CS 32 Week 9
Discussion 2C

UCLA CS

Yiyou Chen / Katie Chang

Topics

Tree

- Level traversal

Trie

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

Radix Tree

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

Hash

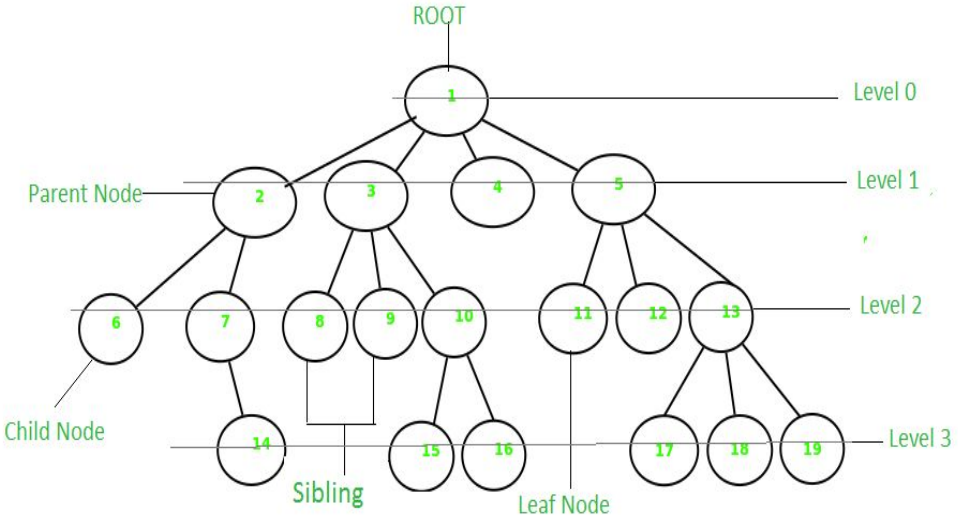
- Hash function
- Complexity
- C++ hash: Unordered_set, unordered_map

Tree: traversal by level

Last time, we discussed traversal by recursion (DFS).
E.g. Preorder: 1 2 6 7 14 3 8 9 … 19

How about traversal by level?

E.g. 1 2 3 4 5 6 7 … 19



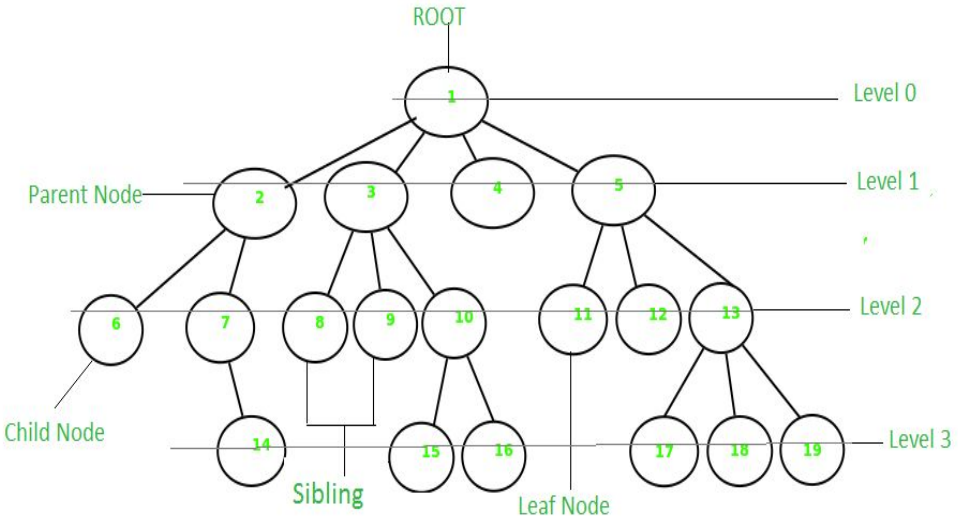
Tree: traversal by level

Last time, we discussed traversal by recursion (DFS).
E.g. Preorder: 1 2 6 7 14 3 8 9 … 19

How about traversal by level?

