CS240 Exam Review W15

1 Hashing

Main Idea:
Use of a Hash Function to store keys in an array.
For an array T of size M, we store key k at $T[h(k)]$.
Search, insert and delete should cost $O(1)$

1.1 Chaining

Main Idea: **Create an unsorted linked list for each entry**
Each entry in the array is a *bucket*.

Recall: load balance $\alpha = \frac{n}{M}$
Assuming uniform hashing with average bucket size α

Search: $\Theta(1+\alpha)$ average, $\Theta(n)$ worst case
Insert: $O(1)$ worst case
Delete: Same as Search

If we maintain $M \in \Theta(n)$ average cost will be $O(1)$.

1.2 Open Addressing

Main Idea:
Each hash entry holds only one item, key k can go in multiple locations

1.2.1 Linear Probing

Our hash function finds the first empty spot starting at $h(k)$.
$h(k,i) = (h(k)+i) \bmod M$, for some hash function h.

Lazy deletion/Tombstones: We mark deleted elements as deleted instead of deleting the element. Deleted elements are treated as empty when inserting and occupied during a search.

$\Rightarrow$ Easy to form clusters with linear probing.

1.2.2 Double Hashing

We use two hash functions that are independent.
Assume the probability that a key k has $h_1(k) = a$ and $h_2(k) = b$ is $\frac{1}{M^2}$

Our hash function:
$h(k,i) = h_1(k) + i \cdot h_2(k) \bmod M$
Similar to linear probing but with a different probe sequence.

1.2.3 Cuckoo Hashing

Main Idea: always insert a new item into $h_1(k)$.
We will reinsert the other item when there is a collision.

We could have a loop, in this case we rehash with a larger M.
A key k can only be at $T[h(k_1)]$ or $T[h_2(k)]$.

1.2.4 Complexity of open addressing strategies

(Big Theta)

Strategy	Search	Insert	Delete
Linear Probing	$\frac{1}{(1-\alpha)^2}$	$\frac{1}{(1-\alpha)^2}$	$\frac{1}{(1-\alpha)}$
Double Hashing	$\frac{1}{(1-\alpha)}$	$\frac{1}{(1-\alpha)}$	$\frac{1}{\alpha}\log(\frac{1}{(1-\alpha)})$
Cuckoo Hashing	1	$\frac{\alpha}{(1-2\alpha)^2}$	1

1.2.5 Extendible Hashing

Main Idea: Minimize disk transfers by hashing to blocks of size S.

The directory $\Rightarrow$ internal memory and blocks $\Rightarrow$ external memory.
The directory contains an array of size 2^d where $d \leq L$ is the order.

Every block stores its own local depth $k_B \leq d$.

`Search:`
Given k, compute h(k).
The leading d digits will give us the index.
Load that block into main memory and search the block for k.

`Insert:`
Search for h(k).
If B has space, insert, done.
If block is full and local depth $k_B < d$, block split

1. split B into 2 blocks
2. seperate items to the $k_B + 1$th bit
3. increment the local depth
4. update references in directory

If block is full and local depth $k_B = d$, directory grow

1. double directory (increment d)
2. update references in directory
3. split B

`Delete:`
Reverse of insert,

1. Search for block B, remove k
2. If empty, block merge
3. If every block B has $k_B \leq d - 1$, directory shrink

$\Rightarrow$ main disadv: extra cpu cost of $O(\log S)$ or $O(S)$

2 #JustDictionaryThings

2.1 Interpolation Search

Main Idea:
We use the value of the key to guess its location (similar to a phone book)
Similar to binary search but instead of halving the search we can cut it down even more:
Use a factor $c = \frac{k-L}{R-L}$

Binary Search Index $= l + \left\lfloor \frac{1}{2}(R-L) \right\rfloor$
Interpolation Search Index $= l + \left\lfloor \frac{k-L}{R-L}(R-L) \right\rfloor$

$\Rightarrow O(\log\log n)$ on average, worst case: $O(n)$

2.2 Gallop Search

Main Idea:
Doubling upper limit of search until the value is larger than key k, then go back and binary search to find k in our new range

The length of the array is unknown, we make $O(\log m)$ comparisons (m: location of k in A).
$\Rightarrow$ Gallop Search is efficient if we expect to find it in the beginning of the array.

