CS 161
Section 4
[CA Name]
Agenda

• Recap
  • Trees
  • Hashing
• Handout
Trees (BSTs and Red-Black)
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.
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)

Write pseudocode (or actual code) to implement this!
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.

The NIL children are treated as black nodes.
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 \( \text{NIL} \).
  - (excluding \( x \), including \( \text{NIL} \)).

- **Claim:**
  - There are at least \( 2^{b(x)} - 1 \) non-NIL nodes in the subtree underneath \( x \).
    - (Including \( x \)).
  - [Proof by induction – see lecture notes]

Then:

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

Using the Claim

Then:

\[
\begin{align*}
    n + 1 & \geq 2^{\frac{\text{height}}{2}} \\
    \Rightarrow \text{height} & \leq 2\log(n + 1)
\end{align*}
\]
## Running time comparison (Red-Black Trees)

<table>
<thead>
<tr>
<th></th>
<th>Sorted Arrays</th>
<th>Linked Lists</th>
<th>Red-Black 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>Search $+O(1)$</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>
<tr>
<td><strong>Extract-min</strong></td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(\log(n))$</td>
</tr>
</tbody>
</table>
Hashing
This is a **hash table** (with chaining)

- Array of $n$ buckets.
- Each bucket stores a linked list.
  - We can insert into a linked list in time $O(1)$
  - To find something in the linked list takes time $O(\text{length(list)})$.
- $h:U \rightarrow \{1,\ldots,n\}$ can be any function:
  - but for concreteness let’s stick with $h(x) =$ least significant digit of $x$.

For demonstration purposes only!
This is a terrible hash function! Don’t use this!

**INSERT:**

```
13  22  43  9
```

**SEARCH 43:**

Scan through all the elements in bucket $h(43) = 3$. 

$n$ buckets (say $n=9$)
The game

1. An adversary chooses any $n$ items $u_1, u_2, \ldots, u_n \in U$, and any sequence of INSERT/DELETE/SEARCH operations on those items.

2. You, the algorithm, chooses a random hash function $h: U \rightarrow \{1, \ldots, n\}$.

3. HASH IT OUT #hashpuns

INSERT 13, INSERT 22, INSERT 43, INSERT 92, INSERT 7, SEARCH 43, DELETE 92, SEARCH 7, INSERT 92
What if we have lots of collisions?

Solution: Randomness
Example

• Say that $h$ is \textit{uniformly random}.
  • That means that $h(1)$ is a \textit{uniformly random} number between 1 and $n$.
  • $h(2)$ is also a \textit{uniformly random} number between 1 and $n$, independent of $h(1)$.
  • $h(3)$ is also a \textit{uniformly random} number between 1 and $n$, independent of $h(1)$, $h(2)$.

• ...

• $h(n)$ is also a \textit{uniformly random} number between 1 and $n$, independent of $h(1)$, $h(2)$, …, $h(n-1)$. 
Expected **number of items in** $u_i$’s **bucket**?

- $E[\cdot] = \sum_{j=1}^{n} P\{ h(u_i) = h(u_j) \}$
- $= 1 + \sum_{j \neq i} P\{ h(u_i) = h(u_j) \}$
- $= 1 + \sum_{j \neq i} 1/n$
- $= 1 + \frac{n-1}{n} \leq 2$. 

**That’s what we wanted.**

All that we needed was that this is $1/n$.
Universal hash family
Let’s stare at this definition

- $H$ is a *universal hash family* if:
  - When $h$ is chosen uniformly at random from $H$,

$$P_{h \in H}\{ h(u_i) = h(u_j) \} \leq \frac{1}{n}$$

for all $u_i, u_j \in U$ with $u_i \neq u_j$,
Sounds good? But How to pick the Hash functions?

Solution: Universal Hash Family
Why Do We Need Hash Family?

• Remember that after we select the hash function, we need to store the hash function for search/delete.
  • What will happen if we did not remember the hash function?

• The set of all hash functions contains $n^M$ functions. So to store the index of a function, we need $\log(n^M) = M\log(n)$ bits 😞.

• A smaller universal hash family it’s easier to remember which function from the family we’re using.
An example of small universal hash family

• Here’s one:
  • Pick a prime $p \geq M$.
  • Define
    \[ f_{a,b}(x) = ax + b \mod p \]
    \[ h_{a,b}(x) = f_{a,b}(x) \mod n \]
  • Claim:
    \[ H = \{ h_{a,b}(x) : a \in \{1, \ldots, p - 1\}, b \in \{0, \ldots, p - 1\} \} \]

is a universal hash family.
So the whole scheme will be

Choose a and b at random and form the function $h_{a,b}$

We can store $h$ in space $O(\log(M))$ since we just need to store $a$ and $b$.

Probably these buckets will be pretty balanced.
Conclusion:

• We can build a hash table that supports INSERT/DELETE/SEARCH in O(1) expected time,
  • if we know that only n items are every going to show up, where n is waaayyyyyyy less than the size M of the universe.

• The space to implement this hash table is \( O(n \log(M)) \) bits.
  • O(n) buckets
  • O(n) items with \( \log(M) \) bits per item
  • \( O(\log(M)) \) to store the hash fn.

• M is waaayyyyyyy bigger than n, but \( \log(M) \) probably isn’t.
Thank you!