CS 32 Week 9 Discussion 1B

this week's topics:

tries, radix trees, hash tables!

TA: **Yiyou Chen** / LA: **Ian Galvez**

a couple things before we start...

- ***First, please find a partner / form a group!*** (Yes, I mean right now.) We'll be doing practice problems throughout the class.
- After that, please fill out the LA feedback form again now that it's the end of the quarter: https://tinyurl.com/S22LAFeedback
  - You can also use the QR code aha >>
  - It really helps me improve as an LA, and improve our section! I want to provide resources that y'all need/want
- Be sure to select the ***Student to LA End of Quarter Feedback*** option!

Today's Topics

Trie

- Insertion
- Query (look up exact match)
- Find all prefix matches

Radix Tree

- Insertion
- Query (look up exact match)
- Find all prefix matches

Hash

- Hash function
- Complexity
- C++ hash: unordered_set, unordered_map

Trie

Trie is a data structure particularly for “dictionary-like” usage that supports looking up words.

Each node saves a “character”, and the path from root to a node saves a prefix.

To know if there is a word ends at current node, one can save an ending flag for each node or add a special ending symbol (e.g. ‘$’, ‘!’, ‘~’...)

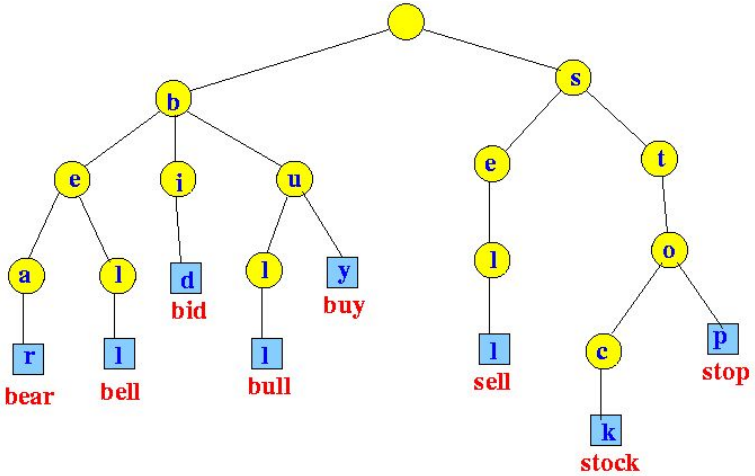


Image: http://www.mathcs.emory.edu

Trie: Nodes

```
//assume lower case letters
struct TrieN {
  TrieN* child[26];
  bool endmark;
  TrieN() {
    for (int i = 0; i < 26; ++i)
      child[i] = nullptr;
    endmark = false;
  }
};

TrieN* root = new TrieN;
```

Doesn't have to use array, can also use vector, list, map, set, etc.

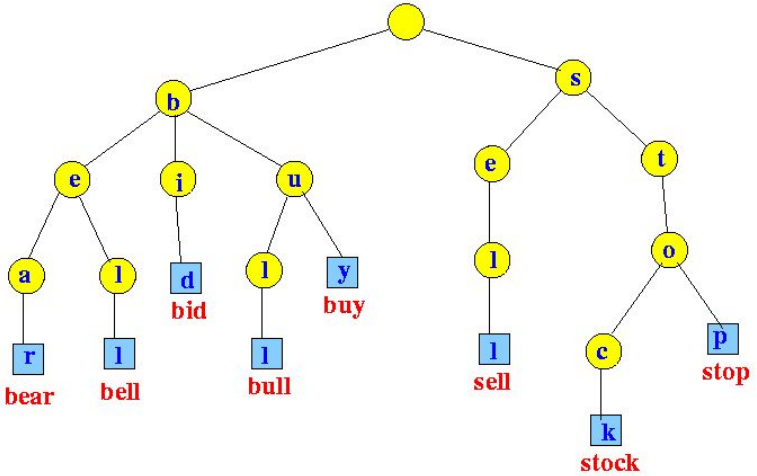


Image: http://www.mathcs.emory.edu

Trie: Insert a word

```
void Trie_Insert(TrieN* root, string s) {

}
```

Trie: Insert a word

```
void Trie_Insert(TrieN* root, string s) {
  if (s.empty()) { //finished insert
    root->endmark = true; //word ends here
    return ;
  }
  int cur_char = s[0] - 'a';
  if (root->child[cur_char] == nullptr)
    //prefix not in dict, create it
    root->child[cur_char] = new TrieN;

  Trie_Insert(root->child[cur_char], s.substr(1));
}
```

Trie: Look up a word (exact match)

```
bool Trie_Query(TrieN* root, string s) {

}
```

Trie: Look up a word (exact match)

```
bool Trie_Query(TrieN* root, string s) {
  if (s.empty()){ //end
    return root->endmark; //true iff endmark=1
  }
  int cur_char = s[0] - 'a';
  if (root->child[cur_char] == nullptr)
    return false;
  return Trie_Query(root->child[cur_char], s.substr(1));
}
```

Trie: Find all prefix matches

Given an input prefix input_s, find all words with input_s as prefix.

```
void Trie_Prefix(TrieN* root, const string& input_s, string s, vector<string>& ret){

}
```

```
vector<string> v;
Trie_Prefix(root, "cbac", "cbac", v);
```

Return v: all words that have “cbac” as their prefix.

