CS/COE 1501
Algorithm Implementation

2-Searching

Fall 2018

Sherif Khattab

ksm73@pitt.edu

6307 Sennott Square

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Review: Searching through a collection

- Given a collection of keys *C*, how to we search for a given key *k*?
  - Store collection in an array
    - Unsorted
    - Sorted
  - Linked list
    - Unsorted
    - Sorted
  - Binary search tree

- Differences?
- Runtimes?

Symbol tables

- Abstract structures that link *keys* to *values*
  - Key is used to search the data structure for a value
  - Described as a class in the text, but probably more accurate to think of the concept of a symbol table in general as an interface
    - Key functions:
      - put()
      - contains()

A closer look

- BinarySearchST.java and BST.java present symbol tables based on sorted arrays and binary search trees, respectively
- Can we do better than these?
- Both methods depend on comparisons against other keys
  - I.e., k is compared against other keys in the data structure
- 4 options at each node in a BST:
  - Node ref is null, k not found
  - k is equal to the current node's key, k is found
  - k is less than current key, continue to left child
  - k is greater than the current key, continue to right child

Digital Search Trees (DSTs)

- Instead of looking at less than/greater than, lets go left right based on the bits of the key, so we again have 4 options:
  - Node ref is null, k not found
  - k is equal to the current node's key, k is found
  - current bit of k is 0, continue to left child
  - current bit of k is 1, continue to right child

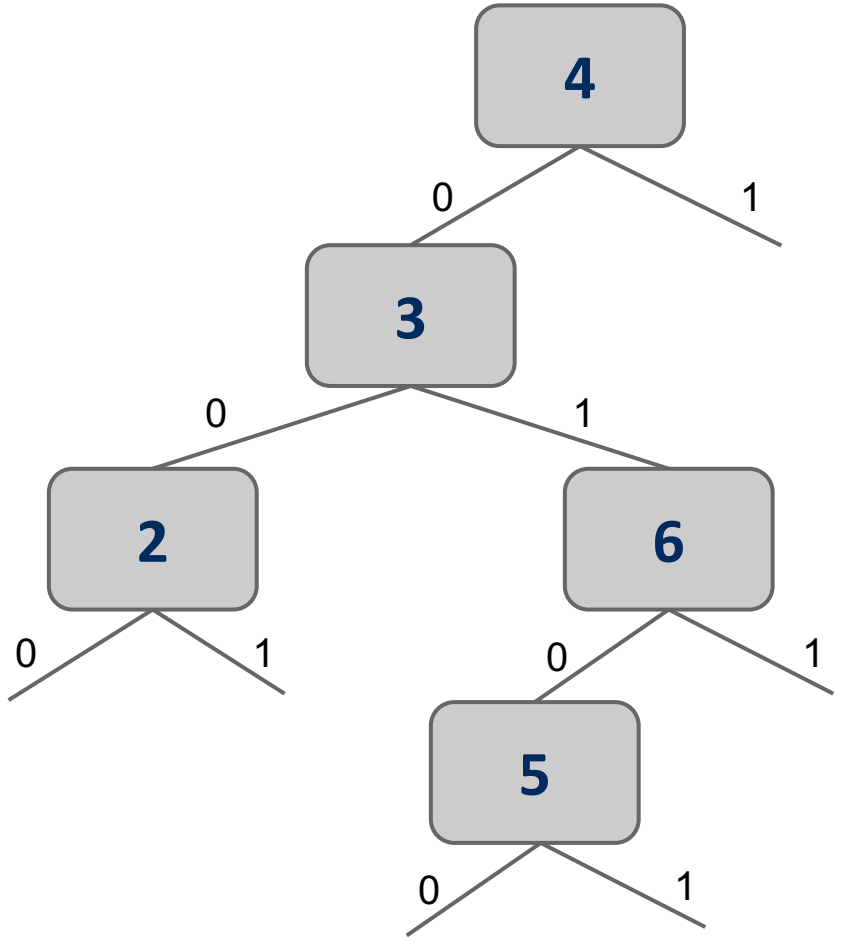
DST example

Insert:

4	0100
3	0011
2	0010
6	0110
5	0101

Search:

3	0011
7	0111



Analysis of digital search trees

- Runtime?

- We end up doing many comparisons against the full key,

  can we improve on this?

Radix search tries (RSTs)

- Trie as in retrieve, pronounced the same as “try”
- Instead of storing keys as nodes in the tree, we store them implicitly as paths down the tree
  - Interior nodes of the tree only serve to direct us according to the bitstring of the key
  - Values can then be stored at the end of key’s bit string path

