Problem 1 : Storing the dictionary

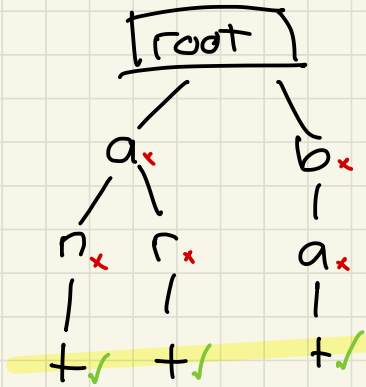
x/✓ valid word bool can tell us if our ending path is a word when searching

- map?
- set?
- hash table?
- trie?

fast traversal but large heap.

- Radix, Patricia, & DAFSAs are better versions

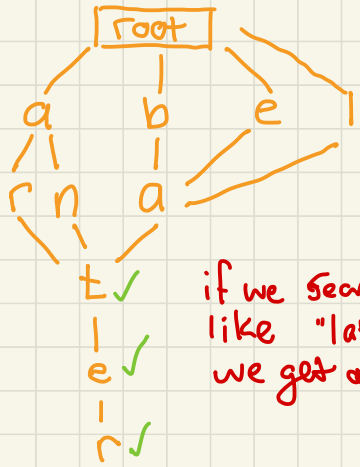
words:
ant
bat
art



... don't need to include letter nodes if a word with that path doesn't exist.

duplicate nodes @ same level... DAFSA cleans up shared suffixes

words
ant
bat
art
eater
late



How do you condense suffixes without causing false positive searches?

if we search for a string like "lat" or "bate", we get a false positive

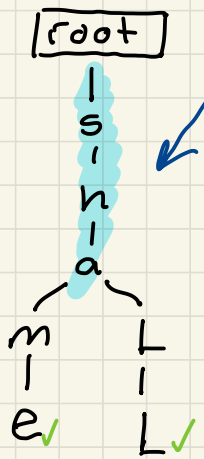
trie → LetterNode graph

```
struct LetterNode {
    bool isWordValid
    unordered_map<char, LN*>
}
```

"LN"

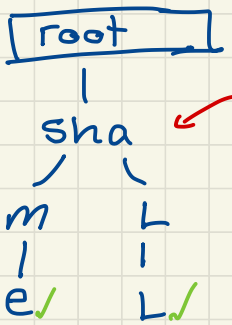
hash map w/ const look up time

"lean" trie



Patricia trie

↳ condense prefix nodes with only 1 parent & 1 child



this requires the node to be a string. Strings are large!! Would this actually be worth it?

You save on lookup time at (maybe) the cost of heap.

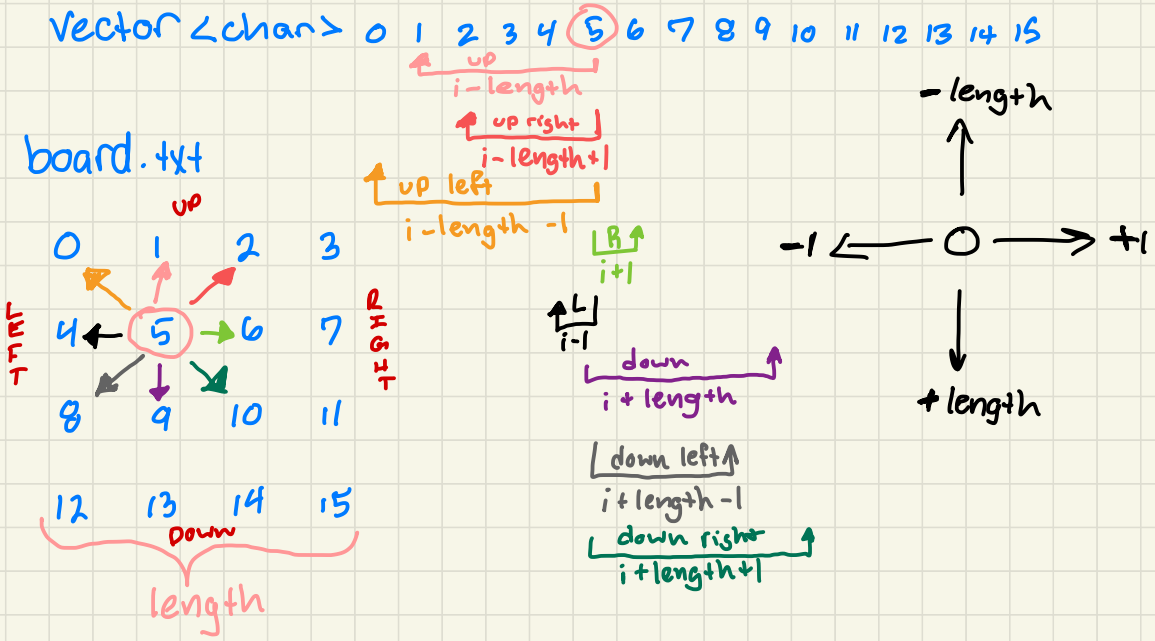
Conclusion:

- store dictionary as a "lean" trie, a minimize function can be implemented later if heap space is an issue

Problem 2: Storing the board

- array or vector? can't assume board size @ compile time. Vector it is.

- 1d or 2d? 2d entities can be traversed in a 1d vector. Stick with 1dimension until 2d are needed.



Boundary Checks

up: (index - length) > 0

1 - 4 = -3

down: (index + length) < vec.size()

14 + 4 = 18

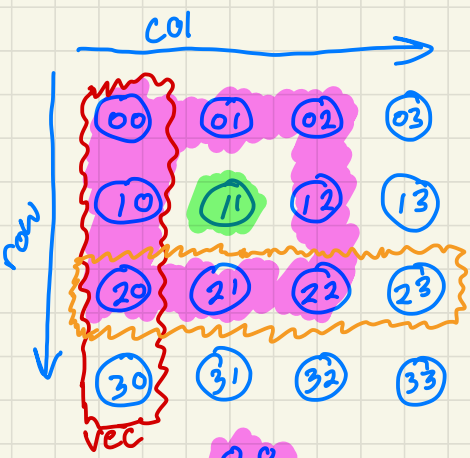
right: (index + 1) < vec.size()

7 + 1 = 8 ...

uh-oh... this causes word wrap

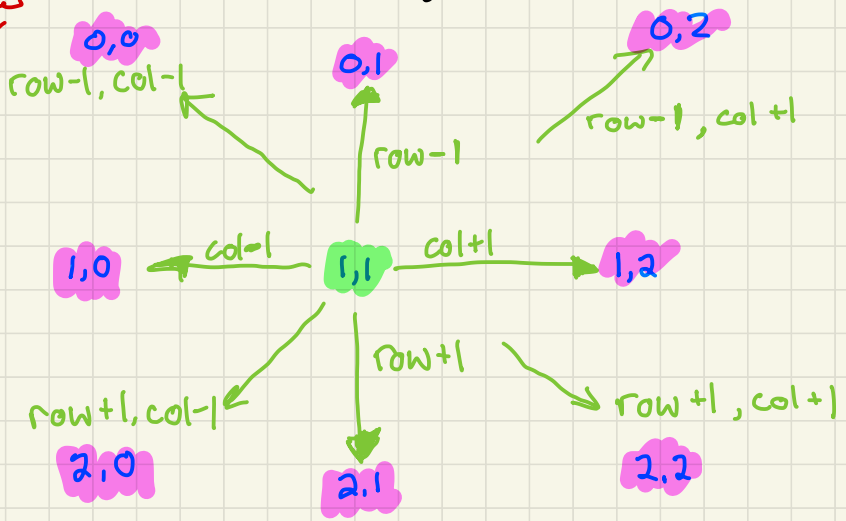
if i have to infer the boggle board line length after importing the board into a 1d vector, then i might as well just use a 2d vector with "row" "col" indexing.

New plan... vector<vector<char>>



vec[row, col] // indexing

- vertical boundaries are based on the length vec
- horizontal boundaries are based on the length of vec[n]
- combining both allows us to handle assymetrical boggle boards



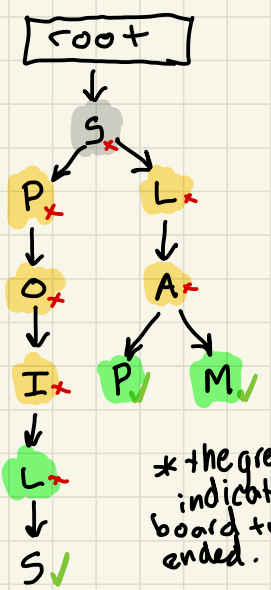
Problem 3: board traversal alg

S Dictionary

SLAM

SLAP

SPOILS



*the green highlight indicates where a board traversal path ended.

