CSCI 241

Lecture 12
Binary Search Trees: insertion and removal
Announcements

• As usual:
  • Quiz 3 today
  • Lab 4 due Sunday
  • Lab 5 out Monday

• A2 out today! Due Monday 5/11.
Announcements

- Feedback survey results
Next week: Experiment!

• Videos for Monday and Wednesday's lecture topics will be posted over the weekend (all out by end of Monday).

• About 5 video segments cover two lectures, not totaling more than 100 minutes.

• I will also provide practice exercises for each segment.

• M&W class periods (attendance optional): Q&A, exercise solutions, more exercises.

• It's not office hours though - no code help.
Goals

• Know how to perform (and code) three tree traversals: pre-order, in-order, and post-order.

• Know the definition and uses of a binary search tree.

• Be prepared to implement, and know the runtime of, the following BST operations:
  • search
  • insert
  • remove
Inserting into a BST
Inserting into a BST

\[
t: \begin{array}{c}
10 \\
8 \\
4 \\
9 \\
11 \\
16 \\
17 \\
\end{array}
\]

insert(t, 11)
Inserting into a BST

\[ t: \begin{array}{c}
10 \\
8 \\
4 \\
9 \\
11 \\
17 \\
16 \\
\end{array} \]

\[ \text{insert}(t, 11) \]
Inserting into a BST

\[
\text{t:} \quad 10
\]

\[
\begin{array}{c}
4 \\
8 \\
9 \\
11 \\
16 \\
17
\end{array}
\]

\[
\text{insert(t, 11)}
\]

\[
11 > 10
\]
Inserting into a BST

$t$: 10

insert(t, 11)
11 > 10
insert(right, 11)
Inserting into a BST

```
insert(t, 11)
```

```
t: 10
    8
    9
    11
  16
  17
```

```
11 > 10
insert(right, 11)
```
Inserting into a BST

\[
\begin{array}{c}
\text{t:} \\
\quad 10 \\
\quad \quad 8 \\
\quad \quad 16 \\
\quad 4 \\
\quad 9 \\
\quad 11 \\
\quad 17 \\
\end{array}
\]

\[
\begin{align*}
\text{insert}(t, 11) \\
11 & > 10 \\
\text{insert(right, 11)} \\
11 & < 16
\end{align*}
\]
Inserting into a BST

$t$:  

```
  10
 /   \
8     16
 /     /
4     9 11 17
```

- $\text{insert}(t, 11)$
  - $11 > 10$
  - $\text{insert}(\text{right, 11})$
  - $11 < 16$
  - $\text{insert}(\text{left, 11})$
Inserting into a BST

\[ t: \begin{array}{c}
4 \\
8 \\
10 \\
9 \\
16 \\
11 \\
17 \\
\end{array} \]

- \text{insert}(t, 11)
- \text{11 > 10}
- \text{insert(right, 11)}
- \text{11 < 16}
- \text{insert(left, 11)}
Inserting into a BST

Insert(t, 11)

11 > 10

insert(right, 11)

11 < 16

insert(left, 11)

11 == 11
Inserting into a BST

$t: \begin{array}{c}
\begin{array}{ccc}
4 & 8 & 10 \\
9 & 11 & 16 \\
& 17 & \\
\end{array}
\end{array}$

insert(t, 11)
11 > 10
insert(right, 11)
11 < 16
insert(left, 11)
11 == 11
found it! no duplicates, allowed; nothing to do.
return.
Inserting into a BST - the nonexistent case

\[
\text{insert}(t, 5)
\]
Inserting into a BST - the nonexistent case

\[
\text{insert}(t, 5)
\]
Inserting into a BST - the nonexistent case

Insert(t, 5)

5 < 10
Inserting into a BST - the nonexistent case

\[
\begin{align*}
\text{t:} & \quad 10 \\
8 & \quad 16 \\
4 & \quad 9 & \quad 11 & \quad 17 \\
\end{align*}
\]

\text{insert(t, 5)}

5 < 10

\text{insert(left, 5)}
Inserting into a BST - the nonexistent case

\[ t: \quad 10 \]
\[ \begin{array}{c}
8 \\
\downarrow \\
4 \\
\end{array} \quad \begin{array}{c}
9 \\
\downarrow \\
11 \\
\end{array} \quad \begin{array}{c}
16 \\
\downarrow \\
17 \\
\end{array} \]

insert(\(t, 5\))
\[ 5 < 10 \]
insert(left, 5)
Inserting into a BST - the nonexistent case

\[
\begin{align*}
t: & \quad 10 \\
8 & \quad \text{insert}(t, 5) \\
5 & < 10 \\
9 & \quad \text{insert}(\text{left}, 5) \\
5 & < 8 \\
4 & \\n11 & \\n17 &
\end{align*}
\]
Inserting into a BST - the nonexistent case

```
 10
 /  \
8    16
/  \
4    9
    11
    17
```

