Last Time

- Linear Time Sorting
  - Counting Sort
  - Radix Sort
  - Bucket Sort
Today

- Some C++ review
- Binary Search Trees
Homework Stats

- Mean: 83.79 (w/o late penalties)
- Standard Deviation: 15
- Median: 87.5
Header files.
Header files.

Return values: `vector<int>` vs. `vector<int>*`
C++ Review

- Header files.
- Return values: `vector<int>` vs. `vector<int>>*</vector<int>`
- Global variables.
- Header files.
- Return values: vector<int> vs. vector<int>*
- Global variables.
- &(*ptr) == ptr
C++ Review

- Header files.
- Return values: `vector<int>` vs. `vector<int>*`
- Global variables.
- `&(*ptr) == ptr`
- `i++` vs. `++i`
Data Structures

- Data structure is a set of elements and the relationship between them.
- The appropriate data structure for a task is determined by the functions it needs to support.
Data Structures

- Data structure is a set of elements and the relationship between them.
- The appropriate data structure for a task is determined by the functions it needs to support.
- A dictionary supports (minimally) **Insert**, **Search** and **Delete**.
- Other data structures might need **Minimum**, **Maximum**, etc.
Data structure is a set of elements and the relationship between them.

The appropriate data structure for a task is determined by the functions it needs to support.

A dictionary supports (minimally) INSERT, SEARCH and DELETE.

Other data structures might need MINIMUM, MAXIMUM, etc.

For a restricted domain, $D = \{1, \ldots, k\}$, we can use an array, $A[1..k]$. This allows INSERT, SEARCH and DELETE to be $O(1)$. 
Data Structures

- Data structure is a set of elements and the relationship between them.
- The appropriate data structure for a task is determined by the functions it needs to support.
- A dictionary supports (minimally) INSERT, SEARCH and DELETE.
- Other data structures might need MINIMUM, MAXIMUM, etc.
- For a restricted domain, \( D = \{1, \ldots, k\} \), we can use an array, \( A[1..k] \). This allows INSERT, SEARCH and DELETE to be \( O(1) \).
  - Aside: Hashing attempts to map an unrestricted domain to a restricted one.
Binary Search Trees (BSTs) are a simple and efficient implementation of a **dictionary**.

- A BST is a rooted binary tree.
- The **keys** are at the nodes.
- For every node, $v$, the keys of the left subtree $\leq key(v)$
- For every node, $v$, the keys of the right subtree $\geq key(v)$
Binary Search Trees (BSTs) are a simple and efficient implementation of a dictionary.

- A BST is a rooted binary tree.
- The keys are at the nodes.
- For every node, $v$, the keys of the left subtree $\leq \text{key}(v)$
- For every node, $v$, the keys of the right subtree $\geq \text{key}(v)$
- Binary Search Trees are not, by definition, balanced.
Binary Search Tree Data Structure

- **key**
- **parent**
- **left**
- **right**
Binary Search Tree Example

```
- 10
  - 5
    - 1
    - 8
  - 20
  null
```

root
Not a Binary Search Tree Example

```
root 10
null

5 20
```

![Binary Search Tree Diagram](image-url)
Not a Binary Search Tree Example
• The height of a BST is at most $n - 1$.
• The height of a BST is at least $\log n$. 
Binary Search Trees are sorted.
Constructing a sorted array is $\Theta(n)$

```
TRAVVERSE(T)

if T.root then
    TRAVVERSE(T.left)
    PRINT T.key
    TRAVVERSE(T.right)
end if
```
Sorting with a BST

- Binary Search Trees are sorted.
- Constructing a sorted array is $\Theta(n)$

**TRAVVERSE(T)**

```plaintext
if T.root then
    TRAVVERSE(T.left)
    PRINT T.key
    TRAVVERSE(T.right)
end if
```

- Can we prove that this is correct and $\Theta(n)$?
Searching a BST

**Search**($T, x$)

if $T$ then
  if $T.key = x$ then
    return $T$
  else
    if $T.key < x$ then
      return **Search**($T.left, x$)
    else
      return **Search**($T.right, x$)
    end if
  end if
else
  return null
end if
Searching a BST

