CS 240: Data Structures and Data Management **Fall 2021**

Final Help Session

Reminder: Final Exam is at either December 21, 4 pm, or December 22, midnight, depending on your chosen timeslot.

Note: This is a sample of problems designed to help prepare for the final exam. These problems do *not* encompass the entire coverage of the exam, and should not be used as a reference for its content.

1 True/False

For each statement below, write true or false.

a) False.

The step size for linear probing is always 1, so a second hash function is not involved.

b) False.

Suffix trees preprocess the text, not the pattern.

c) False.

While Binary Search is comparison-based, Interpolation Search is not. Interpolation Search requires its values to have a "subtraction" operation (in addition to a comparison operation), such as numbers, and does not work for values that do not have a defined subtraction operation (even if they can be compared).

d) False.

Depending on the distribution of points, the number of nodes can increase exponentially with height (up to $n/2$ nodes), while all of these nodes are visited. An example is if we had $n/2$ pairs of points, with early partitions separating the different pairs ($n/2$ internal nodes), but the points in each pair are close enough that $\Omega(h)$ partitions are needed to separate them. Range search can be as bad as $\Omega(nh)$ then.

e) True.

If the pattern appears at the start of the text, then the runtime is in $O(m)$, with no randomness.

f) False.

It is only valid if the sequence in the decoding algorithm covers all characters. For example, any string that begins with \$ cannot be decoded with BWT.

g) True.

m259xu

This document is for the exclusive use of m259xu.

The mismatch is always at the final character of the pattern, with the smallest overlapping shift being a shift by one step (since all other characters are the same). Thus, the pattern would be shifted by one step after each mismatch, similar to brute-force. Note that the runtime for KMP will still be $\Theta(n)$, because the character comparisons at each step will begin at the mismatched position, as opposed to the start of the pattern like brute-force.

h) False.

Chaining involves linked lists at every index, and it is always possible to add a new element to a linked list.

i) False.

2-4 Trees require that all leaves are at the same level, but an AVL Tree does not necessarily satisfy this condition.

j) True.

The dictionary for MTF changes as the characters are being encoded/decoded, since entries are moved to the front.

2 Multiple Choice

a) ii) $f(i) = i$

An arithmetic sequence guarantees $O(1)$ runtime for interpolation search, regardless of whether the target is in the array or not.

b) ii) $h_1(k) = (k \bmod 6) + 1$

This is the only hash function which does not map to 0, the other options can all map to 0. Double Hashing should avoid having h_2 mapping to a value that is not co-prime to the table size, and should especially avoid mapping to multiples of the table size.

c) iv) $h_1(k) = \lfloor \frac{1}{2} \cdot (k \bmod 13) \rfloor$

For cuckoo hashing, on the other hand, it's important for the hash functions to be able to map to every entry of the hash tables, preferably with uniform distribution. The function $(k \bmod 6) + 1$ does not map to 0, while $2 \cdot (k \bmod 4)$ does not map to odd values. The $k^2 \bmod 7$ function is a poor choice since it is completely dependent on $h_0(k)$.

The $\lfloor \frac{1}{2} \cdot (k \bmod 13) \rfloor$ function is not perfectly uniform, since there is only one residue, 12 mod 13, that maps to 6, while there are two residues mod13 for each of the other possible entries. This is still close enough to being a uniform hash function, and is certainly more uniform than $(k \bmod 6) + 1$ and $2 \cdot (k \bmod 4)$.

d) iii) 6

The root has side length $128 - 0 = 128 = 2^7$, while the deepest internal node has side length $92 - 88 = 28 - 24 = 4 = 2^2$. It takes $7 - 2 = 5$ edges to reach this deepest internal node from the root, and then there should be an additional level for the children of this internal node, resulting in a height of $5 + 1 = 6$.

m259xu

This document is for the exclusive use of m259xu.

e) ii) The root of the compressed trie always tests the first bit.

The bit being tested by the root is equal to the length of the longest common prefix over all strings (and is therefore equal to the earliest bit position in which the strings may differ). If there is a non-empty common prefix over all strings, then the root would not test the first bit.

Note that for a suffix tree, due to the presence of the empty suffix $, the root will always test the first bit.

f) iv) Suffix Array

Searching in a suffix array when $P \notin T$ requires $O(\log n)$ iterations, but each iteration would take $O(1)$ time, since the best-case is for the string comparison to terminate at the very first character every time, e.g., if the first character of P never appears in T, so the best-case runtime is $O(\log n)$.

Rabin-Karp and Knuth-Morris-Pratt must go through the entire T if $P \notin T$. Boyer-Moore Bad Character can shift after every character comparison, with each shift moving the pattern $\sqrt{n}$ steps forward, i.e., the pattern shifts do not overlap (such as when the characters in T do not appear in P), but this best-case runtime is in $\Theta\left(\frac{n}{\sqrt{n}}\right) = \Theta(\sqrt{n})$

g) I wonder...

3 Hashing

a) Double Hashing should probe the entire table before declaring failure if $h_1(k)$ is co-prime to m. In order to cause a failure with only two keys, we need to ensure that $h_1(k)$ is *not* co-prime with m for the second key. Since p is prime, we need $h_1(k)$ to be a multiple of p, which is only achieved when $h_1(k) = p$, since $h_1(k)$ cannot be any larger. So we must select the second key such that its h_0 value matches with the first key, while the h_1 value is p.

Possible Solution: 0, $p(p-1)$

b) In order to cause a failure in cuckoo hashing using four keys, we need to make sure that the union of possible slots for the four keys has cardinality 3 or less,

Two keys with the same h_0 must have their difference as a multiple of p, and two keys with the same h_1 must have their difference as a multiple of $p-1$. Two keys with both h_0 and h_1 as the same must, therefore, have a difference as a multiple of $p(p-1)$. Since the universe of keys is from 0 to p^2-1, this is only possible if one key is from $[0, p-1]$ and the other is from $[p(p-1), p^2-1]$

With four keys, we can have two pairs of keys such that each pair would share the same h_0 and h_1. But even then, this would result in four possible slots combined if the hash values differ between the two pairs, so we further need to ensure that all four keys must either have the same h_0 or the same h_1. Having the same h_0 means that even the two keys that lie in the range $[0, p-1]$ must have a difference as a multiple of p, which is impossible. So we need to ensure that the four keys all have the same h_1, so the two keys that lie in the range $[0, p-1]$ must have a difference of $p-1$ (only possible for keys 0 and $p-1$), and then the two keys that lie in the range $[p(p-1), p^2-1]$ would also have a difference of $p-1$ (only possible for keys $p(p-1)$ and p^2-1).

