **Key** and **Value** are associated with each node.
- **Root** is the starting point.
- **Value** is in a node corresponding to the last character.
- **Key** is a sequence of characters from root to value.
- **Nodes** corresponding to characters at the end of the key do not exist, so cross them until set the value of the last one.
- **One node** for each key character.

Examples:
- *she*: value is 0
- *by*: value is 4
- *sells*: value is 1
- *the*: value is 5
- *sea*: value is 2
- *shells*: value is 3
- *shore*: value is 7
Trie – Analysis

• **Theorem:** The number of array accesses when searching in a trie or inserting a key into a trie is at most $1 + \text{the length of the key}$.

• **Proof:** Immediate from the code.
  – The `put()` and `get()` implementations carry the `d` argument.
  – `d` starts at 0, increments at each level, and is used to stop the recursion when it reaches the key length.
Trie – Analysis

• **Theorem:** The number of links in a trie is between $RN$ and $RNw$, where $w$ is the average key length.

• **Proof:** Immediate from the code.
  
  – Every key in the trie has a node containing $R$ links, so the number of links is at least $RN$.
  
  – If the first characters of all of the keys are different, then there is a node with $R$ links for every key character, so the number of links is $R$ times the total number of characters, or $RNw$.

• In other words, lots of wasted space.
de la Briandais Trie

• Old version: Node = value + $R$ links
• New version: Node = value + 2 links
  – Link 1: reference to child node (move to next level)
  – Link 2: reference to sibling node (stay on same level)

• Notice that now we need to store the key as a character, rather than use them as indices.
de la Briandais Trie

bye, by, get, got, gets
de la Briandais Trie - Analysis

• In the worst case, with an $R$-character alphabet and a string of length $w$, our search time is $Rw$, because we search through all $R$ characters on level $i$ before moving to level $i + 1$.

• In the average case, our search time is $\sim \log_R(w)$, since we won’t search through all of the characters before moving levels, and based on the level-sparsity rationale from the regular trie analysis.
  – For small $N$, search time is just $\sim w$. 
Any Questions?