Chapter 2: Basic Data Structures
Basic Data Structures

- Stacks
- Queues
- Vectors, Linked Lists
- Trees (Including Balanced Trees)
- Priority Queues and Heaps
- Dictionaries and Hash Tables
Two Definitions

Depth of a node $v$ in a tree is:
- 0 if $v$ is root, else
- $1 + \text{depth of parent of } v$

Height of a node $v$ is:
- 0 if external node
- $1 + \text{max height of child of } v$

Height of a tree is the maximum depth of an external node of the tree + 1
- Equivalently, maximum depth among all nodes + 1
- If we don’t add the one, then tree with height $h$ has $h+1$ levels!
Heaps (§2.4.3)

A heap is a binary tree storing keys at its internal nodes and satisfying the following properties:

- **Heap-Order:** for every internal node \( v \) other than the root, \( key(v) \geq key(parent(v)) \)

- **Complete Binary Tree:** let \( h \) be the height of the heap
  - for \( i = 0, \ldots, h - 2 \), there are \( 2^i \) nodes of depth \( i \)
  - at depth \( h-1 \), the internal nodes are to the left of the external nodes

The **last node** of a heap is the rightmost internal node of depth \( h-1 \).
Height of a Heap (§2.4.3)

Theorem: A heap storing \( n \) keys has height \( O(\log n) \)

Proof: (we apply the complete binary tree property)

- Let \( h \) be the height of a heap storing \( n \) keys
- Since there are \( 2^i \) keys at depth \( i = 0, \ldots, h - 2 \) and at least one key at depth \( h - 1 \), we have
  \[
  n \geq 1 + 2 + 4 + \ldots + 2^{h-2} + 1 = 2^{h-1} - 1 + 1 = 2^{h-1}
  \]
- Thus, \( n \geq 2^{h-1} \), i.e., \( h \leq (\log n) + 1 \)

<table>
<thead>
<tr>
<th>depth</th>
<th>keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>( h-2 )</td>
<td>( 2^{h-2} )</td>
</tr>
<tr>
<td>( h-1 )</td>
<td>1</td>
</tr>
</tbody>
</table>
Insertion into a Heap (§2.4.3)

- The insertion algorithm consists of three steps
  - Find the insertion node $z$ (the new last node)
  - Store $k$ at $z$ and expand $z$ into an internal node
  - Restore the heap-order property (discussed next)
Upheap

- After the insertion of a new key $k$, the heap-order property may be violated.
- Algorithm upheap restores the heap-order property by swapping $k$ along an upward path from the insertion node.
- Upheap terminates when the key $k$ reaches the root or a node whose parent has a key smaller than or equal to $k$.
- Since a heap has height $O(\log n)$, upheap runs in $O(\log n)$ time.
Removal from a Heap (§2.4.3)

- Method `removeMin` of the priority queue (ADT) corresponds to the removal of the root key from the heap.

- The removal algorithm consists of three steps:
  - Replace the root key with the key of the last node \( w \).
  - Compress \( w \) and its children into a leaf.
  - Restore the heap-order property (discussed next).
Downheap

- After replacing the root key with the key \( k \) of the last node, the heap-order property may be violated.
- Algorithm downheap restores the heap-order property by swapping key \( k \) along a downward path from the root.
- Downheap terminates when key \( k \) reaches a leaf or a node whose children have keys greater than or equal to \( k \).
- Since a heap has height \( O(\log n) \), downheap runs in \( O(\log n) \) time.
Heap-Sort (§2.4.4)

Consider a priority queue with $n$ items implemented by means of a heap
- the space used is $O(n)$
- methods `insertItem` and `removeMin` take $O(\log n)$ time
- methods `size`, `isEmpty`, `minKey`, and `minElement` take time $O(1)$ time

Using a heap-based priority queue, we can sort a sequence of $n$ elements in $O(n \log n)$ time

The resulting algorithm is called heap-sort

Heap-sort is much faster than quadratic sorting algorithms, such as insertion-sort and selection-sort
Vector-based Heap Implementation (§2.4.3)

- We can represent a heap with $n$ keys by means of a vector of length $n + 1$.
- For the node at rank (index) $i$
  - the left child is at rank $2i$
  - the right child is at rank $2i + 1$
- Links between nodes are not explicitly stored.
- The leaves are not represented.
- The cell of at rank 0 is not used.
- Operation `insertItem` corresponds to inserting at rank $n + 1$.
- Operation `removeMin` corresponds to removing at rank $n$ (and placing in position 1).
- Yields in-place heap-sort.
Building a Heap with n nodes

- Just perform n insertions: $O(n \log n)$
- If you know the values of all keys beforehand, can be done in $O(n)$ time (see next few slides)
Merging Two Heaps

- We are given two heaps and a key $k$.
- We create a new heap with the root node storing $k$ and with the two heaps as subtrees.
- We perform downheap to restore the heap-order property.
We can construct in $O(n)$ time a heap storing $n$ given keys by using a bottom-up construction with $\log n$ phases.

