Boggle Player

Group: 34b

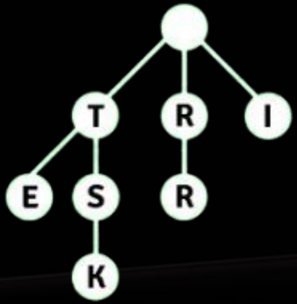
Authors: Anthony, Darian, Matthew

Goal and Motivation

To make something that finds words and works. Preferably quickly and efficiently.

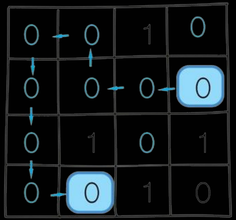
Our motivation was to get a good grade and have the highest scoring boggle player.

Initial Approach



Data Structures

1. Trie
   a. TrieNode class: Each node in the Trie contains an array of children nodes for each letter of the alphabet. Each node includes isEndOfWord attribute, indicating the end of a word.
   b. Insert: During Trie creation, each character in a word is added as a node. O(m) time to insert, where m is the word length.



Algorithms

1. Depth-First Search
   a. Decomposition: Smaller words. Starting at length 1 word, and 1 cell.
   b. Base Case: If the word is a word & is unique & length > 2, add to PQ.
   c. Composition: Expand path, adding characters adjacent of cell to the word, until it reaches max length 8.

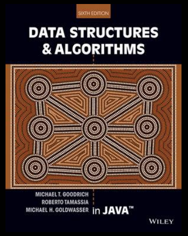
Initial Approach

Ideas devised by group

- Use of Bufferedreader instead of Scanner. Using Arrays to implement a Trie

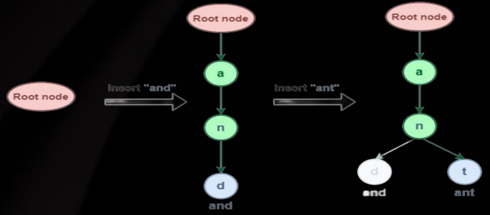
Ideas discussed in course

- Trie data structure, Depth-first search (*Data Structures and Algorithms in Java, Goodrich*)

