Last Time

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

```
root 10
    5 20
     8
   1
```
Binary Search Tree Example
Not a Binary Search Tree Example

root

null

10

5

20

1

8
Not a Binary Search Tree Example

root 10
null

5 20 8

1
• The height of a BST is at most $n - 1$.
• The height of a BST is at least $\log n$. 
Sorting with a BST

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

**`TRAVERSE(T)`**

```
if T.root then
   TRAVERSE(T.left)
   PRINT T.key
   TRAVERSE(T.right)
end if
```
Binary Search Trees are sorted.

Constructing a sorted array is \( \Theta(n) \)

\[
\text{TRaverse}(T)
\]

\[
\text{if } T.\text{root} \text{ then}
\]
\[
\text{TRaverse}(T.\text{left})
\]
\[
\text{PRINT } T.\text{key}
\]
\[
\text{TRaverse}(T.\text{right})
\]
\[
\text{end if}
\]

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

**Search**\( (T, x) \)

\[
\text{if } T \text{ then}
\]
\[
\quad \text{if } T\.\text{key} = x \text{ then}
\]
\[
\quad \quad \text{return } T
\]
\[
\quad \text{else}
\]
\[
\quad \quad \text{if } T\.\text{key} < x \text{ then}
\]
\[
\quad \quad \quad \text{return } \text{Search}(T\.\text{left}, x)
\]
\[
\quad \quad \text{else}
\]
\[
\quad \quad \quad \text{return } \text{Search}(T\.\text{right}, x)
\]
\[
\quad \text{end if}
\]
\[
\text{end if}
\]
\[
\text{else}
\]
\[
\quad \text{return } \text{null}
\]
\[
\text{end if}
\]
Searching a BST

- $\text{Search}(T,x)$ 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

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

**Successor**\( (T) \)

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

If \( T \) has no right child, and is a left child, then return \( T.\text{parent} \)

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.

- **\texttt{Successor}(T) = O(\text{height})**
Inserting an Element into a BST

**Insert(T,x)**

if T then
  if T.key ≤ x then
    Insert(T.left, x)
  else
    Insert(T.right, x)
  end if
else
  T ← NewNode(x).
end if

- Insert is a lot like Search.
- 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 \( = O(\log n) \)
Building a BST

- 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)

**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?
$\text{DELETE}(T, v)$

- 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 $\text{SUCCESSOR}(v)$, then $\text{DELETE}(v)$.

- How long does each case take?
- How can we be sure $\text{DELETE}(v)$ terminates?
**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 $\text{Successor}(v)$, then $\text{DELETE}(v)$.

- How long does each case take?
- How can we be sure $\text{DELETE}(v)$ 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.
Bye

- Next time
  - Heaps.