CS24 - Problem Solving with Computers II

BST Review and Runtime Analysis

BST (Review)

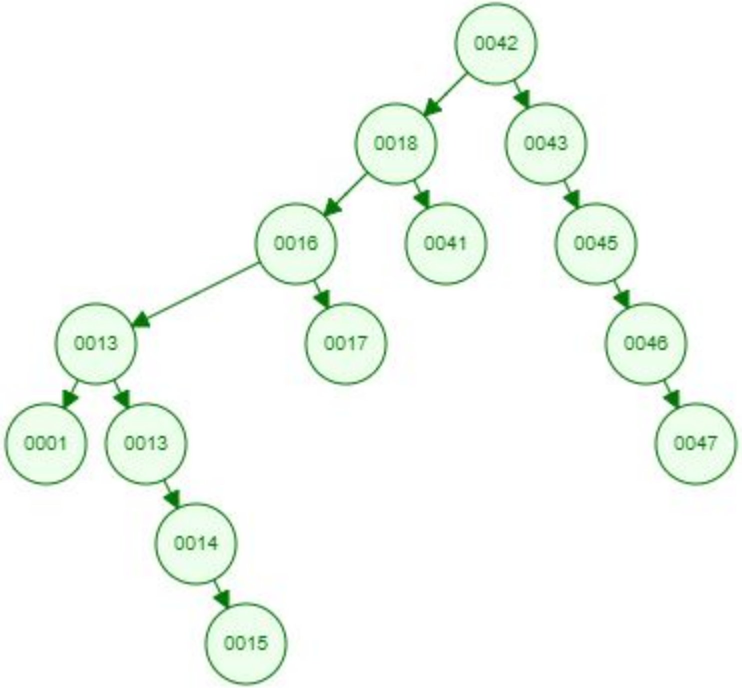
42, 18, 16, 43, 45, 46, 47, 13, 1, 17, 13, 14, 15, 41

Note: 2x 13s... In this case,

if(a<b) insertLeft();

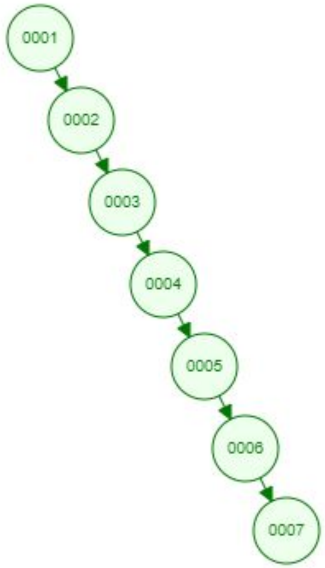
else insertRight();

Other option: Add a “Count”



BST (Review)

1, 2, 3, 4, 5, 6, 7, 8, 9 10, 11



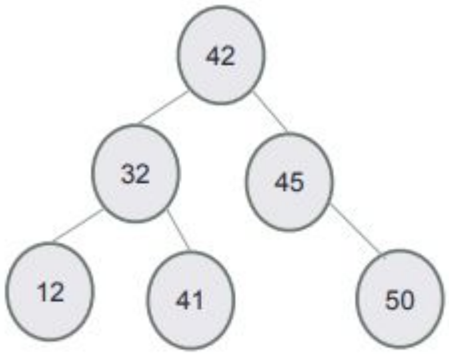
BST (Review)

