Lecture 7

Binary Search Trees and Red-Black Trees
Announcements

• Homework 3 is due today.
• Homework 4 is out today. From HW4 onwards you are allowed pair submissions (but solo is OK too).
• Midterm approaching: Thu, Feb 15 (6pm – 9pm)
• Midterm covers up to (and incl.) lecture 7 – today
Roadmap

- Sorting
  - Asymptotic Analysis
  - Randomized Algorithms
  - Recurrences

- Data Structures
  - Randomized Algorithms
  - Dynamic Programming

- Graphs!
  - Longest, Shortest, Max and Min...
  - The Future!

1 lecture

2 lectures

5 lectures

10 lectures

1st class

Divide and conquer

The Future!

Midterm

More detailed schedule on the website!

We are here
But first!

- A brief wrap-up of divide and conquer.
How do we design divide-and-conquer algorithms?

- So far we’ve seen lots of examples.
  - Karatsuba
  - MergeSort
  - Select
  - QuickSort
  - Polynomial Multiplication (HW1)
  - Dog Safety (HW2)
  - Sorting Frogs (HW3)
  - Sections: Maximum Sum Subarray, ...

- Let’s take a minute to zoom out and look at some general strategies.
One Strategy

1. Identify natural sub-problems
   • Arrays of half the size
   • Things smaller/larger than a pivot

2. Imagine you had the magical ability to solve those natural sub-problems...what would you do?
   • Just try it with all of the natural sub-problems you can come up with! Anything look helpful?

3. Work out the details
   • Write down pseudocode, etc.
One Strategy

1. Identify natural sub-problems
2. Imagine you had the magical ability to solve those natural sub-problems...what would you do?
3. Work out the details

Think about how you could arrive at MergeSort or QuickSort via this strategy!
Other tips

• Small examples.
  • If you have an idea but are having trouble working out the details, try it on a small example by hand.

• Gee, that looks familiar...
  • The more algorithms you see, the easier it will get to come up with new algorithms!

• Bring in your analysis tools.
  • E.g., if I’m doing divide-and-conquer with 2 subproblems of size n/2 and I want an O(n logn) time algorithm, I know that I can afford O(n) work combining my sub-problems.

• Iterate.
  • Darn, that approach didn’t work! But, if I tweaked this aspect of it, maybe it works better?

• Everyone approaches problem-solving differently...find the way that works best for you.
No one recipe for algorithm design

• This can be frustrating on HW....
• Practice helps!
  • The examples we see in Lecture and in HW are meant to help you practice this skill.
  • Sections are the BEST place to practice!

• There are even more algorithms in the book!
  • Check out Algorithms Illuminated Chapter 3, or CLRS Chapter 4, for even more examples of divide and conquer algorithms.
Roadmap

1st class
Divide and conquer

Sorting
- Longest, Shortest, Max and Min
- Randomized Algorithms
- Asymptotic Analysis
- Recurrences

Data structures
- Randomized Algorithms
- Dynamic Programming
- Greedy Algorithms

Graphs!

The Future!

5 lectures
10 lectures
2 lectures
1 lecture

MIDTERM

We are here

More detailed schedule on the website!
Today

• Begin a brief foray into data structures!
  • See CS 166 for more!
• Binary search trees
  • You may remember these from CS 106B
  • They are better when they’re balanced.

this will lead us to...

• Self-Balancing Binary Search Trees
  • Red-Black trees.
Some data structures for storing objects like 5 (aka, nodes with keys)

- (Sorted) arrays:

- Linked lists:

- Some basic operations:
  - INSERT, DELETE, SEARCH
Sorted Arrays

- **O(n) INSERT/DELETE:**
  - First, find the relevant element (we’ll see how below), and then move a bunch elements in the array:

  ![Sorted Array with 4.5 inserted](image)

- **O(log(n)) SEARCH:**
  - eg, insert 4.5

  ![Sorted Array after insertion](image)

  eg, Binary search to see if 3 is in A.
(Not necessarily sorted)

Linked lists

- **O(1) INSERT:**

  eg, insert 6
  ![Diagram](image)

  ![Diagram](image)

- **O(n) SEARCH/DELETE:**

  eg, search for 1 (and then you could delete it by manipulating pointers).
# Motivation for Binary Search Trees

<table>
<thead>
<tr>
<th></th>
<th>Sorted Arrays</th>
<th>Linked Lists</th>
<th>(Balanced) Binary Search Trees</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Search</strong></td>
<td>$O(\log(n))$</td>
<td>$O(n)$</td>
<td>$O(\log(n))$</td>
</tr>
<tr>
<td><strong>Delete</strong></td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(\log(n))$</td>
</tr>
<tr>
<td><strong>Insert</strong></td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(\log(n))$</td>
</tr>
</tbody>
</table>
Binary tree terminology

Each node has two children.