Possible Solution: 0, $p-1$, $p(p-1)$, p^2-1

The only possible solutions are a permutation of these four keys.

m259xu

This document is for the exclusive use of m259xu.

c) The interesting thing about the choice of $h_0(k) = k \bmod p$ and $h_1(k) = \lfloor k/p \rfloor$ is that there are no two keys in the universe for which both h_0 and h_1 match between the two keys. Given a pair of values for (h_0, h_1), there is only one key whose values correspond to that pair.

Therefore, if we consider a set of x values from h_0 and y values from h_1, then there are xy keys whose hash values fall into this set. Note that the actual number of slots in this set is only $x + y$.

For six keys to fail at insertion, we can select 3 values of h_0 and 2 values of h_1 (or vice versa) so that there are $3 \times 2 = 6$ keys that map only to these $3 + 2 = 5$ slots. Regardless of which hash values are picked, there will always exist a set of six keys that achieve this, forcing insertion to fail.

Suppose we choose the h_0 values as 0, 1, and 2, and the h_1 values as 0 and 1. Then the six keys are:

- Key with $h_0(k) = 0$ and $h_1(k) = 0$. This key is 0.
- Key with $h_0(k) = 0$ and $h_1(k) = 1$. This key is p.
- Key with $h_0(k) = 1$ and $h_1(k) = 0$. This key is 1.
- Key with $h_0(k) = 1$ and $h_1(k) = 1$. This key is $p + 1$.
- Key with $h_0(k) = 2$ and $h_1(k) = 0$. This key is 2.
- Key with $h_0(k) = 2$ and $h_1(k) = 1$. This key is $p + 2$.

Possible Solution: 0, 1, 2, p, $p+1$, $p+2$

4 Huffman Compression

a) `pswd: lull`

b) In Huffman compression, characters with greater frequency have shorter or equal-length encodings compared to characters with lower frequency. Here, the letter "ℓ" appears three times while "u" only appears once, but the former has a longer code (010) than the latter (11).

If we swapped the codes for ℓ and u, we would get a shorter string, so the original string was not optimal. Therefore, the string and dictionary combination could not have been generated by Huffman.

5 Boyer-Moore

a) No, the modification is not valid. A counterexample would be $P =$ `bba` and $T =$ `bbba`. Then we would have:

b	b	b	a	
b	b	(a)		
		b	b	a

The first character comparison is a mismatch, so the pattern would shift so that the mismatch in the text aligns with the first occurrence of `b` in the pattern (instead of the last). But this causes the pattern to shift too far, beyond the end of the text, causing the algorithm to terminate with failure. But the pattern `bba` actually does appear in the text `bbba`.

m259xu

This document is for the exclusive use of m259xu.

Let's say we have a mismatch between $T[i]$ and $P[j]$. If we shift the pattern such that $T[i]$ aligns with $P[\ell]$, with $\ell < j$, then the pattern shifts being skipped are those in which $T[i]$ is aligned with indices of P that are after ℓ and before j. The last-occurrence function is valid since $P[\ell]$ is the last occurrence of the character at $T[i]$, so the characters between $P[\ell]$ and $P[j]$ would not match with $T[i]$ anyway.
But if we use the first occurrence of $T[i]$ instead, to align $T[i]$ with $P[\ell]$, then there may be later occurrences of this character that appear between $P[\ell]$ and $P[j]$, and those pattern shifts would be skipped despite possibly having a correct match for the whole pattern.

b) Yes, this is valid. If there is a mismatch between $T[i]$ and $P[j]$, we still use the last-occurrence function to determine the shift. The pattern shift here does not depend on which direction we scanned the pattern with. Consulting the last-occurrence function ensures that we do not skip any potentially correct pattern shifts, so the resulting shift is still correct.

The main reason why Boyer-Moore with Bad Character scans characters from the end of the pattern is because this makes it more likely for the first mismatch to be near the end of the pattern. Therefore, when $T[i]$ mismatches with $P[j]$, it is more likely for the last occurrence of $T[i]$ in P to be at the left side of $P[j]$, where a greater distance between the two results in a greater shift. On the other hand, scanning from the start of the pattern would make it likely for the first mismatch to arise near the start of the pattern. If the last occurrence of $T[i]$ in P is to the right of $P[j]$, then we would perform the tiny brute-force shift (only one step forward). And even if the last occurrence of $T[i]$ in P is to the left of $P[j]$, the distance would likely be small, resulting in a small shift.

Basically, the main reason for scanning from the end is because it tends to result in large pattern shifts from the last-occurrence function, leading to faster search time. Scanning from the front is still valid and correct, but tends to be much slower.

c) Yes, this is valid. Let $L()$ be the last-occurrence function of P, and $L'()$ be the last-occurrence function of $P[0 \ldots m-2]$. The values for L and L' are the same for all characters except for the last character of P, i.e., the character at $P[m-1]$. The value of $L'(P[m-1])$ would indicate its last occurrence in $P[0 \ldots m-2]$, which would be its second-to-last occurrence in P (this would be -1 if it only appears once in P).

When searching for P in T, if there is a mismatch between $T[i]$ and $P[j]$, we consult the last-occurrence function for the character at $T[i]$. If this character is not equal to $P[m-1]$, then L and L' will return the same value and the shift is valid.

On the other hand, if this character is $P[m-1]$, this implies that $j < m-1$ (since $P[j]$ mismatched with $T[i]$). Let $L'(P[m-1]) = \ell$. Since index ℓ is the second-to-last occurrence of this character in T, with its last occurrence being at index $m-1$, this character does not appear in any index between ℓ and j (since j is before $m-1$). If using L' causes any pattern shifts to be skipped, then these skipped pattern shifts would align $T[i]$ with some index between ℓ and j. But $T[i]$ would mismatch for those shifts, so these pattern shifts will always fail. Thus, it is valid to use L' instead of L, since no correct pattern shifts can be skipped.

In general, when checking the last occurrence of the character at $T[i]$ and we find that it is located to the right of $P[j]$ (so we would normally perform the brute-force shift), it would actually be valid to use the second-to-last occurrence of this character instead. And if that is also to the right of $P[j]$, we can then use the third-to-last occurrence instead, and so on. Preprocessing all these values for all characters, however, is rather time-consuming, so we generally settle for just the last-occurrence function. But it's safe to use the second-to-last occurrence for the character at $P[m-1]$, since any mismatch for which that character appears in T would have its last occurrence in P to the right of $P[j]$.

m259xu

This document is for the exclusive use of m259xu.