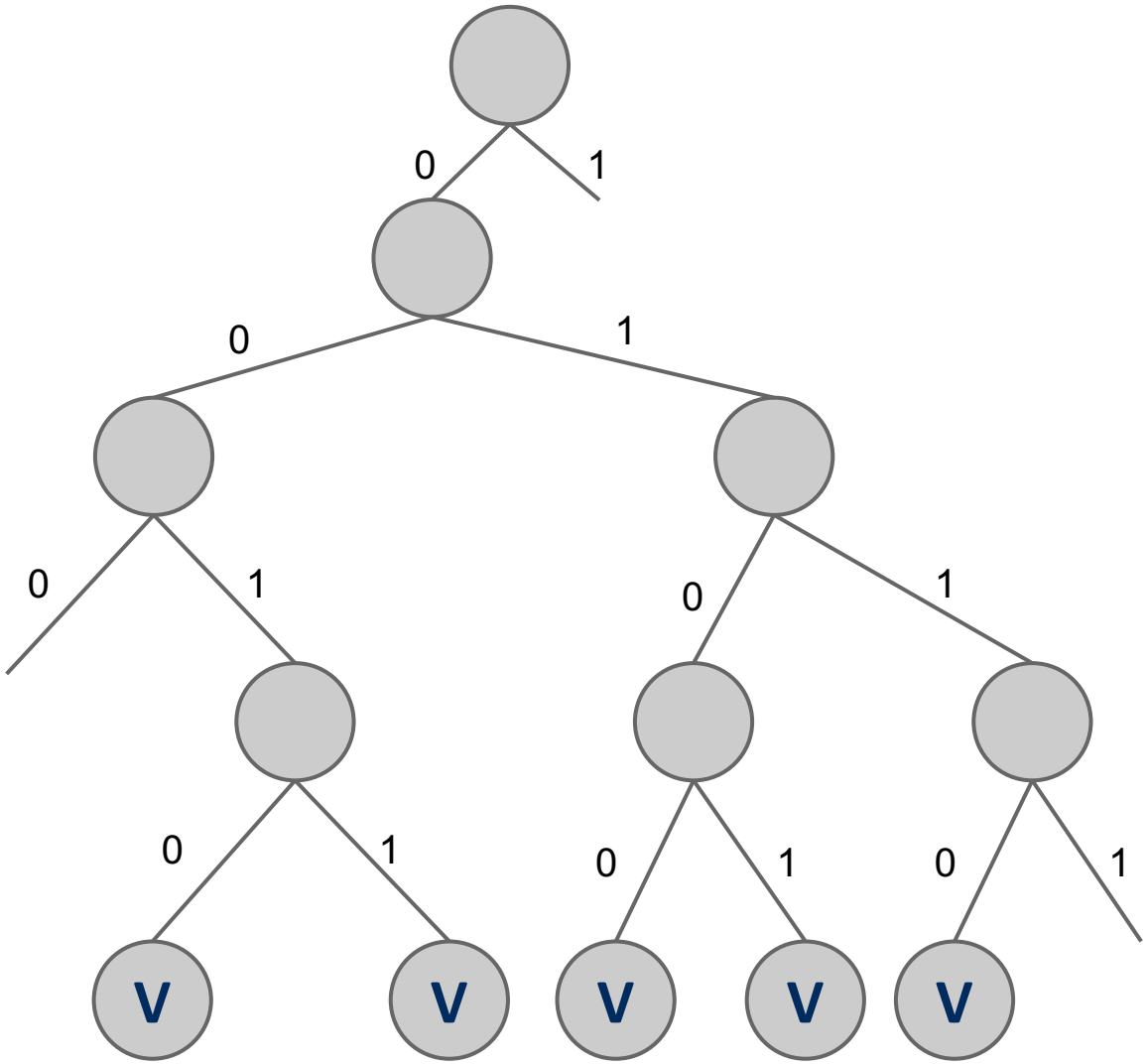
RST example

Insert:

4	0100
3	0011
2	0010
6	0110
5	0101

Search:

3	0011
7	0111



RST analysis

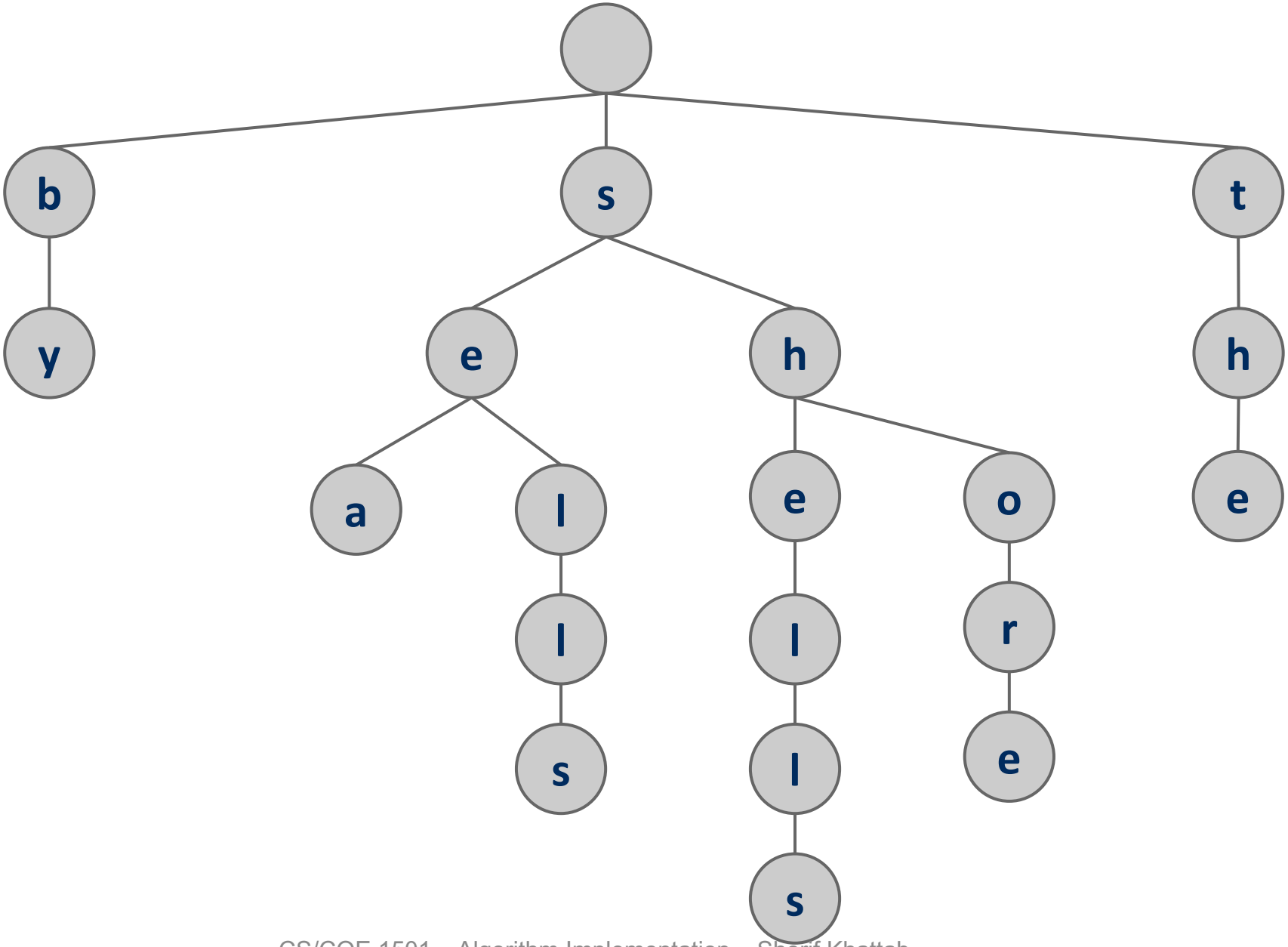
- Runtime?