2.3 Skip Lists

Main Idea: **Hierachy of ordered linked lists starting with $-\infty$ and ending in ∞.**
Traversing the skiplist with:
after(p): next node on same list
below(p): same node on lower list

`Search:`
1. Start at top left $-\infty$
2. If $k <$ after(p), continue right, ($p =$ after(p))
3. $p =$ below(p) (while not null)

`Insert:`
1. Randomly compute the height of k. ($i = $ #heads)
2. Search for where k should go
3. Insert k in all S_j for $0 \leq j \leq i$ (tower of height i).

`Delete:`
1. Search for k
2. Delete Tower
3. Leave only one list that contains $-\infty$ and ∞

`Summary of Skip-Lists`
- Expected space: $O(n)$
- Expected height: $O(logn)$
- Search: $O(\log n)$ expected
- Insert: $O(\log n)$ expected
- Delete: $O(\log n)$ expected

3 Multi-Dimensional Data

We focus on 2-dimensional data, i.e points in Euclidean plane.

Recall: **One-Dimensional Range Search**
for k: number of reported items
Nodes visited:

- $O(\log n)$ boundary nodes
- $O(k)$ inside nodes
- No outside nodes

(Note: finding any of these nodes takes $O(\log n)$ time)

Colouring:

- Grey nodes = Boundary nodes
- Black nodes = Inside nodes
- White nodes = Outside nodes

$\Rightarrow$ Range search in a balanced BST is $O(\log n + k)$

3.1 Quad Trees

Main Idea:
Partition a square R into four equal quadrants, each being a child of R. Continue to split for any subsquare that contains more than one point.

Notes:

- Points on the lines belong to the left/bottom side.
- Each leaf stores at most one point
- Usually denote the quadrants as NW, NE, SE, SW

`Search:`
At each level, if leaf not reached, identify which child it belongs to, recurse on that child.

`Insert:`
Search, split leaf if there are two points

`Delete:`
Search, remove the point, discard any unneeded branches/splits

`Summary of Quad Trees Range Search:`

- height is unbounded by n
- height of quadtree: $h \in \Theta(\log_2 \frac{d_{max}}{d_{min}})$
- Worst-case to build initial tree = $\Theta(\#\text{nodes} \cdot \text{height}) = \Theta(nh)$
- Worst-case complexity of range search = $O(\#\text{nodes} \cdot \text{height}) = O(nh)$
- Space wasteful
- Can have very large height for bad distribution of points

3.2 KD-Trees

Main Idea:
Partition points into two (roughly) equal subsets. Starting with a vertical line, then using horizontal lines, alternating until every point is in a seperate region.

`Building a kd-tree`

1. Split on median: $\lfloor \frac{n}{2} \rfloor$ with vertial line.
2. Split alternating horizontal/vertical
3. Continue until each point is in a seperate region

`Summmary of kd-trees`

- Sort n points according to x-coord (root is median)
- Complexity: $\Theta(n \log n)$
- Height of tree: $\Theta(\log n)$
- Space: $O(n)$

`Range Search`

- Runtime: $O(\#\text{gray nodes} + \text{time to report children of black nodes}) = O(\sqrt{n} + \text{output})$

3.3 Range Trees

Main Idea:
A tree of trees

4 Tries

Main Idea:
A binary tree based on bitwise comparisons. Left child = 0, right child = 1. Keys that are stored are flagged.

