Tries

(“try”-s)

CMSC 420

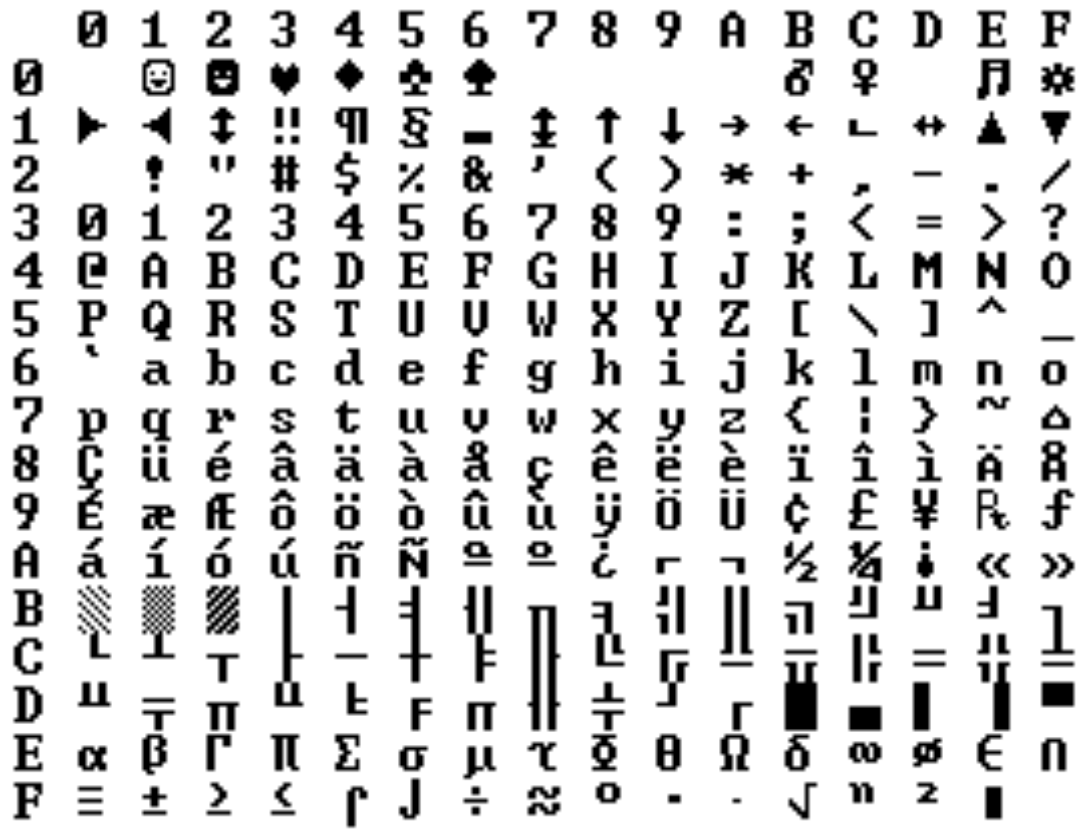
A brief history of character sets

- 1963: **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange (**ASCII**)
  - Used numbers from 32 through 127 to encode *unaccented* English characters and some common symbols, like >, <, _, etc
  - The numbers from 0 through 31 were used for **control characters** that are not all particularly useful nowadays, like BELL, printer feed, etc.

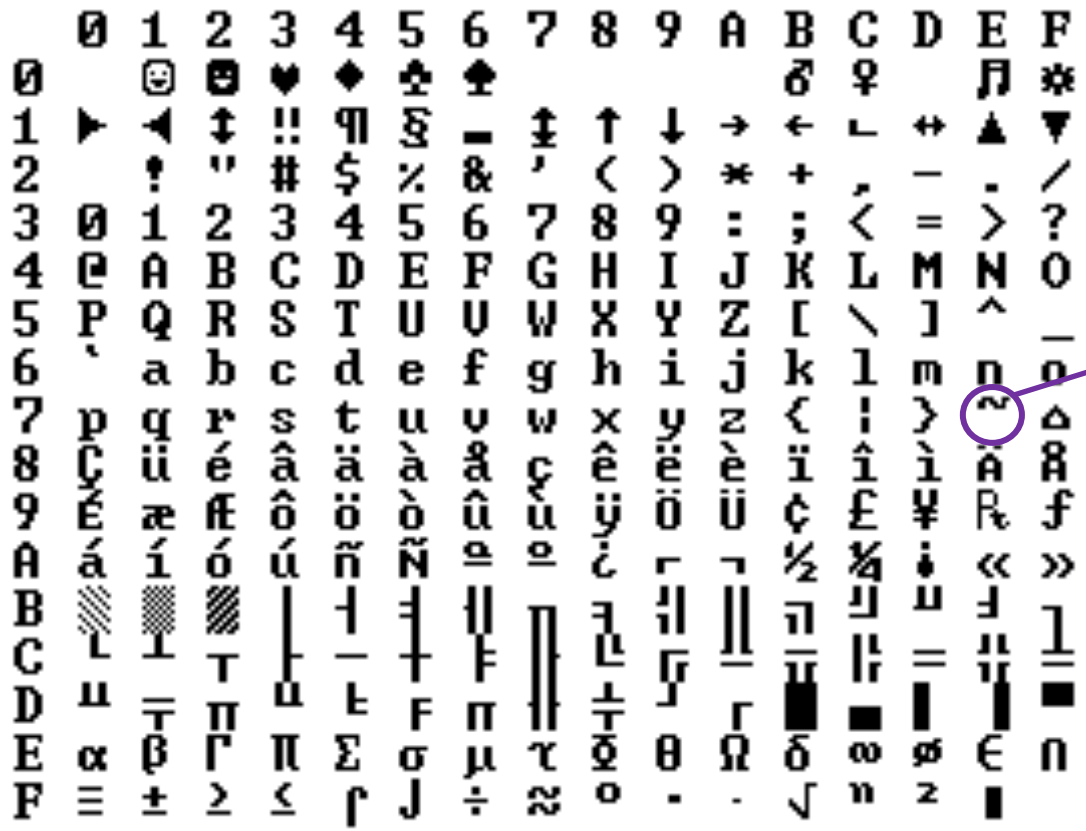
A brief history of character sets

- 1963: **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange (**ASCII**)
  - Used numbers from 32 through 127 to encode *unaccented* English characters and some common symbols, like >, <, _, etc
  - The numbers from 0 through 31 were used for **control characters** that are not all particularly useful nowadays, like BELL, printer feed, etc.
- Need only $\log_2 128 = 7$ bits to encode those, so with 1 extra bit, you can do whatever you want!
- Example: the IBM-PC, which was using an Intel 8088 CPU used a standard known as OEM…

OEM standard

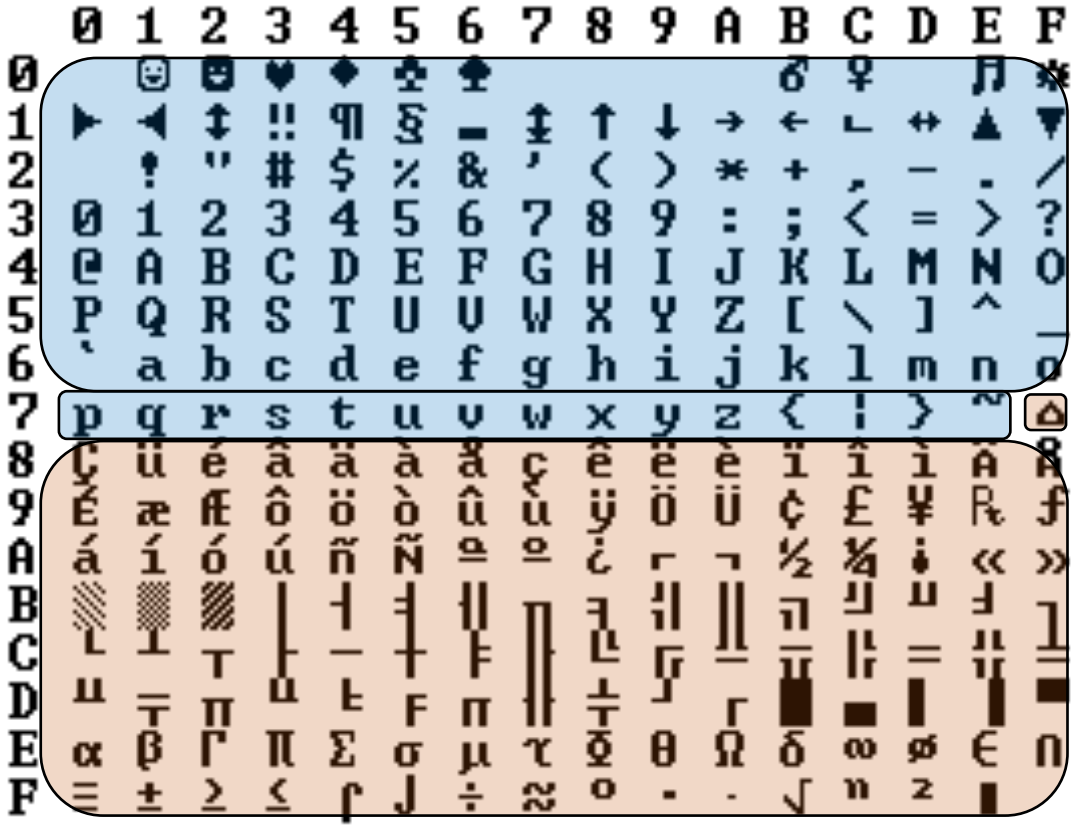


OEM standard



Last ASCII character, since $127_{(10)} = 0x7E$

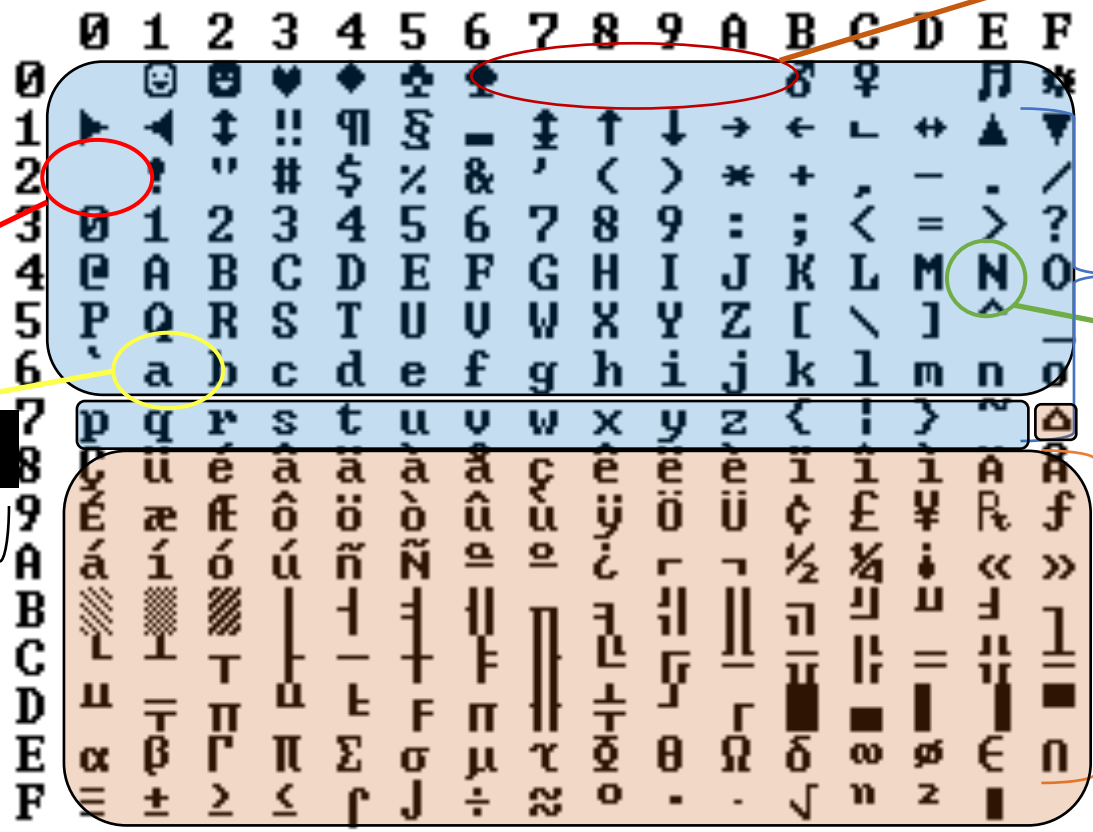
OEM standard



ASCII

I’m the IBM-PC and I do what I want with my top bit

OEM standard



Other whitespace chars (\t, LF, …)

Space character at 0x20 = $32_{(10)}$

ASCII

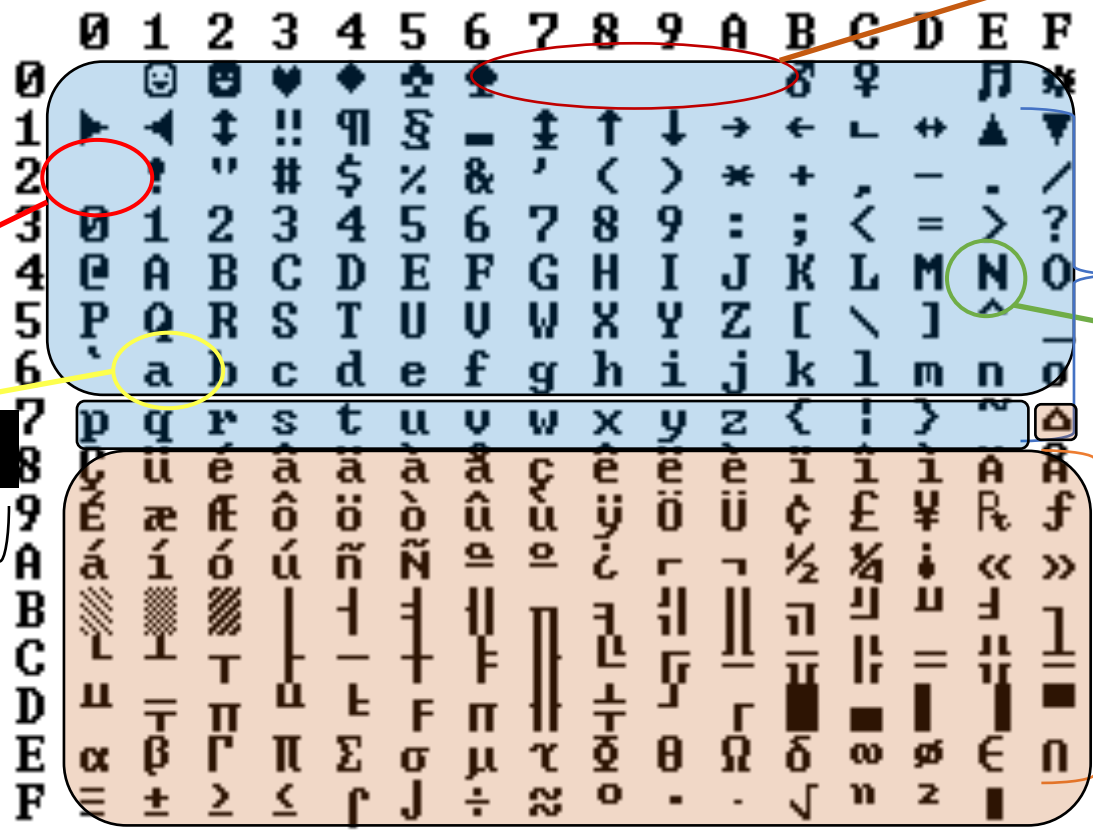
Capital ‘N’ at 0x4E = $79_{(10)}$

Lowercase ‘a’: 0x97

I’m the IBM-PC and I do what I want with my top bit

Justifies $h_{char}(c) = (int)c - 97$ when lowercase characters are used!