- Would this structure work as well for other key data types?
  - Characters?
  - Strings?

Larger branching factor tries

- In our binary-based Radix search trie, we considered one bit at a time
- What if we applied the same method to characters in a string?
  - What would this new structure look like?
- Let's try inserting the following strings into an trie:
  - she, sells, sea, shells, by, the, sea, shore

Another trie example



Analysis

- Runtime?

Further analysis

- Miss times
  - Require an average of $\log_R(n)$ nodes to be examined
    - Where R is the size of the alphabet being considered
    - Proof in Proposition H of Section 5.2 of the text
  - Average # of checks with 2^{20} keys in an RST?
  - With 2^{20} keys in a large branching factor trie, assuming 8-bits at a time?

Implementation Concerns

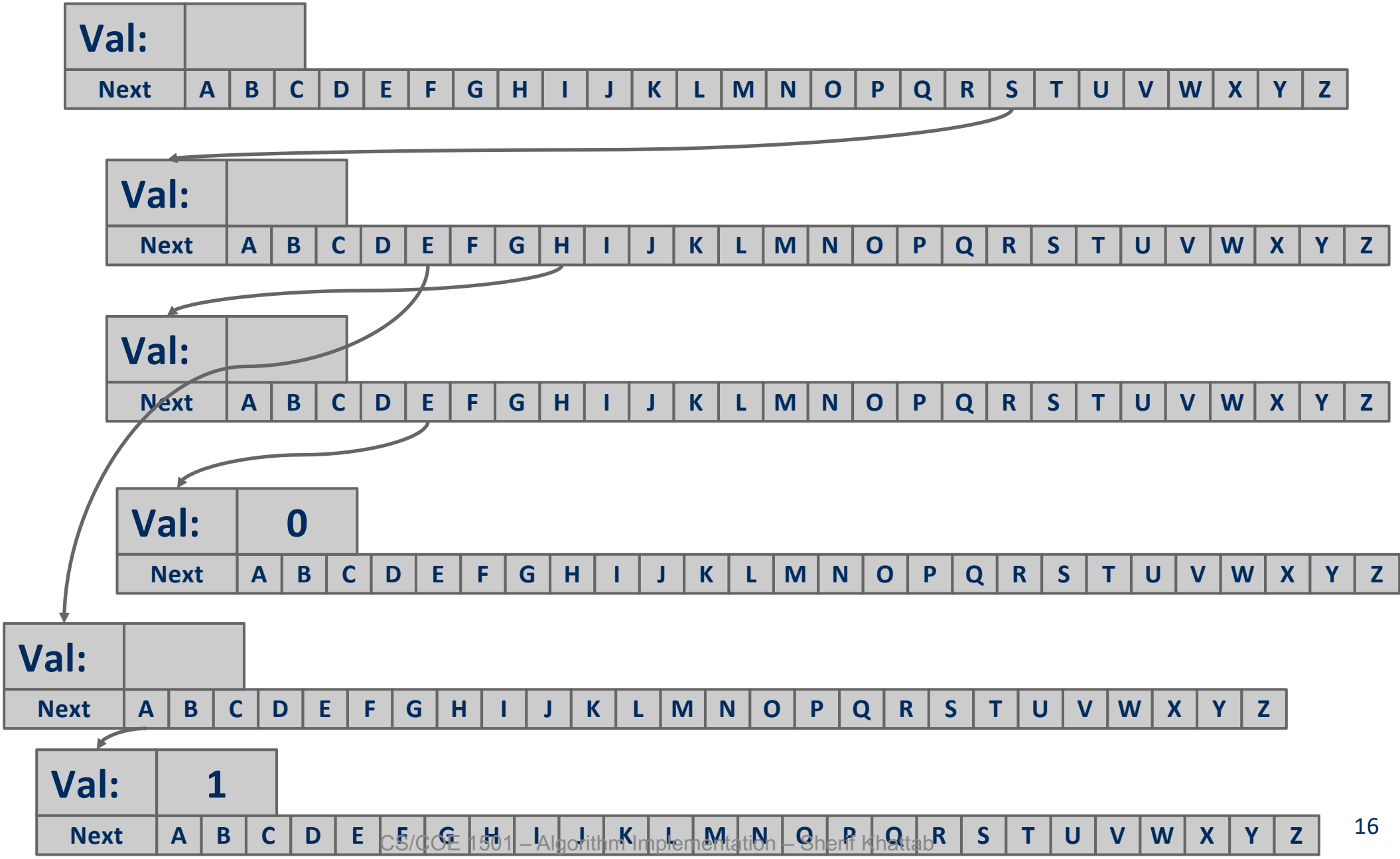
- See TrieSt.java
  - Implements an R-way trie
- Basic node object:

Where R is the branching factor

```
private static class Node {
    private Object val;
    private Node[] next = new Node[R];
}
```

- Non-null val means we have traversed to a valid key
- Again, note that keys are not directly stored in the trie at all