`Search`

1. Left for 0, right for 1,
2. If flagged, found

`Insert`

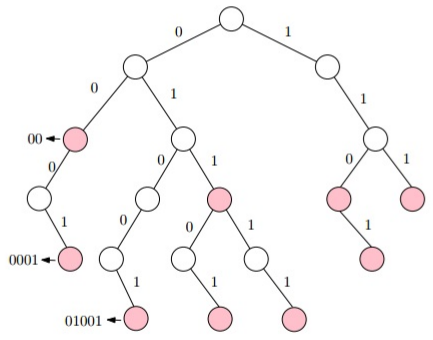
1. Search
2. If we finish at leaf, already there
3. If at internal node, flag.
4. If new path, insert extra nodes, flag (might need to flag where the old one ended if at leaf).

`Delete`

1. Search
2. If at internal node, unflag.
3. If at leaf, delete nodes until we reach a flagged node or one that has 2 children.

Tries

- Example: A trie for $S = \{00, 0001, 01001, 011, 01101, 01111, 110, 1101, 111\}$

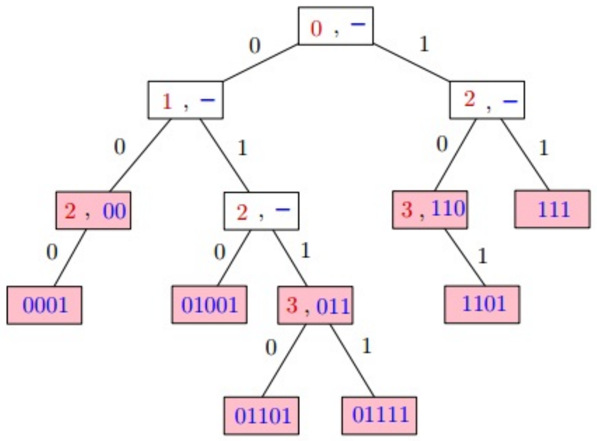


Summary of Tries

- Search: $\Theta(|x|)$
- Insert: $\Theta(|x|)$
- Delete: $\Theta(|x|)$
- $|x|$ is the length of the binary string (# of bits)

4.1 Compressed Tries (Patricia Tries)

Main Idea:
The nodes store the index where the bit differs



4.2 Suffix Tries + Suffix Trees

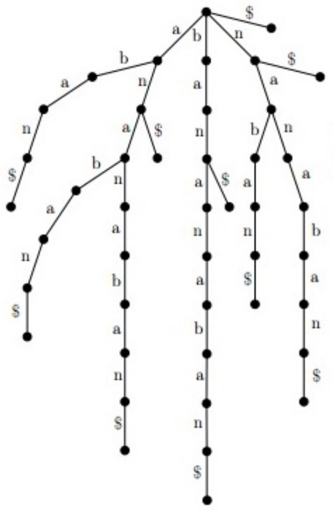
Main Idea:

A trie that stores all suffixes of a text T. A suffix tree is just a compressed trie.

Suffix Trie: Example

T =bananaban

{bananaban, ananaban, nanaban, anaban, naban, aban, ban, an, n}

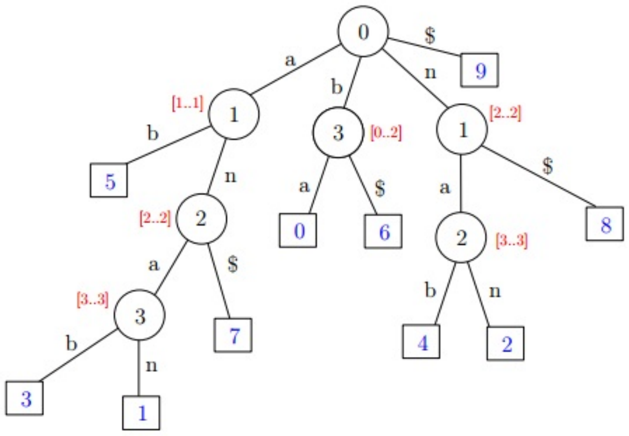


i	0	1	2	3	4	5	6	7	8	9
$T[i]$	b	a	n	a	n	a	b	a	n	\$

Suffix Tree (compressed suffix trie): Example

T =bananaban

{bananaban, ananaban, nanaban, anaban, naban, aban, ban, an, n}



i	0	1	2	3	4	5	6	7	8	9
$T[i]$	b	a	n	a	n	a	b	a	n	\$

5 Pattern Matching

Main Idea:
Trying to find our string of length m in our text of length n

5.1 Brute-force Algorithm

Main Idea:
Check if every character of our string matches at every position in our text.

