

COMP2521 19T0

Week 2, Thursday: Trees!

Jashank Jeremy

jashank.jeremy@unsw.edu.au

recursion
trees

COMP2521
19T0 lec04

cs2521@
jashankj@

Recursion

Linked Lists

DivConq

Trees

Recursion

A linked list can be described recursively!

```
struct node {  
    Item item;  
    node *next;  
};
```

“... this value, and the rest of the values”

```
size_t list_length (node *curr)
{
    if (curr == NULL) return 0;           // base case
    return 1 + list_length (curr->next);  // recursive case
}
```

```
int int_list_sum (intnode *curr)
{
    if (curr == NULL) return 0;           // base case
    return curr->item +
        int_list_sum (curr->next);        // recursive case
}
```

Recursive Linked Lists

Functions Amenable to Recursion (II)

```
void int_list_print (node *curr)
{
    if (curr == NULL) return;
    printf ("%d\n", curr->item);
    int_list_print (curr->next);
}

void int_list_print_reverse (node *curr)
{
    if (curr == NULL) return;
    int_list_print_reverse (curr->next);
    printf ("%d\n", curr->item);
}
```

Divide-and-Conquer, Recursively

REMINDER divide and conquer algorithms tend to:

- divide the input into parts,
- solve the problem on the parts recursively, then
- combine the results into an overall solution.

(This is a common 'big-data' approach: map-reduce.)

((“There’s no such thing as ‘big data’.”))

Divide-and-Conquer, Recursively

Maximum of an Unsorted Array (I)

Iteratively:

```
int array_max (int a[], size_t n)
{
    int max = a[0];
    for (size_t i = 0; i < n; i++)
        if (a[i] > max) max = a[i];
    return max;
}
```

complexity: $O(n)$

Divide-and-Conquer, Recursively

Maximum of an Unsorted Array (II)

Recursively:

```
int array_max (int a[], size_t l, size_t r)
{
    if (l == r) return a[l];
    int m = (l + r) / 2;
    int m1 = array_max (a, l, m);
    int m2 = array_max (a, m + 1, r);
    return (m1 < m2) ? m2 : m1;
}
```

complexity: ...

Divide-and-Conquer, Recursively

Maximum of an Unsorted Array (IIa)

How many calls of `array_max` are necessary?

for length 1, $c(1) = 1$

for length $n > 1$, $c(n) = c(\frac{n}{2}) + c(\frac{n}{2}) + 1$

... overall $c(n) = 2n - 1$ calls

in each recursive call, we do $O(1)$ steps.

$$\implies O(n)$$

Iteratively:

```
ssize_t binary_search (int a[], size_t n, int key)
{
    size_t lo = 0, hi = n - 1;
    while (hi >= lo) {
        size_t mid = (lo + hi) / 2;
        if (a[mid] == key) return mid;
        if (a[mid] > key) hi = mid - 1;
        if (a[mid] < key) lo = mid + 1;
    }
    return -1;
}
```

complexity: $O(\log n)$

Recursively:

```
ssize_t binary_search (int a[], size_t n, int key)
{
    return binary_search_do (a, 0, n - 1, key);
}
```

```
ssize_t binary_search_do (int a[], size_t lo, size_t hi, int key)
{
    if (lo > hi) return -1;
    size_t mid = (lo + hi) / 2;
    if (a[mid] == key) return mid;
    if (a[mid] > key) return binary_search_do (a, lo, mid - 1, key);
    if (a[mid] < key) return binary_search_do (a, mid + 1, hi, key);
    assert (!"unreachable");
}
```

complexity: $O(\log n)$

COMP2521
19T0 lec04

cs2521@
jashankj@

Recursion

Trees

Searching

Trees

BTrees

BSTs

Trees

Search is a critical operation, e.g.

- looking up a name in a phone book
- selecting records in databases
- searching for pages on the web

Characteristics of the search problem:

- typically, very large amount of data (very many items)
- query specified by keys (search terms)
- effective keys identify a small proportion of data

Recursion

Trees

Searching

Trees

BTrees

BSTs

We'll abstract the problem to:
a large collection of *items*,
each containing a *key* and other data
(We can think of these as
'key/data' or 'key/value' pairs.)

```
typedef <...> Key;  
typedef struct {  
    Key key;  
    <... data ...>  
}Item;
```

The search problem:

input a key value

output item(s) containing that key

Common variations:

- keys are unique; key value matches 0 or 1 items
- multiple keys in search, items containing any key
- multiple keys in search/item, items containing all keys

We assume: keys are unique, each item has one key.

Cheap, easy gains from searching sorted data.

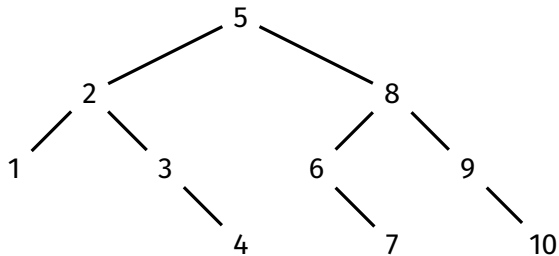
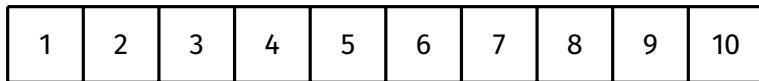
Maintaining sorted sequences is hard...
inserting into a sorted sequence is a two-step problem.

array search $O(\log n)$, insert $O(n)$
... we have to move all the items along

linked list search $O(n)$, insert $O(1)$
... search is *always* linear

Can we do better?