R-way trie example



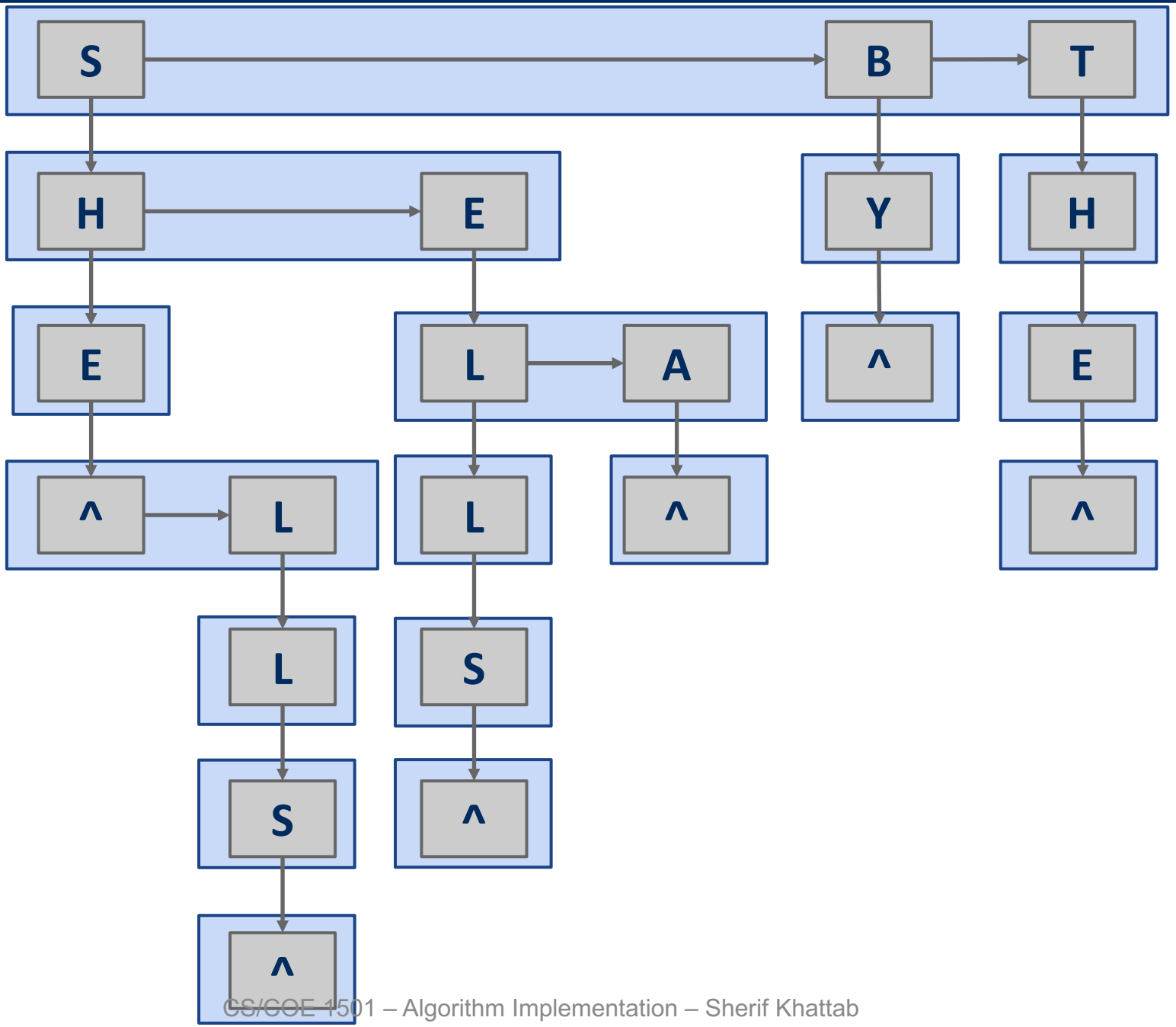
So what's the catch?

- Space!
  - Considering 8-bit ASCII, each node contains 2^8 references!
  - This is especially problematic as in many cases, alot of this space is wasted
    - Common paths or prefixes for example, e.g., if all keys begin with "key", thats 255*3 wasted references!
    - At the lower levels of the trie, most keys have probably been separated out and reference lists will be sparse

De La Briandais tries (DLBs)

- Replace the .next array of the R-way trie with a linked-list

DLB Example



DLB analysis

- How does DLB performance differ from R-way tries?
- Which should you use?

Searching

- So far we've continually assumed each search would only look for the presence of a whole key
- What about if we wanted to know if our search term was a prefix to a valid key?

Final notes

- This lecture does not present an exhaustive look at search trees/tries, just the sampling that we're going to focus on
- Many variations on these techniques exist and perform quite well in different circumstances
  - Red/black BSTs
  - Ternary search Tries
  - R-way tries without 1-way branching
- See the table at the end of Section 5.2 of the text

Wouldn't it be wonderful if...

- Search through a collection could be accomplished in Θ(1) with relatively small memory needs?
- Lets try this:
  - Assume we have an array of length m (call it HT)
  - Assume we have a function h(x) that maps from our key space to {0, 1, 2, …, m-1}
    - E.g., $\mathbb{Z} \rightarrow$ {0, 1, 2, …, m-1} for integer keys
    - Let's also assume h(x) is efficient to compute
- This is the basic premise of *hash tables*

How do we search/insert with a hash map?

- Insert:

```
i = h(x)
HT[i] = x
```

- Search:

```
i = h(x)
if (HT[i] == x) return true;
else return false;
```

- This is a very general, simple approach to a hash table implementation
  - Where will it run into problems?

What do we do if h(x) == h(y) where x != y?

