Trees

Searching

Suppose we want to search for things in a list

One possibility is to keep the items in a 'randomly' ordered list, so insertion is O(1), but then a search takes O(n) time

Or, we could keep them in a sorted list, in which case we can use a binary search which takes O(log(n)) time, but then new items would have to be added in the middle, which takes O(n) time

Trees

When there is a mixture of search and insert operations, and both operations need to be well below O(n), then the items can usefully be stored in an ordered binary tree

A tree is created out of cells, with each cell having two pointers

Balancing

We will create ordered binary trees, without worrying about how well balanced the tree is

Balancing techniques include:

- Reorder or randomise the input data (assuming few updates)
- Occasional re-construction of the tree
- Use a self-balancing (red-black, AVL, 2-3, ...) tree

Tree structure

Here's a struct for holding one node in a tree of ints:

```
struct node {
    struct node *left;
    int key;
    struct node *right;
};
typedef struct node node;
```

This is essentially the same as

`Tree a = Tip | Node (Tree a) a (Tree a)`

in Haskell (using `NULL` for `Tip`)

New node

Here's a function to create a new node (a one-element tree):

```
node *new_node(int n) {
    node *p = malloc(sizeof(node));
    *p = (node) { NULL, n, NULL };
    return p;
}
```

Recursive Insertion

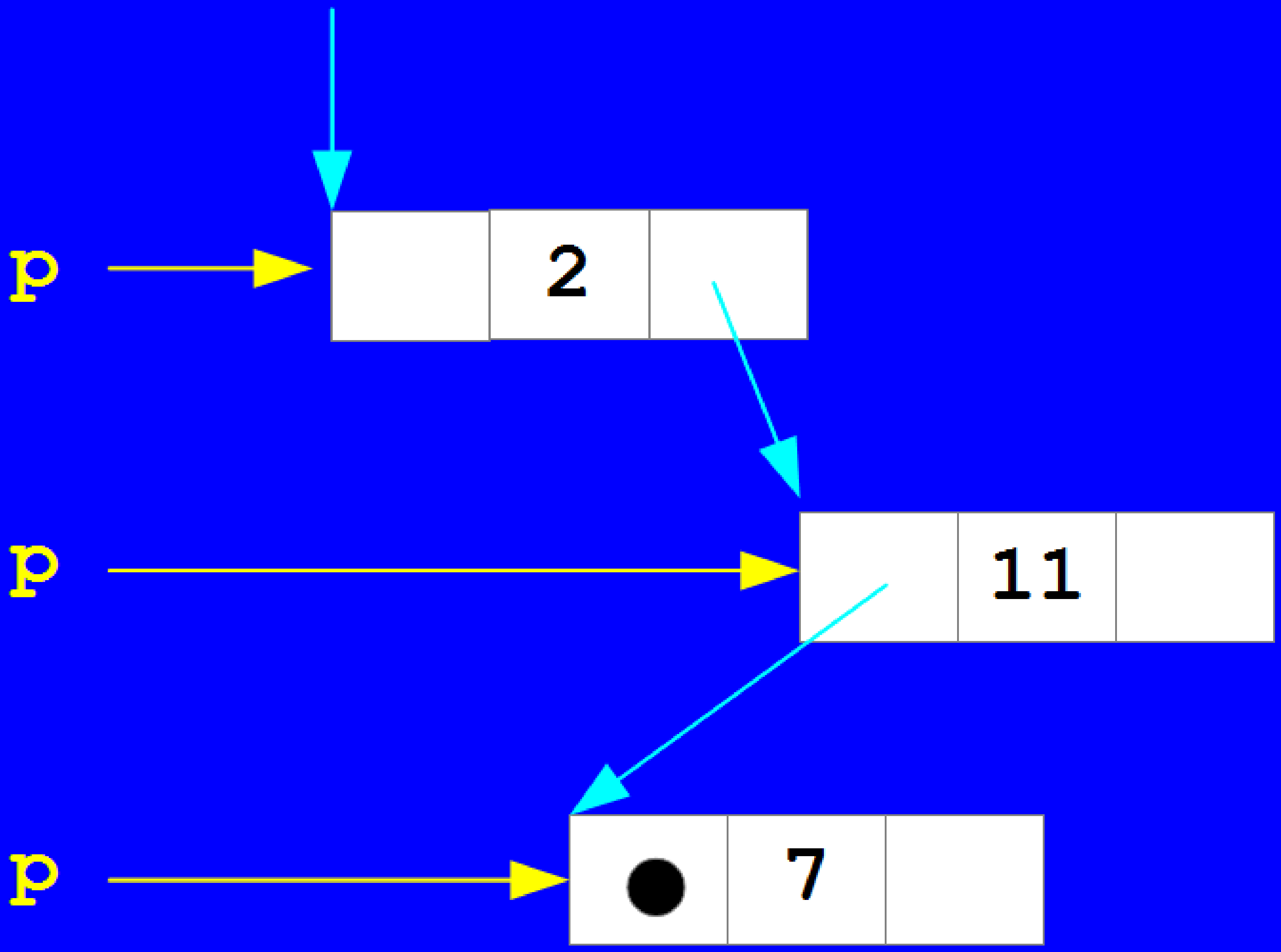
Here's a recursive insertion function:

```
node *insert_node(node *p, int n) {
    if (p == NULL) p = new_node(n);
    else if (n < p->key)
        p->left = insert_node(p->left, n);
    else if (n > p->key)
        p->right = insert_node(p->right, n);
    return p;
}
```