42, 32, 45, 50, 12, 41

42, 45, 32, 50, 12, 41

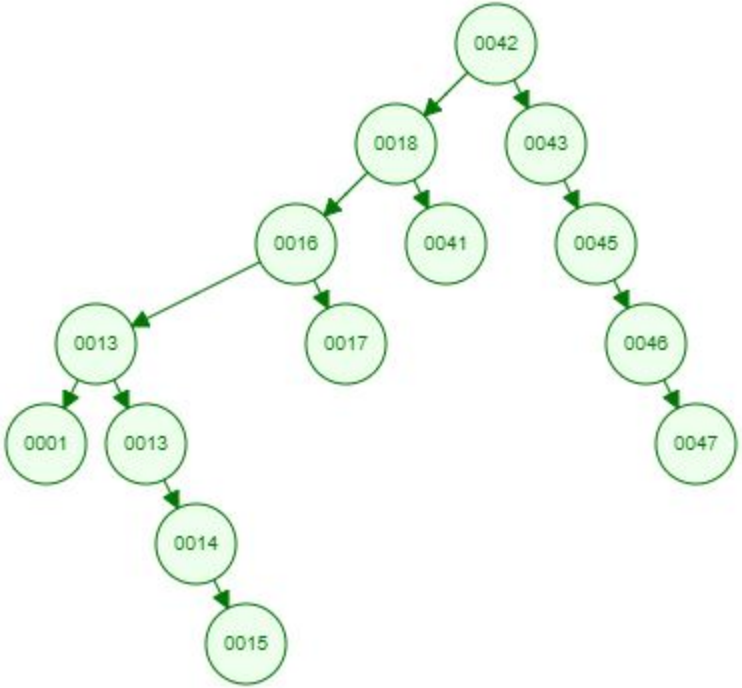
42, 32, 12, 45, 50, 41

...



BST (Review)

Binary search trees allow efficient searching...



Runtime Analysis

Computer scientists need to optimize code to run as efficiently as possible. This is extremely important for coding interviews (and in real life).

Amount of memory required/transferred and execution time need balanced depending on program requirements (sending/receiving JPEGs)

Choosing the correct data structure for the task is usually the most important step

We can manually time how long a program takes, but this isn’t necessarily useful...

Runtime Analysis

```cpp
#include <iostream>
#include <ctime>
using namespace std;

int main() {
    time_t start, end;
    start = time(0);
    int count = 0;

    for(int i=0; i<999999; i++){
        count++;
    }

    end = time(0);
    cout << "It took " << difftime(end, start) << " seconds to complete " << count << " loops." << endl;

    return 0;
}
```

Runtime Analysis

```
#include <iostream>
#include <ctime>
using namespace std;

int main() {
    time_t start, end;
    start = time(0);
    int count = 0;

    for(int i=0; i<999999; i++){
        count++;
    }

    end = time(0);
    cout << "It took " << difftime(end, start) << " seconds to complete " << count << " loops." << endl;

    return 0;
}
```

```
It took 0 seconds to complete 999999 loops.
```

Runtime Analysis

```
#include <iostream>
#include <ctime>
using namespace std;

int main() {
    time_t start, end;
    start = time( _Time: 0);
    int count = 0;

    for(int i=0; i<999999; i++){
        count++;
    }

    end = time( _Time: 0);
    cout << "It took " << difftime(end, start) << " seconds to complete " << count << " loops." << endl;

    return 0;
}
```

```
It took 0 seconds to complete 999999 loops.
It took 1 seconds to complete 999999999 loops.
```

Runtime Analysis

Not very useful to measure time (or even clocks) - dependent on hardware, background processes, etc.

Instead, we want to consider what happens when the number of times the program executes grows. In other words, what happens when the input size gets bigger?