OEM standard



Other whitespace chars (\t, LF, …)

Space character at 0x20 = $32_{(10)}$

ASCII

Capital ‘N’ at 0x4E = $79_{(10)}$

Lowercase ‘a’: 0x97

Justifies $h_{char}(c) = (int)c - 97$ when lowercase characters are used!

I’m the IBM-PC and I do what I want with my top bit

- **This turned out to be a problem…**

ANSI Solution: Code pages

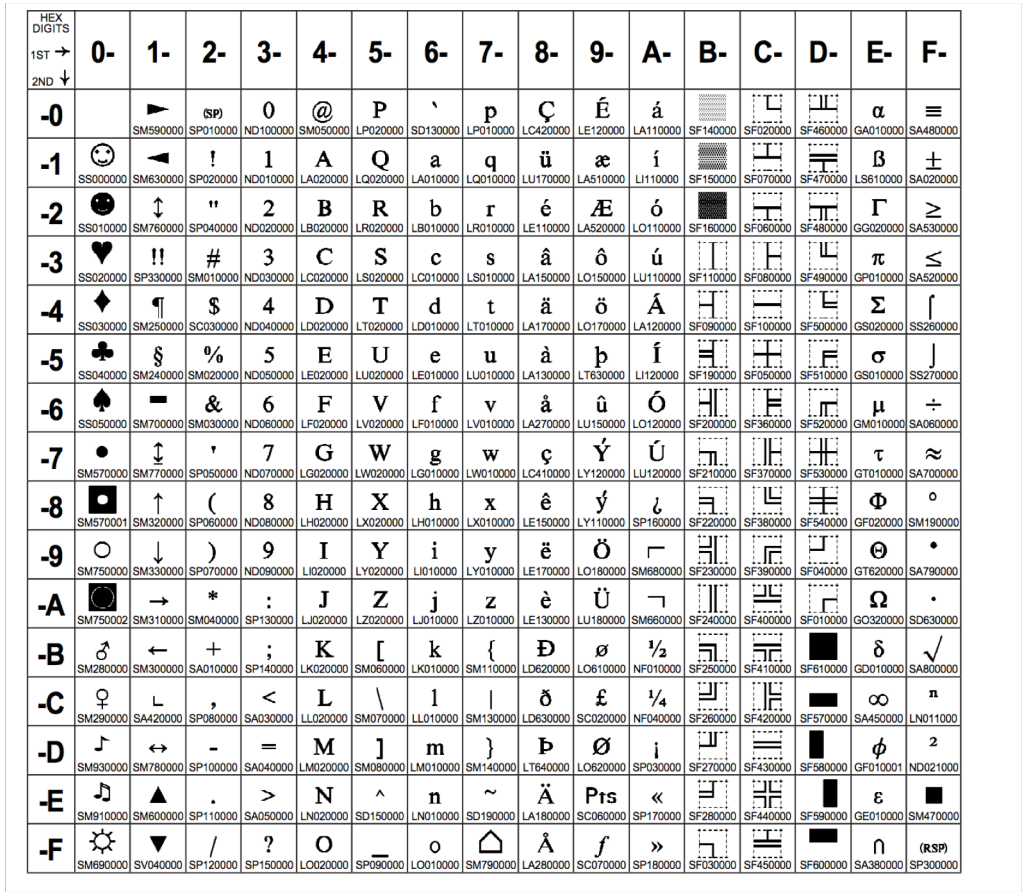
- **Axiom: Everybody agrees on the ASCII subset**
  - So, **7 LSBs interpretable in only one way across all computers in the world.**

ANSI Solution: Code pages

- **Axiom: Everybody agrees on the ASCII subset**
  - So, **7 LSBs interpretable in only one way across all computers in the world.**
- Top bit: Alphabet-specific
  - This led to different *code pages* for different alphabets!
  - E.g Icelandic: Code page #00861, Greek: Code page CP00851
  - So, as long as all Greek computers use ANSI CP00851 (reasonable assumption), they will all be able to transmit strings to one another without problems ☺

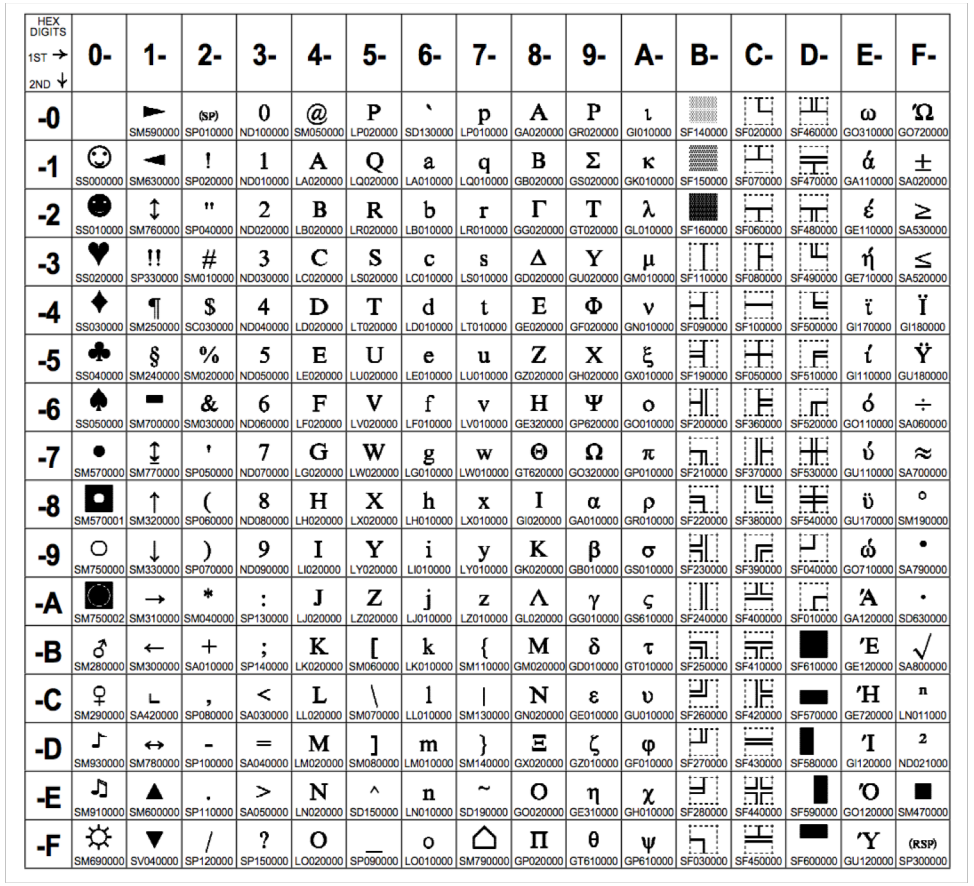
Look at these code pages

Icelandic



HEX DIGITS 1ST → 2ND ↓	0-	1-	2-	3-	4-	5-	6-	7-	8-	9-	A-	B-	C-	D-	E-	F-
-0		► SM590000	(SP) SP010000	0 ND100000	@ SM050000	P LP020000	` SD130000	p LP010000	Ç LC420000	É LE120000	á LA110000	▒ SF140000	┴ SF020000	╨ SF460000	α GA010000	≡ SA480000
-1	☺ SS000000	◄ SM630000	! SP020000	1 ND010000	A LA020000	Q LQ020000	a LA010000	q LQ010000	ü LU170000	æ LA510000	í LI110000	▓ SF150000	┼ SF070000	╬ SF470000	ß LS610000	± SA020000
-2	☻ SS010000	↕ SM760000	" SP040000	2 ND020000	B LB020000	R LR020000	b LB010000	r LR010000	é LE110000	Æ LA520000	ó LO110000	■ SF160000	├ SF060000	╠ SF480000	Γ GG020000	≥ SA530000
-3	♥ SS020000	!! SP330000	# SM010000	3 ND030000	C LC020000	S LS020000	c LC010000	s LS010000	â LA150000	ô LO150000	ú LU110000	│ SF110000	┤ SF080000	╙ SF490000	π GP010000	≤ SA520000
-4	♦ SS030000	¶ SM250000	$ SC030000	4 ND040000	D LD020000	T LT020000	d LD010000	t LT010000	ä LA170000	ö LO170000	Á LA120000	┬ SF090000	┬ SF100000	╘ SF500000	Σ GS020000	⌠ SS260000
-5	♣ SS040000	§ SM240000	% SM020000	5 ND050000	E LE020000	U LU020000	e LE010000	u LU010000	à LA130000	þ LT630000	Í LI120000	╡ SF190000	┼ SF050000	╒ SF510000	σ GS010000	⌡ SS270000
-6	♠ SS050000	▬ SM700000	& SM030000	6 ND060000	F LF020000	V LV020000	f LF010000	v LV010000	å LA270000	û LU150000	Ó LO120000	╢ SF200000	╞ SF360000	╓ SF520000	μ GM010000	÷ SA060000
-7	• SM570000	↨ SM770000	' SP050000	7 ND070000	G LG020000	W LW020000	g LG010000	w LW010000	ç LC410000	ý LY120000	Ú LU120000	╖ SF210000	╟ SF370000	╫ SF530000	τ GT010000	≈ SA700000
-8	◘ SM570001	↑ SM320000	(SP060000	8 ND080000	H LH020000	X LX020000	h LH010000	x LX010000	ê LE150000	ÿ LY110000	¿ SP160000	╕ SF220000	╚ SF380000	╪ SF540000	Φ GF020000	° SM190000
-9	○ SM750000	↓ SM330000	) SP070000	9 ND090000	I LI020000	Y LY020000	i LI010000	y LY010000	ë LE170000	Ö LO180000	⌐ SM680000	╣ SF230000	╔ SF390000	┘ SF040000	Θ GT620000	• SA790000
-A	◙ SM750002	→ SM310000	* SM040000	: SP130000	J LJ020000	Z LZ020000	j LJ010000	z LZ010000	è LE130000	Ü LU180000	¬ SM660000	╨ SF240000	╩ SF400000	┌ SF010000	Ω GO320000	· SD630000
-B	♂ SM280000	← SM300000	+ SA010000	; SP140000	K LK020000	[SM060000	k LK010000	{ SM110000	Ð LD620000	ø LO610000	½ NF010000	╖ SF250000	╦ SF410000	■ SF610000	δ GD010000	√ SA800000
-C	♀ SM290000	∟ SA420000	, SP080000	< SA030000	L LL020000	\ SM070000	l LL010000	| SM130000	ð LD630000	£ SC020000	¼ NF040000	╙ SF260000	╠ SF420000	▬ SF570000	∞ SA450000	n LN011000
-D	♪ SM930000	↔ SM780000	- SP100000	= SA040000	M LM020000	] SM080000	m LM010000	} SM140000	Þ LT640000	Ø LO620000	¡ SP030000	╤ SF270000	═ SF430000	▌ SF580000	φ GF010001	² ND021000
-E	♫ SM910000	▲ SM600000	. SP110000	> SA050000	N LN020000	^ SD150000	n LN010000	~ SD190000	Ä LA180000	Pts SC060000	« SP170000	╘ SF280000	╬ SF440000	▐ SF590000	ε GE010000	■ SM470000
-F	☼ SM690000	▼ SV040000	/ SP120000	? SP150000	O LO020000	_ SP090000	o LO010000	⌂ SM790000	Å LA280000	ƒ SC070000	» SP180000	┐ SF030000	═ SF450000	▀ SF600000	∩ SA380000	(RSP) SP300000

