Module 9: Binary trees

Readings: HtDP, Section 14

We will cover the ideas in the text using different examples and different terminology. The readings are still important as an additional source of examples.

Binary arithmetic expressions

A binary arithmetic expression is made up of numbers joined by binary operations $*$, $+$, $/$, and $-$.

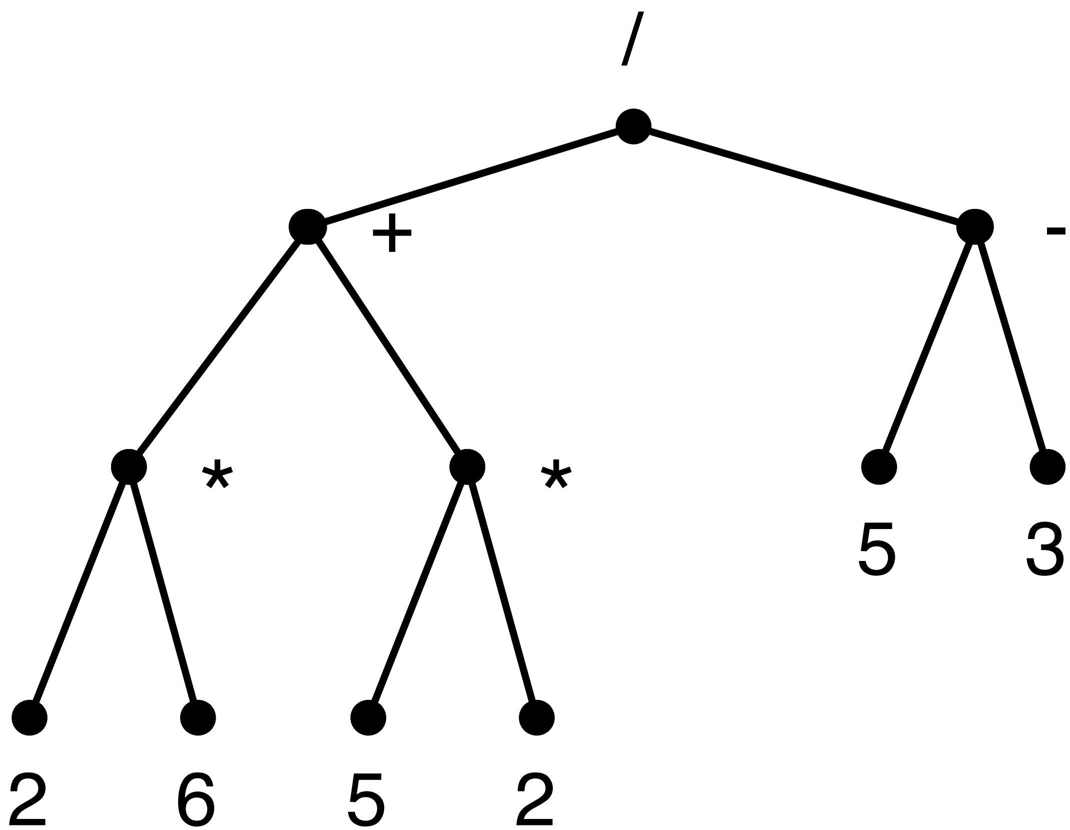
$((2 * 6) + (5 * 2))/(5 - 3)$ can be defined in terms of *two* smaller binary arithmetic expressions, $(2 * 6) + (5 * 2)$ and $5 - 3$.

Each smaller expression can be defined in terms of even smaller expressions.