It uses p as a current-node variable

When you call it, it returns a possibly updated node, which you have to put back where you got it

Visualisation



p = NULL

Alternative Recursive Insertion

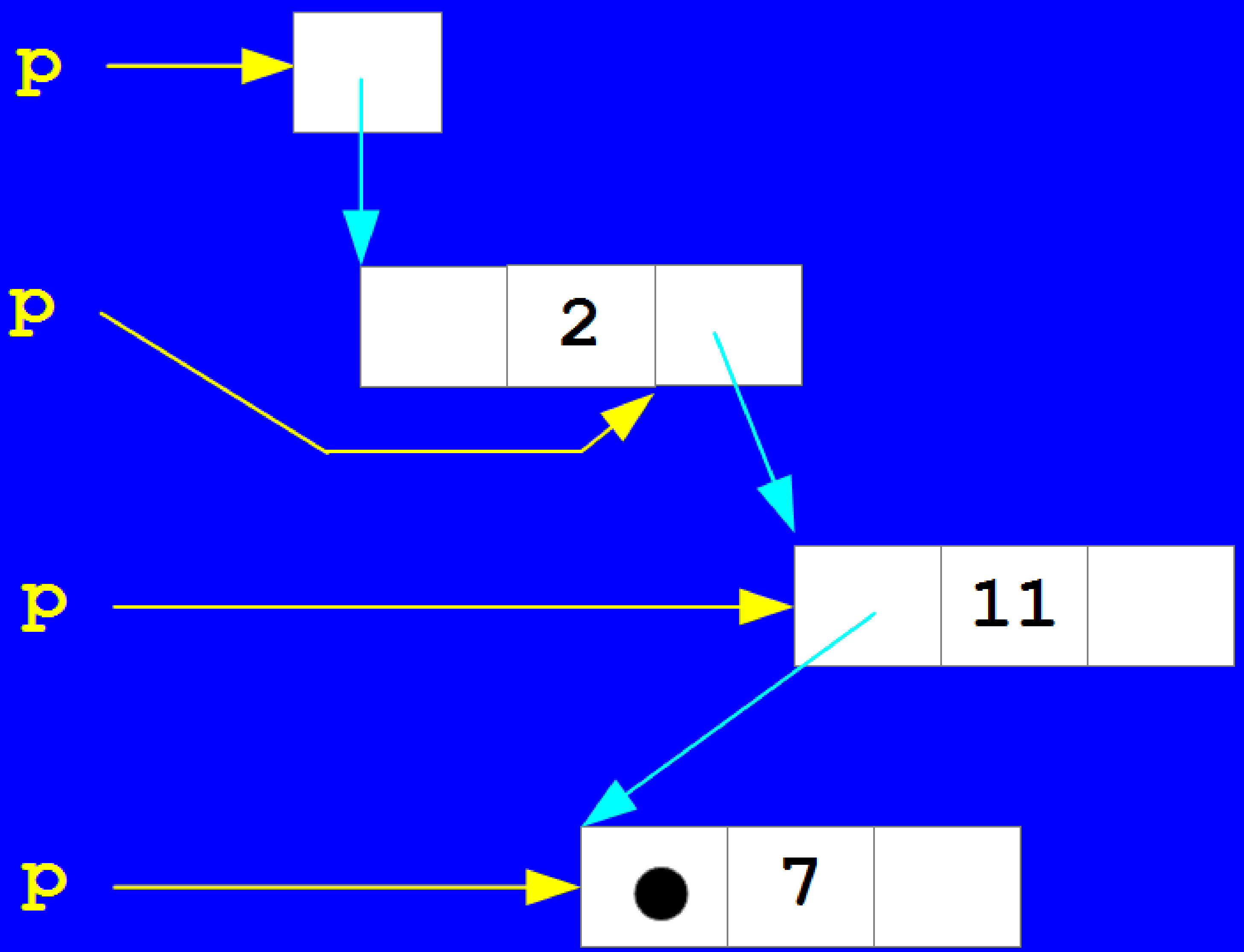
Here's a version which doesn't return anything, but uses a pointer to a pointer:

```
void insert_node(node **p, int n) {
    if (*p == NULL) *p = new_node(n);
    else if (n < (*p)->key)
        insert_node(&(*p)->left, n);
    else if (n > (*p)->key)
        insert_node(&(*p)->right, n);
}
```