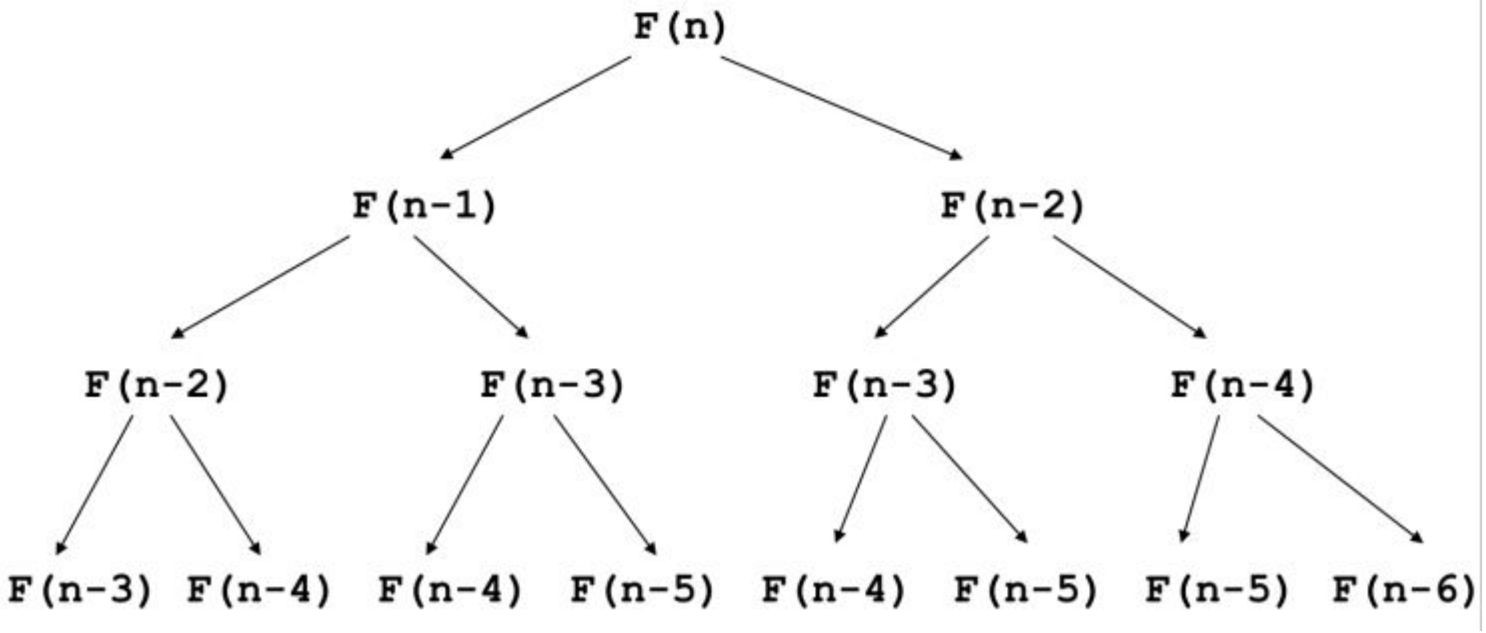
This is done by counting the number of basic (constant time) operations being performed (setting a value, performing arithmetic, comparing values, accessing an array element, pushing/popping, delete, etc)

Runtime Analysis

```
long FibSequenceForLoop(int numberIterations){
    long fibArray[numberIterations];
    fibArray[0] = 1;
    fibArray[1] = 1;
    for(int i=2; i<numberIterations; i++){
        fibArray[i] = fibArray[i-1] + fibArray[i-2];
    }
    return fibArray[numberIterations -1];
}

long FibSequenceRecursive(int numIter){
    if (numIter == 1 || numIter == 2)
        return 1;
    return FibSequenceRecursive( numIter: numIter -1) + FibSequenceRecursive( numIter: numIter -2);
}
```

Runtime Analysis



Runtime Analysis

The actual number of basic operations for each step is ignored, doesn't matter in the big picture...

```cpp
#include <iostream>
using namespace std;

int numOps = 0; // Keeps track of the number of operations

void MyFunction(){
    int n = 100; // One operation
    numOps++;

    for(int i=3; i<n; i++){ // Runs n-3 times
        int x = 1; // One operation
        x++; // One operation
        numOps+=2;
    }numOps+=4; //... setting i and incrementing 3 times = 4 operations
    // Total number of operations = 1+4 (operations) + [n-3 (loops) * 2 (operations)] = 2n - 1 operations
}

int main() {
    int m = 100;   // One operation
    numOps++;
    for(int i=0; i<m; i+=2){ //loop runs m/2 times
        MyFunction();
    }numOps+=(1+m/2); // ... setting i and incrementing m/2 times

    // Total = 1 + m/2(2n-1) + 1 + m/2 = [mn + 2]
    cout << numOps << endl;
}
```

