Data structures

Ziyan Maraikar

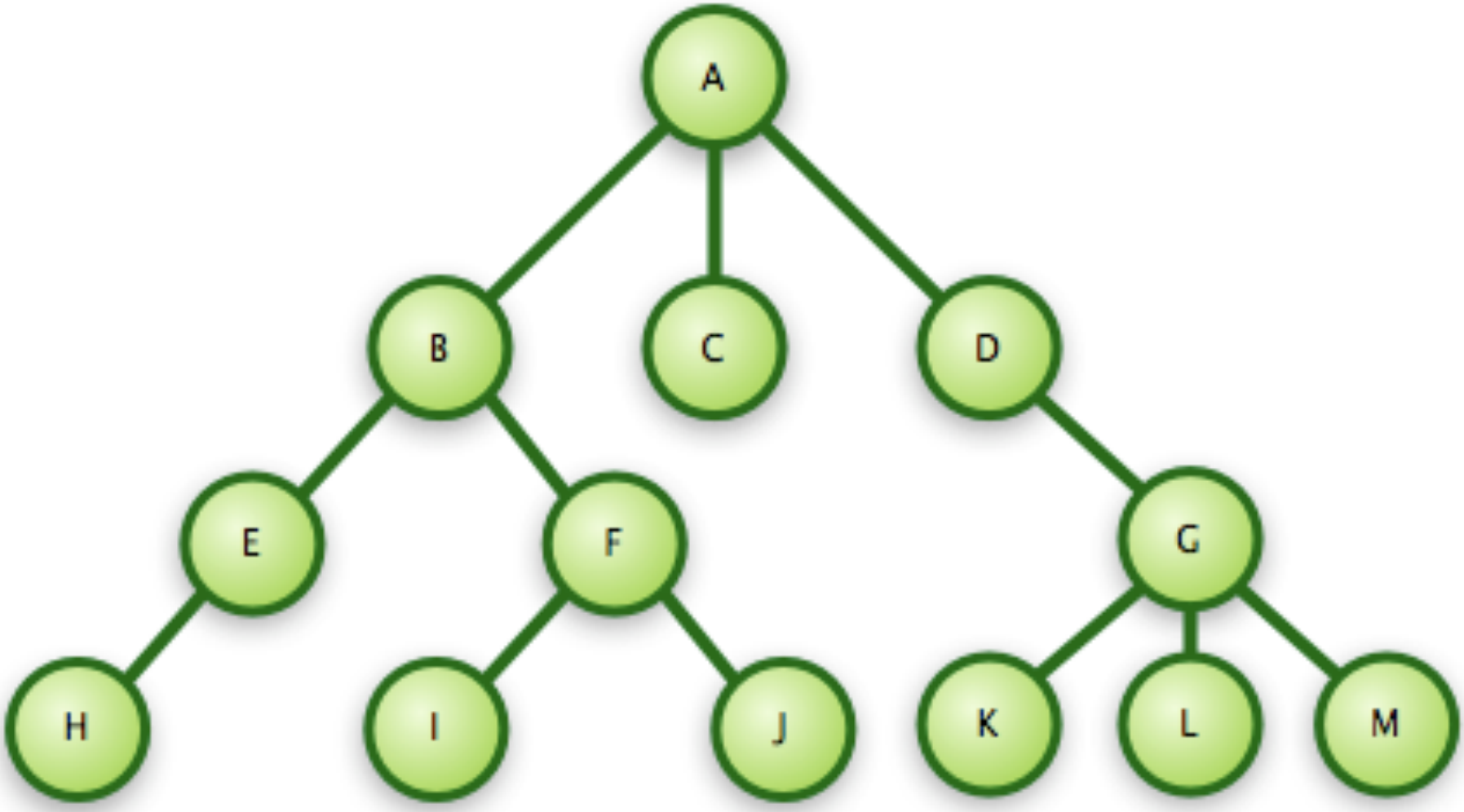
September 6, 2014

Lecture Outline

Trees

- A *tree* stores items in a hierarchical structure. Useful for storing data in a particular order for fast searching and sorting.
- Each *node* in the tree has exactly one *parent* above it and zero or more *children* below it.
- The topmost node is called the *root* and the nodes at the bottom are called *leaves*.

Example



Binary trees

In a *binary tree* a node has two children.

