Lecture 12: String Algorithms

Date: 09/28/2023

Poll: https://vevox.app/#/m/106148115

Sorting

⇒ We can use the same algorithms we have used in sorting integers,

- Insertion Sort
- Selection Sort
- Merge Sort
- Quick Sort
- Count Sort
- Radix Sort

Problem:

1_integer_selection_sort.cpp

```cpp
void selection_sort(vector<int>& nums){
     int n = nums.size();
     for(int i=0; i<n; i++){

         // finding minimum element
         int minIndex = i;
         for(int j=i+1; j<n; j++){
             if(nums[j] < nums[minIndex]) minIndex = j;
         }

         // put minimum number at it's actual index
         swap(nums[i], nums[minIndex]);
     }
}
```

We need to define our own “<” operator for comparing strings.

⇒ Strings are an under-the-hood array of characters. So, we can compare one character one-by-one.

Comparing Characters

⇒ How do we decide which **character** is smaller than the other?
If all characters we had were only lower case English characters, then the answer will be pretty easy.
Just pick the character that comes before in alphabetical order.

⇒ But, there are so many more characters than just “lower case english characters”.
So, we use the ASCII Table.

```
cook@pop-os:~$ ascii -d
    0 NUL    16 DLE    32        48 0    64 @    80 P    96 `   112 p
    1 SOH    17 DC1    33 !      49 1    65 A    81 Q    97 a   113 q
    2 STX    18 DC2    34 "      50 2    66 B    82 R    98 b   114 r
    3 ETX    19 DC3    35 #      51 3    67 C    83 S    99 c   115 s
    4 EOT    20 DC4    36 $      52 4    68 D    84 T   100 d   116 t
    5 ENQ    21 NAK    37 %      53 5    69 E    85 U   101 e   117 u
    6 ACK    22 SYN    38 &      54 6    70 F    86 V   102 f   118 v
    7 BEL    23 ETB    39 '      55 7    71 G    87 W   103 g   119 w
    8 BS     24 CAN    40 (      56 8    72 H    88 X   104 h   120 x
    9 HT     25 EM     41 )      57 9    73 I    89 Y   105 i   121 y
   10 LF     26 SUB    42 *      58 :    74 J    90 Z   106 j   122 z
   11 VT     27 ESC    43 +      59 ;    75 K    91 [   107 k   123 {
   12 FF     28 FS     44 ,      60 <    76 L    92 \   108 l   124 |
   13 CR     29 GS     45 -      61 =    77 M    93 ]   109 m   125 }
   14 SO     30 RS     46 .      62 >    78 N    94 ^   110 n   126 ~
   15 SI     31 US     47 /      63 ?    79 O    95 _   111 o   127 DEL
```

Reference: https://www.johndcook.com/blog/2022/05/28/how-to-memorize-the-ascii-table/

So, we can use order in the ASCII table to compare two characters.
Which means, now I can compare any characters in the ASCII table.

- ‘E’ < ‘G’
- ‘2’ < ‘a’
- ‘}’ > ‘*’

Comparing Strings

⇒ Just like integers, we need to compare characters starting from Most Significant place to Least Significant place. Because, this way once we get a different character we can stop.
e.g. string1 = “CPSC335”, string2 = “CPSC535”

index	string1 character	string2 character	Comparison
0	C	C	=
1	P	P	=
2	S	S	=
3	C	C	=
4	3	5	<

So, I can iterate through two strings until I find a different character, and return the answer based on that character’s comparison.

⇒ There is one more problem, if these two strings are not the same length, then we might not have two characters to compare at some time.
e.g. string1 = “CPSC335”, string2 = “CPSC”

index	string1 character	string2 character	Comparison
0	C	C	=
1	P	P	=
2	S	S	=
3	C	C	=
4	3		

⇒ So, first we need to check if strings are of the same length.

- If not, than just return the answer based on comparing two lengths

2_string_sort.cpp

```
bool operator<(string s1, string s2){
    // check if two lengths are similar
    if(s1.length() != s2.length())
    return s1.length() < s2.length();

    // iterating through all characters
    // from Most Significant to Least Significant
    int n = s1.length();
    for(int i=0; i<n; i++){
        // return answer if we find different character
        if(s1[i] != s2[i])
        return s1[i] < s2[i];
    }

    return false;
}
```

⇒ Now, after successfully implementing a comparison operator for strings, we can use any of the algorithms to sort an array of strings.

But, keep in mind that the comparison of each element (string) is not taking constant time anymore. So, Time Complexity of algorithms will change.

Algorithm	Time Complexity (Worst Case)
Selection Sort	$\Theta(n^2 * str\_len)$
Insertion Sort	$\Theta(n^2 * str\_len)$
Merge Sort	$\Theta(n\log(n) * str\_len)$
Quick Sort	$\Theta(n^2 * str\_len)$

Count Sort	?
Radix Sort	Θ(n * str_len)

⇒ Radix Sort implementation for strings will be the best choice.