Trie: Find all prefix matches

```
void Trie_Prefix_helper(TrieN* root, const string& s, vector<string>& ret) {
  if (root->endmark) ret.push_back(s);
  for (int i = 0; i < 26; ++i) {
    if (root->child[i] != nullptr) {
      //have more matches
      Trie_Prefix_helper(root->child[i], s + (char)('a'+i), ret);
    }
  }
}
void Trie_Prefix(TrieN* root, const string& input_s, string s, vector<string>& ret) {
  if (s.empty()) {
    //return all matchs in subtree root
    Trie_Prefix_helper(root, input_s, ret);
    return ;
  }
  int cur_char = s[0] - 'a';
  if (root->child[cur_char] == nullptr) //not prefix of any word
    return;
  Trie_Prefix(root->child[cur_char], input_s, s.substr(1), ret);
}
```

Radix Tree

A variation of Trie.

We compress the characters of trie when there's just one path.

There can be many ways to implement it,
I'll just show one way.

1 romane
2 romanus
3 romulus
4 rubens
5 ruber
6 rubicon
7 rubicundus

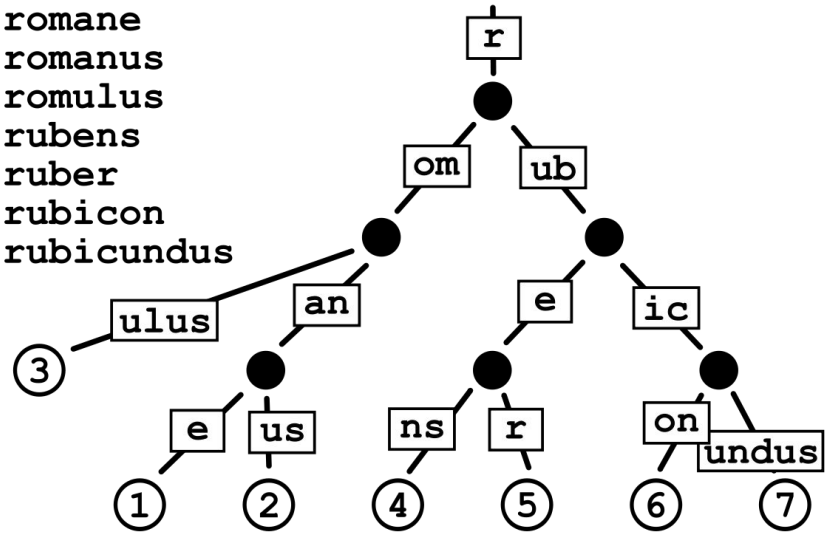


Image: https://en.wikipedia.org/wiki/Radix_tree

Radix Tree

```
struct RadixN {
  bool endmark;
  RadixN* child[26];
  string child_s[26];
  RadixN() {
    for (int i = 0; i < 26; ++i)
      child[i] = nullptr;
  }
};

RadixN* root2 = new RadixN;
```

Doesn't have to use array, can also use vector, list, map, set, etc.

```
1 romane
2 romanus
3 romulus
4 rubens
5 ruber
6 rubicon
7 rubicundus
```

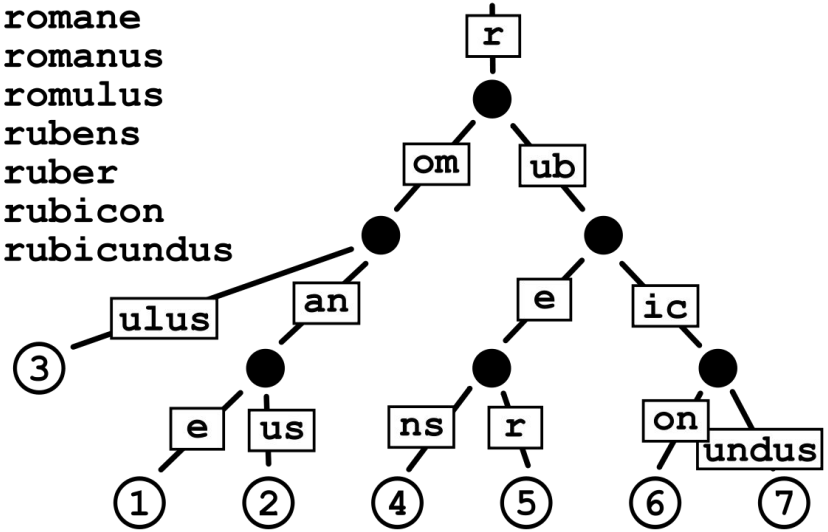


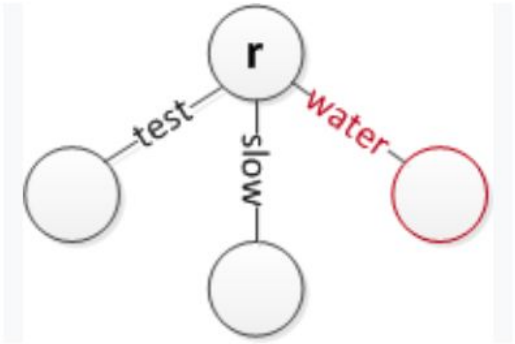
Image: https://en.wikipedia.org/wiki/Radix_tree

