CS 61B Fall 2025

Tries and Sorting

Exam-Level Discussion 11: November 10, 2025

1 A Wordsearch

Given an N by N wordsearch and N words, devise an algorithm (using pseudocode or describe it in plain English) to solve the wordsearch in $O(N^3)$. For simplicity, assume no word is contained within another, i.e., if the word **"bear"** is given, **"be"** wouldn't also be given.

If you are unfamiliar with wordsearches or want to gain some wordsearch solving intuition, see below for an example wordsearch.

Example Wordsearch:

S F T A T R D V R G K E V I N
A T S K L N X F I H D P X H Z
C N E D X A J Z G U N A I R U
J Y I L C V A N E S S A V P O
A B A R L K F J Q U S Y H I C
V Z U I U A T Q K D A A G R D
F S P E I D S T A A S N M Y N
W C T T S S H Q T W H N G A U
S I W H P E A A E N L I T N V
T Y I A G L D B R K E Y T Y K
A L N N J A J G E T Y A P E A
C X F I K I M U S L I T Y P R
E M U A M N T M A Z X A H U E
Y R A E K E Y W I K O Y O P N
R B C A Q J V Q I A C R O E F

Anirudh	Anniyat	Vanessa	Ryan
Ashley	Elaine	Isabel	
David	Stella	Karen	
Ethan	Kevin	Teresa	
Stacey	Dawn		

Hint: Add the words to a **Trie**, and you may find the `longestPrefixOf` operation helpful. Recall that `longestPrefixOf` accepts a `String key` and returns the longest prefix of `key` that exists in the **Trie**, or `null` if no prefix exists.

Solution:
Algorithm: Begin by adding all the words we are querying for into a **Trie**. Next, we will iterate through each letter in the wordsearch and see if any words **start** with that letter. For a word to start with a given letter, note that it can go in one of eight directions — N, NE, E, SE, S, SW, W, NW.

Looking at each direction, we will check if the string going in that direction has a prefix that exists in our **Trie**, which we can do using `longestPrefixOf`. Note that words are not nested inside of others, so **at most** one word can start from a given letter in a given direction. As such, if `longestPrefixOf` returns a word, we know it is the only word that goes in that direction from that letter.

For instance, if we are at the letter "S" in the middle of the top row of the wordsearch above and are considering the direction west, we would want to see if the string `"SOHUMC"` has a prefix that exists in the given wordsearch. To efficiently perform this query, we call `longestPrefixOf("SOHUMC")`, which, in this case, returns `"SOHUM"`, and we proceed by removing `"SOHUM"` from our **Trie** to signal that we found the word `"SOHUM"`.

We will repeat this process until all the words have been found, i.e. when the **Trie** is empty. Finally, note that this is a very open-ended problem, so this is one of **many** possible solutions.

Runtime: We look at N^2 letters. At each letter, we execute eight calls to `longestPrefixOf` which runs in time linear to the length of the inputted string, which can be of at most length N, since that is the height and width of the wordsearch. Thus, if we perform on the order of N work per letter and we look at N^2 letters, the runtime is $O(N^3)$.

2 Longest Prefix

Fill in the **`longestPrefixOf(String word)`** method below such that it returns the longest prefix of **`word`** that is also a prefix of a key in the trie.

For example, if a **`TrieSet t`** contains keys **`{"cryst", "tries", "cr"}`**, then **`t.longestPrefixOf("crystal")`** returns **`"cryst"`** and **`t.longestPrefixOf("crys")`** returns **`"crys"`**.

The code uses the **`StringBuilder`** class to build strings character-by-character. To add a character to the end of the **`StringBuilder`**, use the **`append(char c)`** method. Once all characters have been appended, the resulting String is returned by the **`toString()`** method.

```
  StringBuilder sb = new StringBuilder();
  sb.append('a');
  sb.append('b');
  System.out.println(sb.toString()); // "ab"
```

```
public class TrieSet {
    private Node root;
    private class Node {
        boolean isKey;
        Map<Character, Node> map;
        private Node() {
            isKey = false;
            map = new HashMap<>();
        }
    }

    public String longestPrefixOf(String word) {
        int n = word.length();
        StringBuilder prefix = new StringBuilder();
        Node curr = _____________________;
        for (_____________________________________) {
            ____________________________________________________
            ____________________________________________________
            ____________________________________________________
            ____________________________________________________
            ____________________________________________________
            ____________________________________________________
        }
        return _____________________________________________
    }
}
```

Solution:

```
public String longestPrefixOf(String word) {
    int n = word.length();
    StringBuilder prefix = new StringBuilder();
    Node curr = root;
    for (int i = 0; i < n; i++) {
        char c = word.charAt(i);
        if (!curr.map.containsKey(c)) {
            break;
        }
        curr = curr.map.get(c);
        prefix.append(c);
    }
    return prefix.toString();
}
```

3 All Sorts of Sorts

Show the steps taken by each sort on the following unordered list:

`0, 4, 2, 7, 6, 1, 3, 5`

(a) Insertion sort

Solution:

0 | 4 2 7 6 1 3 5
0 4 | 2 7 6 1 3 5
0 2 4 | 7 6 1 3 5
0 2 4 7 | 6 1 3 5
0 2 4 6 7 | 1 3 5
0 1 2 4 6 7 | 3 5
0 1 2 3 4 6 7 | 5
0 1 2 3 4 5 6 7 |

(b) Selection sort

Solution:

0 | 4 2 7 6 1 3 5
0 1 | 2 7 6 4 3 5
0 1 2 | 7 6 4 3 5
0 1 2 3 | 6 4 7 5
0 1 2 3 4 | 6 7 5
0 1 2 3 4 5 | 7 6
0 1 2 3 4 5 6 | 7
0 1 2 3 4 5 6 7 |

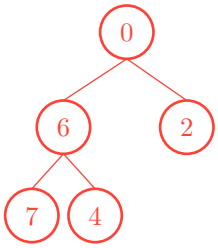
(c) Merge sort

Solution:

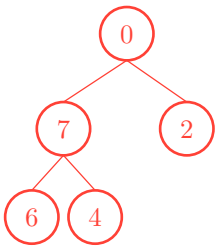
[0 4 2 7 6 1 3 5]
[0 4 2 7] [6 1 3 5]
[0 4] [2 7] [6 1] [3 5]
[0] [4] [2] [7] [6] [1] [3] [5]
[0 4] [2 7] [1 6] [3 5]
[0 2 4 7] [1 3 5 6]
[0 1 2 3 4 5 6 7]

(d) Use heapsort to sort the following array (hint: draw out the heap).
Draw out the array at each step: `0, 6, 2, 7, 4`

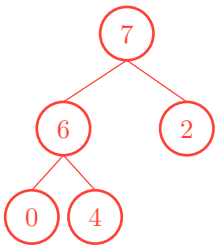
Solution: First, we need to heapify our array. We convert the current array to a max heap:



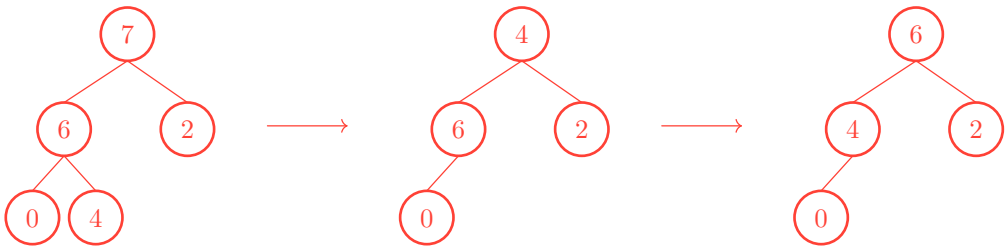
Recall that to heapify our array, we bubble down in reverse level order (bottom to top, right to left). This means we start by bubbling down 4, which in this case gives us the same heap structure. Bubbling down 7, and then 2, leaves the heap unchanged as well. Bubbling down 6 (swapping 6 and 7) then gives us the following:



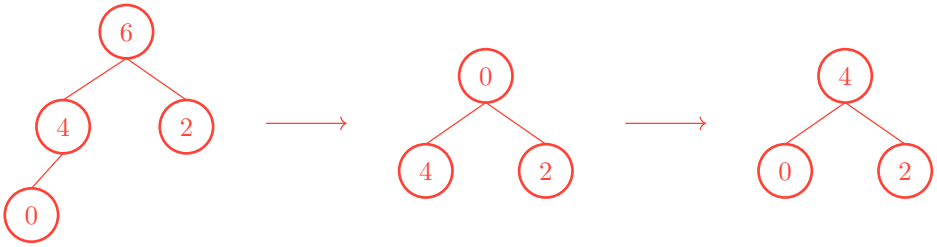
Bubbling down 0 gives us our final heap:



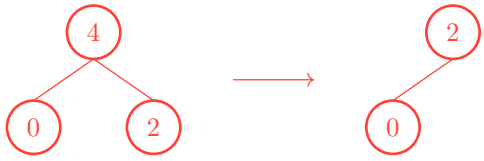
Note that as we heapify, we also modify the underlying array representation as well. This means that our final array looks like **[7, 6, 2, 0, 4]**. We then begin popping off the max value from the heap, placing it at the back of the array. Note that our underlying array representation doesn't consider the popped value as part of the heap any more. We start by popping off 7 and bubbling down:



Our array now looks like this: **[6, 4, 2, 0, 7]**, where the bolded section is considered sorted and not part of the heap. We the continue by popping off 6:



and the array looks like **[4, 0, 2, 6, 7]**. We then pop off 4:



and our array looks like **[2, 0, 4, 6, 7]**. In a similar fashion, we pop off 2 and 0 from our heap, resulting in **[0, 2, 4, 6, 7]** and finally our sorted array: **[0, 2, 4, 6, 7]**