Algorithms 1

SECTIONS 4.3–4.6

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.

movies

movie_id	title	year
0126029	Shrek	2001
0181689	Minority Report	2002
0212720	A.I. Artificial Intelligence	2001
0983193	The Adventures of Tintin	2011
➡ 4975722	Moonlight	2016
➡ 5010201	Dunkirk	2017
➡ 5012394	Maigret Sets a Trap	2016

```
CREATE INDEX ind1 ON movies (year)
```

year	movie_id
2001	0126029
2001	0212720
2002	0181689
2011	0983193
➡ 2016	5012394
2016	4975722
2017	5010201

```
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)
```

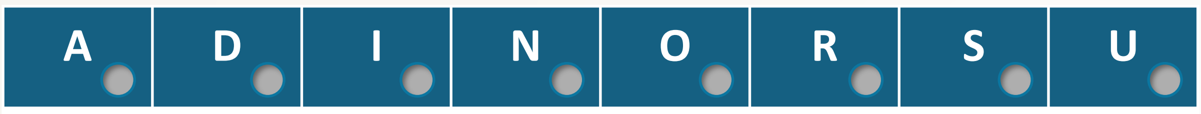
```
AbstractDataType Index:
    # Holds a collection of (key,value) pairs, where there is an ordering on keys.
    # Typically, values are small, e.g. pointers to objects in memory.

    # 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)
    etc.

    # 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)
```

Fastidious Frances (everything pristine all of the time)

SORTED ARRAY



An array of n (key,value) records, sorted by key

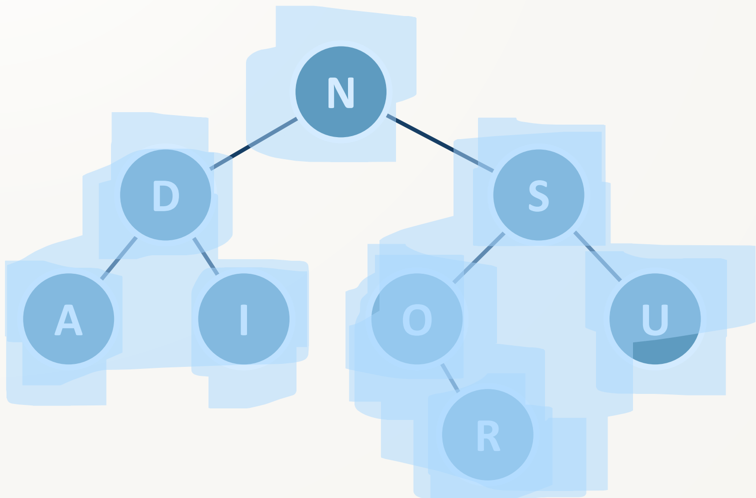
e.g. value is a pointer to a record stored elsewhere

- 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

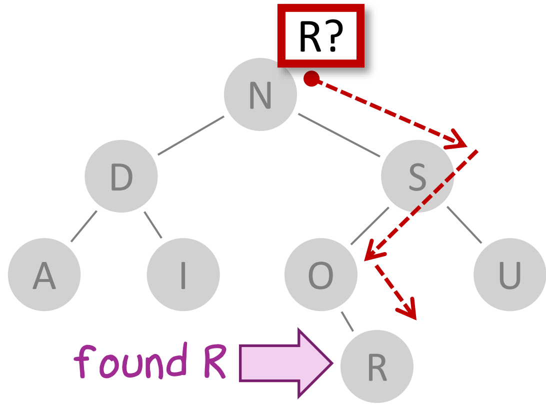
Binary search trees

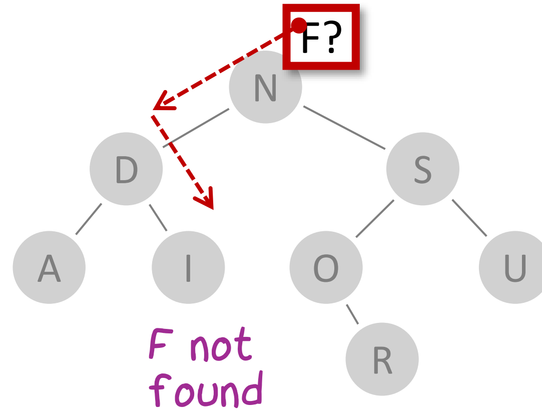
`search(`k`)`

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.