The left child of 3 is 2.

The right child of 3 is 4.

The parent of 3 is 5.

2 is a descendant of 5.

Each node has a pointer to its left child, right child, and parent.

Both children of 1 are NIL. (I won’t usually draw them).

The height of this tree is 3. (Max length of path from the root to a leaf).

For today all keys are distinct.
Binary Search Trees

• A BST is a binary tree so that:
  • Every LEFT descendant of a node has key less than that node.
  • Every RIGHT descendant of a node has key larger than that node.

• Example of building a binary search tree:
Binary Search Trees

• A BST is a binary tree so that:
  • Every LEFT descendant of a node has key less than that node.
  • Every RIGHT descendant of a node has key larger than that node.

• Example of building a binary search tree:
Binary Search Trees

• A BST is a binary tree so that:
  • Every LEFT descendant of a node has key less than that node.
  • Every RIGHT descendant of a node has key larger than that node.

• Example of building a binary search tree:
Binary Search Trees

• A BST is a binary tree so that:
  • Every LEFT descendant of a node has key less than that node.
  • Every RIGHT descendant of a node has key larger than that node.

• Example of building a binary search tree:
Binary Search Trees

• A BST is a binary tree so that:
  • Every LEFT descendant of a node has key less than that node.
  • Every RIGHT descendant of a node has key larger than that node.

• Example of building a binary search tree:

Q: Is this the only binary search tree I could possibly build with these values?

A: No. I made choices about which nodes to choose when. Any choices would have been fine.
Aside: this should look familiar

kinda like QuickSort
Binary Search Trees

- A BST is a binary tree so that:
  - Every LEFT descendant of a node has key less than that node.
  - Every RIGHT descendant of a node has key larger than that node.

Which of these is a BST?
1 minute Think-Pair-Share

Binary Search Tree

```
  5
 / \ 
3   7
/ \ / \ 
2  4 8
```

```
  5
 / \ 
3   7
/ \ / \ 
2  4 8
```

NOT a Binary Search Tree

```
  5
 / \ 
3   7
/ \ / \ 
2  4 8
```

```
  5
 / \ 
3   7
/ \ / \ 
2  4 8
```

NOT a Binary Search Tree
Aside: In-Order Traversal of BSTs

• Output all the elements in sorted order!

• inOrderTraversal(x):
  • if x!= NIL:
    • inOrderTraversal(x.left)
    • print(x.key)
    • inOrderTraversal(x.right)
Aside: In-Order Traversal of BSTs

• Output all the elements in sorted order!

• inOrderTraversal(x):
  • if x!= NIL:
    • inOrderTraversal( x.left )
    • print( x.key )
    • inOrderTraversal( x.right )
Aside: In-Order Traversal of BSTs

• Output all the elements in sorted order!

• inOrderTraversal(x):
  • if x!= NIL:
    • inOrderTraversal( x.left )
    • print( x.key )
    • inOrderTraversal( x.right )
Aside: In-Order Traversal of BSTs

• Output all the elements in sorted order!

• inOrderTraversal(x):
  • if x!= NIL:
    • inOrderTraversal( x.left )
    • print( x.key )
    • inOrderTraversal( x.right )
Aside: In-Order Traversal of BSTs

• Output all the elements in sorted order!

• `inOrderTraversal(x)`:  
  • if `x != NIL`:  
    • `inOrderTraversal(x.left)`  
    • `print(x.key)`  
    • `inOrderTraversal(x.right)`  

```plaintext
2
NUL NUL
3
2 4
7
5
```
Aside: In-Order Traversal of BSTs

• Output all the elements in sorted order!