Searching

We can use the same algorithms we used in integer searching,

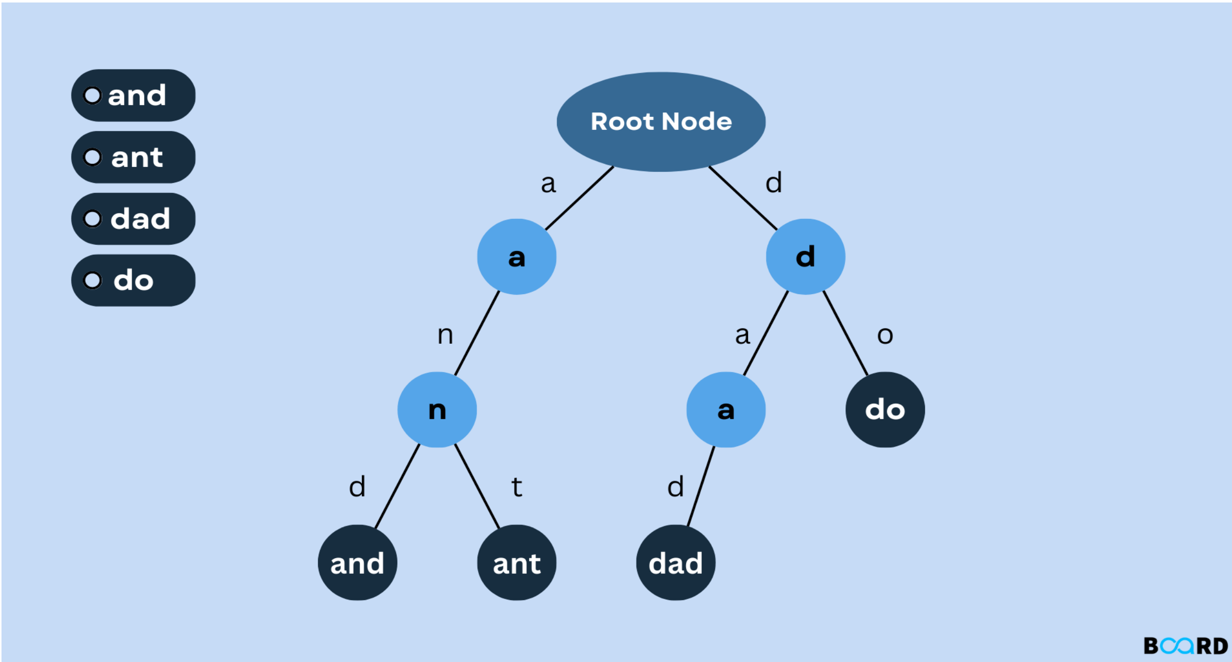
- Linear Search
- Binary Search

But as the comparison of two strings is not constant, time complexity will change.

Algorithm	Time Complexity
Linear Search	Θ(n * str_len)
Binary Search	Θ(log(n) * str_len)

Now, we have another data structure that we can use to make string searching efficient.

Trie



Reference: https://www.boardinfinity.com/blog/trie-data-structure/

We can implement Trie as class,

```
class TrieNode{
public:
    bool isWord;
    map<char, TrieNode*> childs;
};
```

3_trie_search.cpp

```
class Trie{
public:
    TrieNode* root;
    Trie(){
        root = new TrieNode();
    }

    void insert(string s){
        TrieNode* node = root;
        for(int i=0; i<s.length(); i++){
            // if node does not have that children, make a new child
            if(!node->childs[s[i]])
```

```
            node->childs[s[i]] = new TrieNode();

            // otherwise to next child
            node = node->childs[s[i]];
        }

        // make isWord true for the last node only.
        node->isWord = true;
    }

    bool find(string t){
        TrieNode* node = root;
        for(int i=0; i<t.length(); i++){
            // if node does not have that children, we can stop
            if(!node->childs[t[i]])
            return false;
            // otherwise go to next child
            node = node->childs[t[i]];
        }

        // return if last node isWord is true of not
        return node->isWord;
    }
};
```

Time Complexity:

- Insert: Θ(str_len)
- Find: Θ(str_len)

Now, if I am inserting n strings, total time taken by Trie will be,
str1_len + str2_len + ... + strn_len = Θ(n * str_len)
Which is similar to linear search.

⇒ So, why is using Trie better?

Substring Search

Given two strings, find out if the second is a substring of the first.

e.g. string1 = “CPSC335”, string2 = “SC3”
Answer will be **yes**. string2 is a substring of string1, “CP**SC3**35”.

Brute Force

4_brute_force_search.cpp

```
int find(string s, string t){
    int ns = s.length();
    int nt = t.length();

    for(int s_offset=0; s_offset<ns; s_offset++){
        int sameCount = 0;

        // for every offset we need to compare if string matches
        for(int t_ind=0; t_ind<nt; t_ind++){
            // calculate index of s to compare with t's index
            int s_ind = s_offset + t_ind;
            // if s index is out of bound stop the loop
            if(s_ind >= ns) break;
            // calculating how many similar characters are there
            sameCount += (s[s_ind] == t[t_ind]);
        }

        // if whole string matches return the starting offset
        if(sameCount == nt) return s_offset;
    }

    // if string did not match return -1
    return -1;
}
```