Indexing algorithms
We have already spoken of a table having an index.

An index is a data structure – created and maintained within a database system – that can greatly reduce the time needed to locate records.

```
CREATE INDEX ind1 ON my_table (my_column)
```

- *IA Algorithms* presents useful data structures for implementing database indices (search trees, hash tables, and so on).
- **While an index can speed up reads, it will slow down updates.** In some cases it is better to store read-oriented data in a separate database optimised for that purpose.
```
SELECT *  
FROM movies  
WHERE year > 2015

SLOW METHOD  
Scan through all rows of the movies table and pick out those that match

FAST METHOD  
```
cursor = ind1.search_gt(2015)
while not ind1.at_end(cursor):
    m_id = cursor.movie_id
    m = movies.primary_key.search(m_id)
    print(m)
    cursor = ind1.next(cursor)
```
AbstractDataType Index:
# Holds a collection of (key,value) pairs, where there is an ordering on keys.
# Typically, values are small, e.g. pointers to objects in memory.

# Find a key (if it exists) and return a cursor.
# This cursor lets us access the (key,value) we found.
Cursor search(Key k)
Cursor search_gt(Key k)

# Move the cursor; and test if it’s gone past the end of the data.
# (We may also wish to support min() and max() operations.)
Cursor next(Cursor c)
Cursor prev(Cursor c)
bool at_end(Cursor c)

# Modify the contents of the data structure
insert(Key k, Value v)
delete(Key k)

NOTE. Sensible database indexes allow multiple items with the same key. But for consistency with notes & textbook, we’ll assume keys are unique.

cursor = ind1.search_gt(2015)
while not ind1.at_end(cursor):
    m_id = cursor.movie_id
    m = movies.primary_key.search(m_id)
    print(m)
    cursor = ind1.next(cursor)
SORTED ARRAY

An array of \( n \) (key,value) records, sorted by key
- search is fast, \( O(\log n) \), using repeated bisection
- next is trivial, \( O(1) \)
- insert/delete are slow, \( O(n) \)

BALANCED BINARY SEARCH TREE

Each node stores a (key,value) record, call it \((k, v)\)
- Its left subtree consists of records \((k', v')\) with \( k' < k \)
- Its right subtree consists of records \((k', v')\) with \( k' > k \)

Subtree sizes are balanced
SECTION 4.3

Binary search trees
search($k$)
Start at $x$=root node.
If $x$.key=$k$, we’re done.
Otherwise, set $x \leftarrow x$.left or $x \leftarrow x$.right as appropriate, and repeat.

---

**BALANCED BINARY SEARCH TREE**

Each node stores a (key,value) record, call it $(k, v)$
- Its left subtree consists of records $(k', v')$ with $k' < k$
- Its right subtree consists of records $(k', v')$ with $k' > k$
- Subtree sizes are balanced
Each node stores a (key,value) record, call it \((k,v)\)

- Its left subtree consists of records \((k',v')\) with \(k' < k\)
- Its right subtree consists of records \((k',v')\) with \(k' > k\)

Subtree sizes are balanced.
**Balanced Binary Search Tree**

Each node stores a (key,value) record, call it \((k,v)\)
- Its left subtree consists of records \((k',v')\) with \(k' < k\)
- Its right subtree consists of records \((k',v')\) with \(k' > k\)
- Subtree sizes are balanced

*Exercise.* \(O(n)\) to rebalance the tree.

Horrid! Insert is easy enough if we don’t mind an *unbalanced* tree, but *balanced* makes it very tough.

Delete is fiddly, even in an unbalanced tree.
Free-form Binary Search Tree

A binary search tree as before, but we won’t require subtrees to be balanced.

**Crunch-time Charlie**
(quick and dirty, too harried to learn)

**Question.** Where should we insert A? What’s the procedure for insertion?

insert\((k, v)\)

\(x \leftarrow \text{search}(k)\) and if search fails then let \(x\) be the last node searched.

If search fails, create a new node \((k, v)\) and set it to be a child of \(x\).

If search succeeded, update \(x . \text{val} \leftarrow v\)
FREE-FORM BINARY SEARCH TREE
A binary search tree as before, but we won’t require subtrees to be balanced.

Crunch-time Charlie (quick and dirty, too harried to learn)

**QUESTION. How do we delete node S?**

**QUESTION. How do we delete node N?**

**delete(k)**
Deleting a leaf node is easy.

To delete a node with one child, replace it by its child.

If there are two children: find the successor s, delete it, overwrite k’s node with s.
<table>
<thead>
<tr>
<th></th>
<th>Insert/Delete</th>
<th>Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>Balanced BST</td>
<td>Horrid $O(n)$</td>
<td>$O(\text{height})$</td>
</tr>
<tr>
<td>Unbalanced BST</td>
<td>$O(\text{height})$</td>
<td>$\Omega(n)$</td>
</tr>
</tbody>
</table>

... for an index with $n$ items