Greek



HEX DIGITS 1ST → 2ND ↓	0-	1-	2-	3-	4-	5-	6-	7-	8-	9-	A-	B-	C-	D-	E-	F-
-0		► SM590000	(SP) SP010000	0 ND100000	@ SM050000	P LP020000	` SD130000	p LP010000	A GA020000	P GR020000	ι GI010000	▒ SF140000	┴ SF020000	╨ SF460000	ω GO310000	Ω GO720000
-1	☺ SS000000	◄ SM630000	! SP020000	1 ND010000	A LA020000	Q LQ020000	a LA010000	q LQ010000	B GB020000	Σ GS020000	κ GK010000	▓ SF150000	┼ SF070000	╬ SF470000	ά GA110000	± SA020000
-2	☻ SS010000	↕ SM760000	" SP040000	2 ND020000	B LB020000	R LR020000	b LB010000	r LR010000	Γ GG020000	T GT020000	λ GL010000	■ SF160000	├ SF060000	╠ SF480000	έ GE110000	≥ SA530000
-3	♥ SS020000	!! SP330000	# SM010000	3 ND030000	C LC020000	S LS020000	c LC010000	s LS010000	Δ GD020000	Y GU020000	μ GM010000	│ SF110000	┤ SF080000	╙ SF490000	ή GE710000	≤ SA520000
-4	♦ SS030000	¶ SM250000	$ SC030000	4 ND040000	D LD020000	T LT020000	d LD010000	t LT010000	E GE020000	Φ GF020000	ν GN010000	┬ SF090000	┬ SF100000	╘ SF500000	ϊ GI170000	Ϊ GI180000
-5	♣ SS040000	§ SM240000	% SM020000	5 ND050000	E LE020000	U LU020000	e LE010000	u LU010000	Z GZ020000	X GH020000	ξ GX010000	╡ SF190000	┼ SF050000	╒ SF510000	ί GI110000	Ϋ GU180000
-6	♠ SS050000	▬ SM700000	& SM030000	6 ND060000	F LF020000	V LV020000	f LF010000	v LV010000	H GE320000	Ψ GP620000	ο GO010000	╢ SF200000	╞ SF360000	╓ SF520000	ό GO110000	÷ SA060000
-7	• SM570000	↨ SM770000	' SP050000	7 ND070000	G LG020000	W LW020000	g LG010000	w LW010000	Θ GT620000	Ω GO320000	π GP010000	╖ SF210000	╟ SF370000	╫ SF530000	ύ GU110000	≈ SA700000
-8	◘ SM570001	↑ SM320000	(SP060000	8 ND080000	H LH020000	X LX020000	h LH010000	x LX010000	I GI020000	α GA010000	ρ GR010000	╕ SF220000	╚ SF380000	╪ SF540000	ϋ GU170000	° SM190000
-9	○ SM750000	↓ SM330000	) SP070000	9 ND090000	I LI020000	Y LY020000	i LI010000	y LY010000	K GK020000	β GB010000	σ GS010000	╣ SF230000	╔ SF390000	┘ SF040000	ώ GO710000	• SA790000
-A	◙ SM750002	→ SM310000	* SM040000	: SP130000	J LJ020000	Z LZ020000	j LJ010000	z LZ010000	Λ GL020000	γ GG010000	ς GS610000	╨ SF240000	╩ SF400000	┌ SF010000	Ά GA120000	· SD630000
-B	♂ SM280000	← SM300000	+ SA010000	; SP140000	K LK020000	[SM060000	k LK010000	{ SM110000	M GM020000	δ GD010000	τ GT010000	╖ SF250000	╦ SF410000	■ SF610000	Έ GE120000	√ SA800000
-C	♀ SM290000	∟ SA420000	, SP080000	< SA030000	L LL020000	\ SM070000	l LL010000	| SM130000	N GN020000	ε GE010000	υ GU010000	╙ SF260000	╠ SF420000	▬ SF570000	Ή GE720000	n LN011000
-D	♪ SM930000	↔ SM780000	- SP100000	= SA040000	M LM020000	] SM080000	m LM010000	} SM140000	Ξ GX020000	ζ GZ010000	φ GF010000	╤ SF270000	═ SF430000	▌ SF580000	Ί GI120000	² ND021000
-E	♫ SM910000	▲ SM600000	. SP110000	> SA050000	N LN020000	^ SD150000	n LN010000	~ SD190000	O GO020000	η GE310000	χ GH010000	╘ SF280000	╬ SF440000	▐ SF590000	Ό GO120000	■ SM470000
-F	☼ SM690000	▼ SV040000	/ SP120000	? SP150000	O LO020000	_ SP090000	o LO010000	⌂ SM790000	Π GP020000	θ GT610000	ψ GP610000	┐ SF030000	═ SF450000	▀ SF600000	Ύ GU120000	(RSP) SP300000

Look.

Problems with code page scheme

1. Advent of the web and e-mail: browsers and mail programs would need to store all code books in order to be able to decode messages from different countries correctly!
   - Back in the day, when memory was expensive, this required a lot of space.
   - E.g Jason wants to use his Mac, manufactured in China, according to American specifications, to e-mail his mother, who is currently in China, using a laptop purchased in Greece. Oh, and he types the e-mail in Greek!
2. For some languages, e.g Mandarin, 8 bits are simply not enough!

The Unicode standard

- Defined and maintained by the *Unicode Consortium*.
- An effort to develop a "mega-code page" to satisfy all users in the world.
- Every character gets mapped to a single *"code point"* which is just a number in Hex followed by the string "U+" (stands for Unicode).
  - *E.g* U+0388= *λ (Greek lowercase lambda),* U+*0153*=œ (latin 'oe')
  - "**Hello**" = U+0048 **U+0065 U+006C U+006C U+006F**.

The Unicode standard

- Defined and maintained by the *Unicode Consortium*.
- An effort to develop a "mega-code page" to satisfy all users in the world.
- Every character gets mapped to a single *"code point"* which is just a number in Hex followed by the string "U+" (stands for Unicode).
  - *E.g* U+0388= *λ (Greek lowercase lambda),* U+*0153*=œ (latin 'oe')
  - "**Hello**" = U+0048 **U+0065 U+006C U+006C U+006F**.

The Unicode standard

- "**Hello**" = U+0048 **U+0065 U+006C U+006C U+006F**.
- **This hex number does not necessarily represent the number of bits required to store the character in memory!**
  - E.g U+10FFFF is the largest Unicode codepoint and clearly it can not fit into either one or two bytes…
- How the code point itself is mapped to memory (the code point's *encoding*) is left to the destination computer ☺

Implementation of encodings

- **STEP 0: EVERYBODY AGREES ON USING UNICODE. EVERYONE.**

Implementation of encodings

- **STEP 0: EVERYBODY AGREES ON USING UNICODE. EVERYONE.**
- Step 1: Select an encoding.

Implementation of encodings

- **STEP 0: EVERYBODY AGREES ON USING UNICODE. EVERYONE.**
- Step 1: Select an encoding.
- Many different encodings of Unicode exist, and are pre-installed in all major OSs.
  - UCS2 and its successor UTF-16 stores the codepoint U+xxxx using exactly xxxx, so in 16 bits.
  - UTF8 stores every codepoint in MULTIPLES OF 1 BYTE. *(variable-length encoding)*
    - Codepoints from U+0x0000 to U+0x7E ($127_{(10)}$) are stored **within an entire byte of their own (so the MSB is effectively *ignored*!).**
    - Codepoints above $127_{(10)}$ use as many bytes as they might need!

Implementation of encodings

- **STEP 0: EVERYBODY AGREES ON USING UNICODE. EVERYONE.**
- Step 1: Select an encoding.
- Many different encodings of Unicode exist, and are pre-installed in all major OSs.
  - UCS2 and its successor UTF-16 stores the codepoint U+xxxx using exactly xxxx, so in 16 bits.
  - UTF8 stores every codepoint in MULTIPLES OF 1 BYTE. *(variable-length encoding)*
    - Codepoints from U+0x0000 to U+0x7E ($127_{(10)}$) are stored **within an entire byte of their own (so the MSB is effectively *ignored*!).**
    - Codepoints above $127_{(10)}$ use as many bytes as they might need!
- Step 2: The sender needs to tell the recipient what kind of Unicode encoding it used to build the text
  - E-mail headers are a good example
  - In HTML, we can use <meta> tags

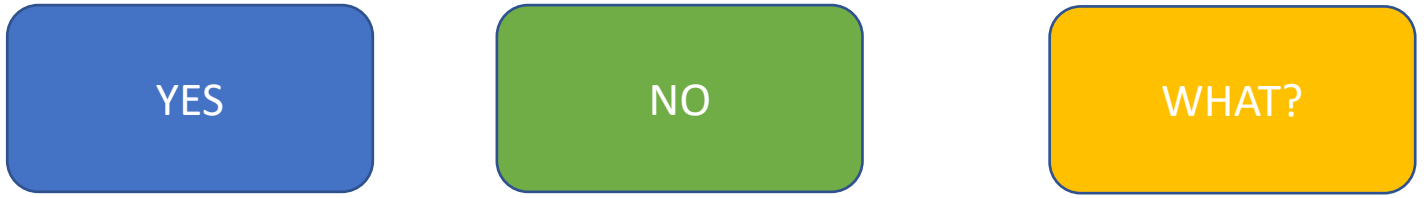
Quiz

Quiz

- UTF-8 is racist

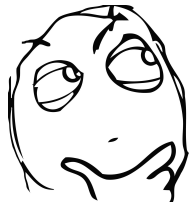
Quiz

- UTF-8 is racist

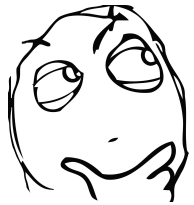


UTF-8 is kind of bigoted ☹

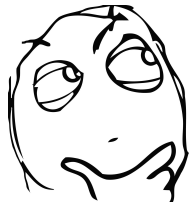
UTF-8 is kind of bigoted ☹

- Or is it? 

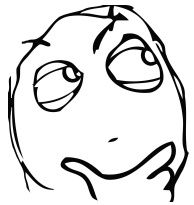
UTF-8 is kind of bigoted ☹

- Or is it? 
- Vast majority of pairs of receivers – senders in the world will send and read characters in English (so, from the set of the first 128 characters that UTF-8 represents in a single byte)
  - This is exactly ASCII! ☺

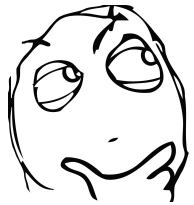
UTF-8 is kind of bigoted ☹

- Or is it? 
- Vast majority of pairs of receivers – senders in the world will send and read characters in English (so, from the set of the first 128 characters that UTF-8 represents in a single byte)
  - This is exactly ASCII! ☺
- As a result, the vast majority of messages can be encoded in one byte per character.
  - We can't hope to do any better when encoding a single character! ☺

UTF-8 is kind of bigoted ☹

- Or is it? 
- Vast majority of pairs of receivers – senders in the world will send and read characters in English (so, from the set of the first 128 characters that UTF-8 represents in a single byte)
  - This is exactly ASCII! ☺
- As a result, the vast majority of messages can be encoded in one byte per character.
  - We can't hope to do any better when encoding a single character! ☺
- Non-English speakers or people who want to use rare characters are disadvantaged: additional storage cost has to be paid!

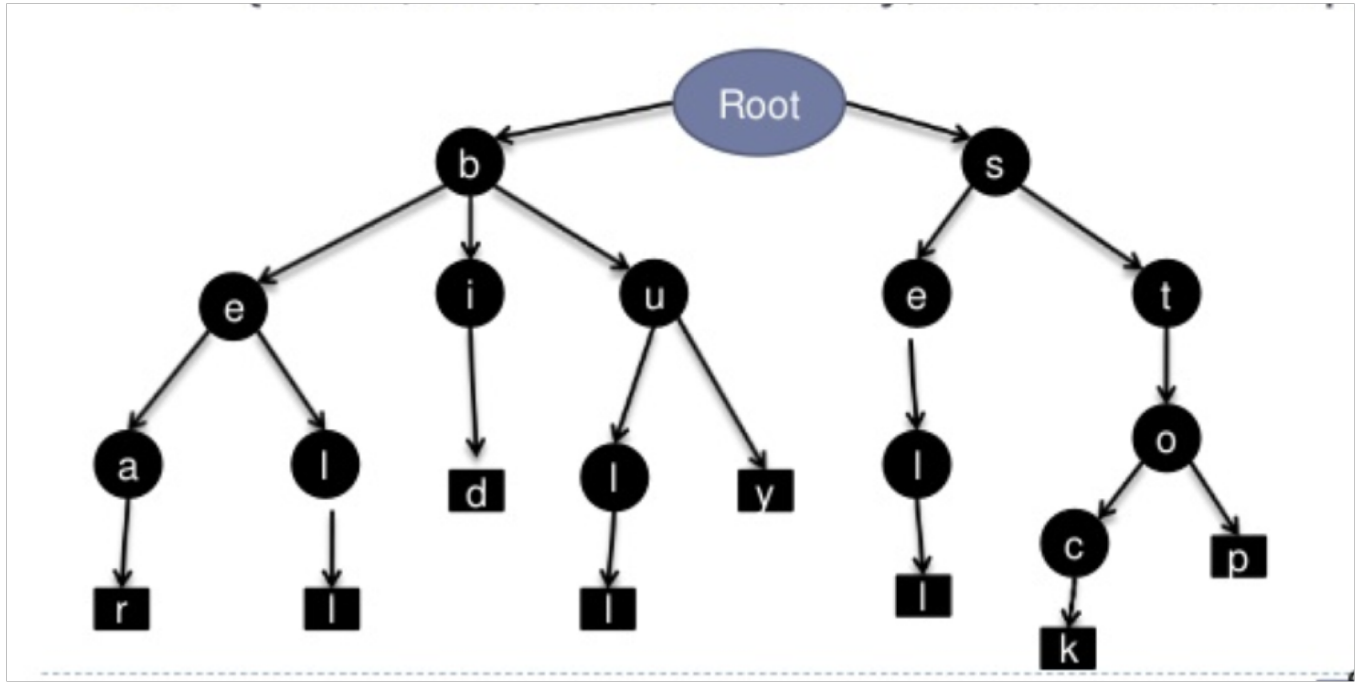
UTF-8 is kind of bigoted ☹

- Or is it? 
- Vast majority of pairs of receivers – senders in the world will send and read characters in English (so, from the set of the first 128 characters that UTF-8 represents in a single byte)
  - This is exactly ASCII! ☺
- As a result, the vast majority of messages can be encoded in one byte per character.
  - We can't hope to do any better when encoding a single character! ☺
- Non-English speakers or people who want to use rare characters are disadvantaged: additional storage cost has to be paid!
  - **Experimentation**: try to load a webpage in Mandarin, making sure that the HTML source reads `<meta http-equiv="utf8"` right after the <html> tag… It should take longer than loading the Washington Post, for instance.

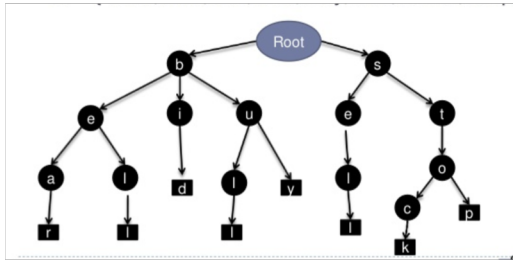
ASCII is good enough for us!