- Called a *collision*

Consider an example

- Company has 500 employees
- Stores records using a hashmap with 1000 entries
- Employee SSNs are hashed to store records in the hashmap
  - Keys are SSNs, so |keyspace| == 10^9
- Specifically what keys are needed can't be known in advance
  - Due to employee turnover
- What if one employee (with SSN x) is fired and replacement has an SSN of y?
  - Can we design a hash function that guarantees h(y) does not collide with the 499 other employees' hashed SSNs?

Can we ever guarantee collisions will not occur?

- Yes, if our keyspace is smaller than our hashmap
  - If |keyspace| <= m, *perfect hashing* can be used
    - i.e., a hash function that maps every key to a distinct integer < m
    - Note it can also be used if n < m and the keys to be inserted are known in advance
      - E.g., hashing the keywords of a programming language during compilation
- If |keyspace| > m, collisions cannot be avoided

Handling collisions

- Can we reduce the number of collisions?
  - Using a good hash function is a start
    - What makes a good hash function?
      1. Utilize the entire key
      2. Exploit differences between keys
      3. Uniform distribution of hash values should be produced

Examples

- Hash list of classmates by phone number
  - Bad?
    - Use first 3 digits
  - Better?
    - Consider it a single int
    - Take that value modulo m
- Hash words
  - Bad?
    - Add up the ASCII values
  - Better?
    - Use Horner's method to do modular hashing again
      - See Section 3.4 of the text

The madness behind Horner's method

- Base 10
  - 12345
  - $= 1 * 10^4 + 2 * 10^3 + 3 * 10^2 + 4 * 10^1 + 5 * 10^0$
- Base 2
  - 10100
  - $= 1 * 2^4 + 0 * 2^3 + 1 * 2^2 + 0 * 2^1 + 0 * 2^0$
- Base 16
  - BEEF3
  - $= 11 * 16^4 + 14 * 16^3 + 14 * 16^2 + 15 * 16^1 + 3 * 16^0$
- ASCII Strings
  - HELLO
  - $= 'H' * 256^4 + 'E' * 256^3 + 'L' * 256^2 + 'L' * 256^1 + 'O' * 256^0$
  - $= 72 * 256^4 + 69 * 256^3 + 76 * 256^2 + 76 * 256^1 + 79 * 256^0$

Modular hashing

- Overall a good simple, general approach to implement a hash map
- Basic formula:
  - h(x) = c(x) mod m
    - Where c(x) converts x into a (possibly) large integer
- Generally want m to be a prime number
  - Consider m = 100
  - Only the least significant digits matter
    - h(1) = h(401) = h(4372901)

Back to collisions

- We've done what we can to cut down the number of collisions, but we still need to deal with them
- Collision resolution: two main approaches
  - Open Addressing
  - Closed Addressing

Open Addressing

- I.e., if a pigeon's hole is taken, it has to find another
- If h(x) == h(y) == i
  - And x is stored at index i in an example hash table
  - If we want to insert y, we must try alternative indices
    - This means y will not be stored at HT[h(y)]
      - We must select alternatives in a consistent and predictable way so that they can be located later

Linear probing

- Insert:
  - If we cannot store a key at index i due to collision
    - Attempt to insert the key at index i+1
    - Then i+2 …
    - And so on …
    - mod m
    - Until an open space is found
- Search:
  - If another key is stored at index i
    - Check i+1, i+2, i+3 … until
      - Key is found
      - Empty location is found
      - We circle through the buffer back to i

Linear probing example

- h(x) = x mod 11
- Insert 14, 17, 25, 37, 34, 16, 26

0	1	2	3	4	5	6	7	8	9	10
	34		14	25	37	17	16	26		

- How would deletes be handled?
  - What happens if key 17 is removed?

Alright! We solved collisions!

- Well, not quite…
- Consider the *load factor* $\alpha = n/m$
- As α increases, what happens to hash table performance?

- Consider an empty table using a good hash function
  - What is the probability that a key x will be inserted into any one of the indices in the hash table?

- Consider a table that has a cluster of c consecutive indices occupied
  - What is the probability that a key x will be inserted into the index directly after the cluster?

Avoiding clustering

- We must make sure that even *after* a collision, all of the indices of the hash table are possible for a key
  - Probability of filled locations need to be distributed throughout the table

Double hashing

- After a collision, instead of attempting to place the key x in i+1 mod m, look at i+h2(x) mod m
  - h2() is a second, different hash function
    - Should still follow the same general rules as h() to be considered good, but needs to be different from h()
      - h(x) == h(y) AND h2(x) == h2(y) should be very unlikely
        - Hence, it should be unlikely for two keys to use the same increment

Double hashing

- h(x) = x mod 11
- h2(x) = (x mod 7) +1
- Insert 14, 17, 25, 37, 34, 16, 26

0	1	2	3	4	5	6	7	8	9	10
	34		14	37	16	17		25		26

- Why could we not use h2(x) = x mod 7?
  - Try to insert 2401

A few extra rules for h2()

- Second hash function cannot map a value to 0
- You should try all indices once before trying one twice

- Were either of these issues for linear probing?

As $\alpha \rightarrow 1$...

- Meaning n approaches m…
- Both linear probing and double hashing degrade to Θ(n)
  - How?
    - Multiple collisions will occur in both schemes
    - Consider inserts and misses…
      - Both continue until an empty index is found
        - With few indices available, close to m probes will need to be performed
          - Θ(m)
        - n is approaching m, so this turns out to be Θ(n)

Open addressing issues

- Must keep a portion of the table empty to maintain respectable performance
  - For linear hashing ½ is a good rule of thumb
    - Can go higher with double hashing

Closed addressing

- I.e., if a pigeon's hole is taken, it lives with a roommate
- Most commonly done with separate chaining
  - Create a linked-list of keys at each index in the table
    - As with DLBs, performance depends on chain length
      - Which is determined by α and the quality of the hash function

In general...