E.g. 1 2 3 4 5 6 7 … 19

Like on a graph or grid, we can use **breadth-first search (BFS).**



Tree: traversal by level

```
struct Node {
  string val;
  vector <Node*> child;
};
Node* root = nullptr;
```

```
void Traversal_Level(Node* root) {
  if (root == nullptr) return; //empty
  queue<Node*> q;
  q.push(root);
  while(!q.empty()) {
    Node* u = q.front();
    q.pop();
    cout << u->val << endl;
    for (auto it: u->child) {
      q.push(it);
    }
  }
}
```

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. ‘$’, ‘!’, ‘~’...)

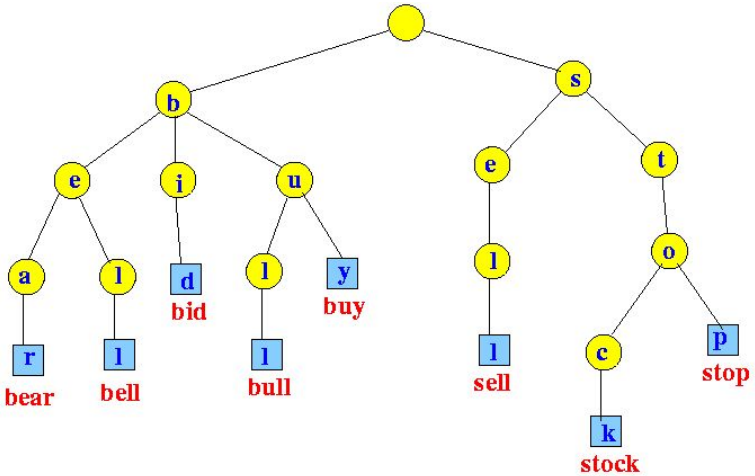


Image: http://www.mathcs.emory.edu

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() {
    for (int i = 0; i < 26; ++i)
      delete child[i];
  }
};