- For our study of strings, **128 characters will be enough**.
- So we limit ourselves to ASCII (technically, 7 of the 8 bits contained in the byte reserved via our assumed UTF-8 encoding).
- ‘A’ in ASCII: $0x41 = 65_{(10)}$
- ‘a’ in ASCII: $0x61 = 97_{(10)}$
- Since $65 + 24 = 89 < 97$, some characters in between lowercase and uppercase alphabet…
- Subtracting 65 from all of our code points allows us to treat ‘A’ at 0…

Tries

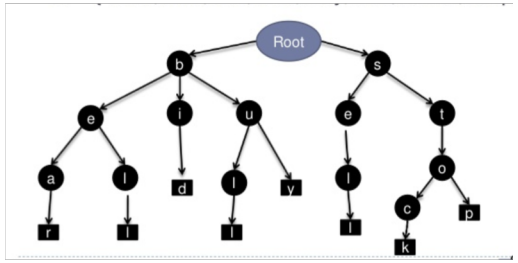


Tries



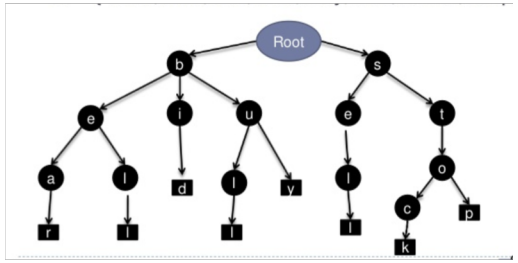
- Pronounced exactly like the verb!

Tries



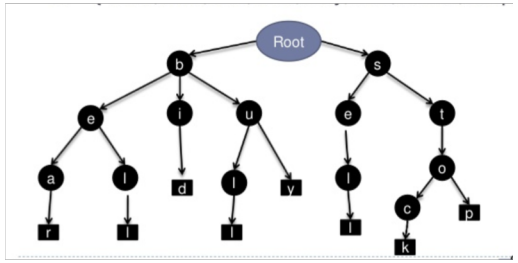
- Pronounced exactly like the verb!
  - E.g “José **tries** to catch the ball”
  - This can be sometimes perplexing, since they are essentially *trees*… don’t let this confuse you.

Tries



- Pronounced exactly like the verb!
  - E.g “José **tries** to catch the ball”
  - This can be sometimes perplexing, since they are essentially *trees*… don’t let this confuse you.
- A trie is a tree-like data structure where paths store strings, and branches are taken based on the relevant **character** of the string that is searched for (or inserted / deleted).
  - An additional bit field tells us if we have reached an actual string that was stored in the trie.

Tries



- Pronounced exactly like the verb!
  - E.g “José **tries** to catch the ball”
  - This can be sometimes perplexing, since they are essentially *trees*… don’t let this confuse you.
- A trie is a tree-like data structure where paths store strings, and branches are taken based on the relevant **character** of the string that is searched for (or inserted / deleted).
  - An additional bit field tells us if we have reached an actual string that was stored in the trie.
- Invented in late 1950s by René de la Briandais, esteemed French person.

Structure of a trie node

- We must always specify the alphabet Σ that we sample characters from!
  - Common choices: ASCII, UNICODE.
- In this course, we will be using ASCII or small subsets thereof.
- Every node will hold a buffer with $|\Sigma|$ −many cells! 
- Question: For UNICODE ($\approx 65K$ usable symbols), how big will those buffers be ***in bytes***?

Structure of a trie node

- We must always specify the alphabet Σ that we sample characters from!
  - Common choices: ASCII, UNICODE.

Structure of a trie node

- We must always specify the alphabet Σ that we sample characters from!
  - Common choices: ASCII, UNICODE.
- In this course, we will be using ASCII or small subsets thereof.

Structure of a trie node

- We must always specify the alphabet Σ that we sample characters from!
  - Common choices: ASCII, UNICODE.
- In this course, we will be using ASCII or small subsets thereof.
- Every node will hold a buffer with $|\Sigma|$ —many cells! 

Structure of a trie node

- We must always specify the alphabet Σ that we sample characters from!
  - Common choices: ASCII, UNICODE.
- In this course, we will be using ASCII or small subsets thereof.
- Every node will hold a buffer with $|\Sigma|$ −many cells! 
- Question: For UNICODE ($\approx 65K$ usable symbols), how big will those buffers be ***in bytes***?

128

65K

>65K

<65K

Structure of a trie node

- We must always specify the alphabet Σ that we sample characters from!
  - Common choices: ASCII, UNICODE.
- In this course, we will be using ASCII or small subsets thereof.
- Every node will hold a buffer with $|\Sigma|$ −many cells! 
- Question: For UNICODE ($\approx 65K$ usable symbols), how big will those buffers be ***in bytes***?

128

65K

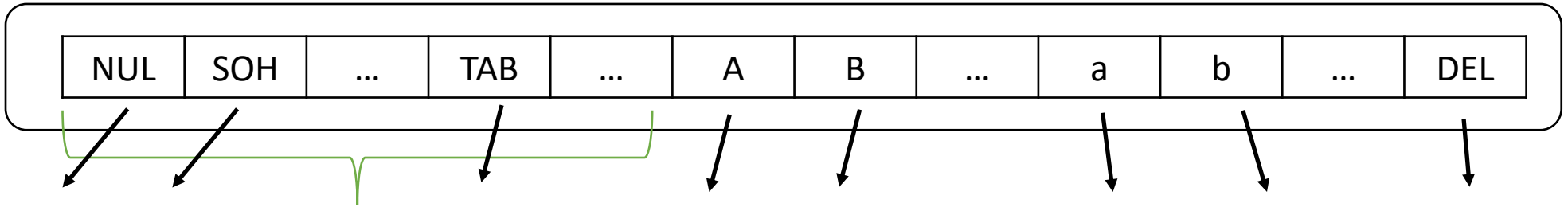
>65K

<65K

Only the first 128 characters are stored in a single byte each!

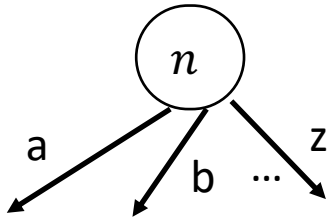
Structure of a trie node

- This is an example of what a trie node would look like if alphabet is ASCII:



Non-printable characters, like **system bell, paper feed, carriage return….**

- To make the art (?) tractable, we will be shortening the nodes to only have outgoing links that correspond to actual characters that are **dynamically used** in the trie



Structure of a trie node

- It turns out that in a trie node, we also need some information that tells us

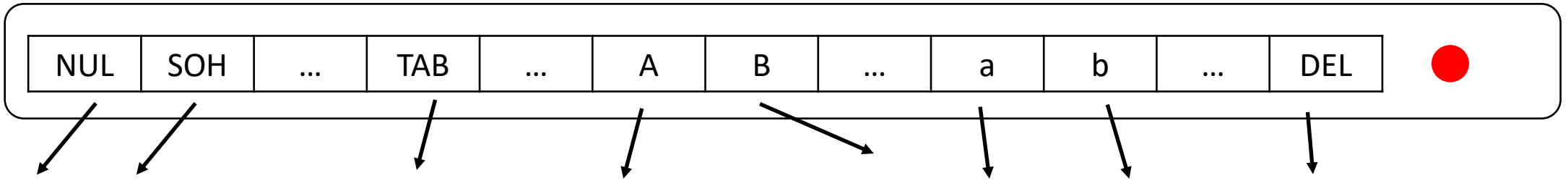
Does the path from the root of the trie correspond to an actual string stored in it?

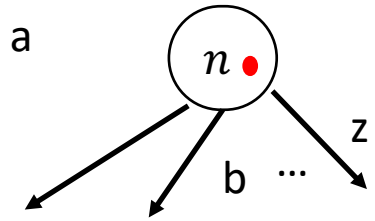
Structure of a trie node

- It turns out that in a trie node, we also need some information that tells us

Does the path from the root of the trie correspond to an actual string stored in it?

- This corresponds to a single bit! “Art” convention: Red for set, blue for unset



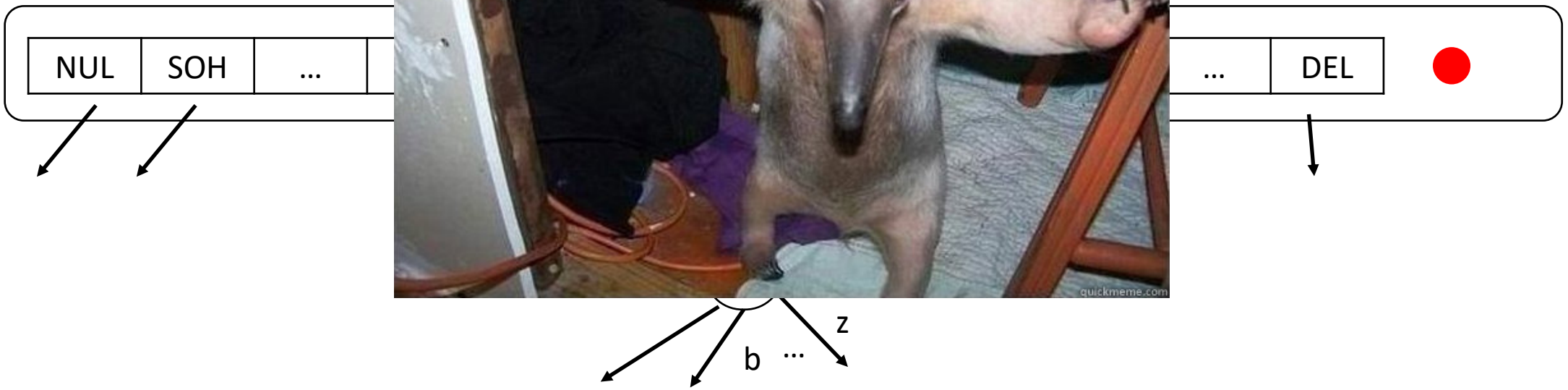


Structure of a trie node

- It turns out that in a trie node, we also need some information that tells us

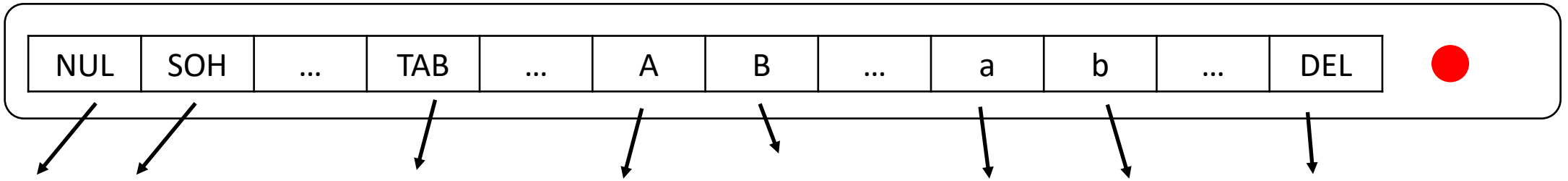
Does the path from t *string stored in it?*

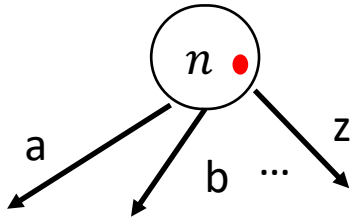
- This corresponds to a blue for unset



Not so fast!

- Don’t I need 2 buffers, to store both characters **and** pointers?



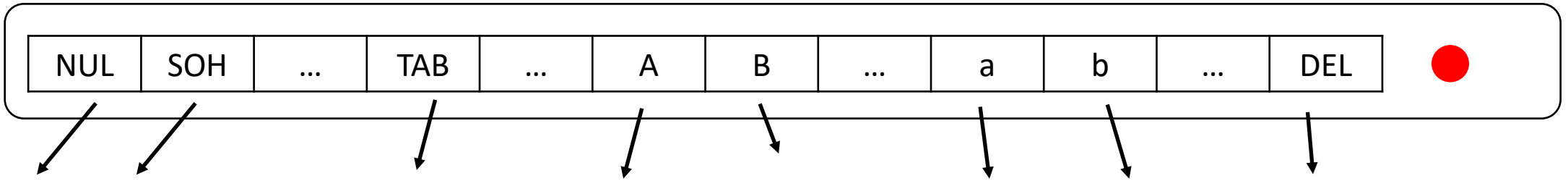


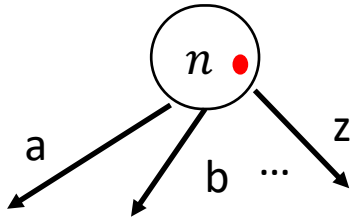
Not so fast!

- Don’t I need 2 buffers, to store both characters **and** pointers?

Yes

No





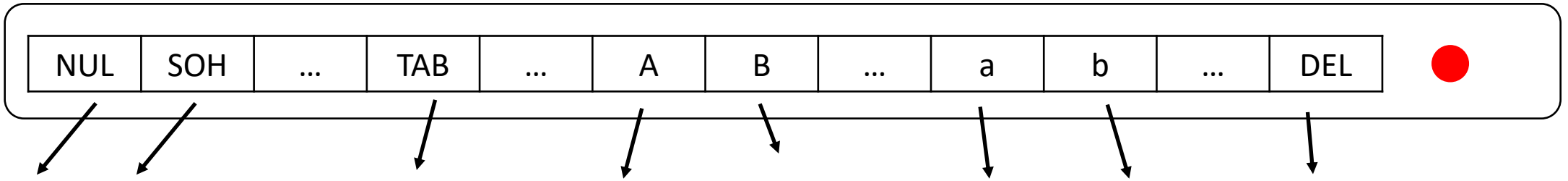
Not so fast!

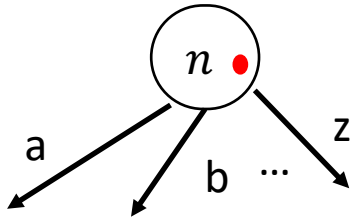
- Don’t I need 2 buffers, to store both characters **and** pointers?