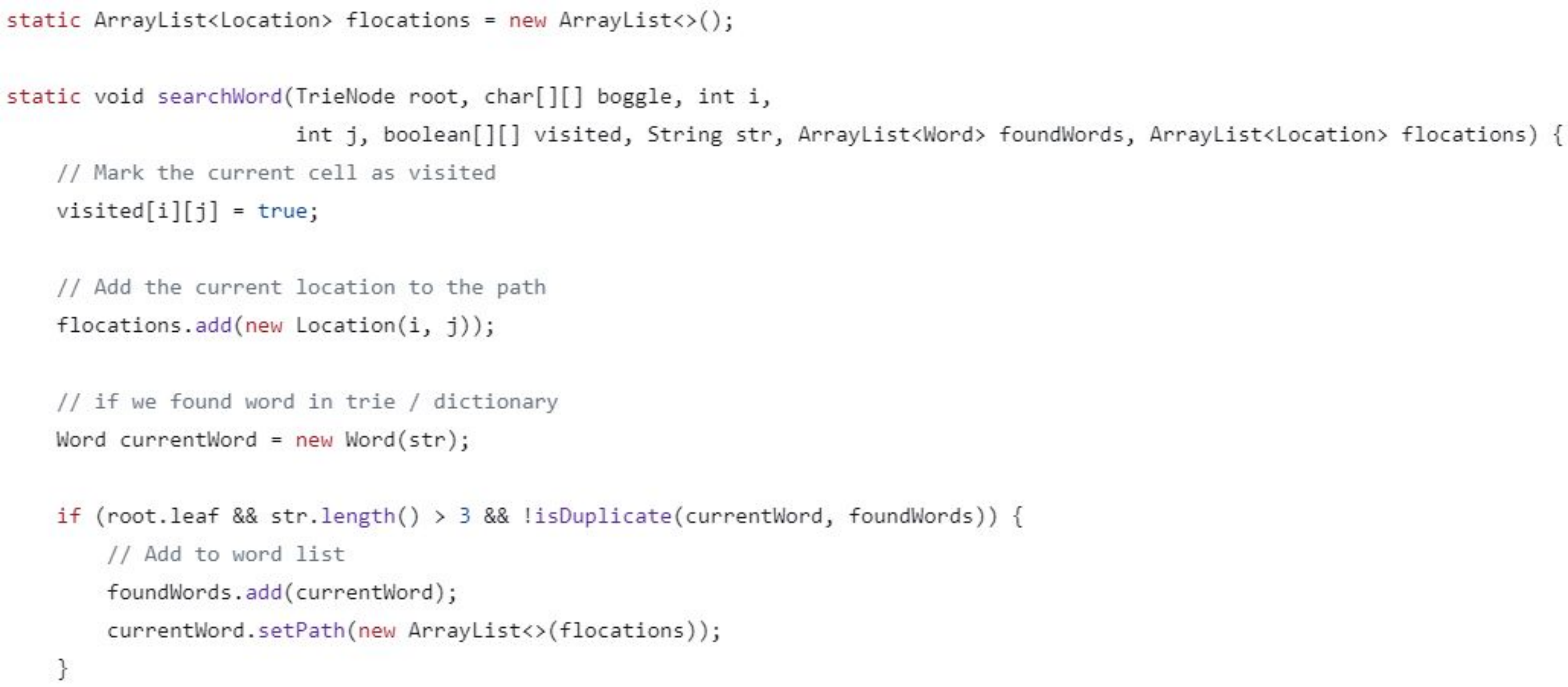


Ideas borrowed from other sources, cite them

- Trie data structure insert and search concepts (*GeeksforGeeks*)

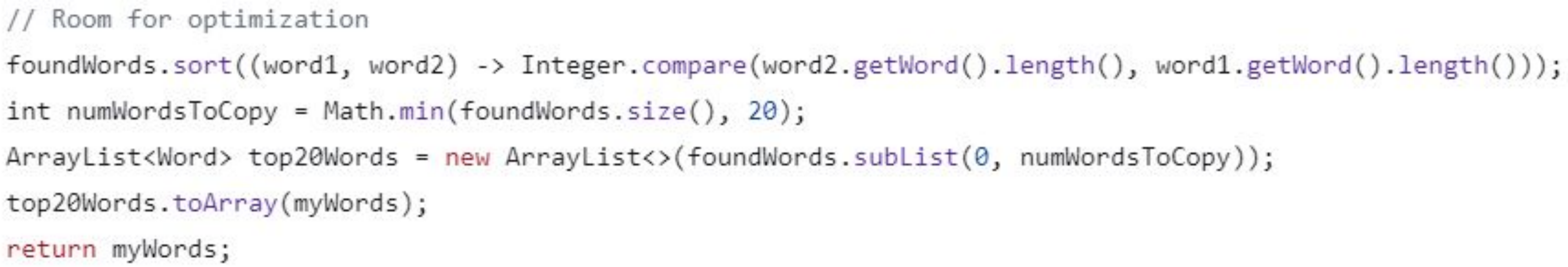


Initial Search Word (Array List)



```
static ArrayList<Location> flocations = new ArrayList<>();

static void searchWord(TrieNode root, char[][] boggle, int i,
                       int j, boolean[][] visited, String str, ArrayList<Word> foundWords, ArrayList<Location> flocations) {
    // Mark the current cell as visited
    visited[i][j] = true;

    // Add the current location to the path
    flocations.add(new Location(i, j));

    // if we found word in trie / dictionary
    Word currentWord = new Word(str);

    if (root.leaf && str.length() > 3 && !isDuplicate(currentWord, foundWords)) {
        // Add to word list
        foundWords.add(currentWord);
        currentWord.setPath(new ArrayList<>(flocations));
    }
```

Initial Return myWords (Sorting Array List)



```
// Room for optimization
foundWords.sort((word1, word2) -> Integer.compare(word2.getWord().length(), word1.getWord().length()));
int numWordsToCopy = Math.min(foundWords.size(), 20);
ArrayList<Word> top20Words = new ArrayList<>(foundWords.subList(0, numWordsToCopy));
top20Words.toArray(myWords);
return myWords;
```

Final Approach

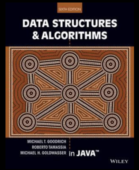
Data Structures:

- HeapPriorityQueue:
    - Used a Heap to store largest found words on the boggle board
- HashSet:
    - Used a HashSet to store found words on the boggle board to check for duplicates

Ideas devised within the group:

- Add a Heap
- Add a HashSet

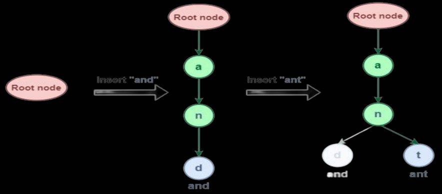
Ideas discussed in the course/book:



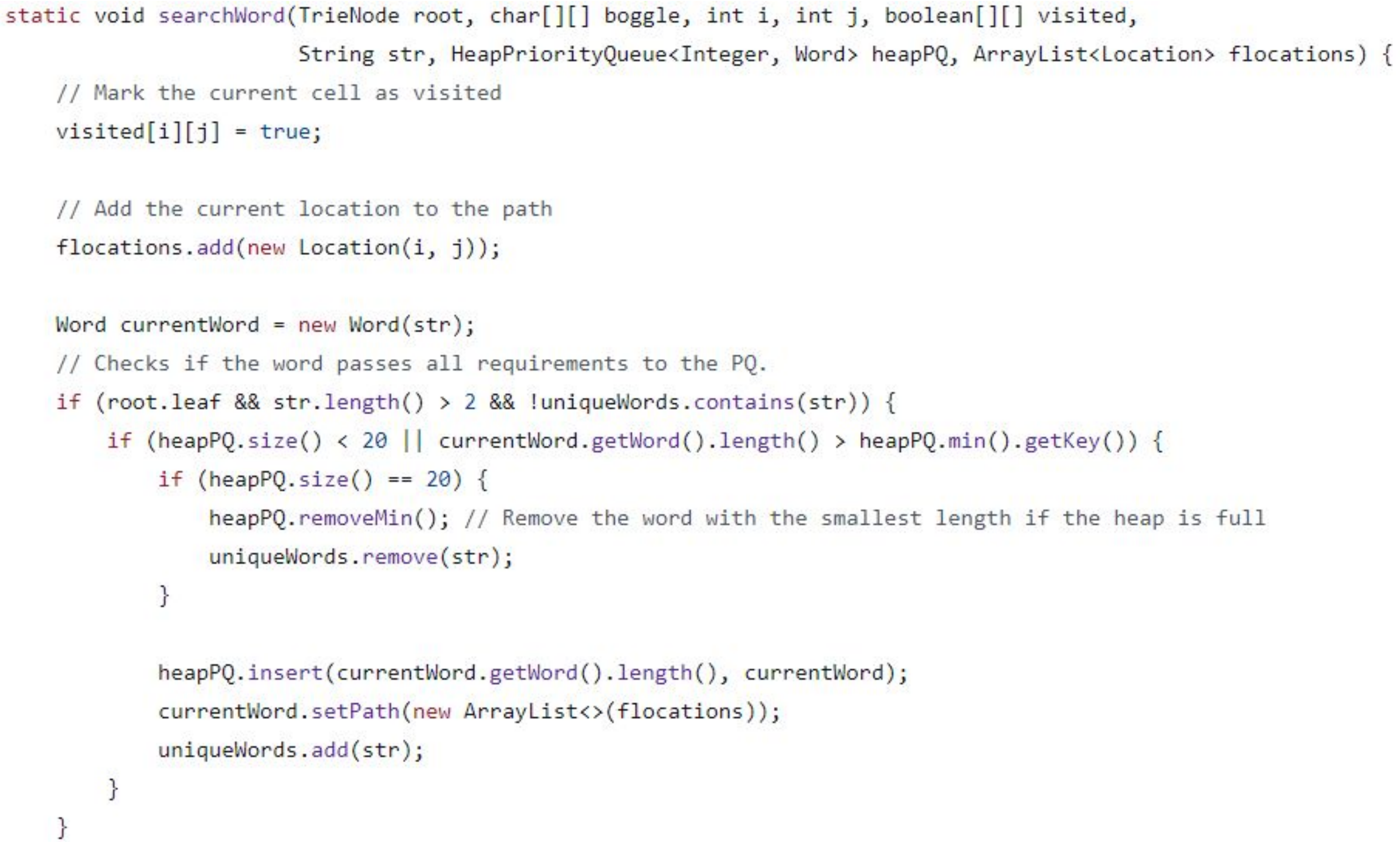
- Dr. Chan's HeapPriorityQueue data structure implementation

Ideas borrowed from other sources:

- Trie data structure insert and search concepts (*GeeksforGeeks*)