Runtime Analysis

The actual number of basic operations for each step is ignored, doesn't matter in the big picture…

m=10

```
1002
```

m=100

```cpp
#include <iostream>
using namespace std;

int numOps = 0; // Keeps track of the number of operations

void MyFunction(){
    int n = 100; // One operation
    numOps++;

    for(int i=3; i<n; i++){ // Runs n-3 times
        int x = 1; // One operation
        x++; // One operation
        numOps+=2;
    }numOps+=4; //... setting i and incrementing 3 times = 4 operations
    // Total number of operations = 1+4 (operations) + [n-3 (loops) * 2 (operations)] = 2n - 1 operations
}

int main() {
    int m = 100;   // One operation
    numOps++;
    for(int i=0; i<m; i+=2){ //loop runs m/2 times
        MyFunction();
    }numOps+=(1+m/2); // ... setting i and incrementing m/2 times

    // Total = 1 + m/2(2n-1) + 1 + m/2 = [mn + 2]
    cout << numOps << endl;
}
```

Runtime Analysis

The actual number of basic operations for each step is ignored, doesn't matter in the big picture...

m=10

1002

m=100

10002

```cpp
#include <iostream>
using namespace std;

int numOps = 0; // Keeps track of the number of operations

void MyFunction(){
    int n = 100; // One operation
    numOps++;

    for(int i=3; i<n; i++){ // Runs n-3 times
        int x = 1; // One operation
        x++; // One operation
        numOps+=2;
    }numOps+=4; //... setting i and incrementing 3 times = 4 operations
    // Total number of operations = 1+4 (operations) + [n-3 (loops) * 2 (operations)] = 2n - 1 operations
}

int main() {
    int m = 100;   // One operation
    numOps++;
    for(int i=0; i<m; i+=2){ //loop runs m/2 times
        MyFunction();
    }numOps+=(1+m/2); // ... setting i and incrementing m/2 times

    // Total = 1 + m/2(2n-1) + 1 + m/2 = [mn + 2]
    cout << numOps << endl;
}
```

Runtime Analysis

What if we change the number of iterations to something non-linear?

```
int main() {
    int m = 30;
    numOps++;
    for(int i=0; i<(pow( x: 2,m)); i+=2){ //loop runs (2^m)/2 times
    MyFunction();
    }numOps+=(1+m/2);

    cout << numOps << endl;
}
```

Runtime Analysis

What if we change the number of iterations to something non-linear?

m=10

```
101895
```

```
int main() {
    int m = 30;
    numOps++;
    for(int i=0; i<(pow( x: 2,m)); i+=2){ //loop runs (2^m)/2 times
    MyFunction();
    }numOps+=(1+m/2);

    cout << numOps << endl;
}
```

Runtime Analysis

What if we change the number of iterations to something non-linear?

m=10

```
101895
```

m=20?

```
int main() {
    int m = 30;
    numOps++;
    for(int i=0; i<(pow( x: 2,m)); i+=2){ //loop runs (2^m)/2 times
    MyFunction();
    }numOps+=(1+m/2);

    cout << numOps << endl;
}
```

Runtime Analysis

What if we change the number of iterations to something non-linear?

m=10

```
101895
```

m=20

```
104333324
```

```
int main() {
    int m = 30;
    numOps++;
    for(int i=0; i<(pow( x: 2,m)); i+=2){ //loop runs (2^m)/2 times
    MyFunction();
    }numOps+=(1+m/2);

    cout << numOps << endl;
}
```

Runtime Analysis

What if we change the number of iterations to something non-linear?