Radix Tree: insertion

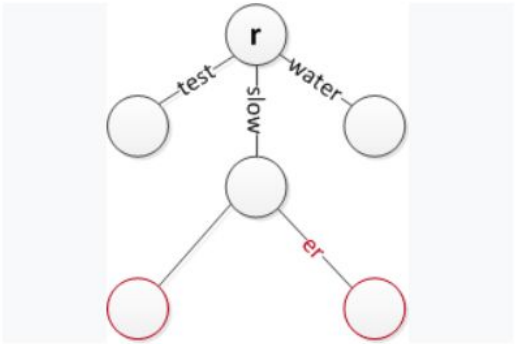
```
void Radix_Insert(RadixN* root, string s) {

}
```

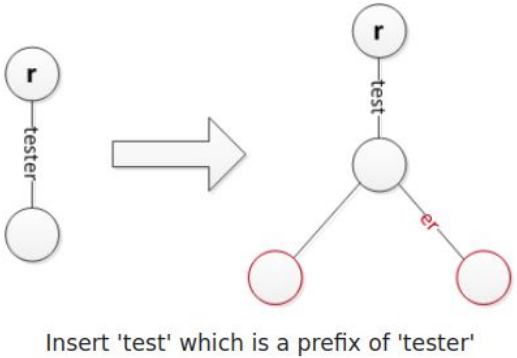
Radix Tree: insertion



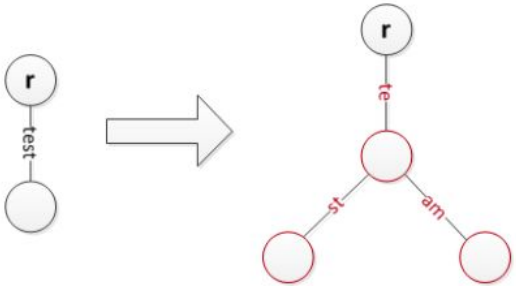
Insert 'water' at the root



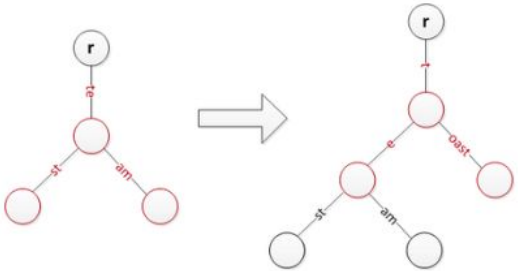
Insert 'slower' while keeping 'slow'



Insert 'test' which is a prefix of 'tester'



Insert 'team' while splitting 'test' and creating a new edge label 'st'



Insert 'toast' while splitting 'te' and moving previous strings a level lower

Radix Tree: insertion

```
134 void Radix_Insert(RadixN* root, string s) {
135   if (s.empty()) { //finishd insertion
136     root->endmark = true;
137     return ;
138   }
139   int cur_char = s[0] - 'a';
140   if (root->child[cur_char] == nullptr) { //insert a new s
141     root->child[cur_char] = new RadixN;
142     root->child_s[cur_char] = s;
143     Radix_Insert(root->child[cur_char], "");
144     return ;
145   }
```

Radix Tree: insertion(continued)

```
147   string transition = root->child_s[cur_char];
148   int match_len = 0;
149   while(match_len < min(transition.length(), s.length()) && transition[match_len] == s[match_len]) //get the matched length
150     ++match_len;
151   if(match_len == transition.length()) { //partial s matched entire transition string
152     Radix_Insert(root->child[cur_char], s.substr(match_len));
153   }
154   else if (match_len == s.length()) { //entire s matched partial transition string
155     RadixN* newnode = new RadixN;
156     newnode->child[transition[match_len] - 'a'] = root->child[cur_char];
157     newnode->child_s[transition[match_len] - 'a'] = transition.substr(match_len);
158     root->child[cur_char] = newnode;
159     root->child_s[cur_char] = s;
160     Radix_Insert(root->child[cur_char], "");
161   }
162   else { //partial s matched partial transition string
163     RadixN* newnode = new RadixN;
164     newnode->child[transition[match_len] - 'a'] = root->child[cur_char];
165     newnode->child_s[transition[match_len] - 'a'] = transition.substr(match_len);
166     root->child[cur_char] = newnode;
167     root->child_s[cur_char] = s.substr(0, match_len);
168     Radix_Insert(root->child[cur_char], s.substr(match_len));
169   }
170 }
```