TrieN* root = new TrieN;
```

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

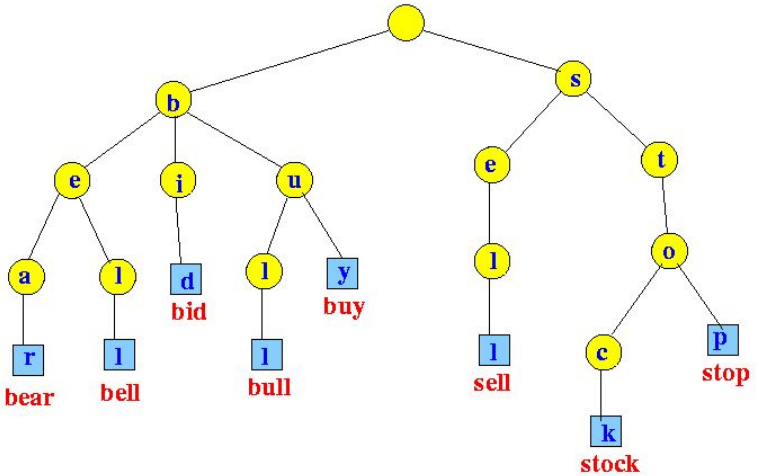


Image: http://www.mathcs.emory.edu

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

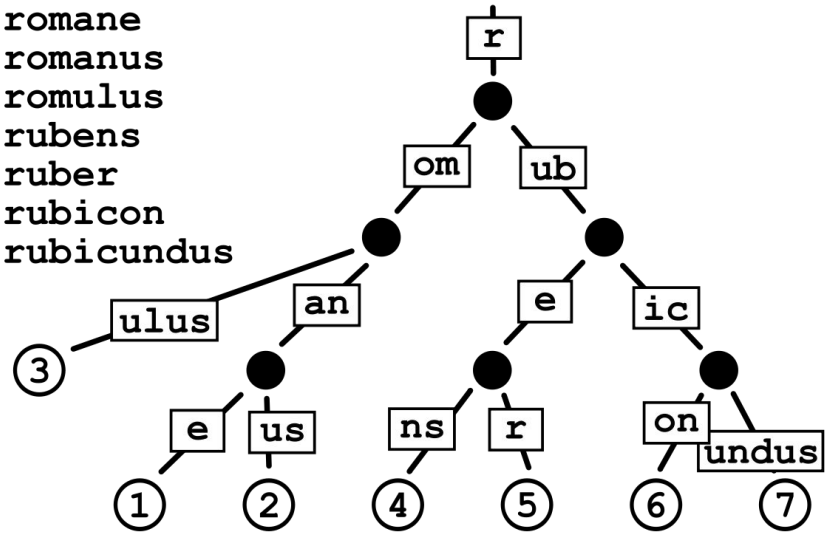


Image: https://en.wikipedia.org/wiki/Radix_tree

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() {
    for (int i = 0; i < 26; ++i)
      delete child[i];
  }
};

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

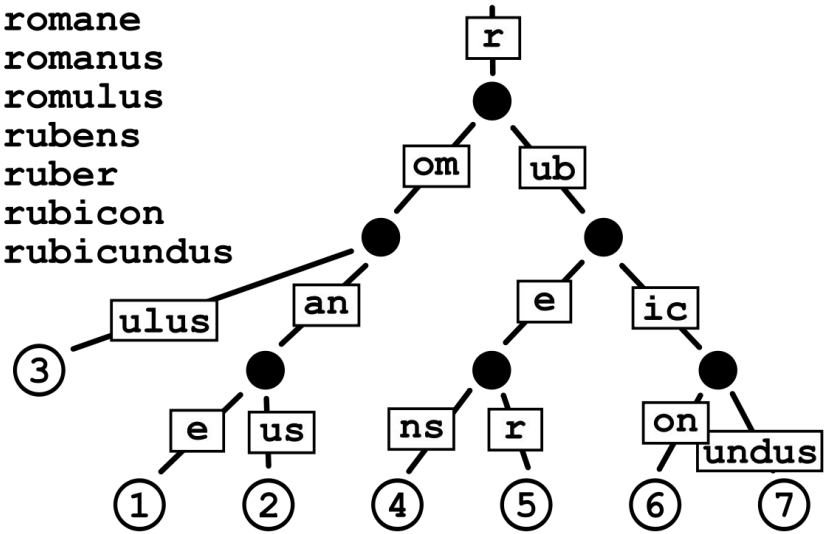


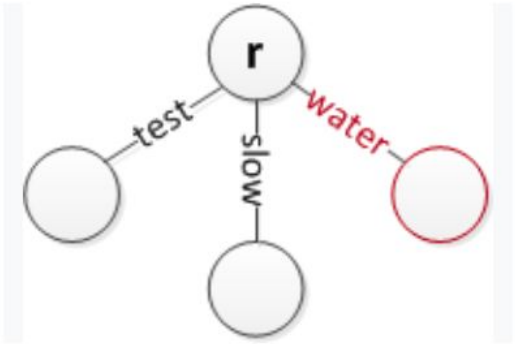
Image: https://en.wikipedia.org/wiki/Radix_tree

Radix Tree: insertion

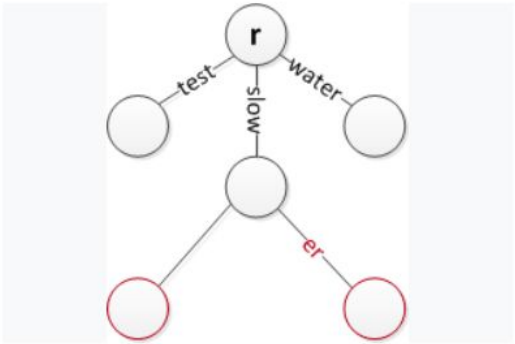
```
void Radix_Insert(RadixN* root, string s) {