m=10

```
101895
```

m=20

```
104333324
```

m=30?

```
int main() {
    int m = 30;
    numOps++;
    for(int i=0; i<(pow( x: 2,m)); i+=2){ //loop runs (2^m)/2 times
    MyFunction();
    }numOps+=(1+m/2);

    cout << numOps << endl;
}
```

Runtime Analysis

What if we change the number of iterations to something non-linear?

m=10

`101895`

m=20

`104333324`

m=30?

```
int main() {
    int m = 30;
    numOps++;
    for(int i=0; i<(pow( x: 2,m)); i+=2){ //loop runs (2^m)/2 times
    MyFunction();
    }numOps+=(1+m/2);

    cout << numOps << endl;
}
```

Runtime Analysis

What if we change the number of iterations to something non-linear?

m=10

```
101895
```

m=20

```
104333324
```

m=30?

```
106837311505
```

```
int main() {
    int m = 30;
    numOps++;
    for(int i=0; i<(pow( x: 2,m)); i+=2){ //loop runs (2^m)/2 times
    MyFunction();
    }numOps+=(1+m/2);

    cout << numOps << endl;
}
```

Runtime Analysis

- Let us consider that an operation can be executed in 1 ns (10^{-9} s).

	Time		
Function	$(n = 10^3)$	$(n = 10^4)$	$(n = 10^5)$
$\log_2 n$	10 ns	13.3 ns	16.6 ns
$\sqrt{n}$	31.6 ns	100 ns	316 ns
n	1 μs	10 μs	100 μs
$n \log_2 n$	10 μs	133 μs	1.7 ms
n^2	1 ms	100 ms	10 s
n^3	1 s	16.7 min	11.6 days
n^4	16.7 min	116 days	3171 yr
2^n	$3.4 \cdot 10^{284}$ yr	$6.3 \cdot 10^{2993}$ yr	$3.2 \cdot 10^{30086}$ yr

- Jordi Cortadella Dept. of Computer Science, UPC

Runtime Analysis

- Let us consider that an operation can be executed in 1 ns (10^{-9} s).

	Time		
Function	$(n = 10^3)$	$(n = 10^4)$	$(n = 10^5)$
$\log_2 n$	10 ns	13.3 ns	16.6 ns
$\sqrt{n}$	31.6 ns	100 ns	316 ns
n	1 μs	10 μs	100 μs
$n \log_2 n$	10 μs	133 μs	1.7 ms
n^2	1 ms	100 ms	10 s
n^3	1 s	16.7 min	11.6 days
n^4	16.7 min	116 days	3171 yr
2^n	$3.4 \cdot 10^{284}$ yr	$6.3 \cdot 10^{2993}$ yr	$3.2 \cdot 10^{30086}$ yr

Note: $\log_2$

- Jordi Cortadella Dept. of Computer Science, UPC

Runtime Analysis

Because of all of this, computer scientists describe time complexity in something known as “Big O” notation. This notation simplifies complexity into its highest order of magnitude for each type of input.

For example,

“mn + 1” would simply be O(mn)

“n^2 + 3n +4” would be O(n^2)

“2^n + n^2 +99999999n” would be O(2^n)

Runtime Analysis

The formal definition of Big-O:

$f = O(g)$ if there is a constant $c > 0$ and $k>0$ such that $f(n) \le c * g(n)$ for all $n >= k$

(maximum runtime of f is $c * g(n)$)

Runtime Analysis

The formal definition of Big-O:

f = O(g) if there is a constant c > 0 an

(maximum runtime of f is c * g(n))