In phase $i$, pairs of heaps with $2^i - 1$ keys are merged into heaps with $2^{i+1} - 1$ keys.
Example
Example (contd.)
Example (end)
Consider an internal node $v$ in the completed heap, and in particular the subtree (in the heap) rooted at $v$. The time to build this subtree (given the completed subtrees rooted at each child of $v$) is proportional to the height of the subtree.

Now consider the path that starts at $v$, and then goes first through the right child of $v$, then repeatedly goes through left children until the bottom of the heap (this path may differ from the actual downheap path) (This goes from $v$ to its inorder external successor).

The length of this path equals the height of the subtree rooted at $T$.

Thus the time cost of constructing the heap is equal to the sum of the lengths of these paths (one path for each $v$).
Analysis

- For any distinct internal nodes, these “proxy paths” share no edges (though they may share nodes).
- Thus the sum of the lengths of the paths is no more than the number of edges in the heap, which is clearly less than $2n$.
- Thus, bottom-up heap construction runs in $O(n)$ time.
- Bottom-up heap construction is thus faster than $n$ successive insertions, provided we have all of the key values prior to building the heap.
Binary Search
Dictionary ADT

A dictionary is simply a data structure that stores key value pairs (like a paper dictionary).

- Ex. hashTable
Binary Search

- Binary search performs operation \texttt{findElement}(k) on a dictionary implemented by means of an array-based sequence, sorted by key
  - similar to looking through a phone book for a name
  - at each step, the number of candidate items is halved
  - terminates after $O(\log n)$ steps

- Example: \texttt{findElement}(7)
A lookup table is a dictionary implemented by means of a sorted sequence
- We store the items of the dictionary in an array-based sequence, sorted by key
- We use an external comparator for the keys

Performance:
- `findElement` takes $O(\log n)$ time, using binary search
- `insertItem` takes $O(n)$ time since in the worst case we have to shift $n$ items to make room for the new item
- `removeElement` take $O(n)$ time since in the worst case we have to shift $n$ items to compact the items after the removal

The lookup table is effective only for dictionaries of small size or for dictionaries on which searches are the most common operations, while insertions and removals are rarely performed (e.g., credit card authorizations)
Now for the Data Structures

Pretty much all variations on the binary search tree theme
Binary Search Tree

- A binary search tree is a binary tree storing keys (or key-element pairs) at its internal nodes and satisfying the following property:
  - Let $u$, $v$, and $w$ be three nodes such that $u$ is in the left subtree of $v$ and $w$ is in the right subtree of $v$. We have $key(u) \leq key(v) \leq key(w)$
- External nodes do not store items

- An inorder traversal of a binary search tree visits the keys in increasing order

![Binary Search Tree Diagram]
Search

- To search for a key \( k \), we trace a downward path starting at the root.
- The next node visited depends on the outcome of the comparison of \( k \) with the key of the current node.
- If we reach a leaf, the key is not found and we return NO_SUCH_KEY.
- Example: `findElement(4)`

**Algorithm `findElement(k, v)`**

```python
if T.isExternal(v):
    return NO_SUCH_KEY
if k < key(v):
    return findElement(k, T.leftChild(v))
else if k = key(v):
    return element(v)
else:
    k > key(v)
    return findElement(k, T.rightChild(v))
```
Consider a dictionary with $n$ items implemented by means of a binary search tree of height $h$

- the space used is $O(n)$
- methods `findElement`, `insertItem` and `removeElement` take $O(h)$ time

The height $h$ is $O(n)$ in the worst case and $O(\log n)$ in the best case
AVL Trees

Diagram of an AVL tree with nodes 3, 4, 6, and 8.
AVL Tree Definition

- **AVL trees are balanced.**
- An AVL Tree is a **binary search tree** such that for every internal node $v$ of $T$, the heights of the children of $v$ can differ by at most 1.

An example of an AVL tree where the heights are shown next to the nodes:
Height of an AVL Tree

**Fact**: The height of an AVL tree storing $n$ keys is $O(\log n)$.

**Proof**: First, we’re trying to bound the height of a tree with $n$ keys. Let’s come at this the reverse way, and find the lower bound on the minimum number of internal nodes, $n(h)$, that a tree with height $h$ must have.