6 Compressed Trie Height

a) To maximize the height of a compressed trie, we need to maximize the depth of just one single leaf. This requires maximizing the number of internal nodes in its path from the root. However, each internal node must have at least two children in a compressed child, where the other child must have at least one leaf in its subtrie. Since there are $n-1$ other leaves in the trie, there can be at most $n-1$ internal nodes in the path to this single leaf.

   This longest path would therefore contain $n-1$ internal nodes followed by the leaf, and would therefore contain $n-1$ edges, which is the maximum height.

   Maximum Height: $n-1$

b) To minimize the height, we want to fill up every level, except possibly the last level. Otherwise, if there is some level ℓ (not bottom level) which is not full, then we can remove a string whose leaf is in the bottom level to replace it with a string that ends at level ℓ,

   It is not necessary to fill up every level to minimize the height, but it is sufficient to try filling up every level (except bottom level) in order to minimize the height.

   Note that each node can have at most $(|\Sigma|+1)$ children, **but** the end-character child must always be a leaf, since there cannot be further characters after the end-character in the string. But if we ignore the end-character edges, then the rest of the trie forms a $|\Sigma|$-ary tree where we can try to fill up every level (except possibly the last level).

   Therefore, for a trie of height h, we have:

$$n \le (\text{Max \# leaves when height is } h) = (\mathit{Max\#}\text{ End-Char Leaves}) + (\mathit{Max\#}\text{ Non-End-Char Leaves})$$

   Every internal node can have an end-character child. Therefore, the maximum number of end-character edges (each of which lead to a leaf) is equal to the maximum number of internal nodes. For a trie of height h, the internal nodes would form a $|\Sigma|$-ary tree of height $(h-1)$.

$$\begin{aligned}(\text{\# End-Char Leaves}) \le (\text{\# Internal Nodes}) &\le (\text{Maximum Size of } \Sigma\text{-ary tree of height } h-1)\\ &= \sum_{\ell=0}^{h-1} |\Sigma|^{\ell}\end{aligned}$$

   For the leaves that do not arise from end-character branches, these are the leaves from the $|\Sigma|$-ary tree formed by excluding the end-character branches. The maximum number of such leaves arise when this $|\Sigma|$-ary tree is full, where there would be $|\Sigma|^h$ leaves. Therefore,

$$\begin{aligned} n &\le (\text{Max \# End-Char Leaves}) + (\text{Max \# Non-End-Char Leaves})\\ &\le \sum_{\ell=0}^{h-1} |\Sigma|^{\ell} + |\Sigma|^h = \sum_{\ell=0}^{h} |\Sigma|^{\ell}\\ \implies n &\le \frac{|\Sigma|^{h+1}-1}{|\Sigma|-1}\\ \implies h &\ge \log_{|\Sigma|}((|\Sigma|-1)n+1)-1\end{aligned}$$

   Since the height must be an integer, we apply the ceiling function to get the minimum height.

   Minimum Height: $\left\lceil \log_{|\Sigma|}((|\Sigma|-1)n+1) \right\rceil - 1$

m259xu

This document is for the exclusive use of m259xu.

7 Karp-Rabin

a) $h(\texttt{TAGCAT}) = 15$.

T	G	C	C	G	A	T	G	T	A	G	C	T	A	G	C	A	T
T	(A)																
	(T)																
		(T)															
								T	A	G	C	(A)					
										(T)							
												T	A	G	C	A	T

Table 1: Table for Rabin-Karp problem.

The parentheses indicate character mismatches, though this is not required by the problem.

b) Map each character to its weight $w(\texttt{A}) = 1$, $w(\texttt{C}) = 2$, $w(\texttt{G}) = 3$, $w(\texttt{T}) = 4$. Then $h(T[i+1 \dots i+m]) = h(T[i \dots i+m-1]) - w(T[i]) + w(T[i+m])$.

c) There are several possible solutions here. It is necessary for there to be $\Omega(n)$ false positives for which $\Omega(m)$ characters are compared in each of these false positives.

A simple observation is that `CC` has the same "weight" as `AG` or `GA`. It is useful for the text to be composed of a single character, since this ensures that all length-m substrings are identical.

One possible answer is for $T = \texttt{C}^n$ (n copies of `C`) while $P = \texttt{C}^{m-2}\texttt{AG}$ ($m-2$ copies of `C` followed by `AG`). Note that every length-m substring of T would be $\texttt{C}^m$. We have:

$$h(P) = \overbrace{2 \cdot (m-2)}^{m-2 \text{ occurrences of } \texttt{C}} + \overbrace{1}^{\texttt{A}} + \overbrace{3}^{\texttt{G}} = 2m - 4 + 4 = 2m$$

$$h(\text{any length-}m \text{ substring of } T) = \overbrace{2m}^{m \text{ occurrences of } \texttt{C}}$$

Since these are the same, every length-m substring of T yields the same hash value as P, leading to a false positive where character comparisons are performed. There are $\Theta(n)$ such substrings (specifically $n-m+1$ substrings), each requiring $\Theta(m)$ character comparisons (specifically $m-1$ comparisons since the first $m-2$ characters match). Thus, we have $\Omega(nm)$ character comparisons.

Another similar solution would be to have $T = \texttt{G}^n$ with $P = \texttt{G}^{m-2}\texttt{CT}$ or $P = \texttt{G}^{m-2}\texttt{TC}$.

m259xu

This document is for the exclusive use of m259xu.

8 KD-Trees

a) See Figure 1.

b) See highlighted nodes of Figure 1.