	=2*(A1^3)		=A1^3+(99999*A1)	
10	2,000	14,000	1,000,990	1,006,990
20	16,000	38,000	2,007,980	1,018,990
30	54,000	74,000	3,026,970	1,036,990
40	128,000	122,000	4,063,960	1,060,990
50	250,000	182,000	5,124,950	1,090,990
60	432,000	254,000	6,215,940	1,126,990
70	686,000	338,000	7,342,930	1,168,990
80	1,024,000	434,000	8,511,920	1,216,990
90	1,458,000	542,000	9,728,910	1,270,990
100	2,000,000	662,000	10,999,900	1,330,990
110	2,662,000	794,000	12,330,890	1,396,990
120	3,456,000	938,000	13,727,880	1,468,990
130	4,394,000	1,094,000	15,196,870	1,546,990
140	5,488,000	1,262,000	16,743,860	1,630,990
150	6,750,000	1,442,000	18,374,850	1,720,990
160	8,192,000	1,634,000	20,095,840	1,816,990
170	9,826,000	1,838,000	21,912,830	1,918,990
180	11,664,000	2,054,000	23,831,820	2,026,990
190	13,718,000	2,282,000	25,858,810	2,140,990
200	16,000,000	2,522,000	27,999,800	2,260,990
210	18,522,000	2,774,000	30,260,790	2,386,990
220	21,296,000	3,038,000	32,647,780	2,518,990
230	24,334,000	3,314,000	35,166,770	2,656,990
240	27,648,000	3,602,000	37,823,760	2,800,990
250	31,250,000	3,902,000	40,624,750	2,950,990
260	35,152,000	4,214,000	43,575,740	3,106,990

Runtime Analysis

The formal definition of Big-O:

f = O(g) if there is a constant c > 0 and k>0 such that f(n) ≤ c * g(n) for all n >= k

(maximum runtime of f is c * g(n))

Big-Omega(Ω)

y f = Ω(g) if there are constants c > 0, k>0 such that c * g(n) ≤ f(n) for n >= k

(minimum runtime of f is c * g(n))

Runtime Analysis

The formal definition of Big-O:

$f = O(g)$ if there is a constant $c > 0$ and $k>0$ such that $f(n) \le c * g(n)$ for all $n >= k$

(maximum runtime of f is $c * g(n)$)

Big-Omega(Ω)

y $f = \Omega(g)$ if there are constants $c > 0$, $k>0$ such that $c * g(n) \le f(n)$ for $n >= k$

(minimum runtime of f is $c * g(n)$)

Big-Theta(Θ)

y $f = \Theta(g)$ if there are constants c1, c2 , k such that $0 \le c1g(n) \le f(n) \le c2g(n)$, for $n >=k$

(f is bound by $c1 * g(n)$ and $c2 * g(n)$

Runtime Analysis

The formal definition of Big-O:

$f = O(g)$ if there is a constant $c > 0$ and $k>0$ such that $f(n) \le c * g(n)$ for all $n >= k$

(maximum runtime of f is $c * g(n)$)

Big-Omega(Ω)

y $f = \Omega(g)$ if there are constants $c > 0$, $k>0$ such that $c * g(n) \le f(n)$ for $n >= k$

(minimum runtime of f is $c * g(n)$)

Big-Theta(Θ)

y $f = \Theta(g)$ if there are constants $c1$, $c2$, k such that $0 \le c1g(n) \le f(n) \le c2g(n)$, for $n >=k$

(f is bound by $c1 * g(n)$ and $c2 * g(n)$

Runtime Analysis of Data Structures

What about a normal linked list?

[3, 8, 1, 4, 9, 12, 18, 6]

Min?
Max?
Search?

Runtime Analysis of Data Structures

What about a normal linked list?

[3, 8, 1, 4, 9, 12, 18, 6]

Min?
Max?
Search?

Must traverse the whole array!

Runtime Analysis of Data Structures

What about a normal linked list?

[3, 8, 1, 4, 9, 12, 18, 6]

Min?
Max?
Search?

Must traverse the whole array!

But what about…
Insert?

Runtime Analysis of Data Structures

What about a normal linked list?

[3, 8, 1, 4, 9, 12, 18, 6]

Min?
Max?
Search?