- Closed-addressing hash tables are fast and efficient for a large number of applications

String Matching

- Have a pattern string p of length m
- Have a text string t of length n
- Can we find an index i of string t such that each of the m characters in the substring of t starting at i matches each character in p
  - Example: can we find the pattern "fox" in the text "the quick brown fox jumps over the lazy dog"?
    - Yes! At index 16 of the text string!

Simple approach

- BRUTE FORCE
  - Start at the beginning of both pattern and text
  - Compare characters left to right
  - Mismatch?
  - Start again at the 2nd character of the text and the beginning of the pattern...

Brute force code

```
public static int bf_search(String pat, String txt) {
    int m = pat.length();
    int n = txt.length();
    for (int i = 0; i <= n - m; i++) {
        int j;
        for (j = 0; j < m; j++) {
            if (txt.charAt(i + j) != pat.charAt(j))
                break;
        }
        if (j == m)
            return i; // found at offset i
    }
    return n; // not found
}
```

Brute force analysis

- Runtime?
  - What does the worst case look like?
    - t = XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXY
    - p = XXXXY
  - m (n - m + 1)
    - Θ(nm) if n >> m
  - Is the average case runtime any better?
    - Assume we mostly mismatch on the first pattern character
    - Θ(n + m)
      - Θ(n) if n >> m

Where do we improve?

- Improve worst case
  - Theoretically very interesting
  - Practically doesn't come up that often for human language
- Improve average case
  - Much more practically helpful
    - Especially if we anticipate searching through large files

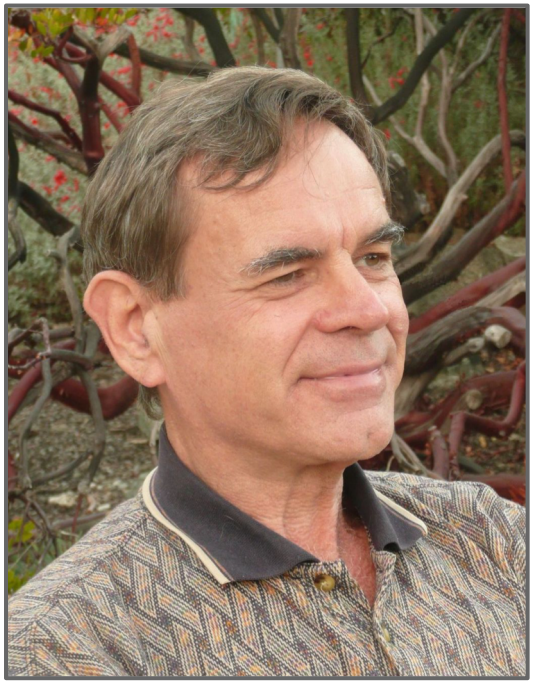
First: improving the worst case

Discovered the same algorithm independently

Knuth

Morris

Pratt

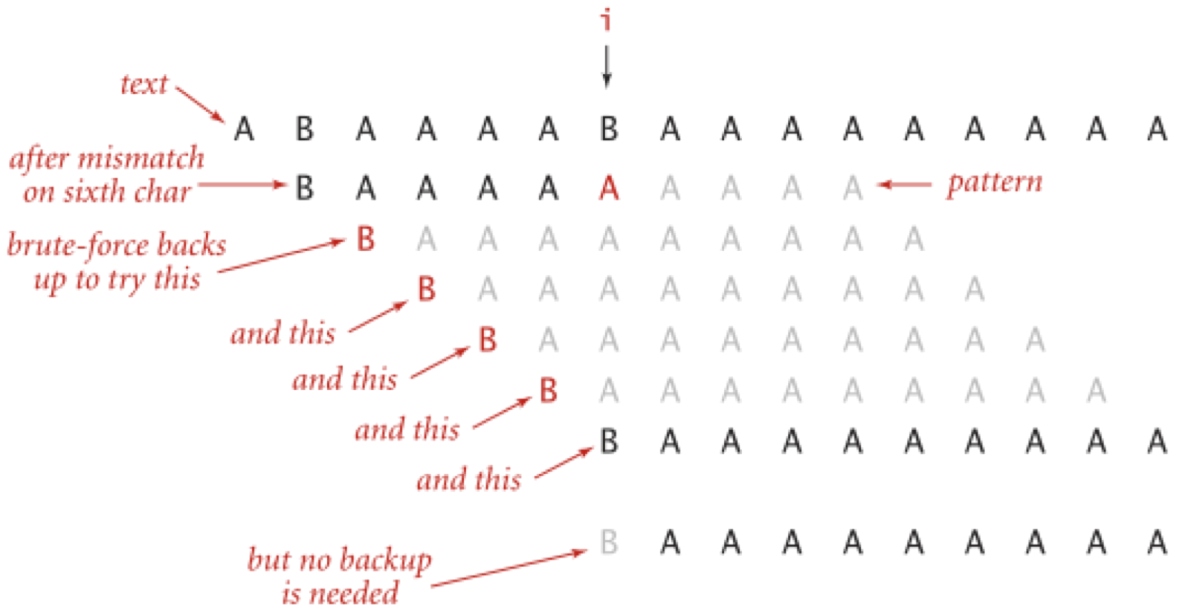


Worked together

Jointly published in 1976

Back to improving the worst case

- Knuth Morris Pratt algorithm (KMP)
- Goal: avoid backing up in the text string on a mismatch
- Main idea: In checking the pattern, we learned something about the characters in the text, take advantage of this knowledge to avoid backing up