Radix Tree: look up exact match

```
RadixN* Radix_Query(RadixN* root, string s) {
  //returns a Node pointer to exact match
}
```

Radix Tree: look up exact match

```
RadixN* Radix_Query(RadixN* root, string s) {
  if (s.empty()) { //matches prefix
    return (root->endmark) ? root : nullptr;
  }
  int cur_char = s[0] - 'a';
  //s doesn't match any prefix
  if (root->child[cur_char] == nullptr)
    return nullptr; //prefix not matched
  string transition = root->child_s[cur_char];
  int len = min(transition.length(), s.length());
  if (transition.substr(0, len) != s.substr(0, len)) //mismatch
    return nullptr;
  if (transition.length() == len) //partial s matches entire transition
    return Radix_Query(root->child[cur_char], s.substr(transition.length()));
  else //entire s matches partial transition
    return nullptr;
}
```

Radix Tree: Find all prefix matches

```
void Radix_Prefix(RadixN* root, string input_s, string s, vector<string>& v) {

}
```

Radix_Prefix(root, prefixst, prefixst, v);
Return v: all words that have prefixst as their prefix.

Radix Tree: Find all prefix matches

```
void Radix_Prefix_helper(RadixN* root, string s, vector<string>& v) {
  if (root->endmark) {
    v.push_back(s);
  }
  for (int i = 0; i < 26; ++i) {
    if (root->child[i] != nullptr)
      Radix_Prefix_helper(root->child[i], s+root->child_s[i], v);
  }
}

void Radix_Prefix(RadixN* root, string input_s, string s, vector<string>& v) {
  if (s.empty()) {
    Radix_Prefix_helper(root, input_s, v);
    return ;
  }
  int cur_char = s[0] - 'a';
  if (root->child[cur_char] == nullptr) //not prefix of any word
    return;
  string transition = root->child_s[cur_char];
  int len = min(transition.length(), s.length());
  if (transition.substr(0, len) != s.substr(0, len)) { //prefix not found
    return ;
  }
  if (len == transition.length()) //partial s matches entire transition
    Radix_Prefix(root->child[cur_char], input_s, s.substr(len), v);
  else //entire s matches partial transition
    Radix_Prefix(root->child[cur_char], input_s + transition.substr(len), "", v);
}
```

Hash tables

For look-up.

Given an input key distribution, we can use hash functions to map them to hash table entries.

A good hash function makes the resulting hash table have an approximate uniform distribution.

To evaluate a hash function, think about the distribution it produces for the given data distribution.

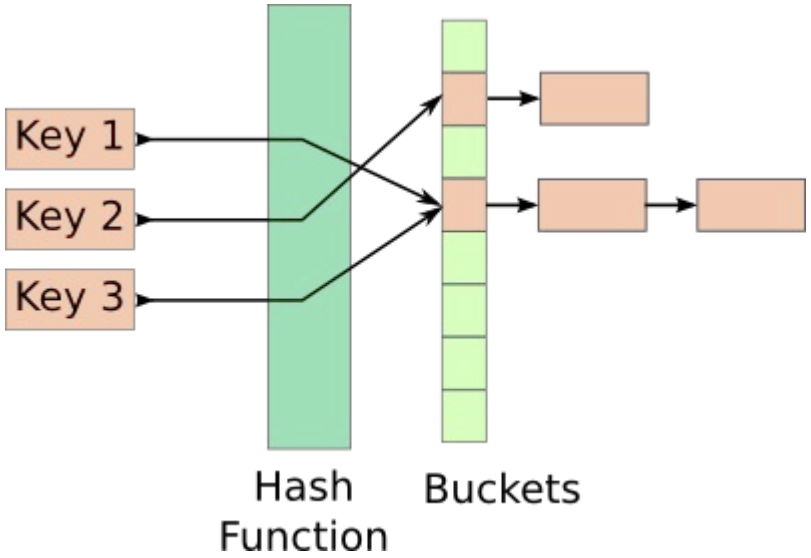


Image: https://vhanda.in/blog

Hash tables: complexity

Load factor = input size N / #buckets

Average case complexity? why?
Worst case complexity? why?

