Lecture 7

Binary Search Trees and Red-Black Trees
Announcements

• HW4 is out today
  • This is a full-length one!
Roadmap

- Asymptotic Analysis
- Randomized Algs
- Recurrences
- Sorting

- 5 lectures
- 2 lectures
- MIDTERM

- Divide and conquer
- Greedy Algs
- Dynamic Programming
- Longest, Shortest, Max and Min...

- Graphs!
- Data structures

- More detailed schedule on the website!
- We are here
But first!

- A brief wrap-up of divide and conquer.

Divide and Conquer:
How do we design divide-and-conquer algorithms?

• So far we’ve seen lots of examples.
  • Karatsuba (and Alien Multiplication)
  • MergeSort
  • Select
  • QuickSort
  • Skyline Problem (HW1)
  • Majority Aliens (HW2)
  • Dancing Ducks (HW2)

• 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
   - The left/right half of buildings in a skyline
   - The lower/upper half of buildings in a skyline
   - 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.
Example: dancing ducks from HW2

Goal: implement PARTITION using FLIP

1 5 2 3 4 6
Example: dancing ducks from HW2

1. Identify natural sub-problems.
   • We already know there’s a pivot, so let’s say “left and right of the pivot.”

2. What would happen if we could magically solve the subproblems?

3. Fill in the details.

   - How can we use the one operation we’re allowed to use?
   - Where do we put the pivot for the recursive calls? How do we do the indexing?
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 $O(n \log n)$ 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.
There is no one algorithm for designing algorithms.

• This can be frustrating on HW....
  • What the heck do dancing ducks have to do with the sorting algorithms we covered in lecture?!??!?!?

• Practice helps!
  • The examples we see in Lecture and in HW are meant to help you practice this skill.

• 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

- Sorting
  - Longest, Shortest, Max and Min...
  - Randomized Algorithms (5 lectures)
  - Asymptotic Analysis
  - Recurrences
- Data structures (2 lectures)
- Graphs!
  - Dynamic Programming
- Greedy Algorithms
- MIDTERM
- The Future!

1st class
- Divide and conquer

More detailed schedule on the website!

We are here

1 lecture
10 lectures
5 lectures
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:

  1 2 3 4 5 7 8

- Linked lists:

  HEAD 3 2 1 8 5 7 4

- 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](image)

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

  ![Sorted Array](image)

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

Linked lists

• \(O(1)\) INSERT:

  eg, insert 6

  ![Diagram showing insertion of 6]

• \(O(n)\) SEARCH/DELETE:

  ![Diagram showing search for 1]

  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 number of edges from the root to a leaf).

For today all keys are distinct.

This is a node. It has a key (7).

These nodes are leaves.

This node is the root.
From your pre-lecture exercise...

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:
From your pre-lecture exercise...

**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:
From your pre-lecture exercise...

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:
From your pre-lecture exercise...

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
```

**NOT a Binary Search Tree**

```
      5
     / \
    3   7
   / \ / \
  2  4 8 1
```
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!

• \text{inOrderTraversal}(x):
  • if $x \neq \text{NIL}$:
    • \text{inOrderTraversal}( x.\text{left} )
    • print( x.key )
    • \text{inOrderTraversal}( x.\text{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 )

```python
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 )

2 3 4
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).

2 3 4 5 7
Sorted!
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}):
  - \text{x} = \text{SEARCH}(\text{key})
  - \text{Insert} a new node with desired key at \text{x}...

You thought about this on your pre-lecture exercise!
(See skipped slide for pseudocode.)
**EXAMPLE:** Insert 4.5

- **INSERT**(key):
  - Let x = **SEARCH**(key)
  - If key > x.key:
    - Make a new node with the correct key, and put it as the right child of x.
  - If key < x.key:
    - Make a new node with the correct key, and put it as the left child of x.
  - If x.key == key:
    - Return
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.
DELETE in a Binary Search Tree
several cases (by example)
say we want to delete 3

Case 1: if 3 is a leaf, just delete it.

Case 2: if 3 has just one child, move that up.

Write pseudocode for all of these!
Case 3: if 3 has two children, replace 3 with its **immediate successor**.
(aka, next biggest thing after 3)

- Does this maintain the BST property?
  - Yes.

- How do we find the immediate successor?
  - SEARCH for 3 in the subtree under 3.right

- How do we remove it when we find it?
  - If [3.1] has 0 or 1 children, do one of the previous cases.
  - What if [3.1] has two children?
    - It doesn’t.
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.

Time = $O$ (height of tree)

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

How long does search take?
1 minute think; 1 minute pair+share

Wait a second...
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.

YOINK!

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
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.

Yes!

1 minute think
1 minute pair+share

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

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.

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

$$n \geq 2^{b(root)} - 1$$

using the Claim

$$\geq 2^{\text{height}/2} - 1$$

$b(root) \geq \text{height}/2$ 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.
INSERT: 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.
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

What if it looks like this?

No!
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.

Example: insert 0

What if it looks like this?

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

-4  -3  -2  -1  3  6  7

Want to insert 0 here.
Example, part I

Diagram:
-3
  -4
  -2
-1
  3
  7
  6
  0
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.
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?

![Red-Black Tree Diagram]

YOINK!

![Before and After Diagram]

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

Want to insert 6 here.
Example, part 2

YOINK!

YOINK!
Example, part 2

YOINK!
Example, part 2

```
Example, part 2

TA-DA!

-1

-3

-4

-2

0

6

3

7
```
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!