Must traverse the whole array!

But what about...
Insert?

We can change tail to point to a new node in one step

Runtime Analysis of Data Structures

What about a sorted linked list?

[1, 3, 4, 6, 8, 9, 12, 18]

Min?
Max?

Runtime Analysis of Data Structures

What about a sorted linked list?

[1, 3, 4, 6, 8, 9, 12, 18]

Min?
Max?

Can be done in one step by grabbing the head or tail node

Runtime Analysis of Data Structures

What about a sorted linked list?

[1, 3, 4, 6, 8, 9, 12, 18]

Min?
Max?

Can be done in one step by grabbing the head or tail node

Search, Insert?

Runtime Analysis of Data Structures

What about a sorted linked list?

[1, 3, 4, 6, 8, 9, 12, 18]

Min?
Max?

Can be done in one step by grabbing the head or tail node

Search, Insert?

Best case, you're inserting or searching for the minimum value...

Runtime Analysis of Data Structures

What about a sorted linked list?

[1, 3, 4, 6, 8, 9, 12, 18]

Min?
Max?

Can be done in one step by grabbing the head or tail node

Search, Insert?

Best case, you're inserting or searching for the minimum value…
Worst case, you're inserting or searching for the maximum value

Runtime Analysis of Data Structures

What about a sorted linked list?

[1, 3, 4, 6, 8, 9, 12, 18]

Min?
Max?

Can be done in one step by grabbing the head or tail node

Search, Insert?

Best case, you're inserting or searching for the minimum value…
Worst case, you're inserting or searching for the maximum value

How about an array?

Runtime Analysis of Data Structures

What about a sorted linked list?

[1, 3, 4, 6, 8, 9, 12, 18]

Min?
Max?

Can be done in one step by grabbing the head or tail node

Search, Insert?

Best case, you're inserting or searching for the minimum value…
Worst case, you're inserting or searching for the maximum value

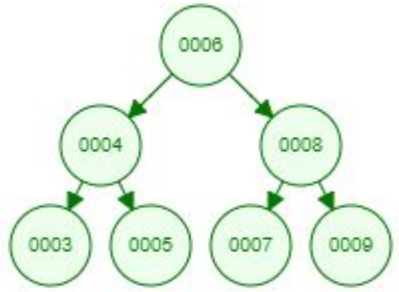
How about an array?

Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Runtime Analysis of Data Structures

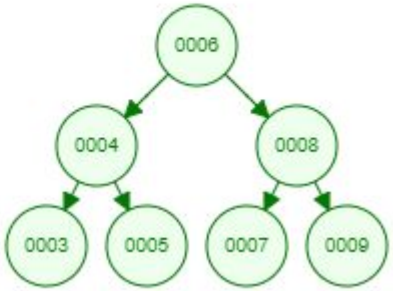
Binary Search Tree… Worst case O(n)?



Runtime Analysis of Data Structures

Binary Search Tree… Worst case O(n)?

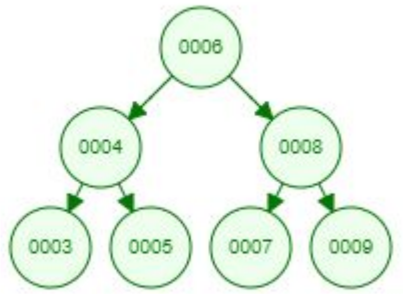
But what if we changed the order that we inputted our numbers?



Runtime Analysis of Data Structures

Binary Search Tree… Worst case O(n)?

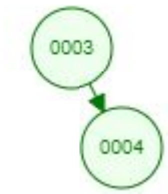
But what if we changed the order that we inputted our numbers?

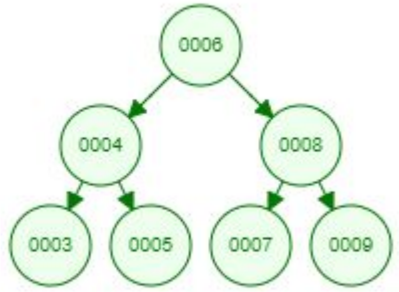


Runtime Analysis of Data Structures

Binary Search Tree… Worst case O(n)?