- pick an index to start at
- find all possible nodes
- for each of those nodes, find all possible node

  -a valid node is a node that is not currently in a traversal path, and that forms a prefix that exists in the dictionary

recursion

- do this until you are at a node with no more options. Check if your current path is in the dictionary AND that dictionary node has its word flag set to true✓ instead of false✗
- if its a valid word, add it to an unordered_set of answers (if its not already in there... no duplicates). Since there are no more valid nodes, return and begin recursing the next node in the for each

Conclusion: this whole block is a DFS alg that can be put into a single function, given a starting index. Thread this function and give each thread a different starting index.

Problem 4: but how many threads?

- timing metrics were taken with chrono.., its not the best for this kind of thing but its good enough for this exercise.

Group A							Average (ms)
Debug	Trie Import Time (ms) :	25.0506	25.1517	26.193	25.1252	25.2659	**25.35728**
	Solver Time (ms) :	1.9846	1.8177	2.021	2.1112	2.0275	**1.9924**
	Total Time (ms) :	27.1762	27.1003	28.3567	27.3886	27.3977	**27.4839**
Release	Trie Import Time (ms) :	2.4332	2.6298	2.8343	2.585	2.6356	**2.62358**
	Solver Time (ms) :	0.1882	0.1927	0.2022	0.183	0.1884	**0.1909**
	Total Time (ms) :	2.6783	2.8864	3.1485	2.8189	2.8854	**2.8835**
Group B			**4 Threads Boggle Solver (Board 1 - 4x4)**				**Average (ms)**
Debug	Thread Pool Init Time (ms) :	0.8784	0.8347	0.6762	0.834	1.1152	**0.8677**
	Trie Import Time (ms) :	30.545	26.8041	29.4216	27.5521	27.3681	**28.33818**
	Solver Time (ms) :	1.7465	1.559	1.5762	1.5344	1.5654	**1.5963**
	Total Time (ms) :	33.3179	29.2961	31.7877	30.0529	30.141	**30.91912**
Release	Thread Pool Init Time (ms) :	0.553	0.7203	0.7567	0.5431	0.6238	**0.63938**
	Trie Import Time (ms) :	3.4294	3.7447	4.3308	3.8123	3.6492	**3.79328**
	Solver Time (ms) :	0.0668	0.0923	0.0823	0.0992	0.0818	**0.08448**
	Total Time (ms) :	4.1189	4.6142	5.2264	4.5133	4.4146	**4.57748**
Group C			**16 Threads Boggle Solver (Board 1 - 16x16)**				**Average (ms)**
Debug	Thread Pool Init Time (ms) :	4.2096	3.4695	3.51	2.0713	3.7923	**3.41054**
	Trie Import Time (ms) :	26.6416	26.3802	26.3059	28.5671	27.0124	**26.98144**
	Solver Time (ms) :	3.2331	3.1902	3.0763	3.2194	3.1624	**3.17628**
	Total Time (ms) :	34.2263	33.1404	32.995	34.0058	34.091	**33.6917**
Release	Thread Pool Init Time (ms) :	2.1333	3.379	1.7115	2.0407	1.5358	**2.16006**
	Trie Import Time (ms) :	3.5162	3.4386	3.8373	4.0508	3.6824	**3.70506**
	Solver Time (ms) :	0.1017	0.0722	0.0971	0.0865	0.1113	**0.09376**
	Total Time (ms) :	5.81	6.9522	5.7049	6.2431	5.3844	**6.01892**

Solver Time is how long it takes threads to solve the board

- 1 thread (baseline) = 0.1909ms
- 4 threads = 0.08448ms
- 16 threads (hw_rec.) = 0.09376ms

} 4x4 boggle board

4 threads 50% faster than 1 and ~10% faster than 16... maybe diminishing returns?