• inOrderTraversal(x):
  • if x!= NIL:
    • inOrderTraversal( x.left )
    • print( x.key )
    • inOrderTraversal( x.right )
Aside: In-Order Traversal of BSTs

• Output all the elements in sorted order!

• inOrderTraversal(x):
  • if x!= NIL:
    • inOrderTraversal( x.left )
    • print( x.key )
    • inOrderTraversal( x.right )
Aside: In-Order Traversal of BSTs

• Output all the elements in sorted order!

• `inOrderTraversal(x)`:  
  • if `x!= NIL`:  
    • `inOrderTraversal( x.left )`
    • `print( x.key )`
    • `inOrderTraversal( x.right )`
Aside: In-Order Traversal of BSTs

- Output all the elements in sorted order!

  - inOrderTraversal(x):
    - if x!= NIL:
      - inOrderTraversal(x.left)
      - print(x.key)
      - inOrderTraversal(x.right)
Aside: In-Order Traversal of BSTs

• Output all the elements in sorted order!

• inOrderTraversal(x):
  • if x!= NIL:
    • inOrderTraversal( x.left )
    • print( x.key )
    • inOrderTraversal( x.right )
Aside: In-Order Traversal of BSTs

• Output all the elements in sorted order!

• inOrderTraversal(x):
  • if x!= NIL:
    • inOrderTraversal( x.left )
    • print( x.key )
    • inOrderTraversal( x.right )
Aside: In-Order Traversal of BSTs

• Output all the elements in sorted order!

• inOrderTraversal(x):
  • if x!= NIL:
    • inOrderTraversal( x.left )
    • print( x.key )
    • inOrderTraversal( x.right )
Aside: In-Order Traversal of BSTs

• Output all the elements in sorted order!

• inOrderTraversal(x):
  • if x!= NIL:
    • inOrderTraversal( x.left )
    • print( x.key )
    • inOrderTraversal( x.right )

• Runs in time O(n).
Back to the goal

Fast SEARCH/INSERT/DELETE

Can we do these?
SEARCH in a Binary Search Tree
definition by example

EXAMPLE: Search for 4.
EXAMPLE: Search for 4.5
• It turns out it will be convenient to return 4 in this case
• (that is, return the last node before we went off the tree)

How long does this take?
O(length of longest path) = O(height)
INSERT in a Binary Search Tree

EXAMPLE: Insert 4.5

- \text{INSERT}(\text{key}):
  - x = \text{SEARCH}(\text{key})
  - \text{Insert} a new node with desired key at x...

You thought about this on your pre-lecture exercise! (See skipped slide for pseudocode.)
DELETE in a Binary Search Tree

EXAMPLE: Delete 2

- DELETE(key):
  - x = SEARCH(key)
  - if x.key == key:
    - ....delete x....

You thought about this in your pre-lecture exercise too!

This is a bit more complicated...see the skipped slides for some pictures of the different cases.
How long do these operations take?

- **SEARCH** is the big one.
  - Everything else just calls **SEARCH** and then does some small $O(1)$-time operation.

Trees have depth $O(\log(n))$. **Done!**

How long does search take?

Lucky the lackadaisical lemur.

Plucky the Pedantic Penguin.
Search might take time $O(n)$.

- This is a valid binary search tree.
- The version with $n$ nodes has depth $n$, **not** $O(\log(n))$. 
What to do?

• Goal: Fast SEARCH/INSERT/DELETE
• All these things take time $O(\text{height})$
• And the height might be big!!! 😞

• Idea 0:
  • Keep track of how deep the tree is getting.
  • If it gets too tall, re-do everything from scratch.
    • At least $\Omega(n)$ every so often…

• Turns out that’s not a great idea. Instead we turn to…
Self-Balancing Binary Search Trees
Idea 1: Rotations

- Maintain Binary Search Tree (BST) property, while moving stuff around.

Note: A, B, C, X, Y are variable names, not the contents of the nodes.

That's not binary!!

CLAIM: this still has BST property.

B fell down.

No matter what lives underneath A, B, C, this takes time $O(1)$. (Why?)
This seems helpful

YOINK!
Strategy?

• Whenever something seems unbalanced, do rotations until it’s okay again.