- `insert(t, 5)`
- `5 < 10`
- `insert(left, 5)`
- `5 < 8`
- `insert(left, 5)`
Inserting into a BST - the nonexistent case

\[
\begin{align*}
\text{t:} & \quad 10 \\
8 & \quad 16 \\
4 & \quad 9 \\
11 & \quad 17
\end{align*}
\]

insert(t, 5)  
5 < 10  
insert(left, 5)  
5 < 8  
insert(left, 5)
Inserting into a BST - the nonexistent case

```
insert(t, 5)
5 < 10
insert(left, 5)
5 < 8
insert(left, 5)
5 > 4
```
Inserting into a BST - the nonexistent case

Insert: 10
- Insert left: 5 < 10
- Insert left: 5 < 8
- Insert right: 5 > 4

Tree:
- 10
  - 8
    - 4
    - 9
  - 16
    - 11
    - 17
Inserting into a BST - the nonexistent case

$t$: 10

- 8
  - 4
  - 9
- 16
  - 11
  - 17

insert(t, 5)
5 < 10
insert(left, 5)
5 < 8
insert(left, 5)
5 > 4
insert(right, 5)
null - not found. insert it here!
Inserting into a BST - the nonexistent case

\[
\begin{align*}
t &: 10 \\
8 & \quad 16 \\
4 & \quad 9 \quad 11 \quad 17 \\
5 &
\end{align*}
\]

\[
\begin{align*}
\text{insert}(t, 5) \\
5 & < 10 \quad \text{insert(left, 5)} \\
5 & < 8 \quad \text{insert(left, 5)} \\
5 & > 4 \quad \text{insert(right, 5)} \\
\text{null - not found. insert it here!}
\end{align*}
\]
Let’s Build Some Trees

\[ t = \text{new BST}(); \]
\[ t.\text{insert}(-1); \]
\[ t.\text{insert}(8); \]
\[ t.\text{insert}(9); \]
\[ t.\text{insert}(10); \]
\[ t.\text{insert}(11); \]
\[ t.\text{insert}(15); \]
\[ t.\text{insert}(16); \]
\[ t.\text{insert}(16); \]
Write a method to find the smallest value in a BST:

/** Returns min value in BST n. * pre: n is not null */
public int minimum(Node n) {
Write a method to find the smallest value in a BST:

1. Spec
/** Returns min value in BST n. */
* pre: n is not null */
public int minimum(Node n) {
    if (n.left == null)
        return n.value;
    return minimum(n.left);
}

3. Recursive definition:
   Smallest(n) is:
   • the smallest value in the left subtree, or
   • n.value if no left subtree exists.

2. Base case

4. Implement using recursive call
Write a method to find the smallest value in a BST:

1. Spec
   /** Returns min value in BST n. 
    * pre: n is not null */
   public int minimum(Node n) {
      if (n.left == null) 
         return n.value;
      return minimum(n.left);
   }

2. Base case

3. Recursive definition:
   Smallest(n) is:
   • the smallest value in the left subtree, or
   • n.value if no left subtree exists.
Warm-up

Write a method to find the smallest value in a BST:

```java
/** Returns min value in BST n. * pre: n is not null */
public int minimum(Node n) {
    if (n.left == null)
        return n.value;
    return minimum(n.left);
}
```
Deleting a node from a BST

Three possible cases:
1. n has no children (is a leaf)
2. n has one child
3. n has two children
Deleting a node from a BST: Case 1

Three possible cases:

1. **n has no children (is a leaf)**
2. n has one child
3. n has two children

```java
if (n is a leaf)
    replace parent’s child with null
```
Deleting a node from a BST: Case 1

Three possible cases:
1. n has no children (is a leaf)
2. n has one child
3. n has two children

if (n is a leaf)
    replace parent’s child with null
Deleting a node from a BST: Case 1

Three possible cases:

1. **n has no children (is a leaf)**
2. **n has one child**
3. **n has two children**

```java
if (n is a leaf)
    replace parent’s child with null
```
Deleting a node from a BST: Case 2