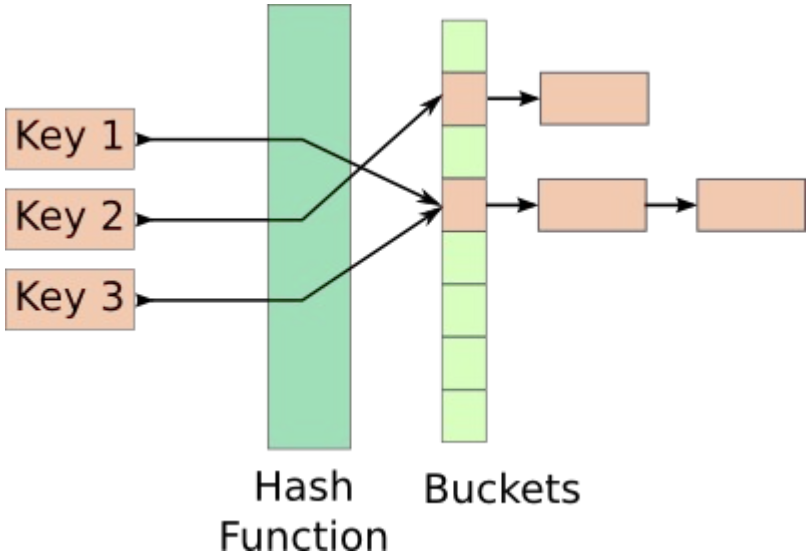


Image: https://vhanda.in/blog

Hash tables: complexity

Load factor = input size N / #buckets

Average case complexity: O(load factor)
Worst case complexity: O(n) linked list. (proof by Pigeonhole principle)

When we have >=O(N) buckets, load factor ~O(1), so the average case time complexity is constant.

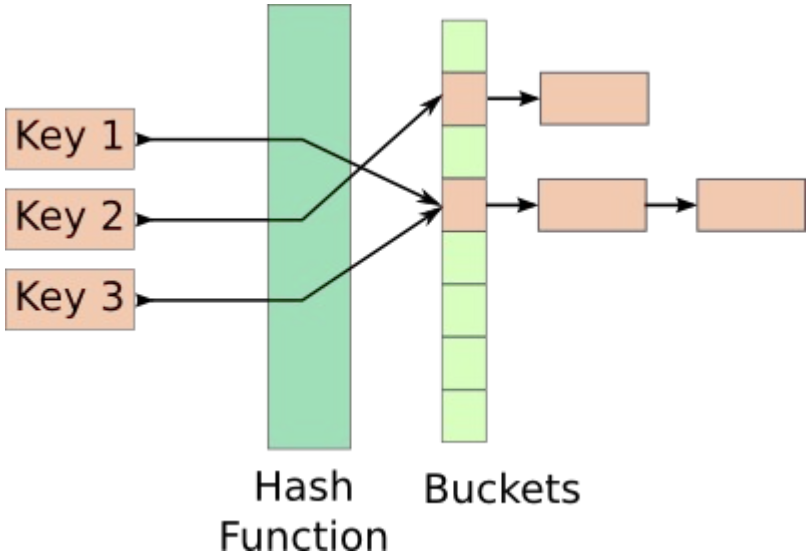


Image: https://vhanda.in/blog

Hash tables: complexity

Load factor = input size N / #buckets

For fixed number of buckets, the load factor(time complexity) grows linearly with input size N.

Therefore, we want to dynamically change #buckets when the input size grows large.

1st idea: create a new hash table with larger(e.g. 2x) number of buckets when load factor reaches a limit, and insert all values from the old hash table to the new one.

Is there any downside of this idea?

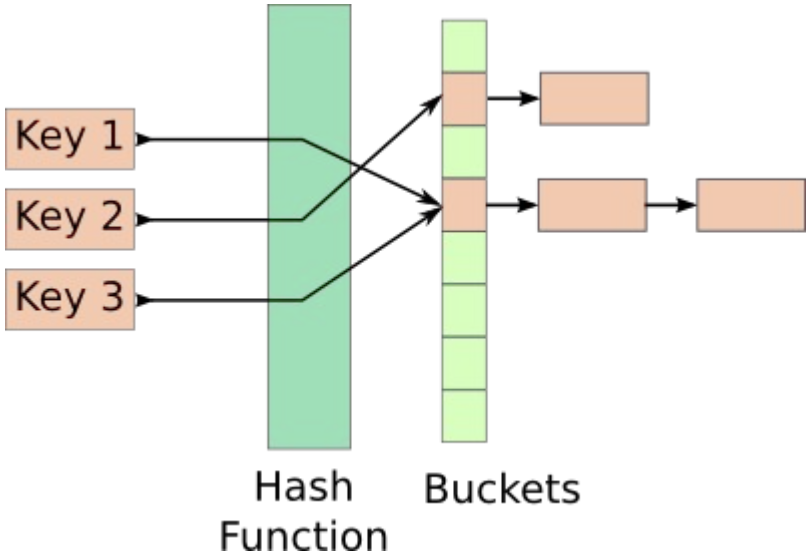


Image: https://vhanda.in/blog

Hash tables: complexity

1st idea: create a new hash table with larger(e.g. 2x) number of buckets when load factor reaches a limit, and insert all values from the old hash table to the new one.

Is there any downside of this idea?

Since the insertion takes O(N) time. The resulting data structure might be inconsistent(unstable) in performance. Once a while, it slows down due to copying.