$\Rightarrow$ This takes $O(n * m)$ time.

5.2 Knuth-Morris-Pratt

Main Idea:
Preprocess our string so that we can find matches in our text with a failure array (basically a DFA)

`Failure Array`

- $F[0] = 0$
- Columns are $P[1...j]$ compared to P
- $F[j]$ is the size of largest suffix of $P[1..j]$ as a prefix in P
- Remeber to use $F[j-1]$ when matching
- Computing the failure array takes $\Theta(m)$

$\Rightarrow$ Running time of KMP $= \Theta(n)$

5.3 Boyer Moore

Main Idea:
Preprocess our string so that we can find a Bad character jump or a Good Suffix jump. We always take the smaller number because it means it's a larger jump.

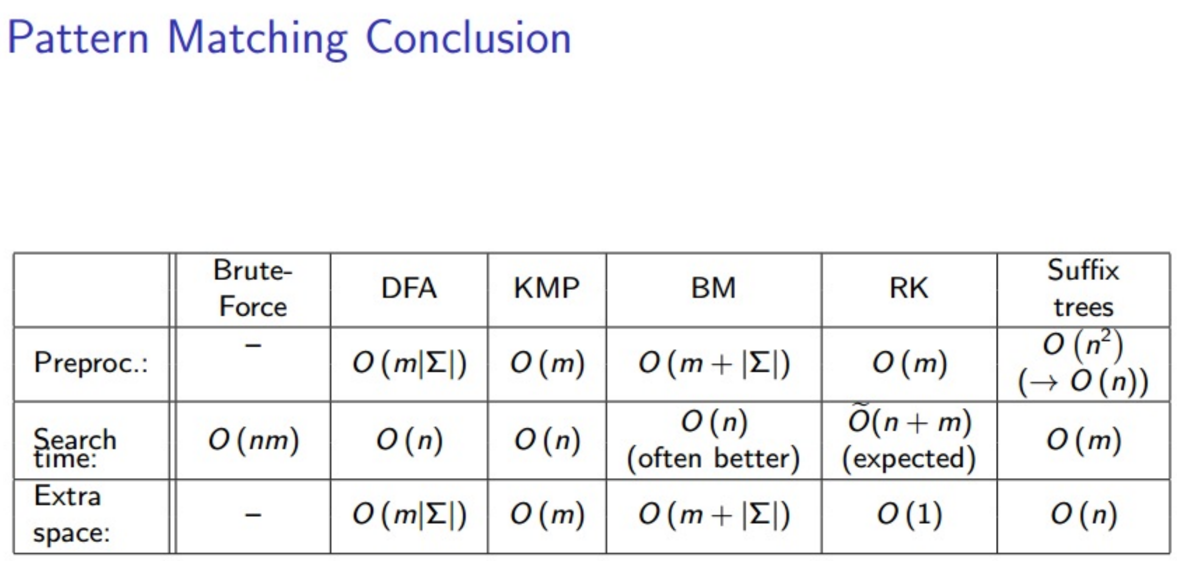
`Last Occurence Function`

1. Preprocess our pattern and build a last occurence table
2. For each of our characters in Σ, we write the last occurence in P, else -1

`Suffix Skip array`

1. Start at the end of the string
2. Ask where the suffix occurs and then subtract 1 from that

When we fail reading, we look into the suffix skip at the index we failed at, and compare with the character we failed to match and skip by the lower number = larger skip.