It updates in place, and only does it once

When we reach the right place, we have a ***pointer*** to NULL, so we can replace it

Visualisation



Iterative Insertion

Here's a complicated iterative version:

```
node *insert_node(node *p, int n) {
    bool done = false;
    if (p == NULL) { p = new_node(n); done = true; }
    while (! done) {
        if (n == p->key) done = true;
        else if (n < p->key) {
           if (p->left != NULL) p = p->left;
           else { p->left = new_node(n); done = true; }
        }
        else {
            if (p->right != NULL) p = p->right;
            else { p->right = new_node(n); done = true; }
        }
    }
    return p;
}
```

Alternative Iterative Insertion

The structure is simpler using a pointer to a pointer:

```
void insert_node(node **p, int n) {
  bool done = false;
  while (! done) {
    if (*p == NULL) {
      *p = new_node(n); done = true;
    }
    else if (n == (*p)->key) done = true;
    else if (n < (*p)->key) p = &(*p)->left;
    else p = &(*p)->right;
  }
}
```

Choosing

Should you use recursive or iterative insertion, and should you use pointers-to-pointers or not?

Your can disregard what anybody says about efficiency - what matters is balance and complexity issues

Use whichever you like - but when you write functions which use ***both*** left and right subtrees instead of just one, recursion stays simple while iteration gets nastier

A wrapper

Functions on trees are inconvenient if we force callers to use the nodes directly, either they have to catch the output, or pass a pointer-to-a-pointer

So we need a wrapping structure for a tree:

```c
struct tree {
    struct node *root;
};
typedef struct tree tree;
```

This can also be a useful place to put global information about the tree

New tree

Here's a reasonable function to create a new tree:

```
tree *new_tree() {
    tree *t = malloc(sizeof(tree));
    t->root = NULL;
    return t;
}
```

Insertion

A wrapped insertion function might be:

```
void insert(tree *t, int n) {
    t->root = insert_node(t->root, n);
}
// ----- OR -----
void insert(tree *t, int n) {
    insert_node(&t->root, n);
}
```

In the iterative cases, the `insert` wrapper function can be combined with `insert_node` to form a single function

Recursive searching

Searching is a bit simpler, and can also be done recursively or iteratively – here's a recursive version:

```
node *find_node(node *p, int n) {
    if (p == NULL) { }
    else if (n < p->key)
        p = find_node(p->left, n);
    else if (n > p->key)
        p = find_node(p->right, n);
    return p;
}
```

Iterative searching

Here's an iterative version:

```
node *find_node(node *p, int n) {
    bool done = false;
    while (! done) {
        if (p == NULL) done = true;
        else if (n == p->key) done = true;
        else if (n < p->key) p = p->left;
        else p = p->right;
    }
    return p;
}
```

Wrapping

Again, we want a wrapper function

It shouldn't export any nodes to the user, it should just return (e.g.) a boolean to say whether the number is in the tree or not

```
bool contains(tree *t, int n) {
    return find_node(t, n) != NULL;
}
```

Maps

A *map* is a structure which maps keys to values

For example, when counting words, you might want to map word strings as keys, to integer counts as values:

```
struct node {
   struct node *left;
   char word[20];
   int count;
   struct node *right;
};
```

The tree would be structured according to the words, and functions would retrieve or update the counts