1	2	3	4	5	6	7	8	9	10
---	---	---	---	---	---	---	---	---	----



Trees are branched data structures,
consisting of **nodes** and **edges**, with no cycles.

Each node contains a value.

Each node has edges to $\leq k$ other nodes.

For now, $k = 2$ — binary trees

Trees can be viewed as a set of nested structures:
each node has k (possibly empty) **subtrees**.

A node is a **parent** if it has outgoing edges.

A node is a **child** if it has incoming edges.

The **root** node has no parents.

A **leaf** node has no children.

A node's **depth** or **level** is
the number of edges from the root to that node.

The root node has depth 0;
all other nodes have one more than their parent's depth

Recursion

Trees

Searching

Trees

BTrees

BSTs

For a given number of nodes, a tree is said to be **balanced** if it has minimal height, and **degenerate** if it has maximal height.

A **k -ary tree**'s internal nodes have k children.
A tree is **ordered** if data/keys in nodes are constrained.

Recursion

Trees

Searching

Trees

BTrees

BSTs

- representing hierarchical data structures (e.g., expressions in a programming language)
- efficient search (e.g., in sets, symbol tables)

For much of the course,
we'll look at **binary trees** (where $k = 2$).

Binary trees are either empty,
or are a node with two subtrees,
where each node has a value,
and the subtrees are binary trees.

$$\begin{array}{lcl} \text{BTree} & := & \text{Empty} \\ & | & \text{Node } x \text{ BTree } l \text{ BTree } r \end{array}$$

A binary tree with n nodes has a height of
at most $n - 1$, if degenerate; or
at least $\lfloor \log_2 n \rfloor$, if balanced.

Cost for **insertion**:

balanced $O(\log_2 n)$, degenerate $O(n)$
(we always traverse the height of the tree)

Cost for **search/deletion**:

balanced $O(\log_2 n)$, degenerate $O(n)$
(worst case, key $\notin \tau$; traverse the height)

A binary tree!

For all nodes in the tree:

the values in the **left** subtree are **less than** the node value

the values in the **right** subtree are **greater than** the node value

Structure tends to be determined
by order of insertion:

[4, 2, 1, 3, 6, 5, 7] vs [6, 5, 2, 1, 3, 4, 7]

Exercise: Happy Little Trees

Starting with an initially-empty binary search tree ... show the tree resulting from inserting values in the order given, and give its resulting height —

1 [4, 2, 6, 5, 1, 7, 3]

2 [5, 3, 6, 2, 4, 7, 1]

3 [1, 2, 3, 4, 5, 6, 7]

Recursion

Trees

Searching

Trees

BTrees

BSTs

```
struct btree_node {  
    Item item;  
    btree_node *left;  
    btree_node *right;  
};
```

As before: the empty tree is NULL.

Recursion

Trees

Searching

Trees

BTrees

BSTs

```
// return the node if found, or NULL otherwise
btree_node *btree_search (btree_node *tree, Item key)
{
    if (tree == NULL) return NULL;
    int cmp = item_cmp (key, tree->item);
    if (cmp == 0) return tree;
    if (cmp < 0) return btree_search (tree->left, key);
    if (cmp > 0) return btree_search (tree->right, key);
}
```

EXERCISE Try writing an iterative version.

We're (recursively) inserting value v into tree τ .

Cases:

- τ empty
 \Rightarrow make a new node with v as the root of the new tree
- the root of τ contains v
 \Rightarrow tree unchanged (assuming no duplicates)
- $v < \tau \rightarrow \text{item}$
 \Rightarrow do insertion into $\tau \rightarrow \text{left}$
- $v > \tau \rightarrow \text{item}$
 \Rightarrow do insertion into $\tau \rightarrow \text{right}$

EXERCISE Try writing an iterative version.

Recursion

Trees

Searching

Trees

BTrees

BSTs

- `btree_size :: BTree → size`
return the number of nodes in a tree
- `btree_height :: BTree → size`
return the height of a tree

‘serialisation’ of a structure:
flattening it in a well-defined way,
such that the original structure can be recovered

Depth-first:

- pre-order traversal (**NLR**)
... visit node, then left subtree, then right subtree
- in-order traversal (**LNR**)
... visit left subtree, then node, then right subtree
- post-order traversal (**LRN**)
... visit left subtree, then right subtree, then node

Breadth-first:

- level-order traversal
... visit node, then all its children

Recursion

Trees

Searching

Trees

BTrees

BSTs

Insertion is easy!
find location, create node, link parent
Deletion is much harder!
find node, unlink and delete, ...?

Recursion

Trees

Searching

Trees

BTrees

BSTs

Insertion is easy!

find location, create node, link parent

Deletion is much harder!

find node, unlink and delete, ...?

One option: don't delete nodes : –)

instead, just mark them as deleted, and ignore them

Recursion

Trees

Searching

Trees

BTrees

BSTs

Insertion is easy!

find location, create node, link parent

Deletion is much harder!

find node, unlink and delete, ...?

One option: don't delete nodes : -)

instead, just mark them as deleted, and ignore them

Otherwise, we must *promote* a child (carefully).

A child with no subtrees: drop.

A child with one subtree: promote that subtree.

A child with two subtrees: ...

replace node with leftmost of right subtree