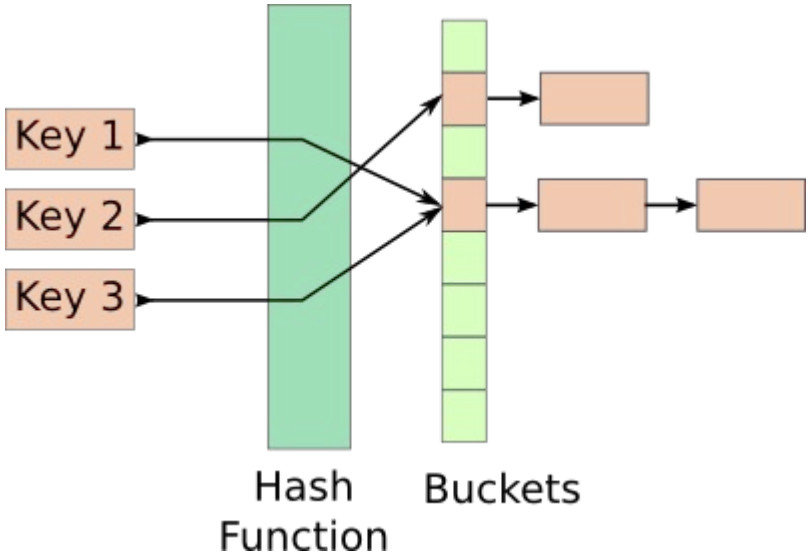


Image: https://vhanda.in/blog

Hash tables: complexity

A better idea: create a new hash table with larger(e.g. 2x) number of buckets when load factor reaches a limit, and insert a constant number of values from the old hash table to the new table upon each new insertion.

Why is this better?

What is the number of values we need to insert from old table to the new table along each new insertion if we x1.5 the size of the buckets when load factor limit is reached?

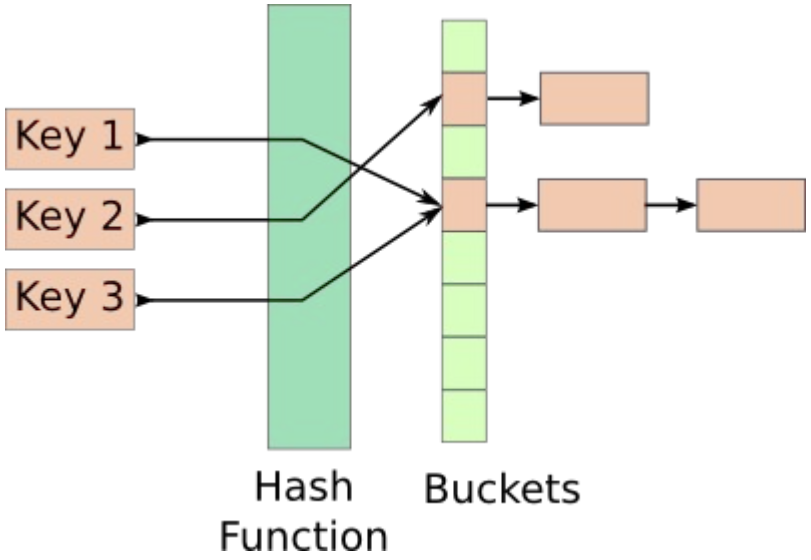


Image: https://vhanda.in/blog

Hash tables: complexity

A better idea: create a new hash table with larger(e.g. 2x) number of buckets when load factor reaches a limit, and insert a constant number of values from the old hash table to the new table along with each new insertion.

Why is this better?
Time complexity is constant throughout.

What is the number of values we need to insert from old table to the new table along each new insertion if we x1.5 the size of the buckets when load factor limit is reached?
2

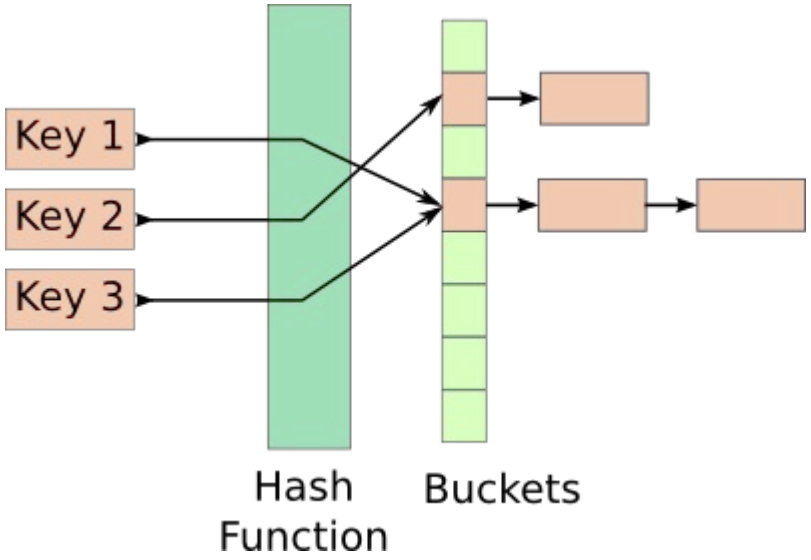


Image: https://vhanda.in/blog

C++ stl: unordered_set, unordered_map

Since both uses hash_table for look up (like a dictionary), both are unsorted, and both remove duplicates. Average Case: O(1). Worst Case: O(n). For nearly all operations.