}
```

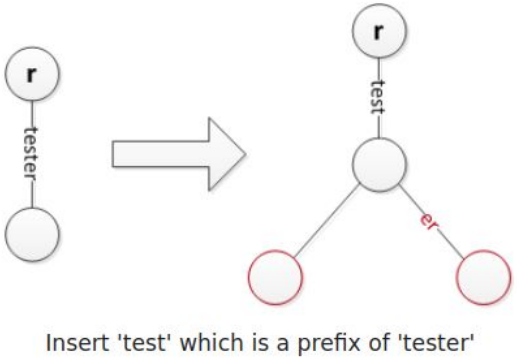
Radix Tree: insertion



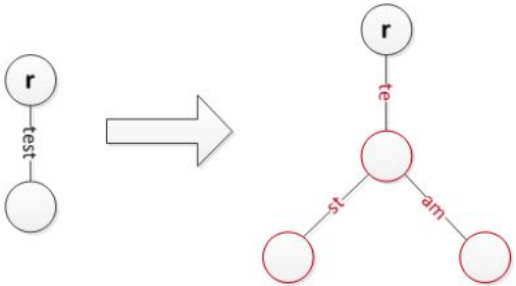
Insert 'water' at the root



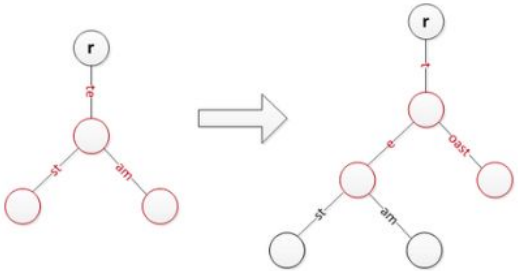
Insert 'slower' while keeping 'slow'



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.

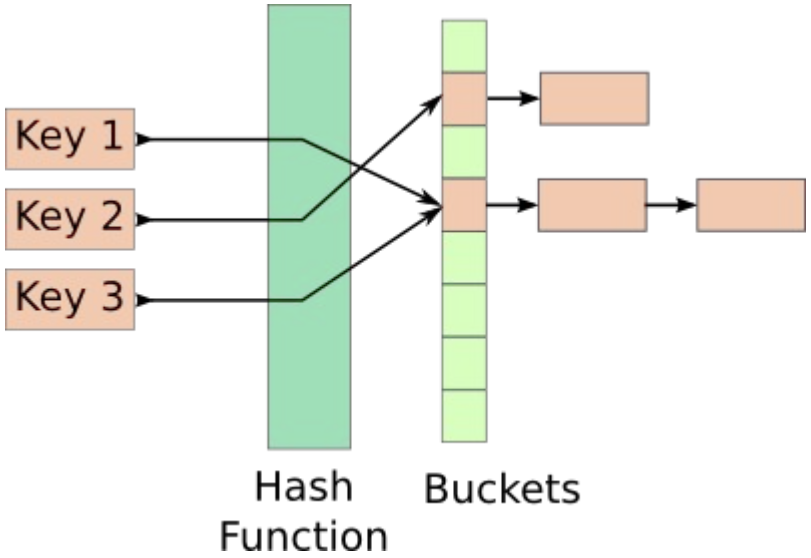


Image: https://vhanda.in/blog

Hash tables: complexity

Load factor = input size N / #buckets

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

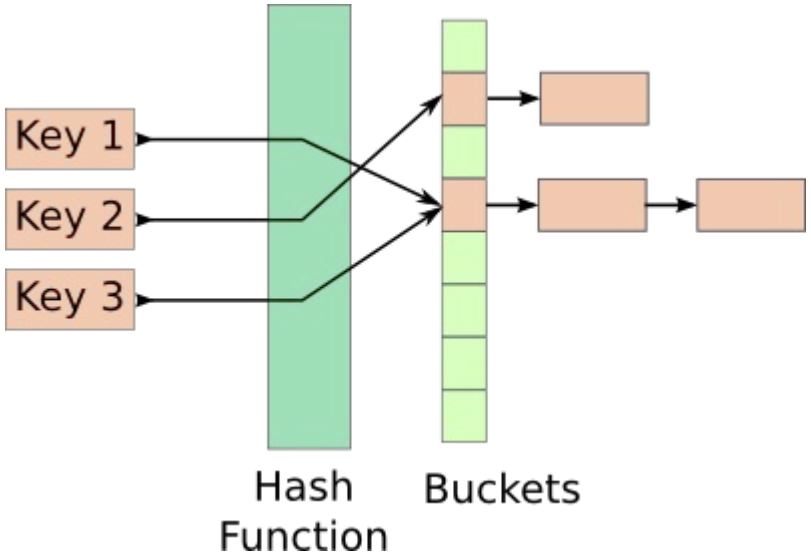


Image: https://vhanda.in/blog

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.

