eecs 281 Data Structures and Algorithms

Discussion 3: Week of Sep 21, 2011

Agenda

- Floating Point Numbers
- Hashing
- Recurrence Relations
- Binary Search Tree

Floating Point Numbers

```
double d1 = 1.1234;
double d2 = 2.1234;
cout << d2 – d1 << endl;
// 1 or 1.0000 ?
```


Floating Point Numbers

```
double d1 = 1.1234;
double d2 = 2.1234;
cout << d2 – d1 << endl;
// outputs 1

printf("%.4f\n",d2 – d1);
// outputs 1.0000
```

Floating Point Numbers

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
 double d1 = 1.1234;
 double d2 = 2.1234;
 cout << setprecision (3);
 cout << d1 << endl;
 cout << d2 << endl;
 cout << setprecision (9) << d1 << endl;
 cout << fixed;
 cout << setprecision (3) << d1 << endl;
 cout << setprecision (9) << d1 << endl;
 return 0;
}
```


Floating Point Numbers

```
if (1 == 1.0000) {
  cout << "Equal" << endl;
}
else {
  cout << "Not equal" << endl;
}
```


Floating Point Numbers

```
if (1 == 1.0000000000000001) {
  cout << "Equal" << endl;
}
else {
  cout << "Not equal" << endl;
}
```


Float Limitation

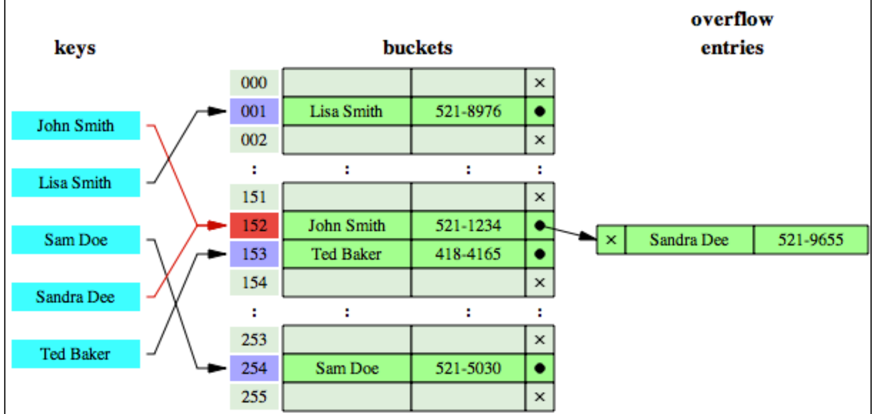
Float value	Hex	Decimal
1.99999976	0x3FFFFFFE	1073741822
1.99999988	0x3FFFFFFF	1073741823
2.00000000	0x4000000	1073741824
2.00000024	0x4000001	1073741825
2.00000048	0x4000002	1073741826

- To store values between 1.99999988 and 2, you need to either use a double or parse the input as characters and use a type that has enough bits to fit

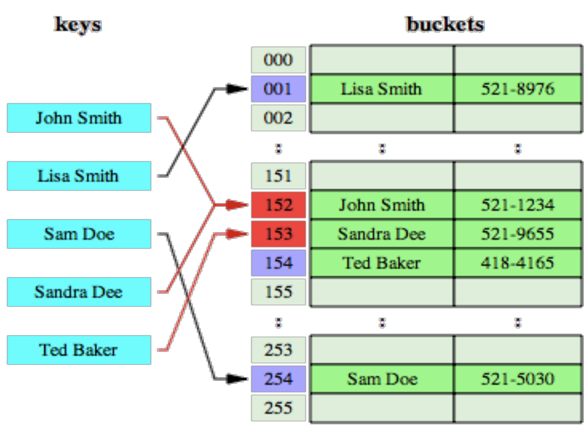
Hashing

- Associative container
  - No concept of “previous/next”
  - Insert/delete/lookup in O(1) time
- Hash a key into index, store the value into hash_map[index]
- Collision resolution
  - Separate chaining
  - Probing/open addressing
    - linear
    - quadratic

Separate Chaining



Open addressing



Quiz for Hashing

- Given n integers, find the one with highest frequency. Return the smallest one if there’s a tie.

Quiz for Hashing

- Given n integers, find the one with highest frequency. Return the smallest one if there's a tie.
- 1. Count the frequency for each integer using hash table
- 2. Key is the number, value is the frequency
- 3. Scan the hash table to return the smallest with highest freq

Quiz for Hashing

- Given n integers, find the one with highest frequency. Return the smallest one if there's a tie.
- 1. Count the frequency for each integer using hash table
- 2. Key is the number, value is the frequency
- 3. Scan the hash table to return the smallest with highest freq
- Complexity? Size of hash table = N

Quiz for Hashing

- Given n integers, find the one with highest frequency. Return the smallest one if there's a tie.
- 1. Count the frequency for each integer using hash table
- 2. Key is the number, value is the frequency
- 3. Scan the hash table to return the smallest with highest freq
- Complexity = O(n + N) Size of hash table = N

Quiz for Hashing

- Given n integers, find the one with highest frequency. Return the smallest one if there's a tie.
- 1. Count the frequency for each integer using hash table
- 2. Key is the number, value is the frequency
- 3. Scan the hash table to return the smallest with highest freq
- Complexity = O(n + N) Size of hash table = N
- What's our hash function?

Quiz for Hashing

- Given n integers, find the one with highest frequency. Return the smallest one if there's a tie.
- 1. Count the frequency for each integer using hash table
- 2. Key is the number, value is the frequency
- 3. Scan the hash table to return the smallest with highest freq
- Complexity = O(n + N) Size of hash table = N
- What's our hash function?
  - int hash(int value) { return value; }

Quiz for Hashing

- Given n integers, find the one with highest frequency. Return the smallest one if there's a tie.
- 1. Count the frequency for each integer using hash table
- 2. Key is the number, value is the frequency
- 3. Scan the hash table to return the smallest with highest freq
- Complexity = O(n + N) Size of hash table = N
- What's our hash function?
  - int hash(int value) { return value; }
  - Does not work for hash(-1)

Quiz for Hashing

- Given n integers, find the one with highest frequency. Return the smallest one if there's a tie.
- 1. Count the frequency for each integer using hash table
- 2. Key is the number, value is the frequency
- 3. Scan the hash table to return the smallest with highest freq
- Complexity = O(n + N) Size of hash table = N
- What's our hash function?
  - int hash(int value) { return value; }
  - Does not work for hash(-1)
  - How about { return abs(value); } ?

Quiz for Hashing

- Given n integers, find the one with highest frequency. Return the smallest one if there's a tie.
- 1. Count the frequency for each integer using hash table
- 2. Key is the number, value is the frequency
- 3. Scan the hash table to return the smallest with highest freq
- Complexity = O(n + N) Size of hash table = N
- What's our hash function?
  - int hash(int value) { return value; }
  - Does not work for hash(-1)
  - How about { return abs(value); } ?

Quiz for Hashing

- Given n integers, find the one with highest frequency. Return the smallest one if there's a tie.
- 1. Count the frequency for each integer using hash table