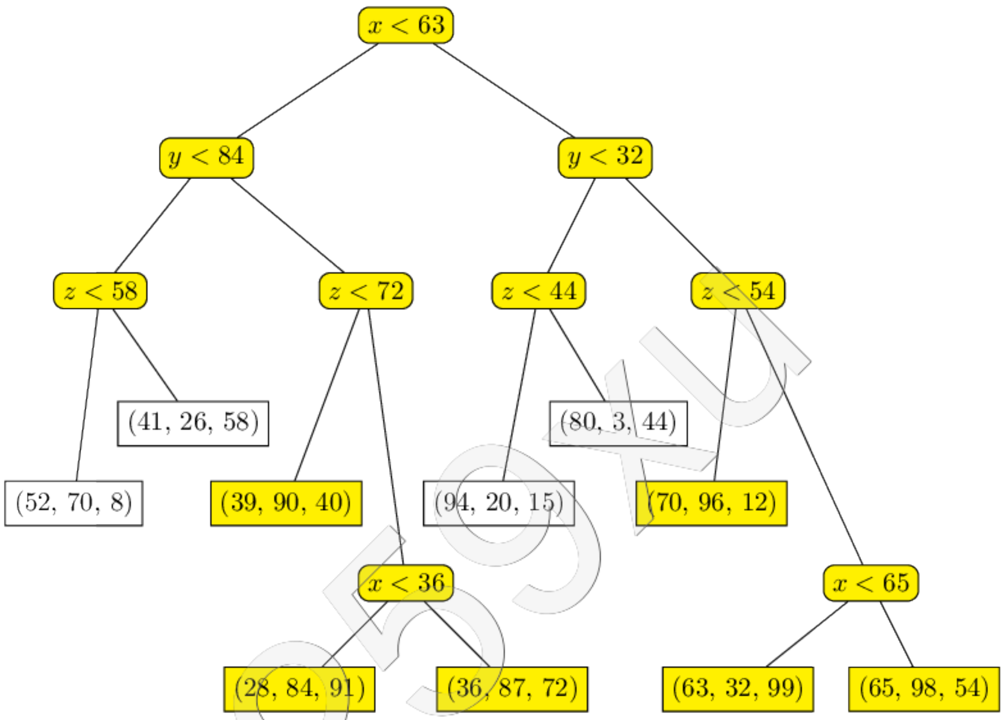


Figure 1: 3D kd-tree

The region corresponding to an internal node is given by the constraints set by its *proper* ancestors. For example, the node labeled $x < 36$ corresponds to the region defined by $x < 63$, $y \geq 84$, and $z \geq 72$. Note that this region includes both $x < 36$ and $x \geq 36$ (provided that $x < 63$), since the label $x < 36$ is only used to differentiate between its two subtrees, and is not relevant for the region of the node itself. Therefore, this node is considered as intersecting the range query and both of its children are visited.

If a region intersects with the range query, then both of its children will be recursed on even if a child region does not intersect with the query. If any node is visited, its sibling must also be visited. Despite there being so many leaves being visited, only $(65, 98, 54)$ is actually reported.

m259xu

This document is for the exclusive use of m259xu.

9 Suffix Arrays

The idea is that the characters in T' also represent the first characters of each of the suffixes in T, so sorting the characters in T' would correspond directly to the elements of A^S.

```
Algorithm 1: RecoverSuffixArray ($T'$, $A^S$)
Input: Scrambled text $T'$, Suffix Array $A^S$ of $T$
Output: Original text $T$
1  if $T'$ does not have an end-character then
2      $T'.append(\$)$;
3  end
4  $SortedT' \leftarrow BucketSort(T, R = 27)$, where $ maps to 0 and the $i$-th English letter maps to $i$ for
   BucketSort;
5  $len \leftarrow A^S.size$;
6  $T \leftarrow$ empty character array of size $len$;
7  for $j = 0$ to $len - 1$ do
8      $ind \leftarrow A^S[j]$;
9      $T[ind] \leftarrow SortedT'[j]$;
10 end
11 Return $T$
```

Lines 7-8 could have been written as $T[A^S[j]] \leftarrow SortedT'[j]$, but this may be harder to read.

Correctness

The characters in T' represent the first letters of the suffixes in T. Since sorting initially considers the first character, this means that the characters in $SortedT'$ represent the first characters of the suffixes of T when the suffixes are sorted.

We now show that lines 8-9 assign the correct characters to T. If $A^S[j] = ind$, this means that the suffix of T which begins at $T[ind]$ would have sorted index j over all suffixes of T, from the definition of suffix array. The first character of this suffix must therefore be the character at $SortedT'[j]$, so lines 8-9 assign the correct character to $T[ind]$.

Finally, we note that the suffix array A^S covers all of the suffixes of T, so repeating lines 8 and 9 for all indices of A^S will retrieve the first characters of all suffixes of T (no duplicate suffixes), so the entire T will be completed when the loop terminates.

Time Complexity

BucketSort in line 4 has runtime $\Theta(n + R) = \Theta(n + 53) = \Theta(n)$. The body of the `for` loop takes $O(1)$ time and it runs n times, so lines 7-9 take $O(n)$ time. All other steps take $O(1)$ time, so the overall runtime is in $\Theta(n)$.

m259xu

This document is for the exclusive use of m259xu.

10 Run-Length Encoding

a) Since Run-Length Encoding simply codes the lengths of the runs, the objective here is to determine how to partition n bits into blocks of runs that results in the longest coded text.

First, we observe that a run of length k is encoded into a codeword of length $\lfloor \log k \rfloor + (1 + \lfloor \log k \rfloor) = 2\lfloor \log k \rfloor + 1$, since the binary representation of k has $(1 + \lfloor \log k \rfloor)$ bits, while there are $\lfloor \log k \rfloor$ leading 0s.

Note that the only instance of k in this expression is in the term $\lfloor \log k \rfloor$. This indicates that a run whose length is in the range $k \in [2^b, 2^{b+1})$ would be encoded into the same length as a run length of exactly 2^b, since $\lfloor \log k \rfloor = b = \lfloor \log(2^b) \rfloor$. For example, a run of length 13 will have a codeword length of $2\lfloor \log(13) \rfloor + 1 = 2(3) + 1 = 7$, which is the same length as the codeword for a run of length 8, since $2\lfloor \log(8) \rfloor + 1 = 2(3) + 1 = 7$.

Since we want to maximize the size of C, it does not help to have runs whose lengths are not powers of 2, since the encoded length is the same if the length was reduced to a power of 2.

We now consider the encoding of runs whose lengths are powers of 2:

- A run of length 1 is encoded to a codeword of length $2\lfloor \log(1) \rfloor + 1 = 1$.
- A run of length 2 is encoded to a codeword of length $2\lfloor \log(2) \rfloor + 1 = 3$.
- A run of length 4 is encoded to a codeword of length $2\lfloor \log(4) \rfloor + 1 = 5$.
- A run of length 8 is encoded to a codeword of length $2\lfloor \log(1) \rfloor + 1 = 7$.

There is no need to check further, since it is clear that doubling the run length would increase the encoded size by $+2$, so the length of the codeword will now always be smaller than the corresponding run length. Thus, the worst-case must arise with either $k = 2$ or $k = 4$, where the codeword is actually longer than the run.

It can be observed that the derivative of $2 \log k + 1$ is $2/n$, which is a decreasing function, so it can only intersect at most twice with k, whose derivative is a constant. The floor function can be ignored when considering only powers of 2. One intersection is at $k = 1$, and the other is between 4 and 8, so the codeword length can only exceed the run length between these two points. Therefore, it suffices to consider only $k = 2$ and $k = 4$.