Even for Lucky this is pretty vague. What do we mean by “seems unbalanced”? What’s “okay”?

Lucky the Lackadaisical Lemur
Idea 2: have some proxy for balance

- Maintaining **perfect balance** is too hard.
- Instead, come up with some **proxy for balance**:
  - If the tree satisfies **[SOME PROPERTY]**, then it’s pretty balanced.
  - We can maintain **[SOME PROPERTY]** using rotations.

There are actually several ways to do this, but today we’ll see...
Red-Black Trees

• A Binary Search Tree that balances itself!
• No more time-consuming by-hand balancing!
• Be the envy of your friends and neighbors with the time-saving...

Red-Black tree!

Maintain balance by stipulating that black nodes are balanced, and that there aren’t too many red nodes.

It’s just good sense!
Red-Black Trees obey the following rules (which are a proxy for balance)

- Every node is colored **red** or **black**.
- The root node is a **black node**.
- NIL children count as **black nodes**.
- Children of a **red node** are **black nodes**.
- For all nodes x:
  - all paths from x to NIL’s have the same number of **black nodes** on them.

I’m not going to draw the NIL children in the future, but they are treated as black nodes.
Examples(?)

• Every node is colored red or black.
• The root node is a black node.
• NIL children count as black nodes.
• Children of a red node are black nodes.
• For all nodes x:
  • all paths from x to NIL’s have the same number of black nodes on them.

Which of these are red-black trees?
(NIL nodes not drawn)

1 minute think
1 minute share

Yes! No! No! No!
Why these rules??????

• This is pretty balanced.
  • The black nodes are balanced
  • The red nodes are “spread out” so they don’t mess things up too much.

• We can maintain this property as we insert/delete nodes, by using rotations.

This is the really clever idea!
This Red-Black structure is a proxy for balance. It’s just a smidge weaker than perfect balance, but we can actually maintain it!
This is “pretty balanced”

• To see why, intuitively, let’s try to build a Red-Black Tree that’s unbalanced.

Let’s build some intuition!

Let’s build some intuition!

Lucky the lackadaisical lemur

One path can be at most twice as long another if we pad it with red nodes.

Conjecture: the height of a red-black tree with \( n \) nodes is at most \( 2 \log(n) \)

Note, this is just a conjecture to build intuition! We’ll prove a rigorous statement on the next slide.
The height of a RB-tree with \( n \) non-NIL nodes is at most \( 2\log(n + 1) \)

- Define \( b(x) \) to be the number of black nodes in any path from \( x \) to NIL.
  - (excluding \( x \), including NIL).

- Claim:
  - There are at least \( 2^{b(x)} - 1 \) non-NIL nodes in the subtree underneath \( x \).
    (Including \( x \)).

- [Proof by induction – on board if time]

Then:
\[
\begin{align*}
    n & \geq 2^{b(root)} - 1 \\
    & \geq 2^{\text{height}/2} - 1
\end{align*}
\]

using the Claim

b(root) \( \geq \) height/2 because of RBTree rules.

Rearranging:
\[
n + 1 \geq 2^{\text{height}/2} \Rightarrow \text{height} \leq 2\log(n + 1)
\]
This is great!

- SEARCH in an RBTree is immediately $O(\log(n))$, since the depth of an RBTree is $O(\log(n))$.

- What about INSERT/DELETE?
  - Turns out, you can INSERT and DELETE items from an RBTree in time $O(\log(n))$, while maintaining the RBTree property.
  - That’s why this is a good property!
INSERT/DELETE

• I expect we are out of time...
  • There are some slides which you can check out to see how to do INSERT/DELETE in RBTrees if you are curious.
  • See CLRS Ch 13. for even more details.

• You are not responsible for the details of INSERT/DELETE for RBTrees for this class.
  • You should know what the “proxy for balance” property is and why it ensures approximate balance.
  • You should know that this property can be efficiently maintained, but you do not need to know the details of how.
• Suppose we want to insert 0 here.

• There are 3 “important” cases for different colorings of the existing tree, and there are 9 more cases for all of the various symmetries of these 3 cases.
**INSERT: Case 1**

- Make a new **red node**.
- Insert it as you would normally.

Example: insert 0

What if it looks like this?
• Suppose we want to insert 0 here.