Yes

No

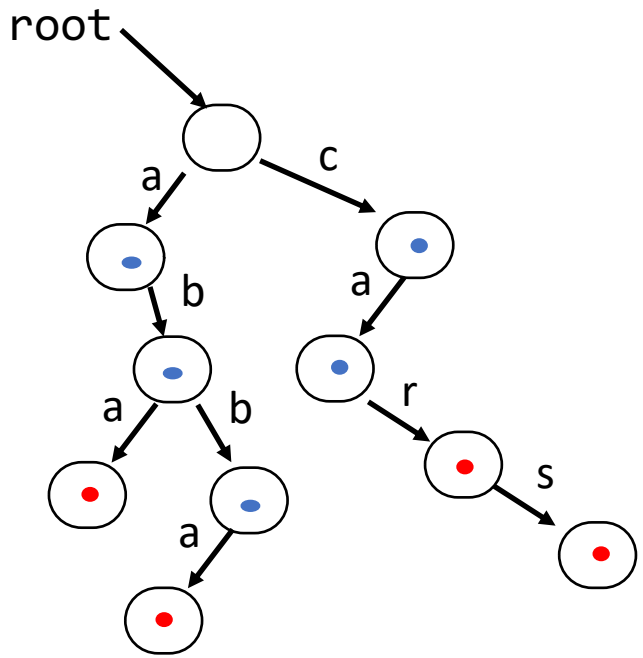
- Every character gets mapped to a number through ASCII! (e.g `(int)`A’ = 65`)
- Can absolutely do something like $buf['b']$! ☺





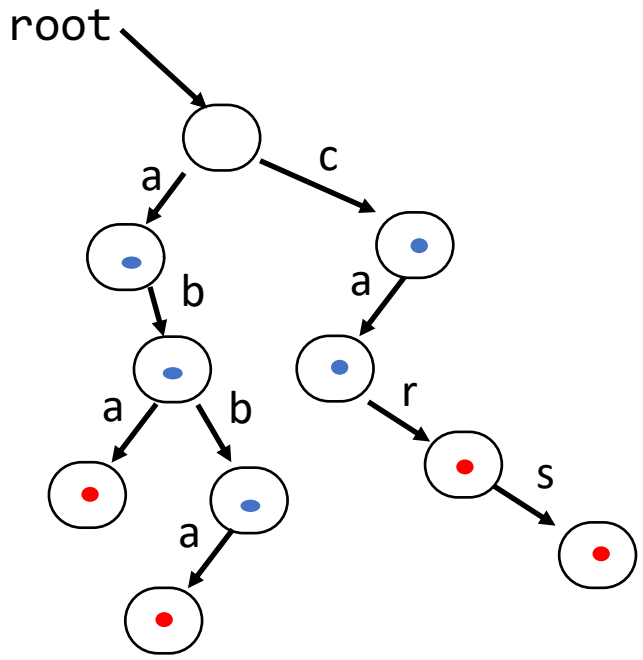
Example

- Assume alphabet to be ASCII: 128 characters.
- The following trie stores the strings “abba”, “aba”, “car”, “cars”:



Example

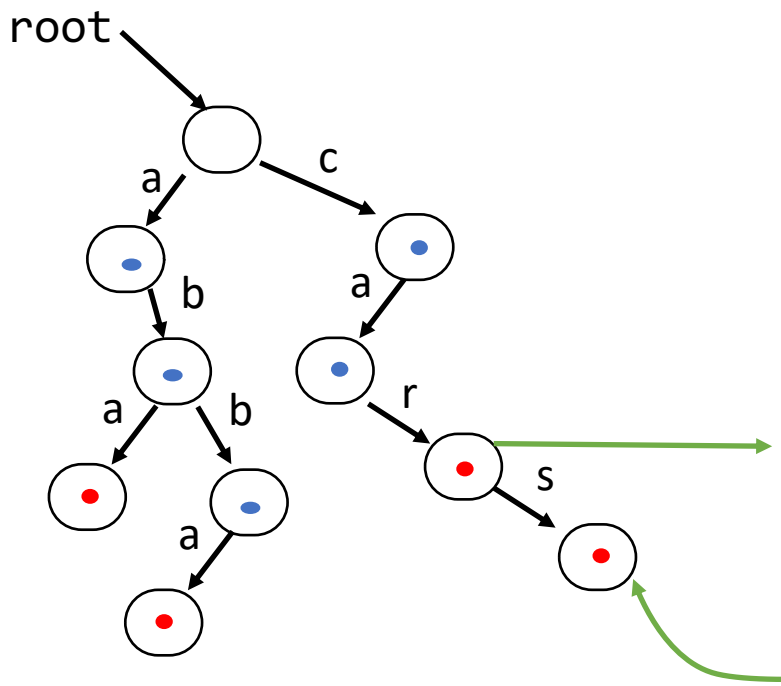
- Assume alphabet to be ASCII: 128 characters.
- The following trie stores the strings “abba”, “aba”, “car”, “cars”:



Do we perhaps see the utility of the “end-of-stored string” bit now? ☺

Example

- Assume alphabet to be ASCII: 128 characters.
- The following trie stores the strings “abba”, “aba”, “car”, “cars”:



Do we perhaps see the utility of the “end-of-stored string” bit now? ☺

Note that encountering a set bit while traversing the tree is **not a sufficient reason** to stop your search for more keys along that path!

Inserting a key

- Quiz: Inserting a new key into a trie always involves the allocation of new nodes.

Yes
(why?)

No
(give me an example)

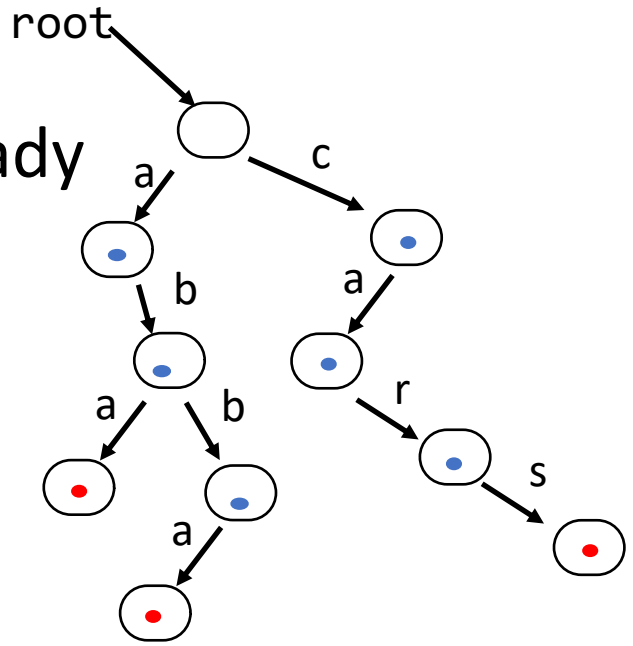
Inserting a key

- Quiz: Inserting a new key into a trie always involves the allocation of new nodes.

Yes
(why?)

No
(give me an
example)

- In our familiar trie, suppose that “car” was not already stored.



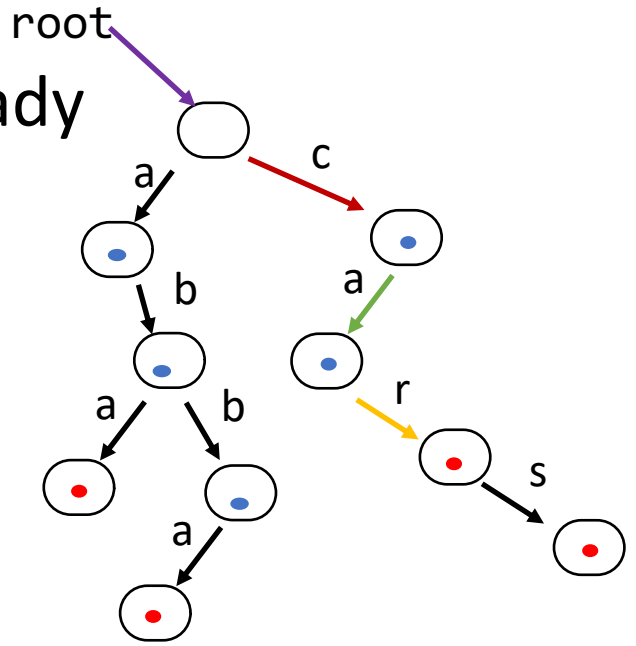
Inserting a key

- Quiz: Inserting a new key into a trie always involves the allocation of new nodes.

Yes
(why?)

No
(give me an example)

- In our familiar trie, suppose that “car” was not already stored.
- Then, we traverse the trie following the links for ‘c’, ‘a’ and ‘r’ and we set the relevant bit!



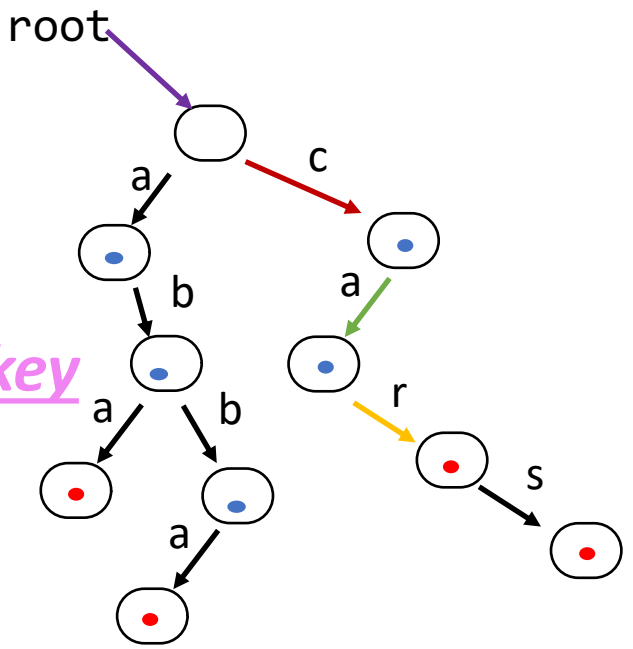
Inserting a key

- Quiz: Inserting a new key into a trie always involves the allocation of new nodes.

Yes
(why?)

No
(give me an example)

- In our familiar trie, suppose that "car" was not already stored.
- Then, we traverse the trie following the links for 'c', 'a' and 'r' and we set the relevant bit!
- Generally: inserting a key that is a ***prefix of an existing key*** never leads to a heap visit!



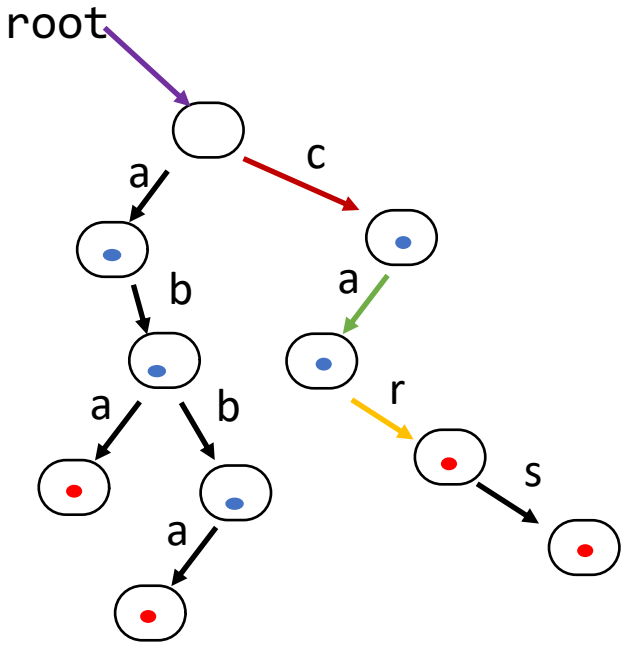
Inserting a key

- Quiz: Inserting a new key into a trie always involves the allocation of new nodes.

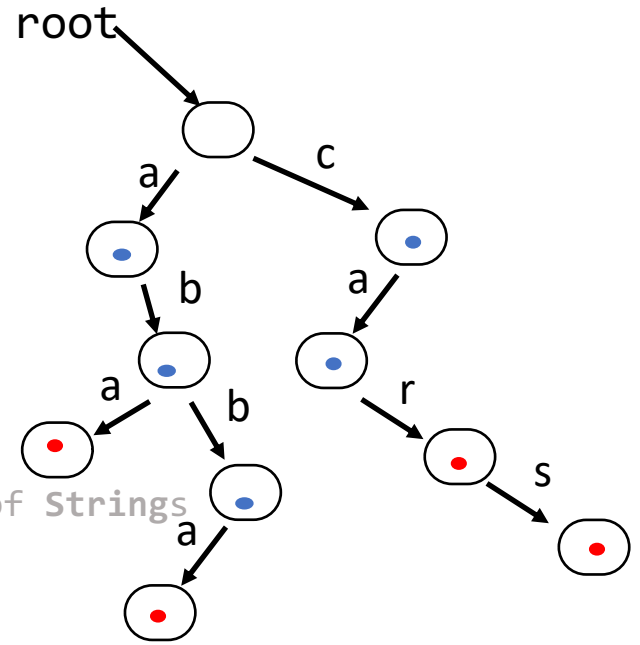
Yes
(why?)

No
(give me an example)

- In our familiar trie, suppose that “car” was not already stored.
- Then, we traverse the trie following the links for ‘c’, ‘a’ and ‘r’ and we set the relevant bit!
- Generally: inserting a key that is a ***prefix of an existing key*** never leads to a heap visit!
    - With ASCII and Unicode-sized buffers, that’s not too bad of an idea….