Comparing the compression ratios of a single run, we see $k = 2$ with a ratio of $3/2 = 1.5$, and $k = 4$ with a ratio of $5/4 = 1.25$. The ratio with $k = 2$ is bigger (i.e., worse), so the worst-case should exploit runs of length 2.

Worst-Case String: $(0011)^{n/4} = 001100110011\ldots$

Of course, the string $(1100)^{n/4}$ would also be correct, since what really matters here are the lengths of the runs, not on which runs are 0s or 1s.

Compression Ratio: $\frac{|C|}{|S|} = \frac{3(n/2)+1}{n} = 1.5 + \frac{1}{n}$

We have $|C| = 3(n/2) + 1$ since S consists of $(n/2)$ runs of length-2, each of which encode to a 3-bit codeword, while the $+1$ is from the very first bit that is always added to C and is not part of any codeword.

m259xu

This document is for the exclusive use of m259xu.

b) For the best-case, a natural guess would be to have all bits as a single run, since the codeword length grows logarithmically with the run length. We can therefore try to prove that reducing the number of runs will not increase the size of $|C|$.

We will later discover that the optimal case is not always a single run alone, but this attempt to improve $|C|$ should reveal the optimal case nonetheless.

Recall that if a run's length is a power of 2, then its encoding will have the same length even if the run length was reduced until it reaches a power of 2. Conversely, we can keep increasing the length of a run as long as it does not reach the next power of 2 to ensure that the corresponding codeword has the same length, thus improving the compression ratio.

Theorem 1. *Let S be a binary string of length n, and let C be the Run-Length Encoding of S.*

- *If n is a power of 2 and S contains exactly two runs, one of length $n-1$ and one of length 1, then $|C|$ has minimum length over all encodings from source length n.*
- *If n is not a power of 2 and S contains a single run of length n, then $|C|$ has minimum length over all encodings from source length n.*

The proof statement may not be obvious at this point, particularly the first case. However, we can generally try to work through of optimizing $|C|$ without actually knowing the results beforehand. Once we arrive at results that demonstrate the optimal cases, we can then adjust the proof statement accordingly.

The idea of the proof is to find scenarios in which two runs may be merged into a single run while ensuring that the resulting compression ratio does not become worse.

Proof. Suppose we have a set of run-lengths in S. We can then apply the following modifications to reduce the number of runs without increasing $|C|$.

i) If there are two or more runs with length 1, and there are at least three runs (so $n \geq 3$), then we can add two of the length-1 runs to one of the other runs (may have length 1 or more). The addition of two bits to this run may cause its run length to reach the next power of 2, but this would only add two bits to its codeword, which is compensated by the two bits saved by removing two runs of length 1. This step is repeated until there are either only two runs, or there is only one run of length 1.

ii) If there are two or more runs of length > 1, they can be combined into a single run. Let the two runs have length $n_1 \in [2^a, 2^{a+1})$ and $n_2 \in [2^b, 2^{b+1})$ with $1 \leq a \leq b$, so their original combined codeword lengths were $(2a+1)+(2b+1) = 2a+2b+2 \geq 2+2b+2 = 2b+4$. Then, after merging the two runs, the new run length is $n_1+n_2 \leq (2^{a+1}-1)+(2^{b+1}-1) \leq 2^{b+1}+2^{b+1}-2 < 2^{b+2}$. The codeword length would therefore be at most $2(b+1)+2 = 2b+4$, which is less or equal to the combined codeword lengths before merging. This process can be repeated until there is at most one run of length > 1.

These two modifications can be applied repeatedly until we either have only one run remaining, or we have two runs with at least one having length 1.

- Case 1: Two runs, one of length 1 and the other of length $n-1$. The current combined codeword length is $(2\lfloor \log(n-1)\rfloor + 1) + 1 = 2\lfloor \log(n-1)\rfloor + 2$.
  - Case 1a: If n is not a power of 2, then $\lfloor \log n\rfloor = \lfloor \log(n-1)\rfloor$. Therefore, we can merge these two into a run of length n, where the total codeword length is $2\lfloor \log(n)\rfloor + 1 = 2\lfloor \log(n-1)\rfloor + 1$, which is an improvement.

m259xu

This document is for the exclusive use of m259xu.

  - Case 1b: If n is a power of 2, then this must be the optimal S since the only other possibility (Case 2b) can be improved to this.
- Case 2: One run of length n. The codeword length is $2\lfloor \log n \rfloor + 1$.
  - Case 2a: If n is not a power of 2, then this is the optimal S, since the only other possibility (Case 1a) can be improved to this.
  - Case 2b: If n is a power of 2, then $\lfloor \log n \rfloor = \lfloor \log(n-1) + 1 \rfloor$. Therefore, we can split the run into two runs of length $(n-1)$ and 1, so the resulting combined codeword length is $(2\lfloor \log(n-1) \rfloor + 1) + 1 = 2(\lfloor \log n \rfloor - 1) + 2 = 2\lfloor \log n \rfloor$, which is an improvement.

If we began the proof with the natural guess that the optimal choice is always a single run of length n, this should be the point where we discover that it's not true for n as a power of 2, and we can adjust the theorem statement accordingly.

Therefore, for n as a power of 2, we can modify all possible run-length choices until we get two runs of lengths $(n-1)$ and 1, which must be optimal. Similarly, for n not as a power of 2, we can modify all possible run-length choices until we get a single run of length n, which must be optimal.

Note that these might not be the *only* choices of S that are optimal. There may be other choices that result in the same minimized length of $|C|$. However, for this problem, we only need to find one such S for each value of n.

□

Best-Case String:

$$\begin{cases} 0^n, & \text{if } n \text{ is not a power of 2;} \\ 0^{n-1}1 & \text{if } n \text{ is a power of 2.} \end{cases}$$

Compression Ratio:

$$\begin{cases} \frac{2\lfloor \log n \rfloor + 2}{n}, & \text{if } n \text{ is not a power of 2;} \\ \frac{2\lfloor \log n \rfloor + 1}{n}, & \text{if } n \text{ is a power of 2.} \end{cases}$$

Note that the proof only compared the combined codeword lengths; the actual C would still have one additional bit at the start, which is not part of any codeword.

m259xu

This document is for the exclusive use of m259xu.

