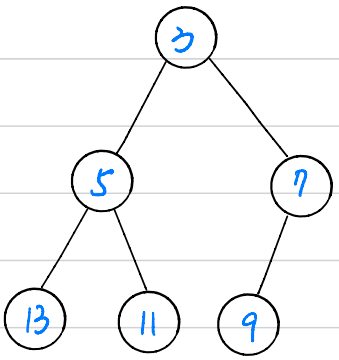
1. Heap and Hash.

① A Max-heap will solve this problem in a snap; however, I decide to render a min-heap instead.



⇒ preorder : 3 . 5 . 13 . 11 . 7 . 9

decreasing.

② Stack : FILO , Priority Queue (STL) : Max-heap

Since one additional member variable is allow, make it into a pair<int, int> with (key, value). Priority queu will compare by the first number of pair, which is key. So keep the element who comes in at last with the largest key value will able to let it leaves first ⇒ LIFO

```
class Stack{
private:
    int key = 0;                          // key as well as the size
    priority_queue < pair < int, int > > heap;

public:
    bool empty(){
        return key == 0;                  // imply no element in priority-queue
    }

    int size(){
        return key;
    }                                     // heap.size() serves the same
```

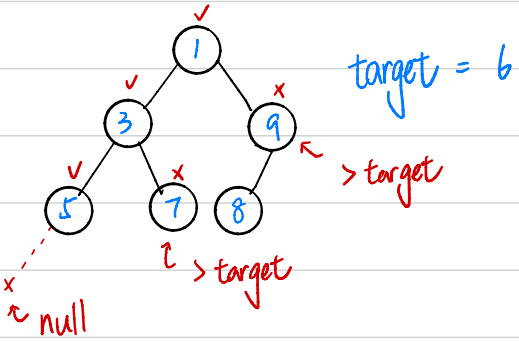
```
    int top(){                            // <key, value>
        return (heap.top()).second;
    }

    void push(int num){                   // <++key, num>
        heap.push(make_pair(++key, num));
    }

    void pop(){
        if(heap.empty())
            cout << "Empty" << endl;
        heap.pop();
        key--;                            // maintain the size of heap.
    }
};
```

⑧ If we want to complete the search in O(k), we have to assume the this heap T is a min-heap or it is impossible.

✓

stop at leaf or key > target

```
Function find(node, target)
    if node is not NULL AND node->key <= target
        print node->key to answer list
        find(node->left, target)
        find(node->right, target)
    end if
End function
```



×

```
Function find(node, target)
    if node is not NULL
        if node->key <= target
            print node->key to answer list
        end if

        find(node->left, target)
        find(node->right)
    end if
End function
```

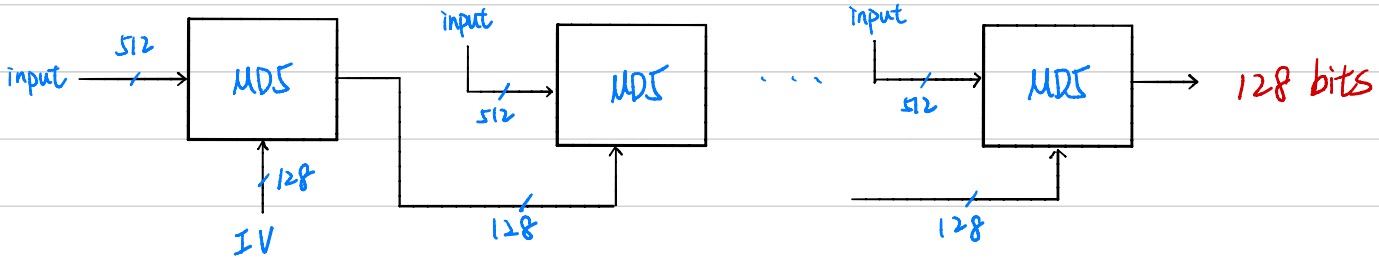
↑ if it's not min-heap then brute-force search.