- Claim that $n(h)$ grows exponentially. From this it will be easy to show that height of tree with $n$ nodes is $O(\log n)$.

So, formally, let $n(h)$ be the minimum number of internal nodes of an AVL tree of height $h$.

We easily see that $n(1) = 1$ and $n(2) = 2$. 

Height of an AVL Tree

For $h > 2$, an AVL tree of height $h$ with minimal number of nodes contains the root node, one AVL subtree of height $h-1$ and another of height $h-2$.

That is, $n(h) = 1 + n(h-1) + n(h-2)$

Knowing $n(h-1) > n(h-2)$, we get $n(h) > 2n(h-2)$. This indicates that $n(h)$ at least doubles every time we increase $h$ by two, which intuitively says that $n(h)$ grows at least exponentially. Let’s show this...

By induction, $n(h) > 2n(h-2)$, $n(h) > 4n(h-4)$, $n(h) > 8n(n-6)$, ..., which implies that $n(h) > 2^i n(h-2i)$

- This holds for any positive integer $i$ such that $h-2i$ is positive.

So, choose $i$ so that we get a base case. That is, we want $i$ such that either $h-2i = 1$ or $h-2i = 2$. I.e. $i = (h/2) - 1$ or $i = (h/2) - (1/2)$. 
Height of an AVL Tree

Thus the \( i \) we seek is \( \left\lceil \frac{h}{2} \right\rceil - 1 \). So, we have

\[
n(h) > 2^{\left\lceil \frac{h}{2} \right\rceil - 1} n(1 \text{ or } 2) > 2^{\frac{h}{2} - 1}.
\]

- Taking logarithms: \( h < 2 \log n(h) + 2 \)
- But \( n(h) \) is the **minimum** number of internal nodes of an AVL tree with height \( h \). So if an AVL tree has height \( h \) and \( n \) nodes, we have \( n(h) \leq n \), so that: \( h < 2 \log n + 2 \)
- Thus the height of an AVL tree is \( O(\log n) \)
Where Are We?

- Ok, so the height of an AVL tree is $O(\log n)$. This is nice...
  - We can search in time $O(\log n)$.
- But...
  - Can we insert and delete nodes from tree in time $O(\log n)$? Remember, we need to preserve the AVL height balancing property!
Yes, we can!

And you’re welcome to read about it in the text and the next few slides....
Yes, we can!

And you’re welcome to read about it in the text and the next few slides….or not.
Insertion in an AVL Tree

- Insertion is as in a binary search tree (may violate AVL property!)
- Always done by expanding an external node.
- Example:

Before insertion of 54:

```
17
  |   
32  48  50  62  88
```

After insertion:

```
17
  |   
32  50  48  62  88
```

Note that only nodes on path from w to root can now be unbalanced.
Restoring Balance

- We use a “search and repair” strategy
- Let z be the first unbalanced node encountered on the path from w up toward the root.
- Let y be the child of z with a higher height (not y must be an ancestor of w), and x be the child of y with a higher height (in case of a tie, go with the child that is an ancestor of w).
Trinode Restructuring

- let \((a,b,c)\) be an inorder listing of \(x, y, z\)
- perform the rotations needed to make \(b\) the topmost node of the three

Case 1: Single rotation
(a left rotation about \(a\))

Case 2: Double rotation
(a right rotation about \(c\), then a left rotation about \(a\))

(other two cases are symmetrical)
Insertion Example, continued

unbalanced...

...balanced
Restructuring (as Single Rotations)

Single Rotations:
Restructuring (as Double)

- double rotations:
Removal in an AVL Tree

- Removal begins as in a binary search tree, which means the node removed will become an empty external node. Its parent, \( w \), may cause an imbalance.
- Example:
Rebalancing after a Removal

Let $z$ be the first unbalanced node encountered while travelling up the tree from $w$. Also, let $y$ be the child of $z$ with the larger height, and let $x$ be the child of $y$ with the larger height.

We perform $\text{restructure}(x)$ to restore balance at $z$.

As this restructuring may upset the balance of another node higher in the tree, we must continue checking for balance until the root of $T$ is reached.
Running Times for

- a single restructure is $O(1)$
  - using a linked-structure binary tree
- find is $O(\log n)$
  - height of tree is $O(\log n)$, no restructures needed
- insert is $O(\log n)$
  - initial find is $O(\log n)$
  - Restructuring up the tree, maintaining heights is $O(\log n)$
- remove is $O(\log n)$
  - initial find is $O(\log n)$
  - Restructuring up the tree, maintaining heights is $O(\log n)$