- 2. Key is the number, value is the frequency
- 3. Scan the hash table to return the smallest with highest freq
- Complexity = O(n + N) Size of hash table = N
- What's our hash function?
  - int hash(int value) { return value; }
  - Does not work for hash(-1)
  - How about { return abs(value); } ?
  - Does not work for (-2^31), can't store 2^31 (> INT_MAX)

Quiz for Hashing

- Given n integers, find the one with highest frequency. Return the smallest one if there's a tie.
- 1. Count the frequency for each integer using hash table
- 2. Key is the number, value is the frequency
- 3. Scan the hash table to return the smallest with highest freq
- Complexity = O(n + N) Size of hash table = N
- What's our hash function?
  - int hash(int value) { return value; }
  - Does not work for hash(-1)
  - How about { return abs(value); } ?
  - Does not work for (-2^31), can't store 2^31 (> INT_MAX)
  - More importantly, input distribution decides output distribution

Quiz for Hashing

- Given n integers, find the one with highest frequency. Return the smallest one if there's a tie.
- 1. Count the frequency for each integer using hash table
- 2. Key is the number, value is the frequency
- 3. Scan the hash table to return the smallest with highest freq
- Complexity = O(n + N) Size of hash table = N
- What's our hash function?
  - int hash(int value) { return value; }
  - Does not work for hash(-1)
  - How about { return abs(value); } ?
  - Does not work for (-2^31), can't store 2^31 (> INT_MAX)
  - More importantly, input distribution decides output distribution
  - { return abs(value % LARGE_PRIME); }

Recurrence Relations

- Usually used to analyze algorithm runtimes.

- Many algorithms loop on a problem set, do something each time, and create smaller sub-problems to solve. This is the idea behind a recurrence relation.

- To solve them – think!
  - How is this problem set changing each time? Decreasing exponentially? Variably? Constantly?
  - How many new sub-problems are being created each time?

Recurrence Relations

- Recurrence relations are those that are defined in terms of themselves.

$S(0) = 0$
$S(n) = n + S(n - 1)$

- What's the closed form of S(n)?

Recurrence Relations

- Recurrence relations are those that are defined in terms of themselves.

$S(0) = 0$
$S(n) = n + S(n - 1)$

- What's the closed form of S(n)?

- $S(n) = n + (n-1) + \dots + 1 + 0$
- $S(n) = n * (n - 1)/2$

Master Theorem

Suppose

$T(n) = a * T(n/c) + f(n)$

where a>=1, c>1 and n/c means either $\lceil n/c \rceil$ or $\lfloor n/c \rfloor$

Then

- 1) If $f(n)=O(n^\wedge(\log_c a-\varepsilon))$ for some $\varepsilon>0$, then $T(n)=\Theta(n^\wedge\log_c a)$
- 2) If $f(n)=\Theta(n^\wedge\log_c a)$, then $T(n)=\Theta((n^\wedge\log_c a)\log n)$
- 3) If $f(n)=\Omega(n^\wedge(\log_c a+\varepsilon))$ for some $\varepsilon>0$ and if $a*f(n/c)\leq kf(n)$ for some constant k<1 and all sufficiently large n, then $T(n)=\Theta(f(n))$

Recurrence Relations

- How would we express the Fibonacci sequence as a recurrence relation?

- What about the tribonacci sequence?

Recurrence Relations

- How would we express the Fibonacci sequence as a recurrence relation?

F(0)=0, F(1)=1, F(n)=F(n-1) + F(n-2)

- What about the tribonacci sequence?

T(0)=1, T(1)=1, T(2)=2, T(n)=T(n-1)+T(n-2)+T(n-3)

Recursive Functions

- We can use the following method to define a function with the **natural numbers** as its domain:

1. Specify the value of the function at zero.
2. Give a rule for finding its value at any integer from its values at smaller integers.

- Such a definition is called **recursive**.

Recursive Functions

- **Example:**

- f(0) = 3
- f(n + 1) = 2f(n) + 3

- f(0) = 3
- f(1) = 2f(0) + 3 = 2⋅3 + 3 = 9
- f(2) = 2f(1) + 3 = 2⋅9 + 3 = 21
- f(3) = 2f(2) + 3 = 2⋅21 + 3 = 45
- f(4) = 2f(3) + 3 = 2⋅45 + 3 = 93

Recursive Functions

- **How can we recursively define the factorial function f(n) = n! ?**

- f(0) = 1
- f(n + 1) = (n + 1)f(n)

- f(0) = 1
- f(1) = 1f(0) = 1•1 = 1
- f(2) = 2f(1) = 2•1 = 2
- f(3) = 3f(2) = 3•2 = 6
- f(4) = 4f(3) = 4•6 = 24

Recursive Function

Iterative version of factorial function

Function does NOT calls itself

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n=0 \\ n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1 & \text{if } n>0 \end{cases}$$

Iteration vs. recursion

- Some things (e.g. reading from a file) are easier to implement iteratively
- Other things (e.g. mergesort) are easier to implement recursively
- Others are just as easy both ways
- When there is no real benefit to the programmer to choose recursion, iteration is the more efficient choice
- It can be proved that two methods performing the same task, one implementing an iteration algorithm and one implementing a recursive version, are equivalent

BST

- Binary Tree
  - Every node has 0, 1, 2 children
- Full Binary Tree
  - Every node other than leaves has 2 children
  - All leave nodes have same path length
  - Also called proper binary tree, strictly binary tree
- Complete Binary Tree
  - Every level above the last level is completely filled
  - Nodes in the last level are as far left as possible
- Binary Search Tree
  - Ordered binary tree

BST