11 Consecutive Trie Strings

There are several possible algorithms, the simplest of which involves ideas from range search:

```
Algorithm 2: Consecutive ($b_1$, $b_2$)
Input: Binary strings $b_1$ and $b_2$
Output: True or False, depending on whether $b_1$ and $b_2$ are consecutive in $T$
1 Perform a 1-dimensional range search on $T$ using the range $[\min(b_1, b_2), \max(b_1, b_2)]$ to get boundary
   nodes and top inside nodes;
2 if there exists a top inside node then
3     Return False;
4 end
5 Return True;
```

Correctness

We need to show that the existence of an inside node is equivalent to b_1 and b_2 not being consecutive. For both directions, observe that a 1D range-search should report all points that are lexicographically between b_1 and b_2 inclusive.

If b_1 and b_2 are consecutive, then the range-search would have no other keys in the range aside from b_1 and b_2 (which are both in boundary nodes). Therefore, there should be no inside nodes.

For the other direction, if b_1 and b_2 are not consecutive, then there exists at least one other key, b_3, which lies between b_1 and b_2. But in the range search, all boundary nodes are internal nodes except for the two nodes at the ends of the paths P_1 and P_2, which correspond to the leaves containing b_1 and b_2 (not necessarily in that order). Since b_3 is in the lexicographic range, it would have to be a leaf as an inside node (since no other boundary nodes are leaves). It follows that there exists at least one inside node: the leaf which contains b_3.

Time Complexity

The runtime to retrieve boundary nodes and top inside nodes is based on the number of boundary nodes. The boundary involving b_1 has at most $|b_1|$ nodes, while the boundary involving b_2 has at most $|b_2|$ nodes, so the runtime is in $O(|b_1| + |b_2|)$. Checking if there exists any top inside nodes takes constant-time for each boundary node, so overall runtime is in $O(|b_1| + |b_2|)$.

There are other algorithms, but they generally all resemble a range-search in some form.

m259xu

This document is for the exclusive use of m259xu.

12 Lempel-Ziv-Welch

a) Since we want the minimum number of codewords w, we want to encode the longest possible substring at each step. Initially, the dictionary only contains single-characters, so the length of the substring to be encoded must be 1. This will add a substring of length 2 to the dictionary. In the next step, the longest substring that is possible to encode is of length 2 (the substring we just added to the dictionary), and this will add a substring of length 3 to the dictionary. In the next step, the most we can possible encode is a length-3 substring, while adding a length-4 substring to the dictionary and so on.

We show that the above scenario is possible by proposing that the source text contains only a single character, repeated n times. Therefore, the substrings added to the dictionary will be composed only of that character, with increasing lengths.

Here, the lengths of the substrings being encoded are 1, then 2, then 3, and so on. We are guaranteed to be able to maintain this sequence until the last step, since we encode the longest possible substring that we have in the dictionary. Since there are a total of w codewords, we have:

$$\sum_{i=1}^{w-1} i < n$$
$$\implies \frac{w(w-1)}{2} < n$$
$$\implies w \in O(\sqrt{n})$$

For the last step, we encode the remaining characters of length $\leq w$ into a single codeword. Therefore,

$$\sum_{i=1}^{w} i \geq n$$
$$\implies \frac{w(w+1)}{2} \geq n$$
$$\implies w \in \Omega(\sqrt{n})$$

Together, we have $w \in \Theta(\sqrt{n})$.

b) There is a consecutive pair of codewords 00100001 01000000, i.e., $33 - 64$ in decimal, which occurs twice in this encoding. In the first such instance, after encoding a substring to 33, the encoder must have added a substring to the dictionary which corresponds to the substring for 33 plus the first character in the substring for 64 (which is the character that encodes to 15 btw). This new substring would be assigned the codeword 65.

Now, in the second instance of $33-64$, this indicates that the encoder once again encountered the substring for 33 followed by the substring for 64. But this means that the substring for 65 was encountered as well, so the encoder should have encoded this substring to 65 instead of the shorter substring to 33. Therefore, this encoding could not have arisen from LZW, revealing that the intelligence must have been false.

m259xu

This document is for the exclusive use of m259xu.

13 Burrows-Wheeler Transform

Note that BWT always adds an end-character to the end of the string (if it's not already there). When we consider a cyclic shift that begins at index i for $i \neq 0$, the last character of the cyclic shift is always at index $i-1$, i.e., one step to the left of the start. For $i = 0$, the last character is the end-character, since it corresponds to the original string.

a) $S = (\texttt{AH})^k\texttt{A}$. The first cyclic shift in sorted order is always the one that begins with the end-character. For this shift, the last character is `A`.

   After that, the remaining shifts either begin with `A` or `H`, where the shifts that begin with `A` would come before the shifts that begin with `H` in sorted order. For the shifts that begin with `A`, the prefixes match up until the end-character is reached, which breaks the ties in sorting, so the later cyclic shifts (i.e., starting at a later index) would come before the earlier cyclic shifts. All of these cyclic shifts would end with `H`, except for the cyclic shift at index 0 (last in sorted order, since the end-character is furthest away), which ends with the end-character `$`.

   Finally, the remaining cyclic shifts all begin with `H`. They all end with `A`.

   Coded Text: $\texttt{AH}^k\texttt{\$A}^k$

b) $S = \texttt{H}^k\texttt{A}^\ell$. The first cyclic shift begins with the end-character, and ends with `A`.

   The next cyclic shifts are those that begin with `A`. For these shifts, they begin with a prefix full of `A`s, until the tie is broken by the end-character. All of these shifts (which begin with `A`) would also end with `A`, except for the shift that begins at the earliest `A` (last in sorted order, since the end-character is furthest away), which ends with `H`.

   Similarly, for the shifts that begin with `H`, the ties are broken by the first `A`. The shifts all end with `H`, except the one that begins at index 0 (last in sorted order), which ends with the end-character `$`.

   Coded Text: $\texttt{A}^\ell\texttt{H}^k\texttt{\$}$

c) $S = \texttt{A}^k\texttt{H}^\ell$. The first cyclic shift begins with the end-character, and ends with `H`.

   The next cyclic shifts all behind with `A`. They all begin with a prefix of `A`s, until the tie is broken by `H`, but here the `H` causes the shift to come later in sorted order. The first of these shifts in sorted order is the one that begins at index 0 (since it's furthest away from the next `H`), which ends with the end-character `$`. The remaining shifts that begin with `A` all end with `A`.

   For the shifts that begin with `H`, the ties are broken by the end-character `$`, which causes the shift to come earlier in sorted order. All of these shifts end with `H`, except the one beginning at the earliest `H` (which is last in sorted order, since the end-character is furthest away), which ends with `A`.