- `SEARCH(T, x)` is $O(\text{height})$
- If balanced, $\text{height} = \log n$, so $O(\log n)$.
- Worst case scenario, a sequential search, $O(n)$.
Finding specific elements in a BST

- \textbf{Minimum}(T) = O(\text{height}). Traverse to the left.
- \textbf{Maximum}(T) = O(\text{height}). Traverse to the right.
- \textbf{Successor}(T) - Find the node with smallest key greater than T.key.
**Successor**

If $T$ has a right child, then return $\text{Minimum}(T\.right)$

If $T$ has no right child, and is a left child, then return $T\.parent$

If $T$ has no right child and is a right child, then traverse up until a left child is found - then this node’s parent.

Else $T$ has no successor.

\[ \text{Successor}(T) = O(\text{height}) \]
Inserting an Element into a BST

\[ \text{Insert}(T, x) \]

\[
\begin{align*}
\text{if } & T \text{ then} \\
& \begin{cases} \\
& \text{if } T.\text{key} \leq x \text{ then} \\
& \quad \text{Insert}(T.\text{left}, x) \\
& \text{else} \\
& \quad \text{Insert}(T.\text{right}, x) \\
& \end{cases} \\
\text{else} \\
& T \leftarrow \text{NewNode}(x). \\
\end{align*}
\]

- \text{Insert} is a lot like \text{Search}.
- \text{Insert}(T, x) = O(\text{height})
To build a BST, **INSERT** $n$ random elements in order to an empty BST.

It takes $O(n \log n)$ to build a BST from such a set of random elements.

Run Insert $n$ times.

$\sum_{i=1}^{n} \log i = n \log n$

Expected height $= O(\log n)$
To build a BST, **INSERT** \( n \) random elements in order to an empty BST.

It takes \( O(n \log n) \) to build a BST from such a set of random elements.

Run Insert \( n \) times.

\[
\sum_{i=1}^{n} \log i = n \log n
\]

Expected height = \( O(\log n) \)

Can a BST be constructed in less than \( O(n \log n) \)?
To build a BST, Insert \( n \) random elements in order to an empty BST.

It takes \( O(n \log n) \) to build a BST from such a set of random elements.

Run Insert \( n \) times.

\[
\sum_{i=1}^{n} \log i = n \log n
\]

Expected height = \( O(\log n) \)

Can a BST be constructed in less than \( O(n \log n) \)?

No. It’s equivalent to a comparison sort. Since Comparison sorting is \( \Omega(n \log n) \), constructing a BST is \( \Omega(n \log n) \).
**Delete**($T, v$)

If $v$ is a leaf, delete $v$.
If $v$ has 1 child, delete $v$, replace $v$ with its child.
If $v$ has 2 children, swap $v$ with **Successor**($v$), then **Delete**($v$).

- How long does each case take?
\textsc{Delete}(T,v)

- If \(v\) is a leaf, delete \(v\).
- If \(v\) has 1 child, delete \(v\), replace \(v\) with its child.
- If \(v\) has 2 children, swap \(v\) with \textsc{Successor}(\(v\)), then \textsc{Delete}(\(v\)).

- How long does each case take?
- How can we be sure \textsc{Delete}(\(v\)) terminates?
\textbf{DELETE}(T,v)

If \( v \) is a leaf, delete \( v \).
If \( v \) has 1 child, delete \( v \), replace \( v \) with its child.
If \( v \) has 2 children, swap \( v \) with \textbf{Successor}(v), then \textbf{DELETE}(v).

- How long does each case take?
- How can we be sure \textbf{DELETE}(v) terminates?
- Show that this holds the BST properties.
Recap

- Binary Search Trees are an efficient, simple dictionary data structure.
- Construction $O(n \log n)$
- Insertion $O(\log n)$
- Search $O(\log n)$
- Deletion $O(\log n)$
- Binary Search Trees are sorted representations of data.
Next time (8/1)
- Heaps.

For Next Class
- Homework 3 Due.
- Read Sections 8.1, 8.2, 8.3, 8.4