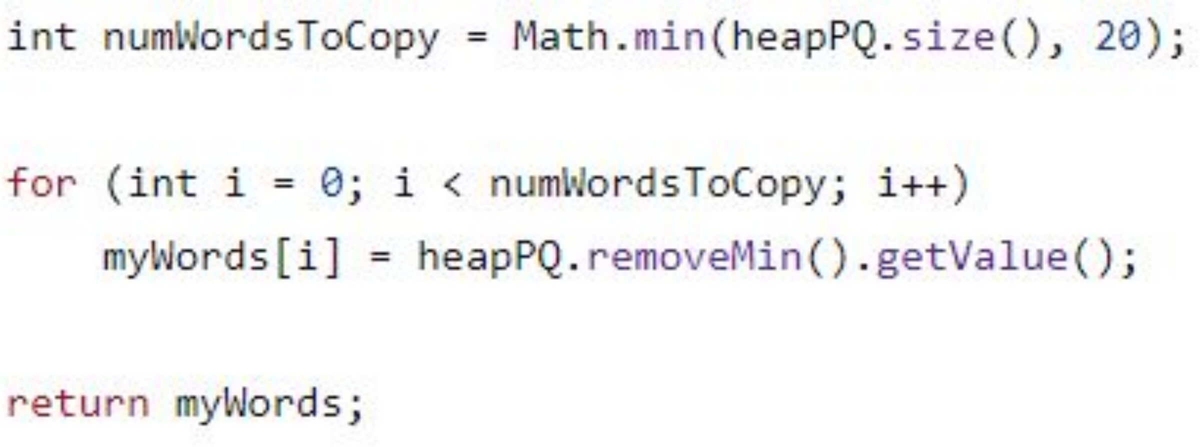


Final Search Word (Heap Priority Queue)



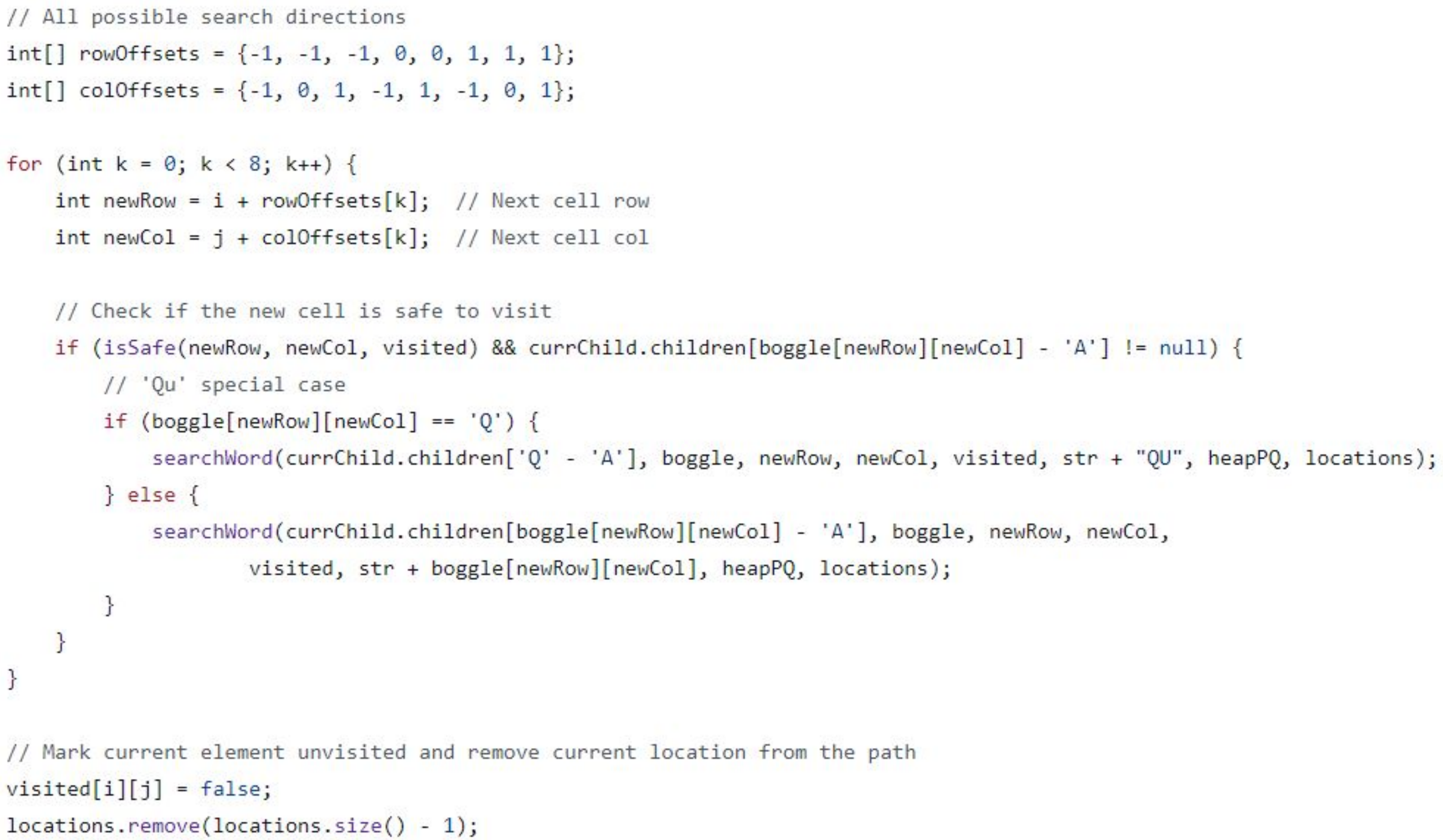
```java
static void searchWord(TrieNode root, char[][] boggle, int i, int j, boolean[][] visited,
                       String str, HeapPriorityQueue<Integer, Word> heapPQ, ArrayList<Location> flocations) {
    // Mark the current cell as visited
    visited[i][j] = true;

    // Add the current location to the path
    flocations.add(new Location(i, j));

    Word currentWord = new Word(str);
    // Checks if the word passes all requirements to the PQ.
    if (root.leaf && str.length() > 2 && !uniqueWords.contains(str)) {
        if (heapPQ.size() < 20 || currentWord.getWord().length() > heapPQ.min().getKey()) {
            if (heapPQ.size() == 20) {
                heapPQ.removeMin(); // Remove the word with the smallest length if the heap is full
                uniqueWords.remove(str);
            }

            heapPQ.insert(currentWord.getWord().length(), currentWord);
            currentWord.setPath(new ArrayList<>(flocations));
            uniqueWords.add(str);
        }
    }
```