Three possible cases:
1. n has no children (is a leaf)
2. n has one child
3. n has two children
Deleting a node from a BST: Case 2

Three possible cases:
1. n has no children (is a leaf)
2. n has one child
3. n has two children

if (n has exactly one child)
Deleting a node from a BST: Case 2

Three possible cases:
1. n has no children (is a leaf)
2. n has one child
3. n has two children

if (n has exactly one child)
   replace parent’s child with n’s child
Deleting a node from a BST: Case 2

Three possible cases:
1. n has no children (is a leaf)
2. n has one child
3. n has two children

if (n has exactly one child)
   replace parent’s child with n’s child
   replace n’s child’s parent with n’s parent
Deleting a node from a BST: Case 2

Three possible cases:
1. n has no children (is a leaf)
2. n has one child
3. n has two children

If (n has exactly one child)
- replace parent’s child with n’s child
- replace n’s child’s parent to n’s parent
Deleting a node from a BST: Case 2

Three possible cases:
1. n has no children (is a leaf)
2. n has one child
3. n has two children

if (n has exactly one child)
   replace parent’s child with n’s child
   replace n’s child’s parent to n’s parent
Deleting a node from a BST: Case 3

Three possible cases:
1. $n$ has no children (is a leaf)
2. $n$ has one child
3. $n$ has two children

if ($n$ has two children)
Deleting a node from a BST: Case 3

Three possible cases:
1. n has no children (is a leaf)
2. n has one child
3. n has two children

if (n has two children)
let k = min node in right subtree
Deleting a node from a BST: Case 3

Three possible cases:
1. n has no children (is a leaf)
2. n has one child
3. n has two children

if (n has two children)
   let k = min node in right subtree
   replace n’s value with k’s value
Deleting a node from a BST:
Case 3

Three possible cases:
1. n has no children (is a leaf)
2. n has one child
3. n has two children

if (n has two children)
let \( k = \text{min node in right subtree} \)
replace n’s value with k’s value
Deleting a node from a BST: Case 3

Three possible cases:
1. n has no children (is a leaf)
2. n has one child
3. n has two children

if (n has two children)
   let k = min node in right subtree
   replace n’s value with k’s value

Can we do that?
Deleting a node from a BST: Case 3

Three possible cases:
1. n has no children (is a leaf)
2. n has one child
3. n has two children

if (n has two children)
let \( k = \text{min node in right subtree} \)
replace n’s value with k’s value

Can we do that?
• k is n’s \textit{successor} (next in an in-order traversal)
Deleting a node from a BST: Case 3

Three possible cases:
1. n has no children (is a leaf)
2. n has one child
3. n has two children

if (n has two children)
let \( k = \text{min node in right subtree} \)
replace n’s value with k’s value

Can we do that?
• k is n’s successor (next in an in-order traversal)
• Everything else in n’s right subtree is bigger than it
Deleting a node from a BST: Case 3

Three possible cases:
1. n has no children (is a leaf)
2. n has one child
3. n has two children

If (n has two children)
   let k = min node in right subtree
   replace n’s value with k’s value

Can we do that?
• k is n’s successor (next in an in-order traversal)
• Everything else in n’s right subtree is bigger than it
• Everything in n’s left subtree is smaller than it
Deleting a node from a BST: Case 3

Three possible cases:
1. n has no children (is a leaf)
2. n has one child
3. n has two children

if (n has two children)
let \( k = \text{min node in right subtree} \)
replace n’s value with k’s value

Can we do that?
• k is n’s successor (next in an in-order traversal)
• Everything else in n’s right subtree is bigger than it
• Everything in n’s left subtree is smaller than it
• k’s value can safely replace n’s…but now we have a duplicate.
Deleting a node from a BST:  
Case 3

Three possible cases:
1. n has no children (is a leaf)
2. n has one child
3. n has two children

if (n has two children)
let k = min node in right subtree
replace n’s value with k’s value
remove k from n’s right subtree
Deleting a node from a BST: Case 3

Three possible cases:
1. \text{n has no children (is a leaf)}
2. \text{n has one child}
3. \text{n has two children}

\begin{itemize}
\item if (n has two children)
\item let \text{k = min node in right subtree}
\item replace n’s value with k’s value
\item remove k from n’s right subtree
\end{itemize}
Deleting a node from a BST: Case 3

Three possible cases:
1. n has no children (is a leaf)
2. n has one child
3. n has two children

if (n has two children)
   let k = min node in right subtree
   replace n’s value with k’s value
   remove k from n’s right subtree (recursively!)
Deleting a node from a BST: Case 3

Three possible cases:
1. n has no children (is a leaf)
2. n has one child
3. n has two children

if (n has two children)
let \( k = \text{min node in right subtree} \)
replace n’s value with k’s value
remove k from n’s right subtree

this has to be either Case 1 or Case 2!
Deleting a node from a BST: Case 3

Three possible cases:
1. n has no children (is a leaf)
2. n has one child
3. n has two children

if (n has two children)
let k = min node in right subtree
replace n’s value with k’s value
remove k from n’s right subtree

this has to be either Case 1 or Case 2!