We define a binary tree inductively as either,

- ★ Empty
- ★ Consisting of a **root node with two chlidren that are binary trees**.

```
type 'a binarytree =
| Empty
| Node of 'a * 'a binarytree * 'a binarytree
```

Example:

```
let t1 = Node (3, Empty, Node (2, Empty, Empty))
```

Tree traversal

We often need to *traverse* a tree visiting each node.

In what order should we visit the nodes? There are three possibilities

preorder Visit the root, then the left subtree, finally the right subtree.

inorder Visit the left subtree, then the root, finally the right subtree.

postorder Visit the the left subtree, then the right subtree, finally the root.

Preorder traversal

```
type 'a binarytree =
  | Empty
  | Node of 'a * 'a binarytree * 'a binarytree

let rec preorder t =
  match t with
  | Node(o, l, r) -> string_of_int o ^ ","
                     ^ preorder l ^ preorder r
  | Empty -> ""
```

Exercise

- ★ Define a function `tree_square` that takes an `int tree` and squares each node.
- ★ Define a function `tree_sum` that takes an `int tree` and returns the sum of all nodes.
- ★ Define a function `flatten_inorder` that takes a `tree` and returns an inorder list of nodes.

Lecture Outline

Map on trees

We already saw the `map` operation on lists

```
(* Apply the function f to each element of list l *)
let map f l =
  match l with
  hd::tl -> f hd :: map f tl
  [] -> []
```

This is an example of a tree map.

```
let rec tree_square t =
  match t with
  | Node(o, l, r) -> Node(square o, tree_square l,
    tree_square r)
  | Empty -> Empty
```

Can we generalise this to a map operation on trees?

Map on trees

Two cases to consider:

Empty just return `Empty`.

Node apply `f` to value at node, and map `f` on left and right subtrees.

```
let rec tree_map f t =
  match t with
  | Empty -> Empty
  | Node(v, l, r) -> Node(f v, tree_map f l, tree_map f r)
```

Fold on trees

We already saw the `fold` operation on lists

```
(* Compose elements of l using function f and identity e *)
let fold f e l =
  match l with
  hd::tl -> f hd (fold f e tl)
  [] -> e
```

This is an example of a tree fold.

```
let rec tree_sum t =
  match t with
  | Node(o, l, r) -> o + tree_sum l + tree_sum r
  | Empty -> 0
```

Fold on trees

Empty return identity.

Node fold `f` on left and right subtrees and combine with value at node.

We need to combine *three* values togther using a *binary* function. In what order should we combine the values using `f`?

```
let rec tree_fold f e t =
  match t with
  | Empty -> e
  | Node(v, l, r) ->
    let lv = tree_fold f e l in
    let lr = tree_fold f e r in
    f v (f lv lr)
```

If `f` is *associative* and *commutative* the order in which we apply `f` will not matter.

Data structures and algorithms

- ★ Define a function `tree_square` using `tree_map`.
- ★ Define `tree_sum` using `tree_fold`.

Lecture Outline

Arithmetic expressions

Consider the structure of arithmetic expressions

$$(1 + 2) \times 3$$

How could we represent them in Ocaml?

Arithmetic expressions

We can define an arithmetic expression recursively as either,

- ★ a constant (number), or
- ★ arithmetic expression(s) composed using an operator.

We can turn this into a type definition.

```
type arexp =
| Const of int
| Plus of arexp * arexp
```

What does this definition look like?

Exercise

```
type arexp =
| Const of int
| Plus of arexp * arexp
```

- ★ Add the `Times` operator to this type.
- ★ Construct the expression $(1 + 2) \times 3$

Evaluating an expression

```
type arexp =
  | Const of int
  | Plus of arexp * arexp
  | Times of arexp * arexp
```

We can use Ocaml's arithmetic operators to evaluate expression.

```
let rec eval_expr e =
  match e with
  | Const a -> a
  | Plus(a, b) -> _______ + _______
  | Times(a, b) ->  _______ * _______
```

Does this pattern look familiar? This is a (variation of) our binary tree definition. Here, the leaf (`Const`) nodes have a value associated with them.

Evaluating an expression

```
let rec eval_expr e =
  match e with
  | Const a -> a
  | Plus(a, b) -> eval_expr a + eval_expr b
  | Times(a, b) -> eval_expr a * eval_expr b
```

This is an inorder traversal of the expression tree!