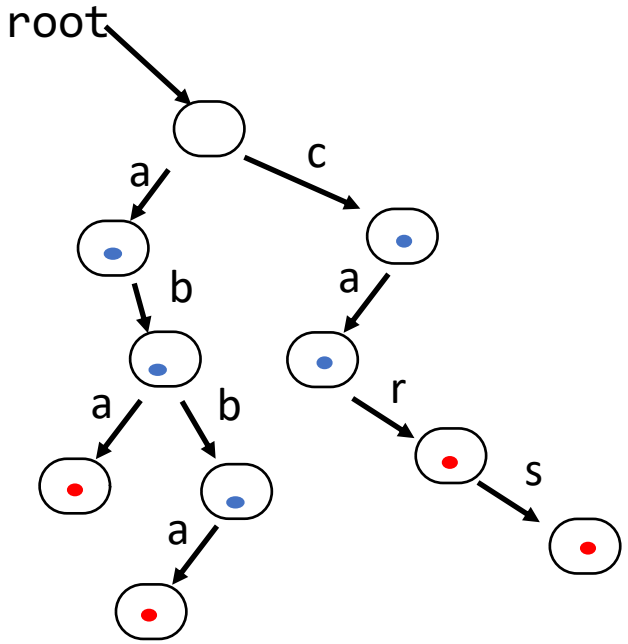


Trie insertion: exercise




```
public class AsciiTrie { // No need to make it a generic ; tries store strings!
        private class Node {
                private Node[] next = new Node[128];
                byte eosFlag; // Wish I could manipulate a single bit…
        }
        private Node root;
        public void insert(String key){ // More general solution: CharSequences instead of Strings
                root = insert(root, key, 0); // Could also make it a method of Node
        }
        private Node insert(Node root, String key, int currStringIndex){
                /* YOU fill this in NOW!
                 * Hints:
                 *
                   * (1) Strings have a method with signature charAt(int index) that returns the char at position
index of the string                                *     (index is 0-based).
                 * (2) Think: When do I complete my insertion?
                 * (3) What if I need to allocate a new node?
                 * (4) What if I don't?
                 */
        }
}
```

Trie insertion: exercise

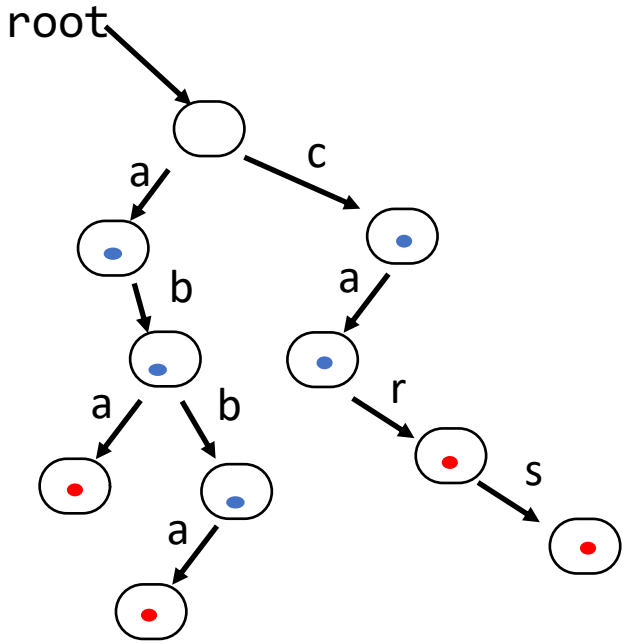



```
public class AsciiTrie {// No need to make it a generic ; tries store strings!
        private class Node {
                private Node[] next = new Node[128];
                byte eosFlag;
        }
        private Node root;
        public void insert(String key){ // More general solution: CharSequences instead of Strings
                root = insert(root, key, 0); // Could also make it a method of Node
        }
        private Node insert(Node node, String key, int currStringIndex){
                assert key != null && key.size() > 0, "Key can't be null or empty.";
                if(node == null) // key is not a prefix of any already stored key…
                        node = new Node(); // Node class defn allocates memory immediately, no need for explicit constructor defn.
                if(currStringIndex == key.size())
                        node.eosFlag = (byte)1;
                else
                        node.next[key.charAt(currStringIndex)] = insert(node.next[key.charAt(currStringIndex)], key, currStringIndex + 1);

                return node;
        }
}
```

This could work! ☺

Trie insertion: exercise

```
public class AsciiTrie {// No need to make it a generic ; tries store strings!
        private class Node {
                private Node[] next = new Node[128];
                byte eosFlag;
        }
        private Node root;
        public void insert(String key){ // More general solution: CharSequences instead of Strings
                root = insert(root, key, 0); // Could also make it a method of Node
        }
        private Node insert(Node node, String key, int currStringIndex){
                assert key != null && key.size() > 0, “Key can’t be null or empty.”;
                if(node == null)
                        node = new Node();
                if(currStringIndex == key.size())
                        node.eosFlag = (byte)1;
                else
                        node.next[key.charAt(currStringIndex)] = insert(node.next[key.charAt(currStringIndex)], key,
                                                                currStringIndex + 1);

                return node;
        }
}
```



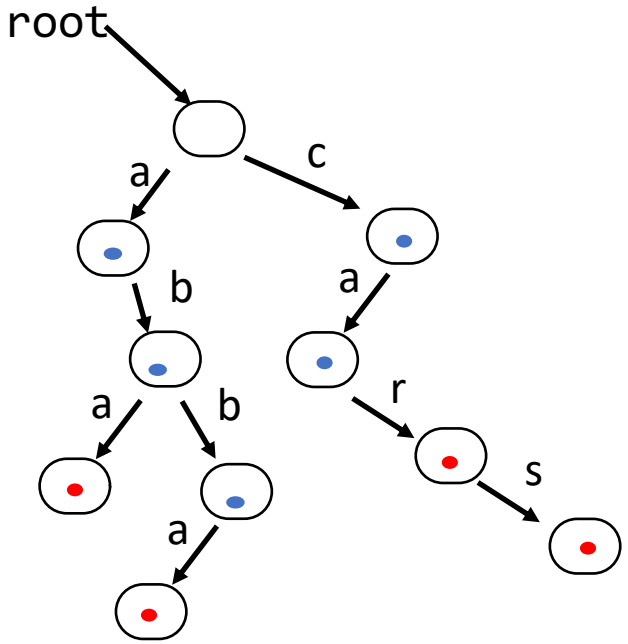
- **key** is not really stored anywhere…
- Can we perhaps get rid of it?

Yes (how?)

No (why?)

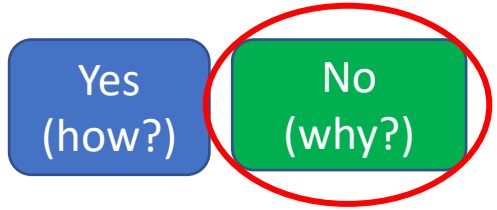
Trie insertion: exercise

```
public class AsciiTrie {// No need to make it a generic ; tries store strings!
        private class Node {
                private Node[] next = new Node[128];
                byte eosFlag;
        }
        private Node root;
        public void insert(String key){ // More general solution: CharSequences instead of Strings
                root = insert(root, key, 0); // Could also make it a method of Node
        }
        private Node insert(Node node, String key, int currStringIndex){
                assert key != null && key.size() > 0, "Key can't be null or empty.";
                if(node == null){ // key is not a prefix of any already stored key…
                        node = new Node();
                if(currStringIndex == key.size())
                        node.eosFlag = (byte)1;
                else
                        node.next[key.charAt(currStringIndex)] = insert(node.next[key.charAt(currStringIndex)], key, currStringIndex
                return node;
        }
}
```



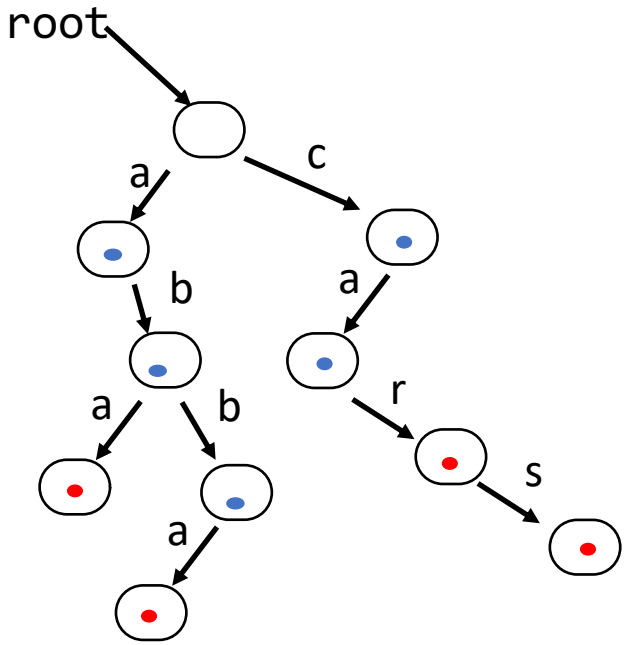
Just because **the reference** isn't stored doesn't mean that:

1. It isn't useful for the insertion algorithm
2. There is no **implicit** storage for the string in the trie (links and node bits!)

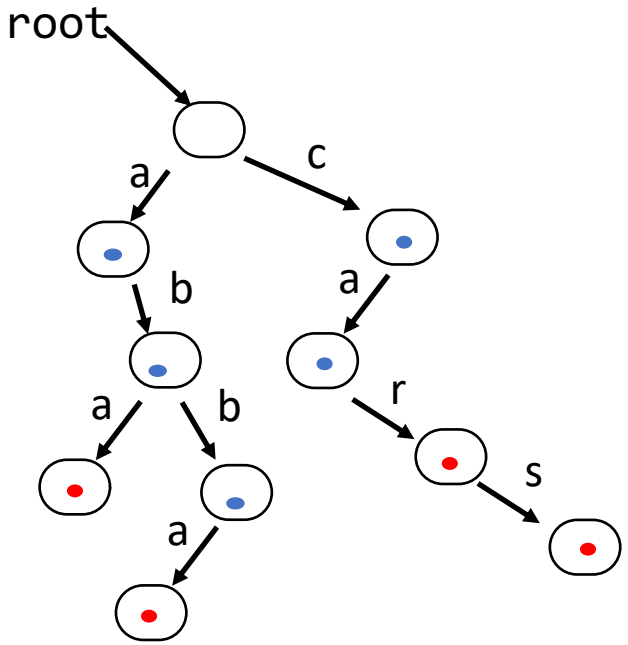


Iterative insertion!

```
public class AsciiTrie {// No need to make it a generic ; tries store strings!
        private class Node {
                private Node[] next = new Node[128];
                byte eosFlag;
        }
        private Node root;
        public void insert(String key){
                /* Fill this in with an iterative implementation
                 * of trie insertion!
                 */
        }
}
```



Iterative insertion!




```
public class AsciiTrie {// No need to make it a generic ; tries store strings!
        private class Node {
                private Node[] next = new Node[128];
                byte eosFlag;
        }
        private Node root;
        public void insert(String key){
                assert key != null && key.size() > 0, “Key can’t be empty.”;
                int currIndex = 0;
                Node curr = root, prev = null;
                while(currIndex < key.size()){
                        char charToFollow = key.charAt(currIndex);
                        if(curr == null)
                                curr = new Node();
                        if(prev != null)
                                prev.next[charToFollow] = curr;
                        currIndex++;
                        prev = curr;
                        curr = curr.next[charToFollow];
                }
                curr.eosFlag = (byte)1;
        }
}
```

Deleting a key

- Quite simple: traverse the trie and, if you find the key, set the bit to blue.
- Nothing to do if the bit is already blue (key not actually in DB)

Deleting a key

- Quite simple: traverse the trie and, if you find the key, set the bit to blue.
- Nothing to do if the bit is already blue (key not actually in DB)
- If you encounter a `null` pointer before you exhaust the key's characters, then you also know that the key is not in DB!

Deleting a key

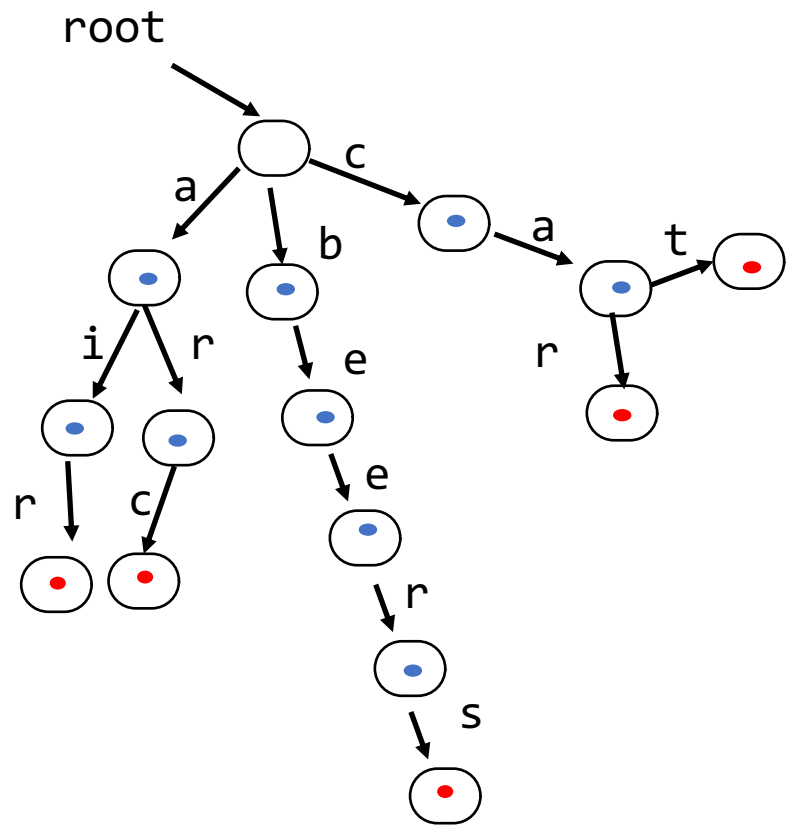
- Quite simple: traverse the trie and, if you find the key, set the bit to blue.
- Nothing to do if the bit is already blue (key not actually in DB)
- If you encounter a `null` pointer before you exhaust the key's characters, then you also know that the key is not in DB!
- In those two cases, make a choice between returning `null` or throwing an `Exception`.
  - No matter what you do, commit to it in your interface!
  - If you throw an `Exception`, it must be **checked** and documented (`@throws`)
- This is an example of "soft" deletion.

Deleting a key

- Quite simple: traverse the trie and, if you find the key, set the bit to blue.
- Nothing to do if the bit is already blue (key not actually in DB)
- If you encounter a `null` pointer before you exhaust the key's characters, then you also know that the key is not in DB!
- In those two cases, make a choice between returning `null` or throwing an `Exception`.
  - No matter what you do, commit to it in your interface!
  - If you throw an `Exception`, it must be **checked** and documented (`@throws`)
- This is an example of "soft" deletion.