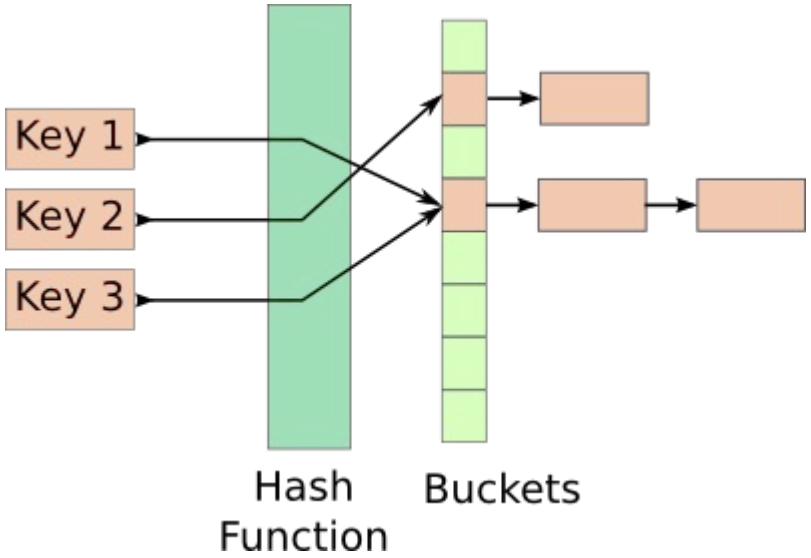


Image: https://vhanda.in/blog

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 can dynamically change #buckets with the input size.

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?

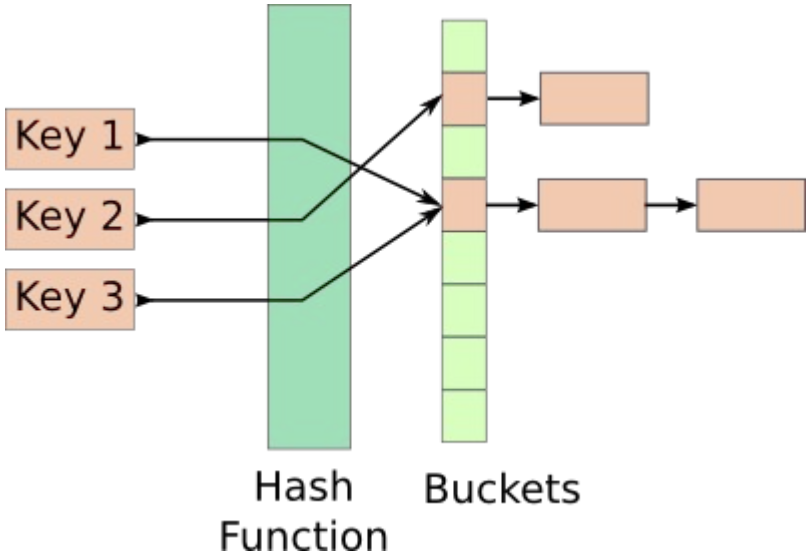


Image: https://vhanda.in/blog

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(lack robustness) in performance. Once a while it slows down due to copying.

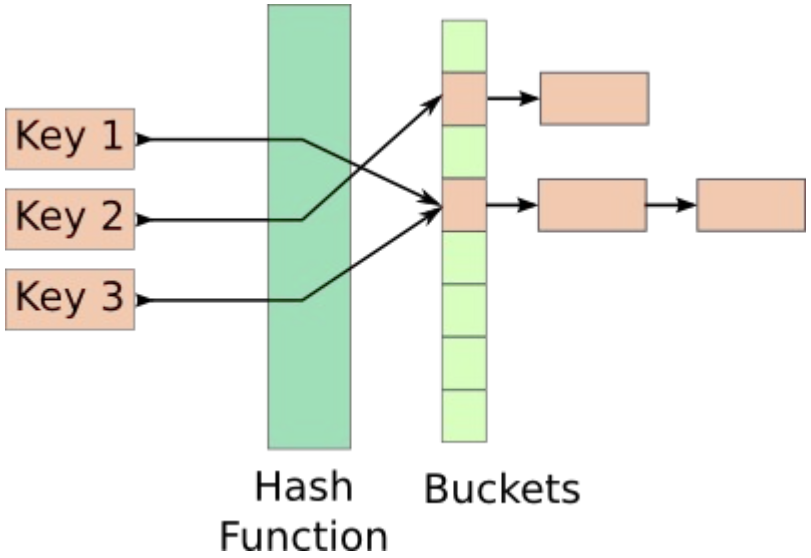


Image: https://vhanda.in/blog

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 along with 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?