Methods of unordered_set:

- insert()- Insert a new {element} in the unordered_set container.
- begin()- Return an iterator pointing to the first element in the unordered_set container.
- end()- Returns an iterator pointing to the past-the-end-element.
- count()- Count occurrences of a particular element in an unordered_set container.
- find()- Search for an element in the container.
- clear()- Removes all of the elements from an unordered_set and empties it.
- cbegin()- Return a const_iterator pointing to the first element in the unordered_set container.
- cend()- Return a const_iterator pointing to past-the-end element in the unordered_set container or in one of it's bucket.
- bucket_size()- Returns the total number of elements present in a specific bucket in an unordered_set container.
- erase()- Remove either a single element or a range of elements ranging from start(inclusive) to end(exclusive).
- size()- Return the number of elements in the unordered_set container.
- swap()- Exchange values of two unordered_set containers.
- emplace()- Insert an element in an unordered_set container.
- max_size()- Returns maximum number of elements that an unordered_set container can hold.

Methods of unordered_map :

- at(): This function in C++ unordered_map returns the reference to the value with the element as key k.
- begin(): Returns an iterator pointing to the first element in the container in the unordered_map container
- end(): Returns an iterator pointing to the position past the last element in the container in the unordered_map container
- bucket(): Returns the bucket number where the element with the key k is located in the map.
- bucket_count: bucket_count is used to count the total no. of buckets in the unordered_map. No parameter is required to pass into this function.
- bucket_size: Returns the number of elements in each bucket of the unordered_map.
- count(): Count the number of elements present in an unordered_map with a given key.
- equal_range: Return the bounds of a range that includes all the elements in the container with a key that compares equal to k.
- find(): Returns iterator to element.
- empty(): checks whether container is empty in the unordered_map container.
- erase(): erase elements in the container in the unordered_map container.

Source: geeksforgeeks

C++ stl: #include <unordered_set>

```
//example from geeksforgeeks.org
unordered_set <string> stringSet ;
stringSet.insert("code") ;
stringSet.insert("in") ;
stringSet.insert("c++") ;
stringSet.insert("is") ;
stringSet.insert("fast") ;
string key1 = "slow" ;
if (stringSet.find(key1) == stringSet.end())
    cout << key1 << " not found" << endl;
else
    cout << "Found " << key1 << endl ;
string key2 = "c++" ;
if (stringSet.find(key2) == stringSet.end())
    cout << key2 << " not found" << endl;
else
    cout << "Found " << key2 << endl ;
```

Outputs:

Source: geeksforgeeks

C++ stl: unordered_set

```
//example from geeksforgeeks.org
unordered_set <string> stringSet ;
stringSet.insert("code") ;
stringSet.insert("in") ;
stringSet.insert("c++") ;
stringSet.insert("is") ;
stringSet.insert("fast") ;
string key1 = "slow" ;
if (stringSet.find(key1) == stringSet.end())
    cout << key1 << " not found" << endl;
else
    cout << "Found " << key1 << endl ;
string key2 = "c++" ;
if (stringSet.find(key2) == stringSet.end())
    cout << key2 << " not found" << endl;
else
    cout << "Found " << key2 << endl ;
```

Outputs:
slow not found
Found c++

Source: geeksforgeeks

C++ stl: #include <unordered_map>

```
unordered_map<string, int> umap;
umap["GeeksforGeeks"] = 10;
umap["Practice"] = 20;
umap["Contribute"] = 30;
umap["GeeksforGeeks"] = 20;
umap.insert(pair<string, int>("Practice", 10));
for (auto x : umap)
  cout << x.first << " " << x.second << endl;
```

Outputs:

Source: geeksforgeeks

C++ stl: unordered_map

```
unordered_map<string, int> umap;
umap["GeeksforGeeks"] = 10;
umap["Practice"] = 20;
umap["Contribute"] = 30;
umap["GeeksforGeeks"] = 20;
umap.insert(pair<string, int>("Practice", 10));
for (auto x : umap)
  cout << x.first << " " << x.second << endl;
```

Outputs:
Contribute 30
Practice 20
GeeksforGeeks 20

Source: geeksforgeeks

let’s take a break, then...

worksheet time!

Could You Repeat That?

(taken from Wk 9 LA Worksheet, Problem 1)

Given a string, find the first non-repeating character in it and return its index. If it doesn't exist, return -1. You may assume the string contains only lowercase letters. ***Use a hash table*** to solve this problem.