```
Node {
    Key;
    Value;
    Node* left;
    Node* right;
}
```

all keys in left subtree < current key < all keys in right subtree

BST

	Average Case	Worst Case
Search		
Insert		
Delete		

BST

	Average Case	Worst Case
Search	O(log n)	O(n)
Insert	O(log n)	O(n)
Delete	O(log n)	O(n)

Inserting a node to BST

```
void insert(node* &root, key, value)
{
 if (root == NULL) {
   root = new node(key,value);
 }
 else if (key < root->key) {
   insert(root->left,key,value);
 }
 else {
   insert(root->right,key,value);
 }
}
```


Inserting a node to BST

```
void insert(node* &root, key, value)
{
 if (root == NULL) {
   root = new node(key,value);
 }
 else if (key < root->key) {
   insert(root->left,key,value);
 }
 else {
   insert(root->right,key,value);
 }
}
```

Can you write it as a non-recursion?

Inserting a node to BST

```
void insert(node* &root, key, value)
{
  if (root == NULL) {
     root = new node(key, value);
     return;
  }
  node* cur = root;
  while (true) {
     if (key < cur->key) {
          if (cur->left == NULL) {
               cur->left = new node(key, value);
               break;
          }
          else
               cur = cur->left;
     }
     else {
          if (cur->right == NULL) {
               cur->right = new node(key, value);
               break;
          }
          else
               cur = cur->right;
     }
  }
}
```


Delete a node from BST

- **Deleting a leaf (node with no children):** Deleting a leaf is easy, as we can simply remove it from the tree.
- **Deleting a node with one child:** Remove the node and replace it with its child.
- **Deleting a node with two children:** Call the node to be deleted *N*. Do not delete *N*. Instead, choose either its in-order successor node or its predecessor node, R. Replace the value of N with the value of R and then delete R. (Why R cannot have more than 2 children?)