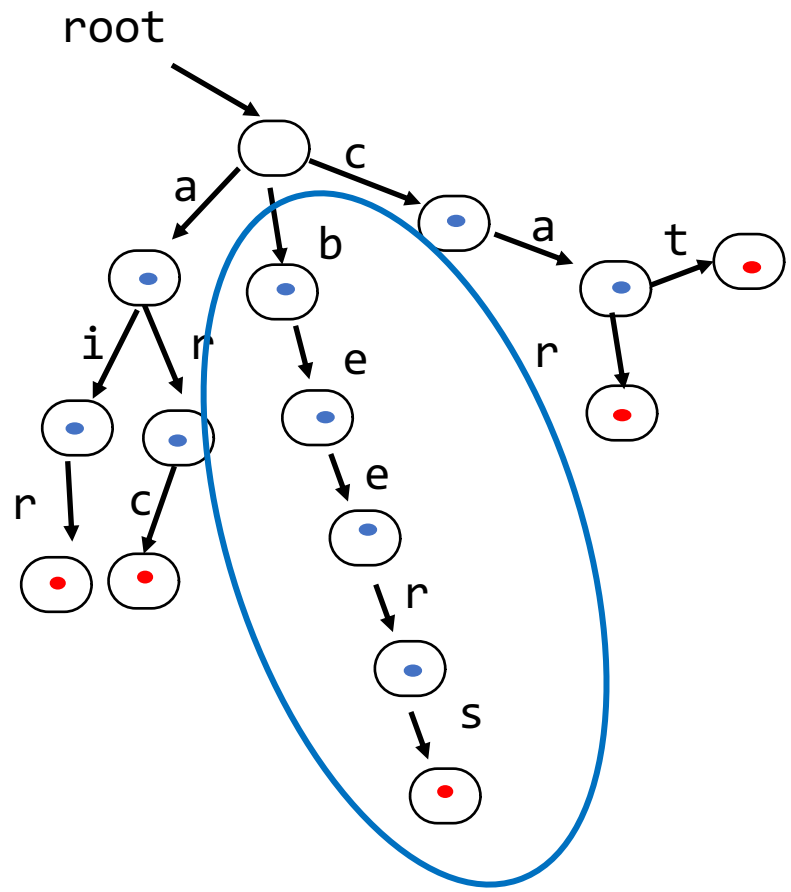
“Soft” vs “hard” deletion in a trie: what would you do?

- Suppose that you want to delete ``beers” from the following trie.



“Soft” vs “hard” deletion in a trie: what would you do?

- Suppose that you want to delete ``beers” from the following trie.



Since “beers” is the only stored string in this long path from the root (35.7% of total nodes), we could throw away this entire path, e.g by setting `root['b'] = null`

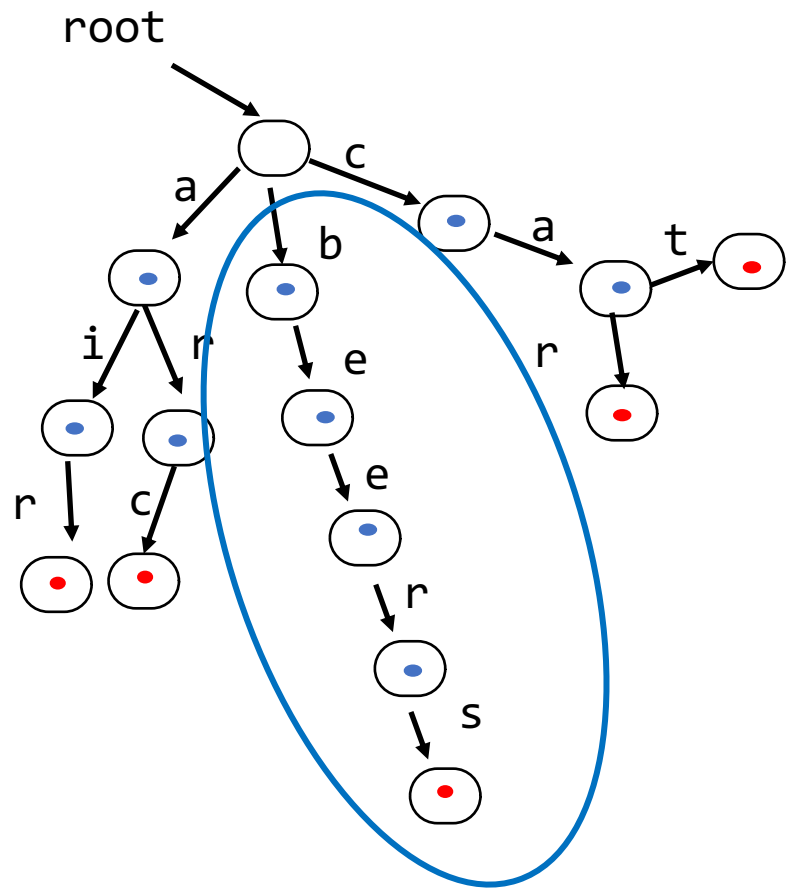
“Soft” vs “hard” deletion in a trie: what would you do?

- Suppose that you want to delete ``beers” from the following trie.

Would **you** do this? (No right or wrong answer here!)

Yes (why?)

No (why?)



Since “beers” is the only stored string in this long path from the root (35.7% of total nodes), we could throw away this entire path, e.g by setting `root['b'] = null`

“Soft” vs “hard” deletion in a trie: what would you do?

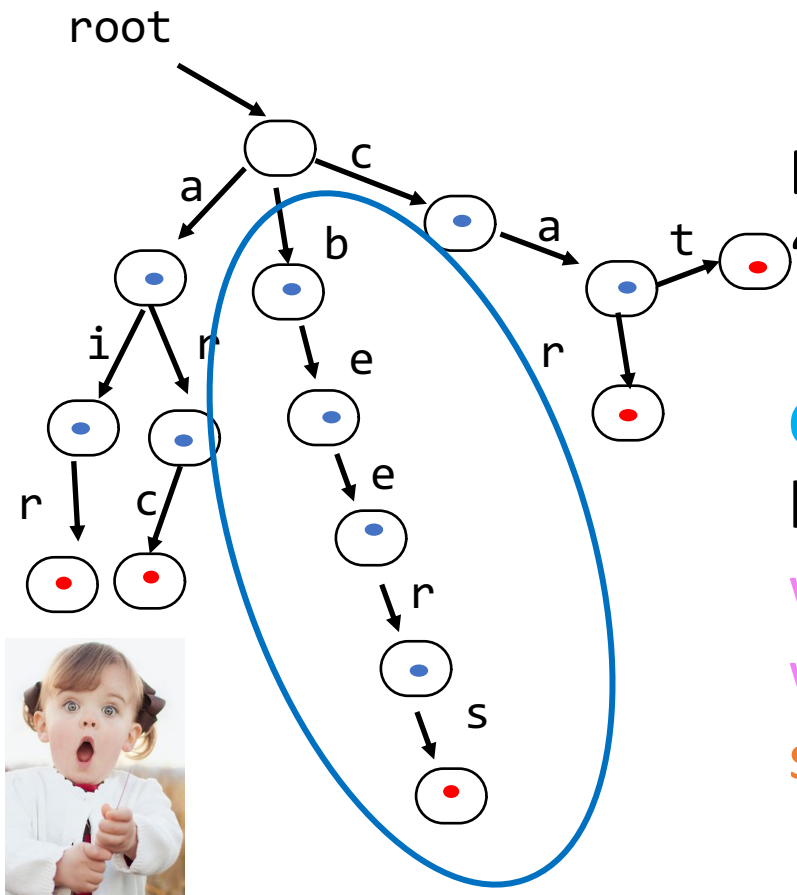
- Suppose that you want to delete ``beers” from the following trie.

Would **you** do this? (No right or wrong answer here!)

Yes (why?)

No (why?)

Jason **wouldn’t**: strings in a trie, particularly long ones, allow for many **prefixes** to be stored with **either no heap visits or few heap visits!**



E.g consider a future insertion of “beer”, “bee”, or “bear!”

Caveat: If for some reason you know that in the future you won’t insert any strings that start with a ‘b’, can delete subtree safely ☺

Complexity of searching for a key in a trie

- Given insertion and deletion, `search(String k)` should be easy for you to implement.

Complexity of searching for a key in a trie

- Given insertion and deletion, `search(String k)` should be easy for you to implement.
- **Most important question in tries**: What is the worst-case complexity of searching for a key in a trie with n keys?

$\approx log_2 n$

$\approx log_k n$, $k > 2$

$\approx \mathcal{O}(n)$

Something else (what?)

Complexity of searching for a key in a trie

- Given insertion and deletion, `search(String k)` should be easy for you to implement.
- **Most important question in tries:** What is the worst-case complexity of searching for a key in a trie with n keys?

$\approx log_2 n$

$\approx log_k n,\ k > 2$

$\approx \mathcal{O}(n)$

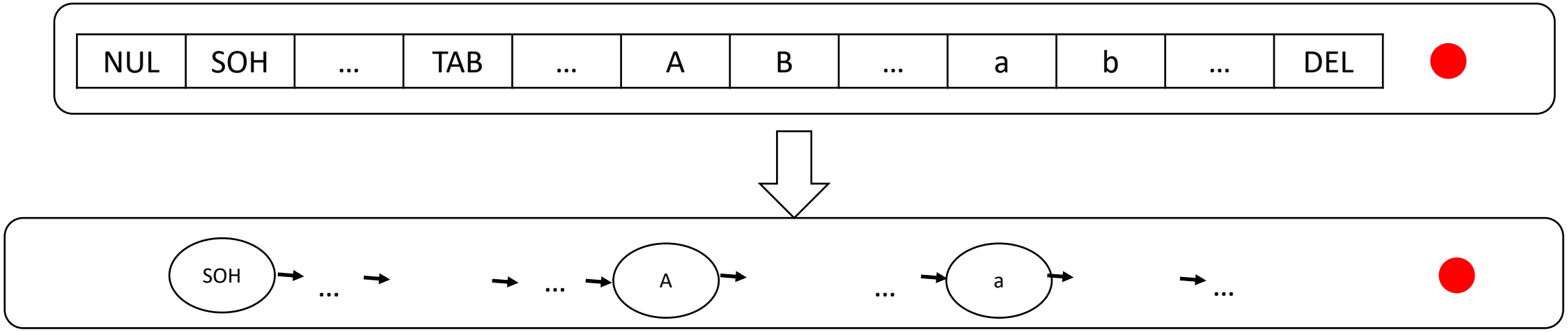
Something else (what?)

LENGTH OF THE LONGEST EXISTING KEY IN THE TRIE!

- Tries: **first data structure we see where search efficiency does not depend on n!!!**

De la Briandais Tries

- Obvious problem with tries: they waste a lot of space when characters are not used =/
- Obvious solution: Replace the inner nodes' buffer with a linked list over the characters that have split keys that go through this node!



- Caveat: we lose the $\mathcal{O}(1)$ access that `charAt()` was giving us ☹

Cool things we can do with tries

- Quiz: Can we use a trie as a dictionary, the way that we've used AVL Trees, Red-Black Trees, etc?

Cool things we can do with tries

- Quiz: Can we use a trie as a dictionary, the way that we've used AVL Trees, Red-Black Trees, etc?

Yes

No

Not sure

Cool things we can do with tries

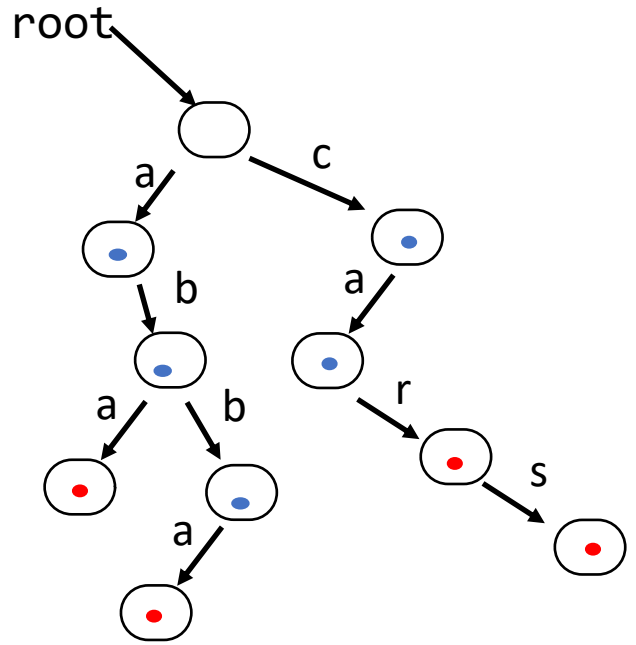
- Quiz: Can we use a trie as a dictionary, the way that we've used AVL Trees, Red-Black Trees, etc?

- Absolutely!
- They offer excellent search efficiency no matter how many strings they hold.
- Insertion on par with search
- Deletion more efficient than B-Trees (at least for small strings)
  - Plus, as in with BSTs, all dictionary operations can be implemented iteratively! ☺

Trie-specific stuff

- Much like with balanced trees, where we could ask for index-specific stuff such as inorder traversal and height, we can also use tries for non-dictionary related things! ☺
- Practice: In the next slide, fill-in the code of the instance method `longestPrefixOf(String key)`, which returns the longest prefix of key `key` in the trie.