   Coded Text: $\texttt{H\$A}^{k-1}\texttt{H}^{\ell-1}\texttt{A}$

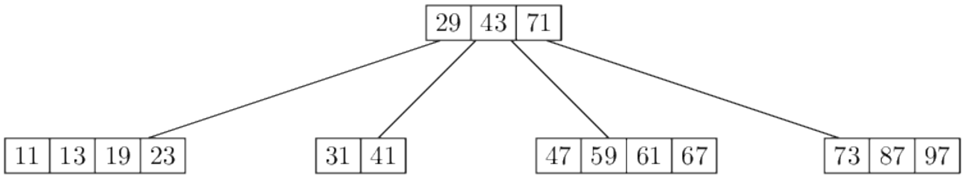
m259xu

This document is for the exclusive use of m259xu.

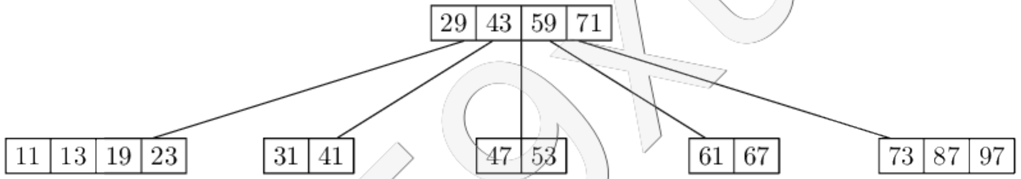
14 B-Trees

A B-Tree of order 5 is a 3-5 Tree, so each node can have a minimum of $(3-1) = 2$ keys and a maximum of $(5-1) = 4$ keys.

a) Insert (13):

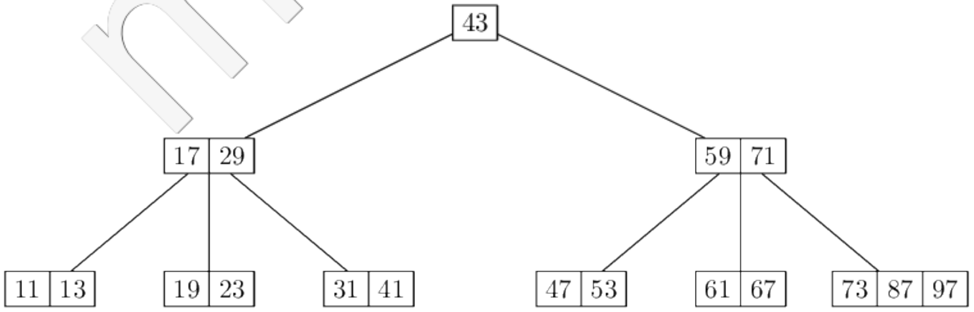


Insert(53), one node split:



The search for 53 ended at the 47/59/61/67 node, so 53 was added to it. After its addition, the node contained 47/53/59/61/67, which exceeds the maximum limit of 4 keys, causing an overflow. The node then splits, with the median 59 moving up to the parent node with 47/53 as its left child and 61/67 as its right child.

Insert(17), two node splits:



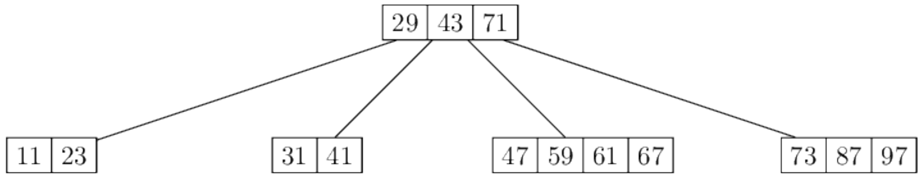
17 is inserted into the 11/13/19/23 node, which becomes 11/13/17/19/23, causing an overflow. The node splits with median 17 moving up to the parent node, but then the parent node becomes 17/29/43/59/71, causing another overflow. This parent node then splits to have 43 move up and form a new root node by itself.

Although each node should have a minimum of 2 keys, the root is always exempt from the minimum limit, and is allowed a minimum of one key, like here.

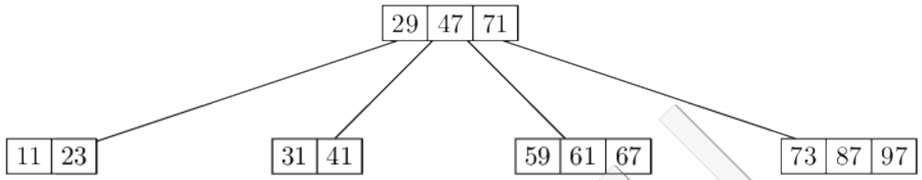
m259xu

This document is for the exclusive use of m259xu.

b) Delete(19):

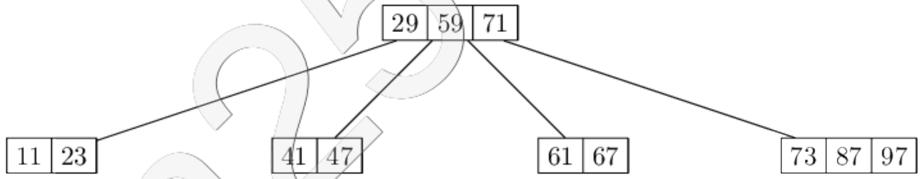


Delete(43), swap with successor (47):



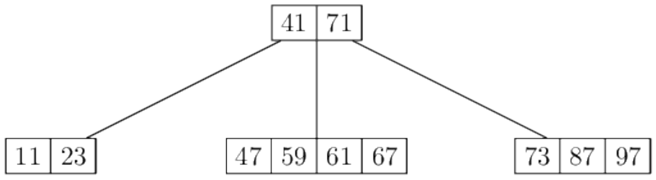
The successor is found by following the right child of the target **key** (this is not necessarily the right-most child of the node containing the target), and then selecting the leftmost key in this subtree (which can be found by following the leftmost child of each node, and then selecting the leftmost key when a leaf is reached).

Delete(31), transfer from right sibling:



Deleting 31 from the 31/41 node leaves only one key behind, causing an underflow (since the minimum is 2 keys). A transfer requires an immediate sibling with non-minimum capacity, i.e., more than 2 keys. The right sibling has 3 keys, so it can perform a transfer with the underflow node.

Delete(29), swap with successor (41), then merge with right sibling:



Note that the 73/87/97 node was not an immediate sibling to the underflow node (47), so it could not have transferred. With no immediate siblings to transfer, the only remaining option was to merge with a sibling (61/67). The underflow node (47), common parent key (59) and sibling node (61/67) merged into a single node. The loss of 59 from the parent node did not cause an underflow, so the deletion was complete.

This document is for the exclusive use of m259xu.