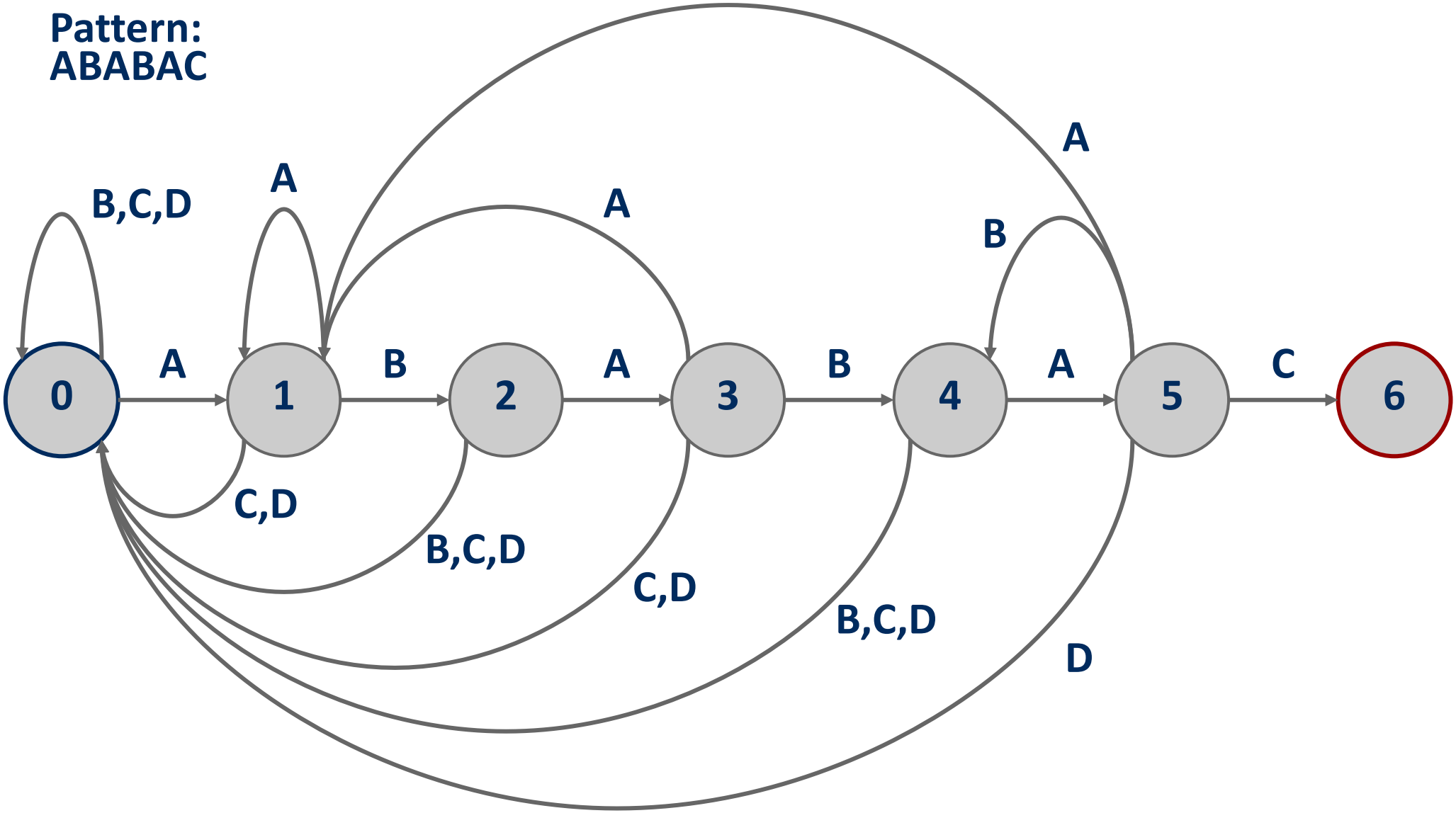


Text pointer backup in substring searching

How do we keep track of text processed?

- Actually, build a deterministic finite-state automata (DFA) storing information about the *pattern*
  - From a given state in searching through the pattern, if you encounter a mismatch, how many characters currently match from the beginning of the pattern

DFA example



Representing the DFA in code

- DFA can be represented as a 2D array:
  - dfa[cur_text_char][pattern_counter] = new_pattern_counter
    - Storage needed?
      - mR

	0	1	2	3	4	5
A						
B						
C						
D						

KMP code

```
public int kmp_search(String pat, String txt) {
    int m = pat.length();
    int n = txt.length();
    int i, j;
    for (i = 0, j = 0; i < n && j < m; i++)
        j = dfa[txt.charAt(i)][j];
    if (j == m) return i - m; // found
    return n; // not found
}
```

- Runtime?