Threading Stress test on 16×16 board

Group D		Single Threaded Boggle Solver (stress test - 16x16)					Average (ms)
Debug	Trie Import Time (ms) :	26.2009	27.7689	25.7078	28.2877	25.4453	**26.68212**
	Solver Time (ms) :	74.225	72.268	71.4945	74.2387	75.115	**73.46824**
	Total Time (ms) :	100.614	100.337	97.3935	102.759	100.785	**100.3777**
Release	Trie Import Time (ms) :	2.4715	2.4648	2.7787	2.7776	2.8215	**2.66282**
	Solver Time (ms) :	5.7517	5.103	5.0687	5.137	5.064	**5.22488**
	Total Time (ms) :	8.2816	7.6272	7.9188	7.9711	7.9414	**7.94802**
Group E		**4 Threads Boggle Solver (stress test - 16x16)**					**Average (ms)**
Debug	Thread Pool Init Time (ms) :	0.8697	0.7745	0.8304	0.7676	0.6836	**0.78516**
	Trie Import Time (ms) :	27.1962	28.5036	26.8004	30.0358	26.9143	**27.89006**
	Solver Time (ms) :	57.3981	55.3949	56.9946	56.6623	57.522	**56.79438**
	Total Time (ms) :	85.7466	84.8525	84.8041	87.6621	85.3031	**85.67368**
Release	Thread Pool Init Time (ms) :	0.7976	0.8424	0.5025	0.6267	0.6001	**0.67386**
	Trie Import Time (ms) :	3.9737	3.5511	3.5012	4.3719	4.144	**3.90838**
	Solver Time (ms) :	1.4703	1.3407	1.4524	1.2494	1.1713	**1.33682**
	Total Time (ms) :	6.3041	5.7963	5.5173	6.309	5.9755	**5.98044**
Group F		**16 Threads Boggle Solver (stress test - 16x16)**					**Average (ms)**
Debug	Thread Pool Init Time (ms) :	2.4742	1.9934	5.7101	3.9191	3.6894	**3.55724**
	Trie Import Time (ms) :	26.5714	27.3084	26.3544	27.1079	26.1164	**26.6917**
	Solver Time (ms) :	189.9	192.999	191.831	191.368	190.685	**191.3566**
	Total Time (ms) :	219.133	222.546	224.228	222.627	220.683	**221.8434**
Release	Thread Pool Init Time (ms) :	2.8472	2.1787	2.0709	3.3525	1.7252	**2.4349**
	Trie Import Time (ms) :	3.8545	4.2451	3.6	3.6127	4.1139	**3.88524**
	Solver Time (ms) :	0.819	0.9277	0.8224	0.9376	0.9999	**0.90132**
	Total Time (ms) :	7.5839	7.412	6.5567	7.9758	6.9185	**7.28938**

- 1 thread (baseline) = 5.22488ms
- 4 thread = 1.33682ms
- 16 thread (hw_rec) = 0.90132ms

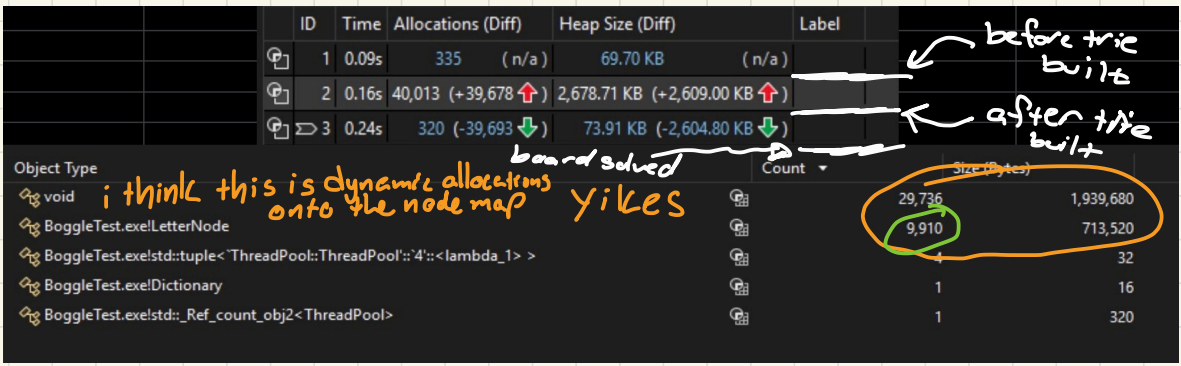
4 threads are still about 75% better than 1, but 16 is 50% better than 4.