Examples:

```
Input:  s = "leetcode"
Output: 0

Input:  s = "loveleetcode"
Output: 2
```

```
int firstUniqueChar(std::string s);
```

Solution

(to Could You Repeat That?)

We know our task is to figure out what the first unique character is. How do we keep track of unique characters using a hash table?

What should our key and value be?
Key: character in the string
Value: occurences in the string

Aha! So once we figure out the occurences of each character, we can look through the string and find which character only has one occurence!

To figure out the occurence of each character, we need to walk through the whole string to count the number of times each character occurs.

Then, we walk through the string again to figure out which one of them is unique, if any!

```
int firstUniqueChar(std::string s) {
        // Map character to the frequency of occurrence
        unordered_map<char, int> counter;
        for(int i = 0; i < s.size(); i++) {
            counter[s[i]]++;
        }
        for (int i = 0; i < s.size(); i++) {
            if (counter[s[i]] == 1) return i;
        }
        return -1;
}
```

Target Practice

(taken from Wk 9 LA Worksheet, Problem 2)

Given an array of integers and a target sum, determine if there exists two integers in the array that can be added together to equal the sum. The time complexity of your solution should be ***O(N)***, where ***N*** is the number of elements in the array. In other words, the brute force method of comparing each element with every other element using nested for loops will not satisfy this requirement.

```
Input: arr[] = [4, 8, 3, 7, 9, 2, 5], target = 15
Output: true (8 and 7 add up to 15)
Input: arr[] = [1, 3, 5, 2, 4], target = 10
Output: false (none of these add up to 10)
```

```
bool twoSum(int arr[], int n, int target);
```

Solution

(to Target Practice)

This is pretty easy to do even without hash tables, but the time complexity for the naive solution is **O(N^2)**.

How do we use hash tables in this case? We can check and fill the hash table as we go!

Here, we use the idea of a “complement” to see whether the complement and the element add up to the sum!

Cool variation for practice: ***Problem 9*** from this week’s worksheet! (it’s the same as this, but you have to find *three* numbers that sum to the target.)

```
bool twoSum(const int arr[], int n, int target) {
        unordered_set<int> numsFound;

    // We will add every number to a set as we iterate
    // through the array. If our set ever contains the
    // 'complement' of the number we are looking at, we
    // have found a pair of numbers whose sum is the target
    // and we will return true. Otherwise, if we reach the
    // end of the array, return false.
    for (int i = 0; i < n; i++) {
        int complement = target - arr[i];
        if (numsFound.find(complement) != numsFound.end()) {
            return true;
        }
        else {
            numsFound.insert(arr[i]);
        }
    }
    return false;
}
```

Cinema Iceman

(taken from Wk 9 LA Worksheet, Problem 3)

Given a vector of strings, group anagrams together. Two words are anagrams of each other if one word can be formed by rearranging the letters of the other. For example, ***cinema*** and ***iceman*** are anagrams because ***cinema*** can be rearranged to form the word ***iceman***. (You may assume only lower case letters will be used.)

```
vector<vector<string>> groupAnagrams(vector<string> strs);
```

For example, given:

```
["eat", "tea", "tan", "ate", "nat", "bat"]
```

Return:

```
[ ["ate", "eat","tea"],
  ["nat","tan"],
  ["bat"] ]
```

Hint: use this hash function to the right!

```
// Given a string, compute its hash value based on
// prime numbers
int calculateHash(string word) {
      int primes[26] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
      37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97,
      101};
      int hash_val = 1;
      for(int i = 0; i < word.size(); i++) {
            // Multiplying by prime numbers ensures that all
            // words with the same characters will have the same
            // product
            hash_val *= primes[word[i]-'a'];
      }
      return hash_val;
}
```

Solution

(to Cinema Iceman)

Think about prime factorization! Every natural number has a one-to-one correspondence to a product of some combination of prime numbers

So if we map each letter to a prime number, then any string containing the same letters will have the same hash or “factorization”.

We can use this fact to realize that we can make the **hash value** the key! Then, the **vector of strings** can be our value that we return!

First, we must fill the hash table with the anagrams!

Then, we go through the hash table and push back all the vectors of strings into another vector, which we return.

```
// the hash function gives the same hash value for words that
are different, but have the same characters. Let's use that to
group our anagrams into a bucket of our hashtable.
// anagrams will be a hash table with the hash value as key,
and a vector of the anagrams as the bucket

vector<vector<string>> groupAnagrams(vector<string> strs){
     unordered_map<int, vector<string>> anagrams;

     for(int i = 0; i < strs.size(); i++) {
          int key = calculateHash(strs[i]);
          anagrams[key].push_back(strs[i]);
     }

     unordered_map<int, vector<string>>::iterator it =
          anagrams.begin();
     vector<vector<string>> res;

     // Loop through Hash Table
     while(it != anagrams.end()) {
     // it->second is the vector of strings (i.e. anagrams)
     // corresponding to the one same key
          res.push_back(it->second);
          it++;
     }
     return res;
}
```