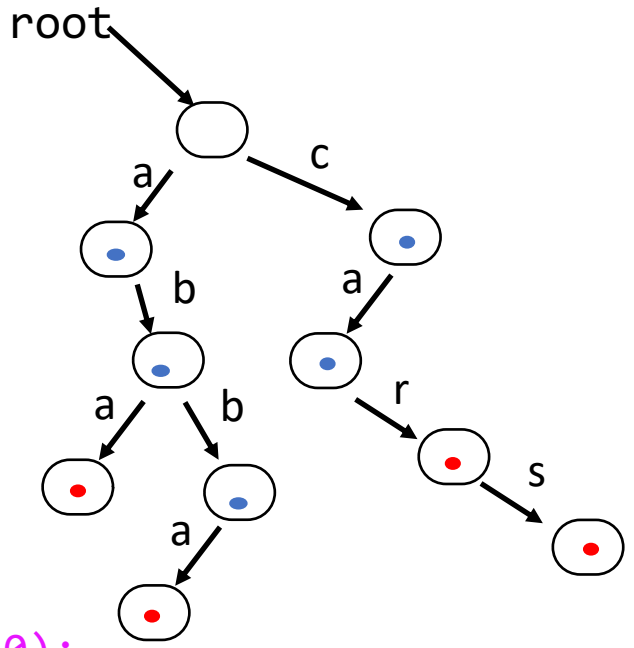
Longest Prefix!




```
public class AsciiTrie {
        private class Node {
                private Node[] next = new Node[128];
                byte eosFlag;
        }
        private Node root;
        public String longestPrefixOf(String key){
                /* Fill this in!
                 * Hints:
                 *  (1) Recall that only nodes with a red bit can be candidates for
                 *   encoding the longest prefix of key!
                 *  (2) key.substr(int startIndex, int length) will return a substring of key that
                 *   starts from startIndex and has a length of “length”. It might be useful.
                 */
        }
}
```

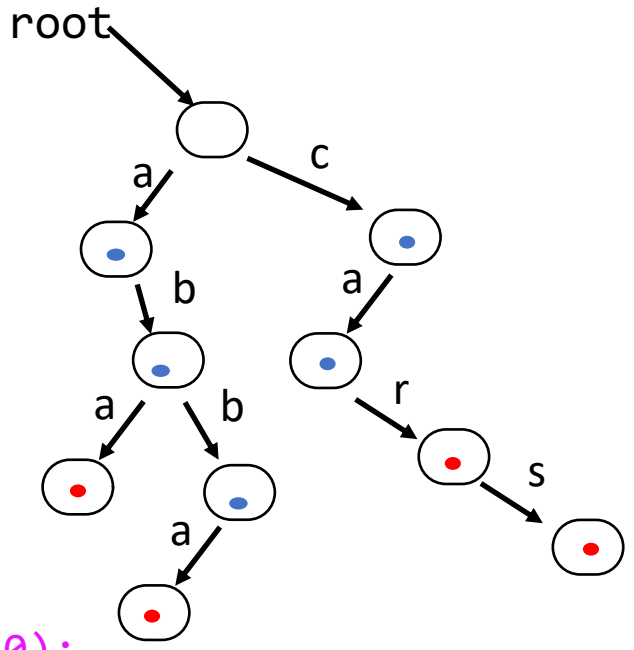
Longest Prefix!

```
public class AsciiTrie {
	private class Node {
		private Node[] next = new Node[128];
		byte eosFlag;
	}
	private Node root;
	public String longestPrefixOf(String key){
		int longestPrefixLength = getMaxPrefixLength(root, key, 0, 0);
		return key.substr(0, longestPrefixLength);
	}
	private int getMaxPrefixLength(Node curr, String key, int currInd, int maxLenRec){
		if(curr == null || currInd == key.length()) return maxLenRec;
		if(curr.eosFlag == (byte)1) maxLenRec = currInd;
		return getMaxPrefixLength(curr.next[key.charAt(currInd)], key, currInd + 1,
                                                                    maxLenRec);
	}
}
```



This works!

Longest Prefix!




```
public class AsciiTrie {
        private class Node {
                private Node[] next = new Node[128];
                byte eosFlag;
        }
        private Node root;
        public String longestPrefixOf(String key){
                int longestPrefixLength = getMaxPrefixLength(root, key, 0, 0);
                return key.substr(0, longestPrefixLength);
        }
        private int getMaxPrefixLength(Node curr, String key, int currInd, int maxLenRec){
                if(curr == null || currInd == key.length()) return maxLenRec;
                if(curr.eosFlag == (byte)1) maxLenRec = currInd;
                return getMaxPrefixLength(curr.next[key.charAt(currInd)], key, currInd + 1,
                                                                        maxLenRec);
        }
}
```

This works… and is also **tail recursive**!

Metacharacter (glob) search!

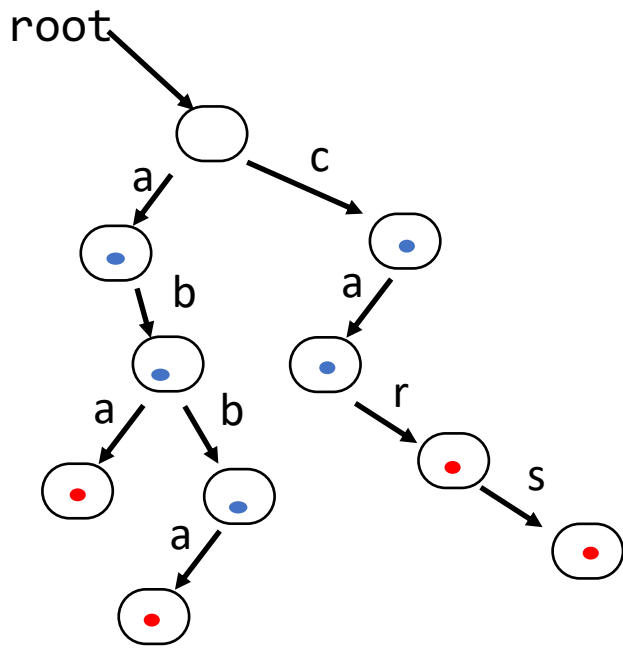
```
[jason@MacBook:~/temp ls
 fil.txt     file.txt    file0.txt   file1.txt   file2.txt   myfile.txt  myfile0.txt myfile1.txt
[jason@MacBook:~/temp ls file*
 file.txt  file0.txt file1.txt file2.txt
[jason@MacBook:~/temp ls myfile*
 myfile.txt  myfile0.txt myfile1.txt
 jason@MacBook:~/temp
```

Metacharacter (glob) search!

```
[jason@MacBook:~/temp ls
 fil.txt     file.txt    file0.txt   file1.txt   file2.txt   myfile.txt  myfile0.txt myfile1.txt
[jason@MacBook:~/temp ls file*
 file.txt  file0.txt file1.txt file2.txt
[jason@MacBook:~/temp ls myfile*
 myfile.txt  myfile0.txt myfile1.txt
 jason@MacBook:~/temp
```

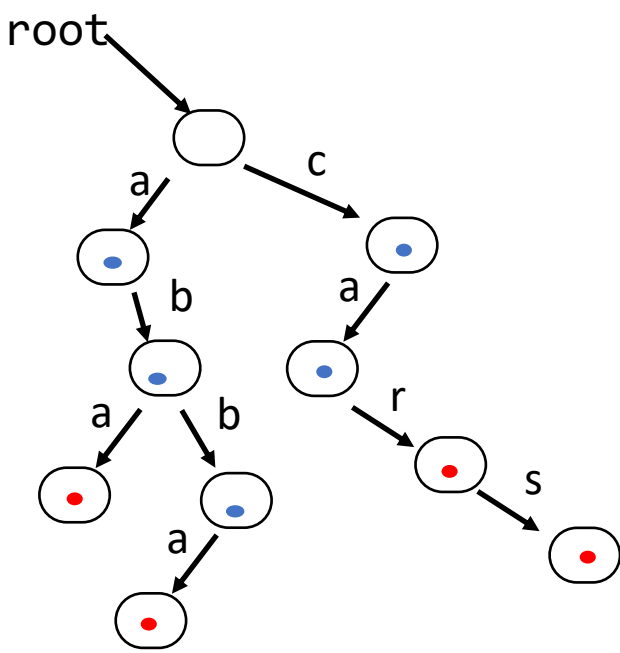
- The metacharacter '*' will match every string that begins with "AVL", ***and will print them in lexicographic order!***
- The BASH shell stores all file names under the current directory in a trie! 
- So let's try to implement metacharacter parsing in a trie ☺

Metacharacter parsing




```
public class AsciiTrie {
        private class Node {
                private Node[] next = new Node[128];
                byte eosFlag;
        }
        private Node root;
        public Collection<String> keysWithPrefix(String prefix){
                /*  Fill the code for this method!
                 *
                 *  Hints:
                 *
                 *  (1) In the trie to the top right, calling keysWithPrefix("ca") would have the same
                 *    effect as ls ca*. Same thing with ls c*.
                 *
                 *  (2) In Java, a Collection is an interface implemented by all kinds of linear
                 *  containers (LinkedList, ArrayList, Queue, Stack, etc)
                 *
                 *  (3) Remember: We want lexicographic ordering!
                 */
        }
}
```

Metacharacter parsing

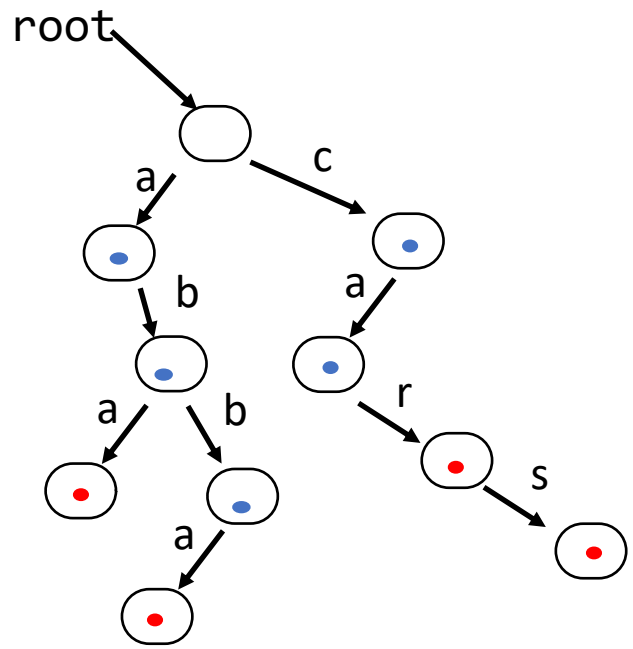



```
public class AsciiTrie {
        private class Node {
                private Node[] next = new Node[128];
                byte eosFlag;
        }
        private Node root;
        public Collection<String> keysWithPrefix(String prefix){
                Node prefixNode = fetchNode(root, prefix, 0);
                ArrayList<String> keys = new ArrayList<String>();
                populate(prefixNode, prefix, keys);
                return keys;
        }
        private void populate(Node curr, String prefix, ArrayList<String> keys){
                if(curr == null) return;
                if(curr.eosFlag == (byte)1) keys.add(prefix);
                for(int i = 0; i < curr.next.length; i++)
                        populate(curr.next[i], prefix + (char)i, keys);
        }
        private Node fetchNode(Node curr, String prefix, int currIndex){
                if(curr == null) return null;
                if(currIndex == prefix.length()) return curr;
                return fetchNode(curr.next[prefix.charAt(currIndex)], prefix, currIndex + 1);
        }
}
```

Tail recursion!

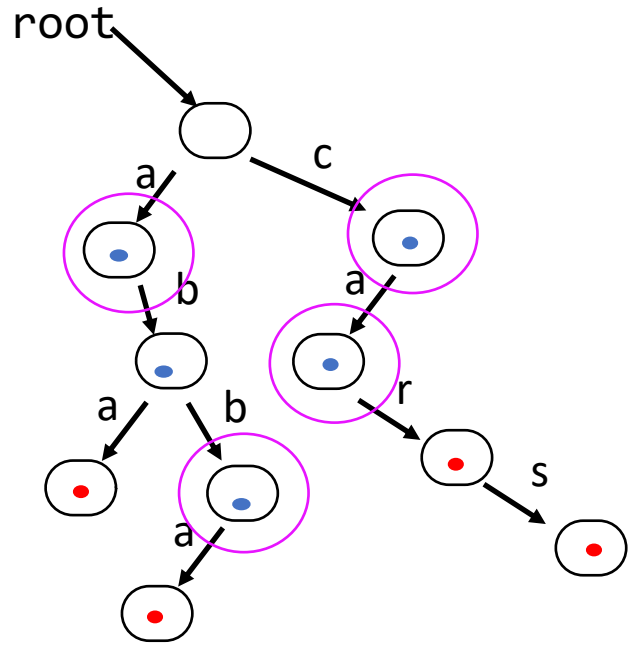
Trie redundancy

- Not everything is perfect in the trie universe!



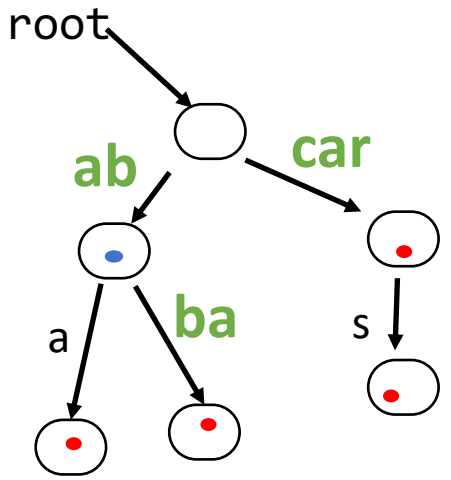
Trie redundancy

- Not everything is perfect in the trie universe!
- Do you agree with me that all the nodes that I circled on the right are effectively useless overhead?

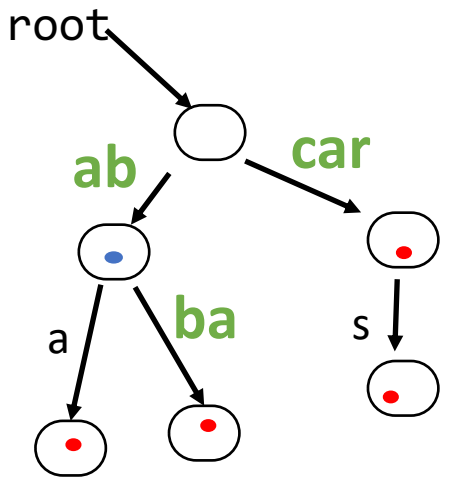


Trie redundancy

- Not everything is perfect in the trie universe!
- Do you agree with me that all the nodes that I circled on the right are effectively useless overhead?
- It would be much better if we could have something like this!

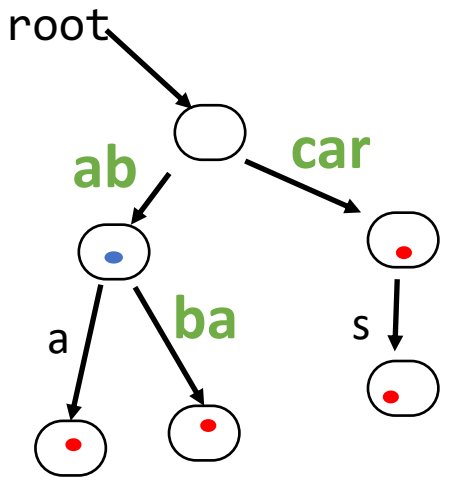


Trie redundancy



- Not everything is perfect in the trie universe!
- Do you agree with me that all the nodes that I circled on the right are effectively useless overhead?
- It would be much better if we could have something like this!
- This we will accomplish with ***Patricia Tries***, a compressed form of trie!

Trie redundancy



- Not everything is perfect in the trie universe!
- Do you agree with me that all the nodes that I circled on the right are effectively useless overhead?
- It would be much better if we could have something like this!
- This we will accomplish with ***Patricia Tries***, a compressed form of trie!

Coming Up Next . . .