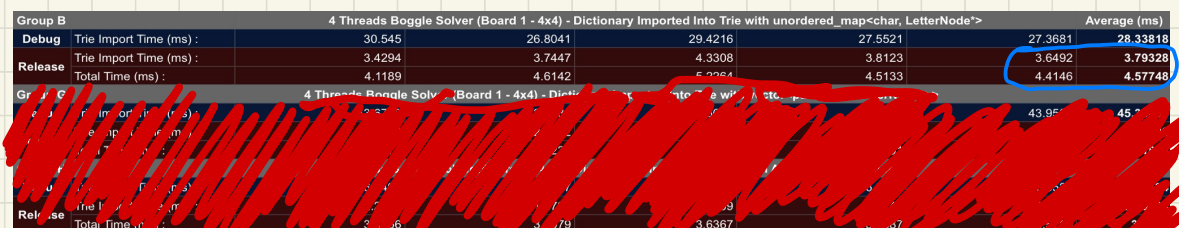
Conclusion:

- because varying board sizes was never specified as a requirement, default to 4 threads...
- ... but if the hw_rec is lower than 4, take the minimum of the two.
- could always dynamically decide how many threads to run based on board size, but i'm treating it as YAGNI for now.

Problem 5: how is my heap?

- memory profiles taken with VS built-in memory profiler.





No SPOILERS ♥ We'll get to this data later.

- threads are now solving the board faster, but my trie import time (building the dictionary) is still a single thread. It accounts for 80% of my execution time.
- that trie process is taking up 2.6 MB, which seems like too much for a relatively small dictionary of only 9910 trie nodes

- lets convert the

```
LetterNode {
  map<char, LetterNode*>
}
```

to

```
LetterNode {
  vector<pair<char, LetterNode*>>
}
```

LetterNode & vector<pair<char, LN*>> heap analysis

ID	Time	Allocations (Diff)	Heap Size (Diff)	Label
1	0.09s	324 (n/a)	68.24 KB (n/a)	
2	0.13s	17,263 (+16,939 ↑)	554.55 KB (+486.31 KB ↑)	
3	0.16s	320 (-16,943 ↓)	73.94 KB (-480.62 KB ↓)	
4	0.16s	320 (+0)	73.94 KB (+0.00 KB)	

Object Type	Count	Size (Bytes)
BoggleTest.exe!LetterNode	9,910	317,120
void	6,986	161,632
BoggleTest.exe!std::tuple<`ThreadPool::`4'::<lambda_1> >	4	32
BoggleTest.exe!std::_Ref_count_resource<Dictionary *,std::default_delete<Dictionary> >	1	24
BoggleTest.exe!Dictionary	1	16
BoggleTest.exe!std::_Ref_count_obj2<ThreadPool>	1	320

i think these are "new" allocations

- switching to a vector saves 80% of heap compared to the map.

How fast is it?