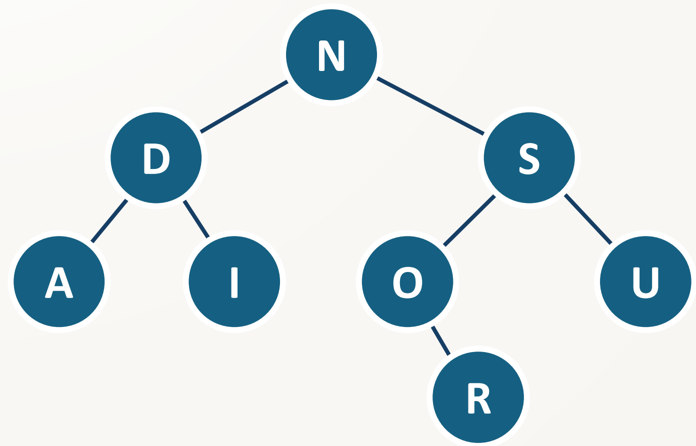
~~BALANCED~~ BINARY SEARCH TREE

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 $k' > k$

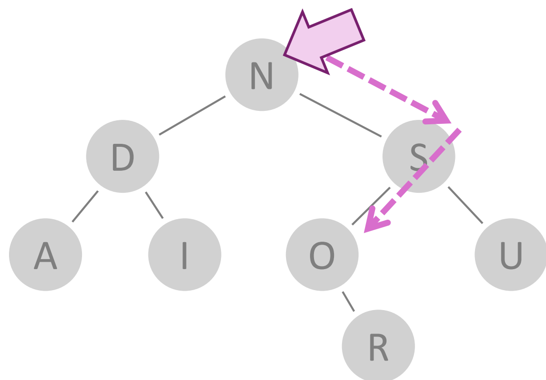
~~Subtree sizes are balanced~~



`next(`x`)`

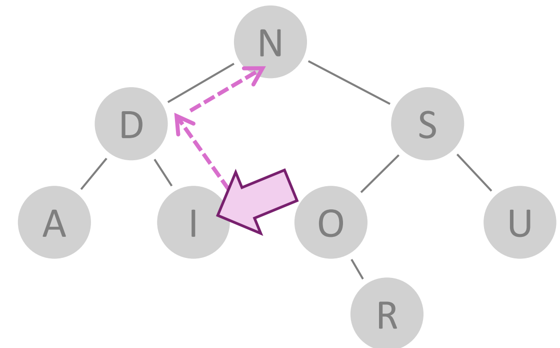
QUESTION. What's the next item after N? What's the procedure to find it?

If x has a right-child, take it, then go **down-left** as far as possible.



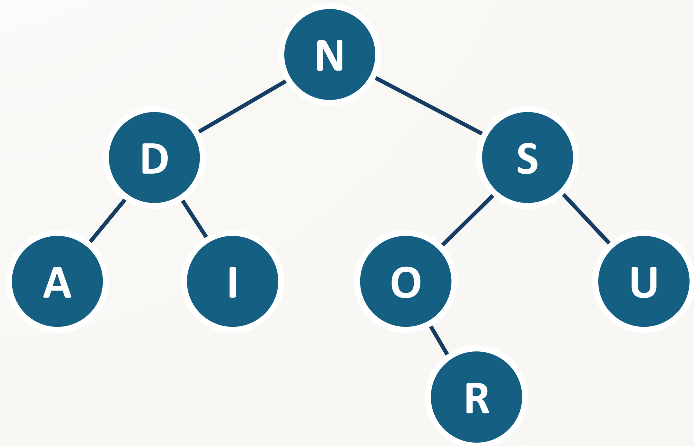
QUESTION. What's the next item after I? What's the procedure to find it?

If x has no right-child, go up until our **first up-right**. (If we reach the root without an up-right, we're at the end.)



~~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~~



`insert`(k,v), `delete`(k)

Exercise.

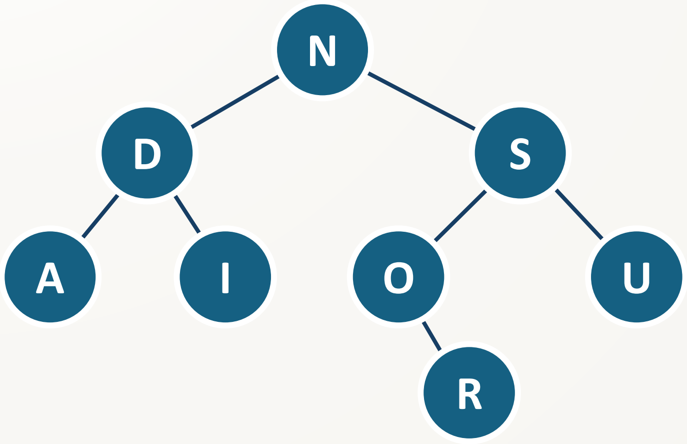
Horrid! O(n) to rebalance the tree.