Final Return myWords (Removing Min)



```
int numWordsToCopy = Math.min(heapPQ.size(), 20);

for (int i = 0; i < numWordsToCopy; i++)
    myWords[i] = heapPQ.removeMin().getValue();

return myWords;
```

Recursive calls (in SearchWord method)



```
// All possible search directions
int[] rowOffsets = {-1, -1, -1, 0, 0, 1, 1, 1};
int[] colOffsets = {-1, 0, 1, -1, 1, -1, 0, 1};

for (int k = 0; k < 8; k++) {
    int newRow = i + rowOffsets[k];  // Next cell row
    int newCol = j + colOffsets[k];  // Next cell col

    // Check if the new cell is safe to visit
    if (isSafe(newRow, newCol, visited) && currChild.children[boggle[newRow][newCol] - 'A'] != null) {
        // 'Qu' special case
        if (boggle[newRow][newCol] == 'Q') {
            searchWord(currChild.children['Q' - 'A'], boggle, newRow, newCol, visited, str + "QU", heapPQ, locations);
        } else {
            searchWord(currChild.children[boggle[newRow][newCol] - 'A'], boggle, newRow, newCol,
                    visited, str + boggle[newRow][newCol], heapPQ, locations);
        }
    }
}

// Mark current element unvisited and remove current location from the path
visited[i][j] = false;
locations.remove(locations.size() - 1);
```

Evaluation

Initial

Points: 117

Time: 7.97E-3

Memory: 4.45E+8

Score: 7.734

Final

Points: 201

Time: 9.3268E-3

Memory: 1.5659E8

Score: 33.4304

Analysis

Why more points

- Initial: Improper storage of top 20 words and exclusion of found longer words.
- Final: Proper storage of top 20 words. Ensured only 20 highest length words were included.

Why Slightly Slower

- HashSet: For the final approach, a Set ensures duplicate words aren't added in **O(1) time**.
- ArrayList: For the Initial approach, an ArrayList was traversed and checked for duplicates in **O(n) time**. *Accidental early termination of word searching allowed for a faster time initially.*

Why more/less memory?

- Initial: ArrayList is used to store **every** found word.
- Final: Heap Priority Queue stores **only** the 20 longest words.

Possible further improvements

Trie compression: Compression nodes into a prefix Trie, allowing for faster search time, and decreased space.

HashMap: Avoid having unused space by mapping each node. Could have faster search time due to direct access.