But what if we changed the order that we inputted our numbers?

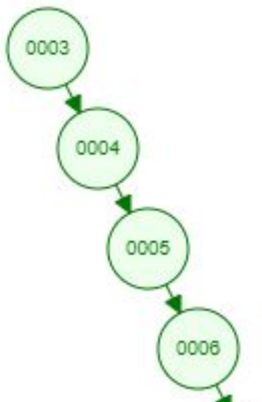


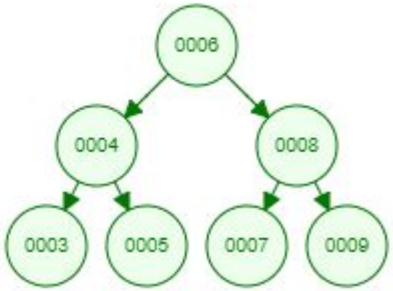


Runtime Analysis of Data Structures

Binary Search Tree… Worst case O(n)?

But what if we changed the order that we inputted our numbers?





Runtime Analysis of Data Structures

There is a better way… Balanced trees

Runtime Analysis of Data Structures

There is a better way… Balanced trees

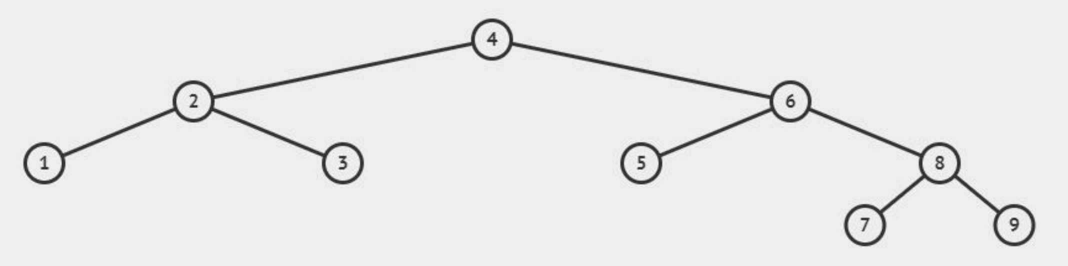
An AVL tree is an example, self balances if one branch becomes too long

Runtime Analysis of Data Structures

Runtime Analysis of Data Structures

Why O(log(n)) ?

Search?



$$\log_b y = x$$

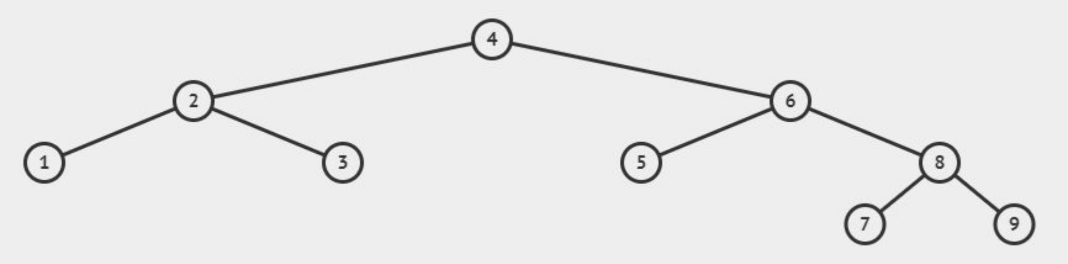
Runtime Analysis of Data Structures

Why O(log(n)) ?

Search?

Insert?

Delete?



$\log_b y = x$ $\quad$ $b^x = y$

Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Runtime Analysis

Practice problems on Gauchospace (with explanations)

This will be on Quiz 4 and the final!!

Next time...

Stacks/Heaps!