15 Maximal Difference

The idea is similar to a 1-dimensional range search, where we can find all nodes in the index range from a BST structured by index. We can check boundary points directly to determine the maximum and minimum values among those in the range. However, there could be many inside nodes, and checking them all might take $\omega(\log n)$ time.

To find the maximum and minimum values from inside nodes efficiently, we can take advantage of the top inside nodes (similar to 2-dimensional range search). Since all points in the subtree of the top inside nodes will fall in the range, we can pre-compute the maximum and minimum value of the entire subtree and store it in the top inside node.

Preprocessing: Construct a balanced BST based on the indices, i.e., the keys encompass the range 0 to $n-1$. Then augment the following fields to each node:

- `val`: the value $A[k]$, where k is the key of the node;
- `submax`: the maximum value from all nodes within its subtree;
- `submin`: the minimum value from all nodes within its subtree.

Range Query: The idea is to perform a 1-dimensional range search and find the maximum and minimum values by checking boundary points directly and utilizing the `submax` and `submin` fields of top inside nodes.

Algorithm 3: MaxDiff (i, j)

Input: Indices i and j with $i < j$
Output: Maximum difference using values of A indexed between i and j

```
1  Perform a 1-dimensional range search on our pre-processed BST using the range $[i, j]$ to get boundary
   nodes and top inside nodes;
   /* Initialize current maximum and minimum to A[i], which is in the range */
2  $currmax \leftarrow A[i]$;
3  $currmin \leftarrow A[i]$;
   /* check boundary nodes */
4  foreach boundary node $bnode$ do
5      if $bnode.key$ is in the range $[i, j]$ then
6          $currmax \leftarrow \max(currmax, bnode.val)$;
7          $currmin \leftarrow \min(currmin, bnode.val)$;
8      end
9  end
   /* check top inside nodes in order to cover all inside nodes */
10 foreach top inside node $tinode$ do
11     $currmax \leftarrow \max(currmax, tinode.submax)$;
12     $currmin \leftarrow \min(currmin, tinode.submin)$;
13 end
14 Return $currmax - currmin$;
```

m259xu

This document is for the exclusive use of m259xu.

Correctness

The correctness of the algorithm depends on the correctness of finding the maximum and minimum values in the input range query.

If an index is in the range, then it should either be a boundary node or an inside node. If it is a boundary node, then Lines 4 to 9 will check its value and update *currmax* and *currmin* accordingly. If it is an inside node, then during the preprocessing stage, its value is used to update the `submax` and `submin` fields of all its ancestors. This includes the top inside node from the current query, whose `submax` and `submin` fields are used to update *currmax* and *currmin* respectively, in Lines 10-13. So all indices in the range are used for updating *currmax* and *currmin*.

For the other direction, if an index is not in the range, then it is either an outside node or a boundary node. Outside nodes are never involved in the *currmax* and *currmin* updates in the algorithm. Boundary nodes are only involved if they are in the range, which is tested on Line 5, so boundary nodes that are not in the range will not contribute to *currmax* or *currmin*.

Therefore, *currmax* and *currmin* will correctly be updated with the maximum and minimum values from all indices in the range, so the maximum difference is correctly calculated.

Complexity

Retrieving the boundary and top inside nodes takes $O(\log n)$ time. There are $O(\log n)$ boundary nodes and $O(\log n)$ top inside nodes that are checked, with a constant number of operations (Lines 5-7 for boundary nodes, and Lines 11-12 for top inside nodes) performed for each of them. Therefore, the overall runtime for each query is in $O(\log n)$ time.

For space complexity, the special BST contains n nodes, and each node stores a key and three fields (which is constant space per node), so the overall space complexity is still in $O(n)$.

Example

With the array presented in the question, our special BST can look something like this:

m259xu

This document is for the exclusive use of m259xu.

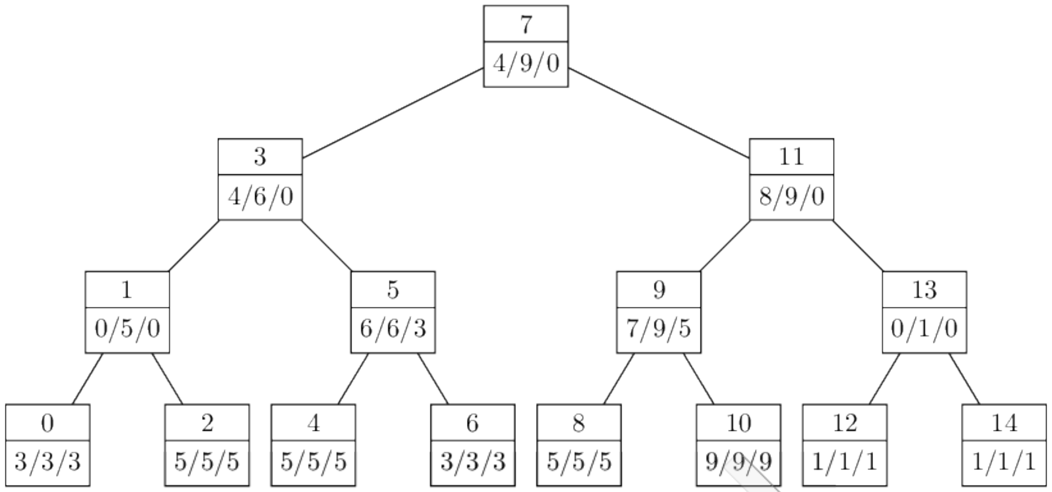


Figure 2: BST for Range Query

For each node, the number on top is the index or key, while the numbers below are `val`/`submax`/`submin`.

Running $MaxDiff(2, 9)$ results in boundary paths $P_1 = (7, 3, 1, 2)$ and $P_2 = (7, 11, 9)$, with 5 and 8 being the only top inside nodes. From the six boundary nodes, we have 7, 3, 2, and 9 in the range. Therefore,

$$\begin{aligned}
\text{Maximum} &= \max(7.\texttt{val}, 3.\texttt{val}, 2.\texttt{val}, 9.\texttt{val}, 5.\texttt{submax}, 8.\texttt{submax}) \\
&= \max(4, 4, 5, 7, 6, 5) = 7; \\
\text{Minimum} &= \max(7.\texttt{val}, 3.\texttt{val}, 2.\texttt{val}, 9.\texttt{val}, 5.\texttt{submin}, 8.\texttt{submin}) \\
&= \max(4, 4, 5, 7, 3, 5) = 3; \\
MaxDiff(2, 9) &= 7 - 3 = 4.
\end{aligned}$$

m259xu

This document is for the exclusive use of m259xu.