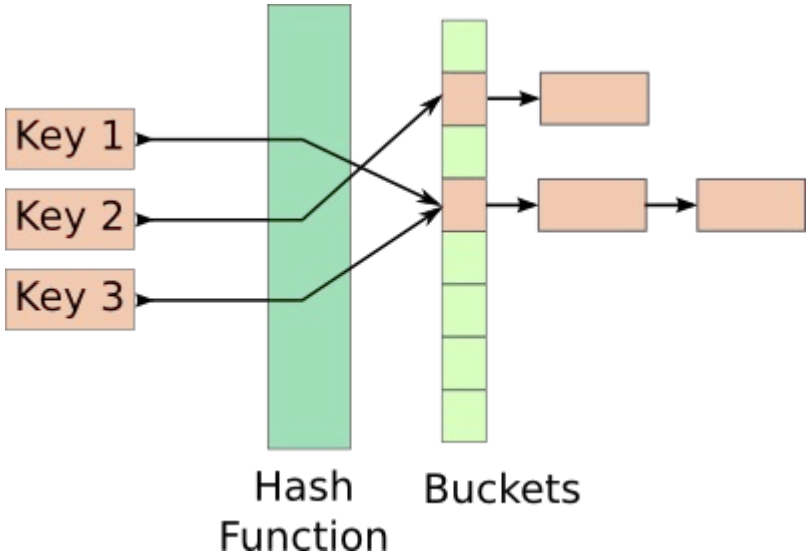


Image: https://vhanda.in/blog

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

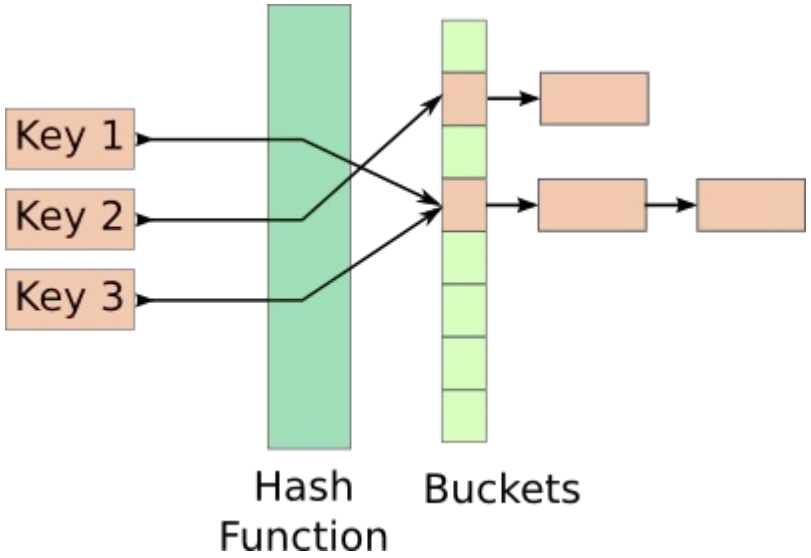


Image: https://vhanda.in/blog

C++ stl: unorderd_set, unorderd_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 <unorderd_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: unorderd_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
C++ not found

Source: geeksforgeeks

C++ stl: #include <unorderd_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: unorderd_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