`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.

BALANCED BINARY SEARCH TREE

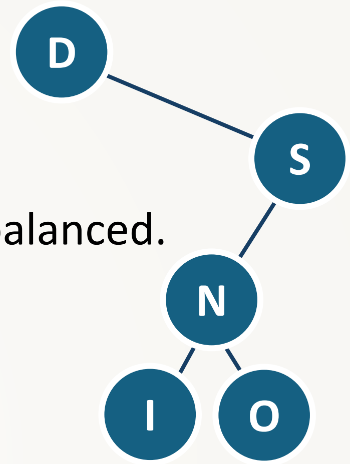
Each node stores a (key,value) record, call it (k, v)
Its left subtree consists of records (k', v') with $k' < k$
Its right subtree consists of records (k', v') with $k' > k$
Subtree sizes are balanced



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

FREE-FORM BINARY SEARCH TREE

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



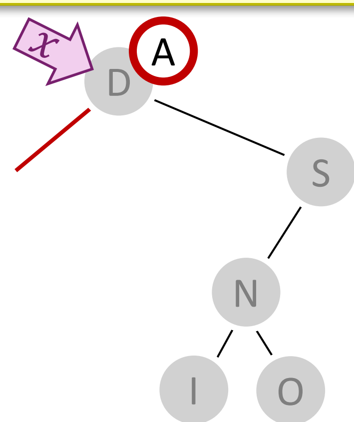
QUESTION. Where should we insert A? What's the procedure for insertion?

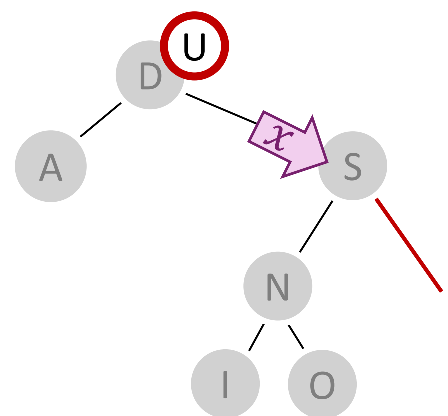
insert(k, v)

$x \leftarrow$ 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.val $\leftarrow v$





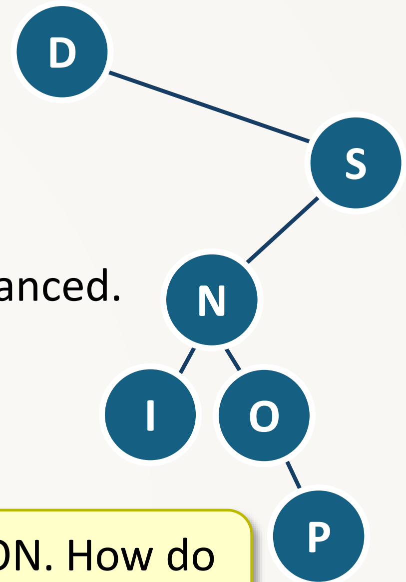
Crunch-time Charlie
(quick and dirty,
too harried to learn)

D I N O P S

Look at the successor of N, which is O.
If we can extract O from the tree, then put in N's spot, we won't mess up any of the ordering relations, since N and O are next to each other in the order.

FREE-FORM BINARY SEARCH TREE

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

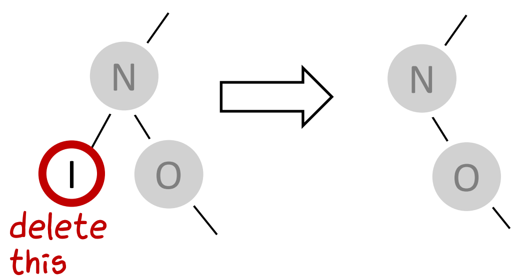


QUESTION. How do we delete node S?

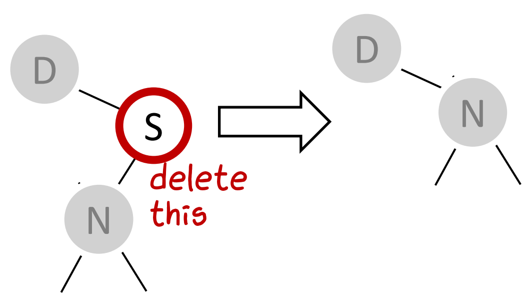
QUESTION. How do we delete node N?