Another approach: Boyer Moore

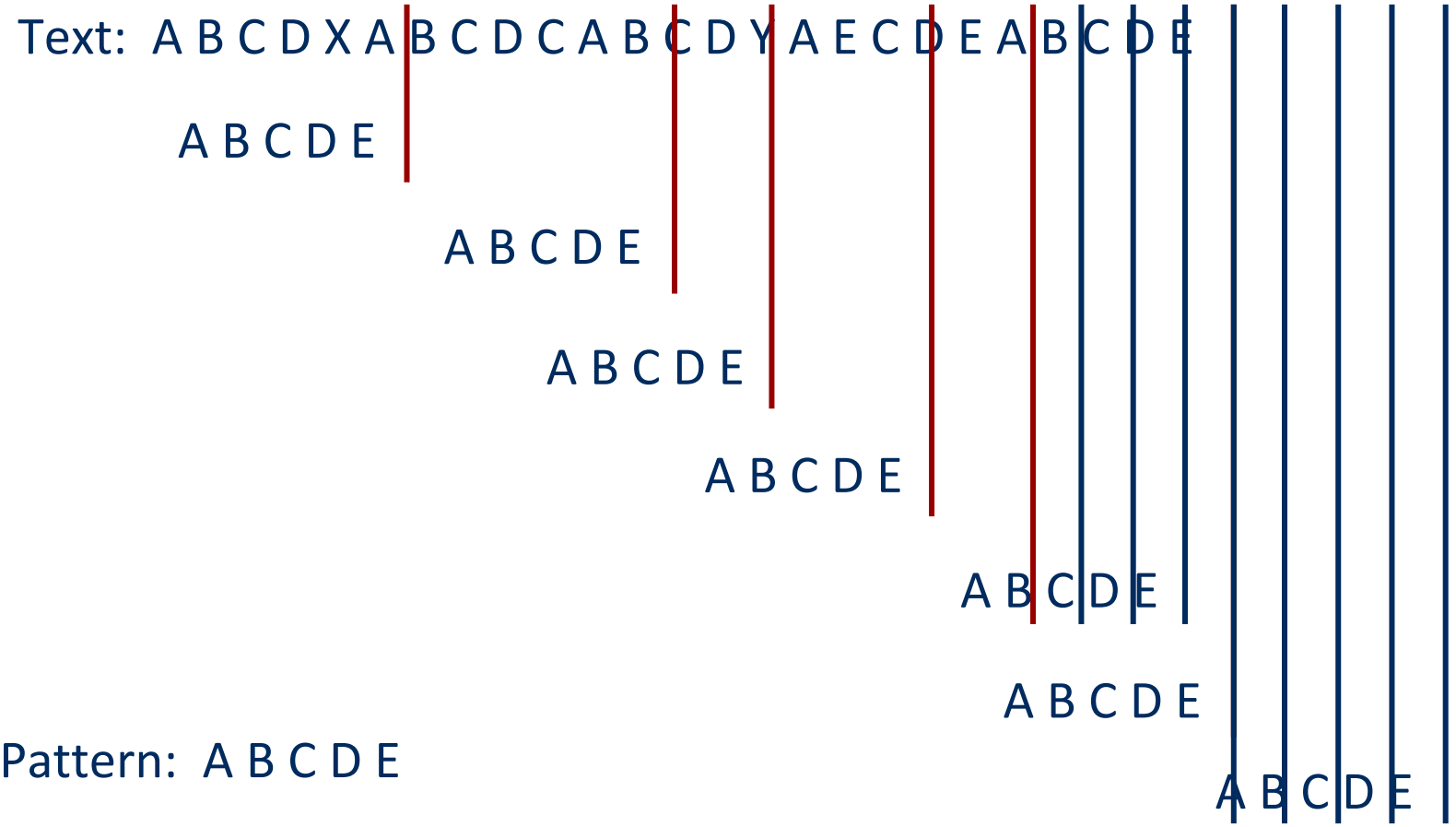
- What if we compare starting at the end of the pattern?
  - t = ABCDVABCDWABCDXABCDYABCDZ
  - p = ABCDE
  - V does not match E
    - Further V is nowhere in the pattern…
    - So skip ahead m positions with 1 comparison!
      - Runtime?
        - In the best case, n/m
- When searching through text with a large alphabet, will often come across characters not in the pattern.
  - One of Boyer Moore's heuristics takes advantage of this fact
    - Mismatched character heuristic

Mismatched character heuristic

- How well it works depends on the pattern and text at hand
  - What do we do in the general case after a mismatch?
    - Consider:
      - `t = XYXYXYZXXXXXXXXXXXXXX`
      - `p = XYXYZ`
    - If mismatched character *does* appear in p, need to "slide" to the right to the next occurrence of that character in p
      - Requires us to pre-process the pattern
        - Create a right array

```
for (int i = 0; i < R; i++)
   right[i] = -1;
for (int j = 0; j < m; j++)
   right[p.charAt(j)] = j;
```

Mismatched character heuristic example



Pattern: A B C D E

right = [0, 1, 2, 3, 4, -1, -1, …]

Runtime for mismatched character

- What does the worst case look like?
  - Runtime:
    - Θ(nm)
      - Same as brute force!
- This is why mismatched character is only one of Boyer Moore's heuristics
  - Another works similarly to KMP
- See BoyerMoore.java

Another approach

- Hashing was cool, let's try using that

```
public static int hash_search(String pat, String txt) {
    int m = pat.length();
    int n = txt.length();
    int pat_hash = h(pat);
    for (int i = 0; i <= n - m; i++) {
            if (h(txt.substring(i, i + m)) == pat_hash)
                return i; // found!
    }
    return n; // not found
}
```

Well that was simple

- Is it efficient?
  - Nope! Practically worse than brute force
    - Instead of nm character comparisons, we perform n hashes of m character strings
- Can we make an efficient pattern matching algorithm based on hashing?

Horner's method

- Brought up during the hashing lecture

```
public long horners_hash(String key, int m) {
        long h = 0;
        for (int j = 0; j < m; j++)
                h = (R * h + key.charAt(j)) % Q;
        return h;
}
```

- horners_hash("abcd", 4) =
  - 'a' * R^3 + 'b' * R^2 + 'c' * R + 'd' mod Q
- horners_hash("bcde", 4) =
  - 'b' * R^3 + 'c' * R^2 + 'd' * R + 'e' mod Q
- horners_hash("cdef", 4) =
  - 'c' * R^3 + 'd' * R^2 + 'e' * R + 'f' mod Q

Efficient hash-based pattern matching

```
text = "abcdefg"
pattern = "defg"
```

- This is Rabin-Karp

What about collisions?

- Note that we're not storing any values in a hash table…
  - So increasing Q doesn't affect memory utilization!
    - Make Q really big and the chance of a collision becomes really small!
      - But not 0…
- OK, so do a character by character comparison on a hash match just to be sure
  - Worst case runtime?
    - Back to brute force esque runtime...

Assorted casinos

- Two options:
  - Do a character by character comparison after hash match
    - Guaranteed correct
    - Probably fast

Las Vegas

  - Assume a hash match means a substring match
    - Guaranteed fast
    - Probably correct

Monte Carlo