• There are 3 “important” cases for different colorings of the existing tree, and there are 9 more cases for all of the various symmetries of these 3 cases.
**INSERT: Case 2**

- Make a new **red node**.
- Insert it as you would normally.
- Fix things up if needed.

**Example: insert 0**
INSERT: Case 2

• Make a new red node.
• Insert it as you would normally.
• Fix things up if needed.

Example: insert 0

Can’t we just insert 0 as a black node?

No!
We need a bit more context

What if it looks like this?

Example: insert 0
We need a bit more context

- Add 0 as a red node.

What if it looks like this?

Example: insert 0
We need a bit more context

- Add 0 as a red node.
- **Claim:** RB-Tree properties still hold.

What if it looks like this?

Example: insert 0

Flip colors!
But what if **that** was red?

What if it looks like this?

Example: insert 0
More context...

What if it looks like this?

Example: insert 0
More context...

What if it looks like this?

Example: insert 0

Now we’re basically inserting 6 into some smaller tree. Recurse!

This one!
Example, part I

Want to insert 0 here.
Example, part I
Example, part I
Example, part I

Need to know how to insert into trees that look like this...

Want to insert 6 here.
• Suppose we want to insert 0 here.

• There are 3 “important” cases for different colorings of the existing tree, and there are 9 more cases for all of the various symmetries of these 3 cases.
INSERT: Case 3

• Make a new red node.
• Insert it as you would normally.
• Fix things up if needed.

Example: Insert 0.
• Maybe with a subtree below it.
Recall Rotations

• Maintain Binary Search Tree (BST) property, while moving stuff around.

YOINK!

That's not binary!!

CLAIM: this still has BST property.
Inserting into a Red-Black Tree

- Make a new **red node**.
- Insert it as you would normally.
- Fix things up if needed.

What if it looks like this?

YOINK!

Argue that this is a good thing to do!
Example, part 2

Want to insert 6 here.
Example, part 2

YOINK!
Example, part 2

-1

-3

-4

-2

6

3

0

7

YOINK!
Example, part 2

TA-DA!
Many cases

- Suppose we want to insert 0 here.
- There are 3 “important” cases for different colorings of the existing tree, and there are 9 more cases for all of the various symmetries of these 3 cases.
Deleting from a Red-Black tree

Fun exercise!

Ollie the over-achieving ostrich
That’s a lot of cases!

• You are **not responsible** for the nitty-gritty details of Red-Black Trees. (For this class)
  • Though implementing them is a great exercise!

• You should know:
  • What are the properties of an RB tree?
  • And (more important) why does that guarantee that they are balanced?
What have we learned?

• Red-Black Trees always have height at most $2\log(n+1)$.
• As with general Binary Search Trees, all operations are $O(\text{height})$
• So all operations with RBTrees are $O(\log(n))$. 
## Conclusion: The best of both worlds

<table>
<thead>
<tr>
<th></th>
<th>Sorted Arrays</th>
<th>Linked Lists</th>
<th>Binary Search Trees*</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Search</strong></td>
<td>O(log(n)) 🎉</td>
<td>O(n) 🙁</td>
<td>O(log(n)) 🎉</td>
</tr>
<tr>
<td><strong>Delete</strong></td>
<td>O(n) 🙁</td>
<td>O(n) 🙁</td>
<td>O(log(n)) 🎉</td>
</tr>
<tr>
<td><strong>Insert</strong></td>
<td>O(n) 🙁</td>
<td>O(1) 😊</td>
<td>O(log(n)) 🎉</td>
</tr>
</tbody>
</table>
Today

• Begin a brief foray into data structures!
  • See CS 166 for more!

• Binary search trees
  • You may remember these from CS 106B
  • They are better when they’re balanced.

this will lead us to...

• Self-Balancing Binary Search Trees
  • Red-Black trees.

Recap
Recap

- Balanced binary trees are the best of both worlds!
- But we need to keep them balanced.
- **Red-Black Trees** do that for us.
  - We get $O(\log(n))$-time INSERT/DELETE/SEARCH
  - Clever idea: have a proxy for balance
Next time

• Hashing!

Before next time

• Pre-lecture exercise for Lecture 8
• More probability yay!