Printing an expression

Exercise: Write a `pretty_print` function that converts an expression to a string.

```
let rec pretty_print e =
  match e with
  | Const a -> pretty_print a
  | Plus(a, b) -> pretty_print a ^ " + " ^ pretty_print b
  | Times(a, b) -> pretty_print a ^ " * " ^ pretty_print b
```

The expression problem

This is a good problem to study how software systems can be extended in different dimensions.

1. How to add new operators to expressions without modifying existing code.
   e.g. Subtraction, division.
2. How to add new functions that operate on expressions without modifying existing code.
   e.g. pretty printing, evaluation.

The study of how to accomplish this is called the *expression problem*.

In which dimension is our solution extensible?

Lecture Outline

Ordered data

- ★ Various data structures have been invented to store data in a particular order.
  Binary search trees (BST) are one such data structure.
- ★ Seaching for an element much more efficient when data is stored in ordered fashion.
- ★ BSTs also supports efficient insertion and deletion of new elements[1].

[1] compared to a sorted list

Binary search tree

A BST imposes an *total ordering* on nodes in a binary tree.

The following constraints hold for any node,

1. its left subtree contains values less than itself.
2. its right subtree contains values less than itself.

How do we ensure that a binary tree meets this constraint?

Constructing a BST

Do we need to change our type definition?

```
type 'a binarytree =
  | Empty
  | Node of 'a * 'a binarytree * 'a binarytree
```

No, the structure of the binary tree does not change.

We must define operations on BSTs so that these constraints are respected.

Operations on a BST

1. Insert a new value.
2. Check whether a BST contains a given value.
3. Delete a value.

In each case we must ensure the BST constraint is maintained.

Insert

- ★ Repeatedly compare the new value with that of the current node.
- ★ Traverse either the left or right subtree.
- ★ On reaching a leaf, insert the new node.

```
let rec insert e t =
  match t with
  | Empty -> Node(e, Empty, Empty)
  | Node(v,l,r) ->
      if e=v then Node(v, l, r)
      else if e<v then Node(v, insert e l, r)
      else  Node(v, l, insert e r)
```

Exercise: Contains

```
let rec contains e t =
  match t with
   | Empty -> _______
   | Node(v, l, r) -> if e<v then _______ else _______
```

Delete

Three possible cases.

The node we are deleting may have,

1. No children[2]:
   replace the node with `Empty`.
2. One child:
   replace the node with its child.
3. Two children:
   replace with *nearest value* in either the left or right subtrees.
   Either,

   PREDECESSOR: Maximum in left subtree, or
   SUCCESSOR: Minimum in right subtree.

[2] non-empty

Delete

```
let rec delete e t =
  match t with
   | Empty -> (* value not found *)
   | Node(v, l, Empty) -> (* replace current subtree with l *)
   | Node(v, Empty, r) -> (* replace current subtree with r *)
   | Node(v, l, r) -> (* replace v with predecessor
                         (maximum in left subtree ) *)
```

Delete

```
let rec delete e t =
  match t with
  | Empty -> Empty
  | Node(v, l, Empty) ->
      if e=v then l
      else if e<v then Node(v, delete e l, Empty)
      else t
  | Node(v, Empty, r) ->
      if e=v then r
      else if e<v then t
      else Node(v, Empty, delete e r)
  | Node(v, l, r) ->
      if e=v then let (p, l') = delete_max l in
      Node(p, l', r)
      else if e<v then Node(v, delete e l, r)
      else Node(v, l, delete e r)
```

Delete maximum

```
(* Deletes the maximum in tree
    and returns the maximum and new tree *)
let rec delete_max t : 'a * 'a binarytree =
  match t with
  | Empty -> failwith "No max"
  | Node(v, l, Empty) -> (v, l)
  | Node(v, l, r) -> let (m, t) = delete_max r in
                     (m, Node(v, l, t))
```

The when keyword

We can make `delete` more succint using Ocaml's `when` keyword.

It lets you add boolean comparisons to match expressions.

```
let rec delete e t =
  match t with
  | Empty -> Empty
  | Node(v, l, Empty) when e=v -> l
  | Node(v, Empty, r) when e=v -> r
  | Node(v, l, r) ->
      if e<v then Node(v, delete e l, r)
      else if e>v then Node(v, l, delete e r)
      else let (m, l') = delete_max l in Node(m, l', r)
```

Note how we avoid reptition of the cases `e<v` and `e>v`.