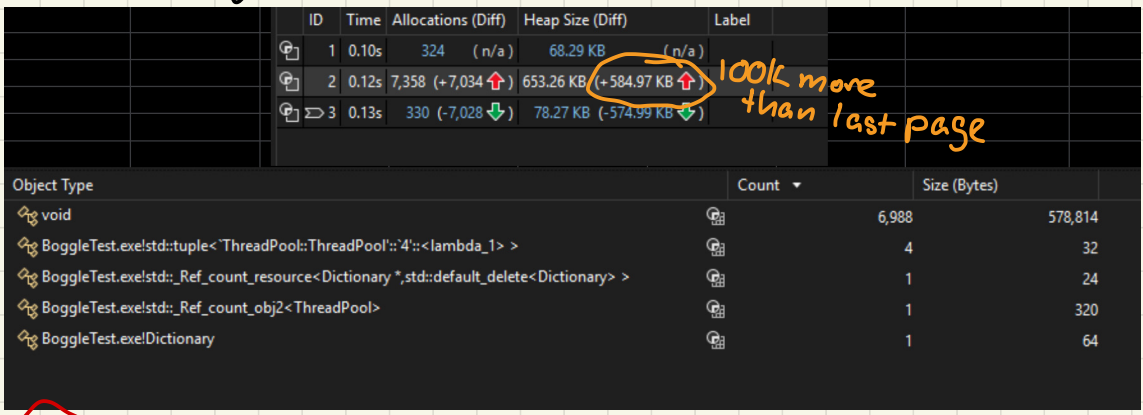
Group B		4 Threads Boggle Solver (Board 1 - 4x4) - Dictionary Imported Into Trie with unordered_map<char, LetterNode*>					Average (ms)
Debug	Trie Import Time (ms) :	30.545	26.8041	29.4216	27.5521	27.3681	28.33818
Release	Trie Import Time (ms) :	3.4294	3.7447	4.3308	3.8123	3.6492	3.79328
	Total Time (ms) :	4.1189	4.6142	5.2264	4.5133	4.4146	4.57748
Group G		4 Threads Boggle Solver (Board 1 - 4x4) - Dictionary Imported Into Trie with vector<pair<char, LetterNode*>>					
Debug	Trie Import Time (ms) :	43.6709	43.7085	45.6689	49.6838	43.9508	45.33658
Release	Trie Import Time (ms) :	2.5092	2.5212	2.4467	2.7347	2.5886	2.56008
	Total Time (ms) :	3.4771	3.6239	3.6026	4.1801	3.7067	3.71808

almost there, i promise

- execution times dropped by about 30%

- it can most certainly be faster with a custom Pool allocator

LetterNodes being preallocated by a custom heap allocator



	ID	Time	Allocations (Diff)	Heap Size (Diff)	Label
	1	0.10s	324 (n/a)	68.29 KB (n/a)	
	2	0.12s	7,358 (+7,034)	653.26 KB (+584.97 KB)	
	3	0.13s	330 (-7,028)	78.27 KB (-574.99 KB)	

100k more than last page

Object Type	Count	Size (Bytes)
void	6,988	578,814
BoggleTest.exe!std::tuple<`ThreadPool::ThreadPool'::`4'::<lambda_1> >	4	32
BoggleTest.exe!std::_Ref_count_resource<Dictionary *,std::default_delete<Dictionary> >	1	24
BoggleTest.exe!std::_Ref_count_obj2<ThreadPool>	1	320
BoggleTest.exe!Dictionary	1	64

Group B							Average (ms)
		4 Threads Boggle Solver (Board 1 - 4x4) - Dictionary Imported Into Trie with unordered_map<char, LetterNode*>					
Debug	Trie Import Time (ms) :	30.545	26.8041	29.4216	27.5521	27.3681	28.33818
Release	Trie Import Time (ms) :	3.4294	3.7447	4.3308	3.8123	3.6492	3.79328
	Total Time (ms) :	4.1189	4.6142	5.2264	4.5133	4.4146	4.57748
Group G		4 Threads Boggle Solver (Board 1 - 4x4) - Dictionary Imported Into Trie with vector<pair<char, LetterNode*>>					
Debug	Trie Import Time (ms) :	43.6709	43.7085	45.6689	49.6838	43.9508	45.33658
Release	Trie Import Time (ms) :	2.5092	2.5212	2.4467	2.7347	2.5886	2.56008
	Total Time (ms) :	3.4771	3.6239	3.6026	4.1801	3.7067	3.71808
Group H		4 Threads Boggle Solver (Board 1 - 4x4) - Dictionary Imported with Custom Allocator (10,000 nodes)					
Debug	Trie Import Time (ms) :	46.3492	44.267	42.135	46.6357	47.6528	45.40794
Release	Trie Import Time (ms) :	2.2395	1.8793	1.9009	2.0749	2.2506	2.06904
	Total Time (ms) :	3.3356	3.1979	3.6367	3.2887	3.5256	3.3969

- heap usage is slightly larger by about 25% (100kB)
- BUT its 20% faster than the vector trie w/o the pool allocator, AND its a total 40ish% faster than where we started (trie with maps, no custom allocator)

Conclusion: faster execution time is worth the extra heap usage. This time likely is gained by not using any "new" keywords as we import the dictionary.

Note: this data uses a pool of 10k nodes, but i set the deliverable to 15k b/c idk if this will be tested with a larger dictionary. The deliverable exits gracefully & logs error if pool is empty.