Self balancing trees

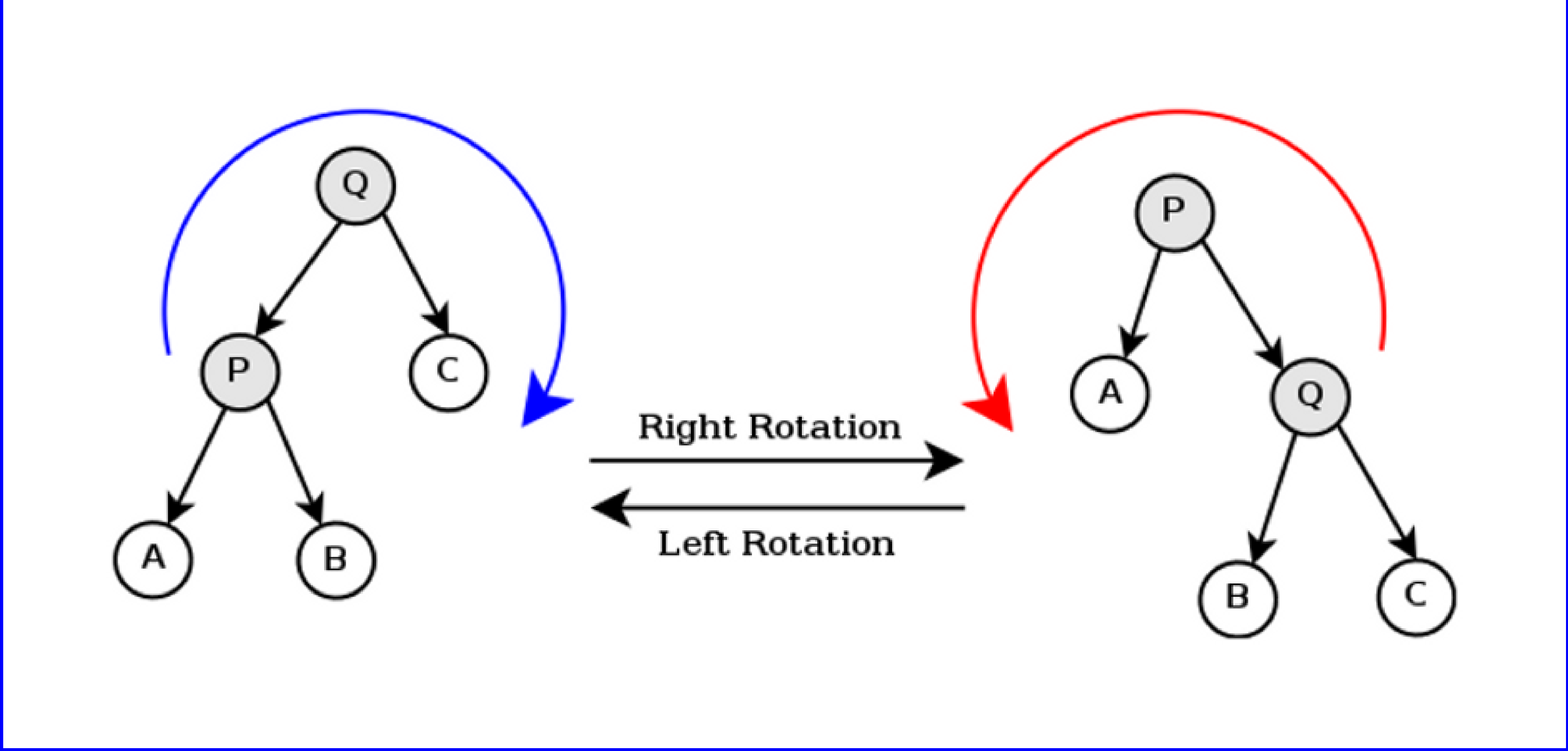
There are many types of self-balancing tree, with red-black trees being the most popular in libraries because you only need one extra bit per node

The different types (AVL trees, 2-3 trees, ...) all use the same ***mechanism*** but have different ***policies***

As Conor says "they are all morally the same"

Rotation

The mechanism used for balancing is rotation:



A rotate function

Here's a function to rotate right:

```
node *rotate_right(node *q) {
    node *p = q->left;
    q->left = p->right;
    p->right = q;
    return p;
}
```

A *policy* is an algorithm which decides what rotations to do and when, according to some extra info in each node, and which guarantees O(log(n)) depth