`delete(`k`)`

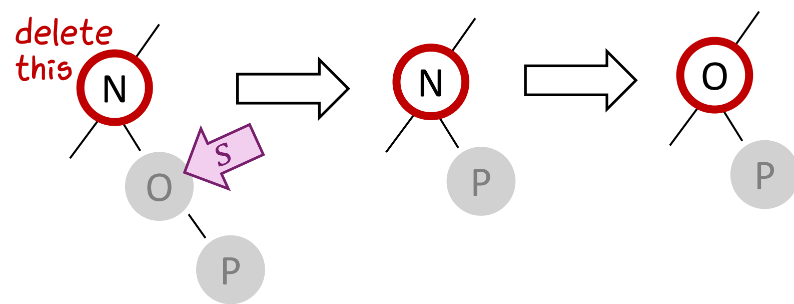
Deleting a leaf node is easy.

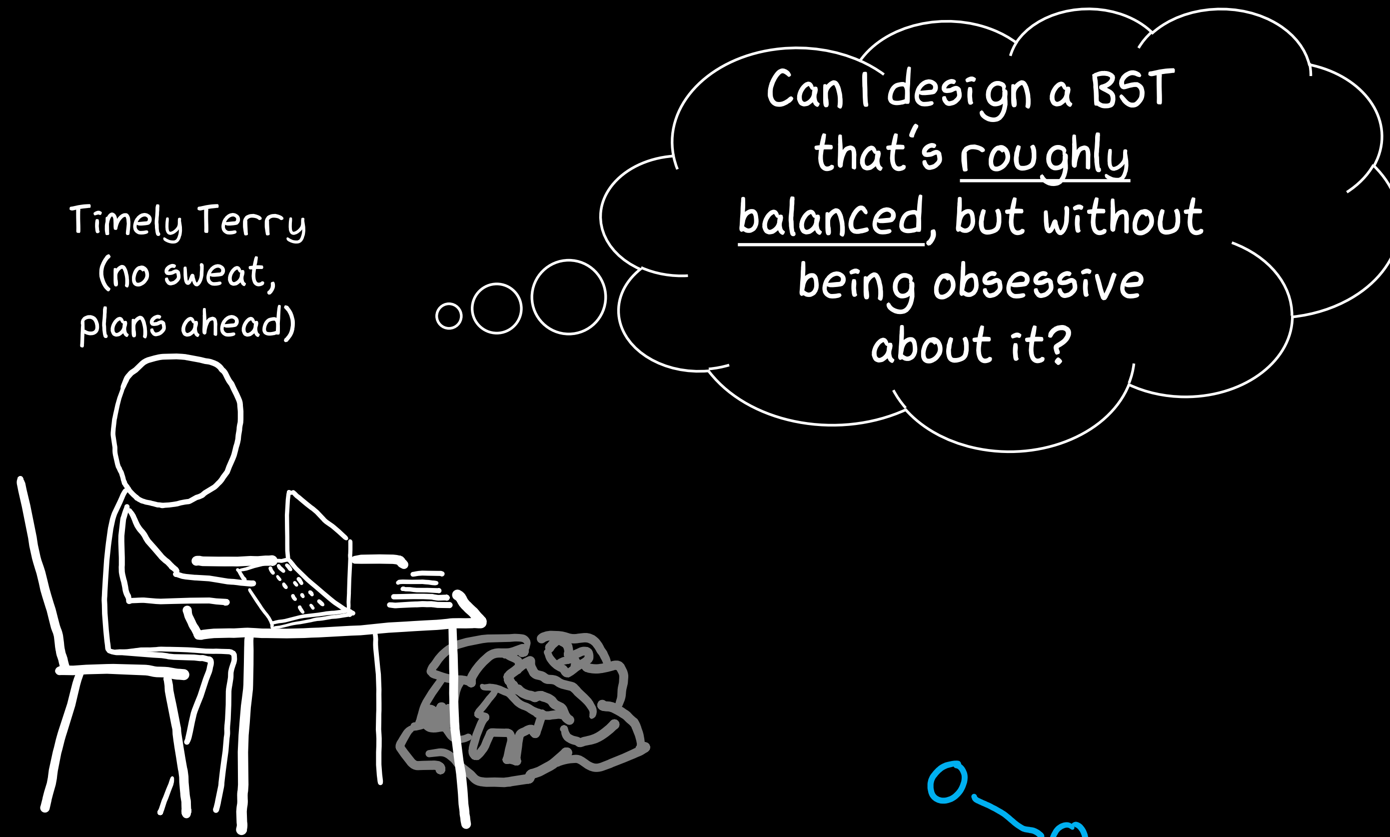


To delete a node with one child, replace it by its child.



If there are two children: find the successor s, delete it, overwrite k's node with s.





	insert/delete	search
balanced BST	horrid $O(n)$	O(height)
unbalanced BST	O(height)	

worst-case height is $O(\log n)$

worst-case height is $\Omega(n)$

… for an index with n items