Pattern Matching Conclusion

	Brute-Force	DFA	KMP	BM	RK	Suffix trees
Preproc.:	–	$O(m\|\Sigma\|)$	$O(m)$	$O(m+\|\Sigma\|)$	$O(m)$	$O(n^2)$ $(\rightarrow O(n))$
Search time:	$O(nm)$	$O(n)$	$O(n)$	$O(n)$ (often better)	$\tilde{O}(n+m)$ (expected)	$O(m)$
Extra space:	–	$O(m\|\Sigma\|)$	$O(m)$	$O(m+\|\Sigma\|)$	$O(1)$	$O(n)$

Notes:

- KMP good for repeated search of small string. ex: Egrep
- BM good cause we can skip more characters
- RK depends on a good hash function (hard to come up with)
- Suffix Trees only search for our pattern. Good for 1 big file, many different searches
- In theory KMP > BM but not in practice.
- If going in blind, use KMP for best worst case.

6 Text compression

Goals of encoding schemes: Processing speed, Reliability, Security(encryption), Size***
We measure data compression with the Compression Ratio:

$$\frac{|C| \cdot \log |\Sigma_C|}{|S| \cdot \log |\Sigma_S|}$$

6.1 Huffman Coding

Main Idea:
Build a binary trie to store the decoding dictionary D. The more frequent characters will have shorter strings (smaller height).

`Building the trie`

1. Determine freq(c) and make mini-tries for each char holding their weight (the freq)
2. Merge two tries with the least weights, now its weight is $w(trie_1 + trie_2)$
3. Repeat until there is only 1 trie; D.

`Huffman Summary`

- Building the decoding trie takes $O(|S| + |\Sigma| \log |\Sigma|)$
- Decoding trie must be passed with the coded text

- Huffman is best we can do for 1 character at a time

6.2 Run Length Encoding

Main Idea: Encoding in blocks of 0s or 1s. First char is either a 1 or 0 and it alternates.

`Encoding`

1. If the string starts with a 1, then we start with a 1, same with 0.
2. Count the lenght of a block, place length(binary(k))-1 0s at the beginning, followed by the binary(k).

6.3 Lempel-Ziv-Welch

Main Idea:
Each character in coded text C either refers to a single character or a substring of S that both encoder and decoder have already seen

`Encoding`

1. Read in
2. Check dictionary
3. If its there, print it out
4. Add the current word + next char into dictionary.
5. Continue until the end

`Decoding`

1. Put what we see in input column. input ⇒ decodes to ⇒ Letter
2. Add the previous + current's first character, starting at code #128
3. If the decode # is the same as the current new code we want to input: Its the previous string + the previous' first character.
4. Continue until the end

6.4 Move to front

Main Idea:
We take advantage of locality in the data, we move an element to the front of the list after it is accessed. HOWEVER: MTF does not compress

`Encoding`

1. Find in our alphabet
2. Move it up to index 0 in our dictionary,
3. Push back things by 1 index
4. Continue encoding

`Decoding`

1. Take char at 0.
2. Move it up to index shown in our dictionary (start from the end),
3. Push back things by 1 index
4. Continue decoding

6.5 The Burrows-Wheeler Transform

Consider car:
car$
ar$c
r$ca
$car

⇒ When BWT is encoded, notice how the letters will be grouped together
Encoding:

1. Place all cyclic shifts in a list L
2. Sort L
3. Extract last characters from sorted shifts

Decoding:

1. BWT index — BWT —— Sorted — Sorted index
2. BWT is stable so multiple letters occur before other of the same letter.
3. START AT THE END OF CHARACTER FILE
4. See that it has sorted index i, so index i in sorted is our first character
5. Continue in the sorted index

⇒ Encoding cost: $O(n^2)$ possible in linear, Decoding cost: $O(n)$

6.6 Important sidenotes

REMEMBER: MTF or BWT don't compress the text, they are only transformations. HOWEVER: Combining BWT $\Rightarrow$ MTF $\Rightarrow$ RLE $\Rightarrow$ Huffman creates a very good compression scheme for english.
Decompression is just the reverse

6.7 Rabin Karp

O(n + m) time for a rolling hash function. in practice actually $O(n * m) \Rightarrow O(n^2)$
- Hash function is hard to come up with - Not sure if the pattern being hashed is actually the pattern we meant to hash
- Know how to run through the algorithms -