1 Strings

1.1 Trie

```cpp
#include "../../headers/headers.h"

/* Implementation from: https://pastebin.com/fyqsH65k */
struct TrieNode
{
    int leaf; // number of words that end on a TrieNode (allows for duplicate words)
    int height; // height of a TrieNode, root starts at height = 1, can be changed with the default value of constructor
    // number of words that pass through this node,
    // ask root node for this count to find the number of entries on the whole Trie
    // all nodes have 1 as they count the words than end on themselves (ie leaf nodes count themselves)
    int count;
    TrieNode *parent; // pointer to parent TrieNode, used on erasing entries
    map<char, TrieNode *> child;
    TrieNode(TrieNode *parent = NULL, int height = 1):
        parent(parent),
        leaf(0),
        height(height),
        count(0), // change to -1 if leaf nodes are to have count 0 insead of 1
        child()
    {}
};

/**
 * Complexity: O(|key| * log(k))
 */
TrieNode *trie_find(TrieNode *root, const string &str)
{
    TrieNode *pNode = root;
    for (string::const_iterator key = str.begin(); key != str.end(); key++)
    {
        if (pNode->child.find(*key) == pNode->child.end())
            return NULL;
        pNode = pNode->child[*key];
    }
    return (pNode->leaf) ? pNode : NULL; // returns only whole word
    // return pNode; // allows to search for a suffix
}

/**
 * Complexity: O(|key| * log(k))
 */
void trie_insert(TrieNode *root, const string &str)
{
    TrieNode *pNode = root;
    root -> count += 1;
    for (string::const_iterator key = str.begin(); key != str.end(); key++)
    {
        if (pNode->child.find(*key) == pNode->child.end())
            pNode->child[*key] = new TrieNode(pNode, pNode->height + 1);
        pNode = pNode->child[*key];
        pNode -> count += 1;
    }
    pNode->leaf += 1;
}

/**
 * Complexity: O(|key| * log(k))
 */
void trie_erase(TrieNode *root, const string &str)
{
    TrieNode *pNode = root;
    string::const_iterator key = str.begin();
    for (; key != str.end(); key++)
    {
        if (pNode->child.find(*key) == pNode->child.end())
            return;
        pNode = pNode->child[*key];
    }
    pNode->leaf -= 1;
    pNode->count -= 1;
    while (pNode->parent != NULL)
    {
        if (pNode->child.size() > 0 || pNode->leaf)
            break;
        pNode = pNode->parent, key--;
        pNode->child.erase(*key);
        pNode->count -= 1;
    }
}
```

1.2 KMP

```cpp
#include "../../headers/headers.h"

vi prefix(string &S)
{
    vector<int> p(S.size());
    p[0] = 0;
    for (int i = 1; i < S.size(); ++i)
    {
        p[i] = p[i - 1];
        while (p[i] > 0 && S[p[i]] != S[i])
            p[i] = p[p[i] - 1];
        if (S[p[i]] == S[i])
            p[i]++;
    }
    return p;
}
```

```
17
18  vi KMP(string &P, string &S)
19  {
20      vector<int> pi = prefix(P);
21      vi matches;
22      int n = S.length(), m = P.length();
23      int j = 0, ans = 0;
24      for (int i = 0; i < n; ++i)
25      {
26          while (j > 0 && S[i] != P[j])
27              j = pi[j - 1];
28          if (S[i] == P[j])
29              ++j;
30
31          if (j == P.length())
32          {
33              /* This is where KMP found a match
34               * we can calculate its position on S by using i - m + 1
35               * or we can simply count it
36               */
37              ans += 1; // count the number of matches
38              matches.eb(i - m + 1); // store the position of those
                    matches
39              // return; we can return on the first match if needed
40              // this must stay the same
41              j = pi[j - 1];
42          }
43      }
44      return matches; // can be modified to return number of matches or
            location
45  }
```