So using the property of min-heap (parent ≤ its children), we only stop when we hit NULL or when current node > target (so as all the children, grandchildren ... or current node). Thus it is O(k)

⑨ Reference: https://en.wikipedia.org/wiki/MD5
https://www.youtube.com/watch?v=g9uoPPW67gE

MD5 is a kind of hash function. It's initially design to be used as a cryptographic hash function, but it is vulnerable to dictionary attack; Thus, people then used it for other purpose such as checksum. It takes in as much input and convert it into a 128-bit cipher. So any data verification which doesn't involve cryptographic purpose, MD5 will be a convenient method since its small output will save space and fast to verify.

It takes in 512 bits or less from input with 128-bits from initial vector or previous work and produce 128-bits' new cipher till the end.

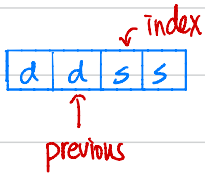


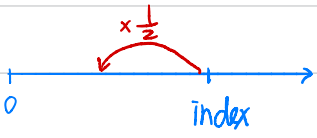
⑤

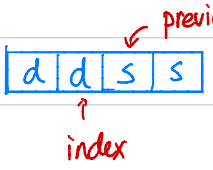
Suppose kth character is different, then the hash value is different for 0 ~ k, same for k+1 ~ len-1. When the hash is same, we leap front $\frac{1}{2}$ while different we leap back $\frac{1}{2}$. So if hash is the same while previous is one step front of current, we know "previous" produce different hash value (leap back) and previous will have different characters; quite the same for "current" as the different one.

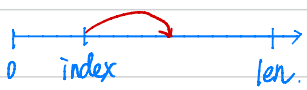
```
Function find_diff(str1, str2)

    len <- length of string - 1
    index <- len / 2   ← set index to middle and previous to 1.
    previous <- -1

    while True; do
        if postfixHash(str1, index) == postfixHash(str2, index)
            if previous = index - 1
                return previous

            previous <- index
            index <- index / 2
        else
            if previous = index + 1
                return index

            previous <- index
            index <- (index + len) / 2
    end while

End function
```

* d: different
  s: same.









6

G[] = {0, 37, 0, 10, 27, 0, 20, 1, 30, 2, 3, 8, 21, 32, 35, 0, 32, 22, 7, 0, 0, 20, 7, 26, 10, 37, 6, 0, 36, 11, 27, 15, 3, 10, 17, 9, 2, 10}

S1[] = {19, 18, 8, 17, 18, 27, 34, 35}
S2[] = {12, 32, 2, 35, 17, 6, 16, 7}

```
Function hash_function(key, T):
    hashed <- 0

    for i in 0 to key.length - 1:
        hashed <- hashed T[i % 8] * ascii(key[i])
    end for

    return hashed % 38
End function

Function compression(key):
    return (G[hash_function(key, S1)] + G[hash_function(key, S2)]) % 38
End function
```

```
albertlin@linxinkaide-MBP HW6 % python3 hash.py
hash the word: auto -> 0
hash the word: bool -> 1
hash the word: case -> 2
hash the word: char -> 3
hash the word: const -> 4
hash the word: continue -> 5
hash the word: default -> 6
hash the word: do -> 7
hash the word: double -> 8
hash the word: else -> 9
hash the word: enum -> 10
hash the word: extern -> 11
hash the word: float -> 12
hash the word: for -> 13
hash the word: goto -> 14
hash the word: if -> 15
hash the word: int -> 16
hash the word: long -> 17
hash the word: register -> 18
hash the word: return -> 19
hash the word: short -> 20
hash the word: signed -> 21
hash the word: sizeof -> 22
hash the word: static -> 23
hash the word: struct -> 24
hash the word: switch -> 25
hash the word: template -> 26
hash the word: union -> 27
hash the word: unsigned -> 28
hash the word: void -> 29
hash the word: volatile -> 30
hash the word: while -> 31
albertlin@linxinkaide-MBP HW6 %
```

The hashing is perfect and minimal since every keyword will only map to one distinct number without any space between the numbers they mapped to. So 32 key words map to 0~31 each (1-1 mapping). There is also no collision with this hash function. So this hash function is perfect as well as minimal. Also, it requires two times the length of string of calculation and with some othe operation. However, since the longest word is 8 in length, it wouldn't take long to compute the hashed value.