Why?
Deleting a node from a BST: Case 3

Three possible cases:
1. n has no children (is a leaf)
2. n has one child
3. n has two children

if (n has two children)
  let k = min node in right subtree
  replace n’s value with k’s value
  remove k from n’s right subtree

this has to be either Case 1 or Case 2!

Why? Rewind to before we removed it:
Deleting a node from a BST: Case 3

Three possible cases:
1. n has no children (is a leaf)
2. n has one child
3. n has two children

if (n has two children)
let \( k = \text{min node in right subtree} \)
replace n’s value with k’s value
remove k from n’s right subtree

this has to be either Case 1 or Case 2!

Why? Rewind to before we removed it:
Deleting a node from a BST: Case 3

Three possible cases:
1. n has no children (is a leaf)
2. n has one child
3. n has two children

if (n has two children)
let k = min node in right subtree
replace n’s value with k’s value
remove k from n’s right subtree

this has to be either Case 1 or Case 2!

Why? Rewind to before we removed it:
• k is the smallest node in n’s right subtree.
Deleting a node from a BST: Case 3

Three possible cases:
1. n has no children (is a leaf)
2. n has one child
3. n has two children

if (n has two children)
let \( k = \text{min node in right subtree} \)
replace n’s value with k’s value
remove k from n’s right subtree

this has to be either Case 1 or Case 2!

Why? Rewind to before we removed it:
• k is the smallest node in n’s right subtree.
• if it had a left child, that child would have to be smaller!
Details

- Need to update root pointer if root is removed.
- Can’t assume n.parent isn’t null - n may be root
- To update parent’s child pointer, you need to know which (L or R) child pointer to update.
- The approach presented differs from that in CLRS and some other resources.
Practice

Do the following operations in sequence:

- remove(9)
- remove(4)
- remove(10)
Practice

Do the following operations in sequence:

remove(9)
remove(4)
remove(10)
30 second kitten break
The Set ADT

/** A collection that contains no duplicates. */

Supports these operations:

- `boolean contains(Object ob);`
- `boolean add(Object ob);`
- `boolean remove(Object ob);`
Set ADT

/** A collection that contains no duplicate * elements. */
interface Set {
  /** Return true if the set contains ob */
  boolean contains(Object ob);

  /** Add ob to the set; return true iff * the collection is changed. */
  boolean add(Object ob);

  /** Remove ob from the set; return true iff * the collection is changed. */
  boolean remove(Object ob);
...

The **Set ADT**

/** A collection that contains no duplicates. */

Supports these operations:

- `boolean contains(Object ob);`
- `boolean add(Object ob);`
- `boolean remove(Object ob);`

Possible concrete implementations?

- Array (sorted, unsorted, linked list)
- BST
The **Set ADT**

/** A collection that contains no duplicates. */

Supports these operations:

- `boolean contains(Object ob);`
- `boolean add(Object ob);`
- `boolean remove(Object ob);`

Runtimes of possible concrete implementations?

<table>
<thead>
<tr>
<th></th>
<th>contains</th>
<th>add</th>
<th>remove</th>
</tr>
</thead>
<tbody>
<tr>
<td>array (unsorted)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>array (sorted)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>linked list (unsorted)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>linked list (sorted)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>binary search tree</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: (unsorted) ArraySet<T>

class ArraySet<T> implements Set<T> {
    T[] a;
    int size;
    /** Return true iff the collection contains x */
    boolean contains(T x) {
        for (int i = 0; i < size; i++) {
            if (a[i].equals(x))
                return true;
        }
        return false;
    }
    return false;
}
/** Add x to the collection; return true iff
    * the collection is changed. */
    boolean add(T x) {
        if (!contains(x)) {
            a[size] = x; // let’s hope a is big enough...
            size++;
            return true;
        }
        return false;
    }
}