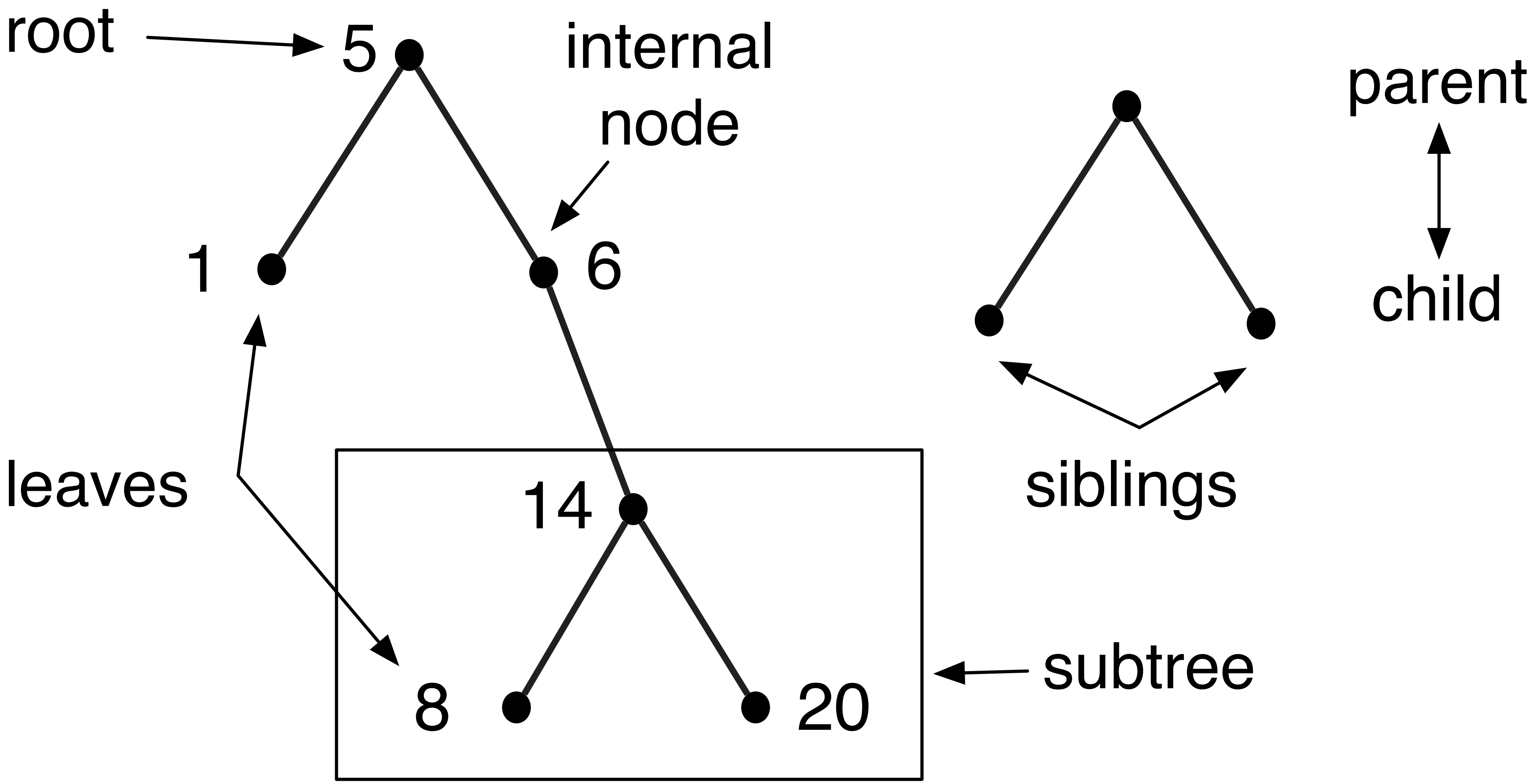
The smallest expressions are numbers.

Visualizing binary arithmetic expressions

$((2 * 6) + (5 * 2))/(5 - 3)$ can be represented as a **tree**:



Tree terminology



Variations on trees

- Number of children of internal nodes:
  - ⋆ exactly two
  - ⋆ at most two
  - ⋆ any number
- Labels:
  - ⋆ on all nodes
  - ⋆ just on leaves
- Order of children (sometimes important)
- Tree structure (from data or for convenience)

Representing binary arithmetic expressions

Internal nodes each have exactly two children.

Leaves have number labels.

Internal nodes have symbol labels.

For subtraction and division, we care about the order of children.

The structure of the tree is dictated by the expression.

How can we group together information for an internal node?

How can we allow different definitions for leaves and internal nodes?

```
(define-struct binode (op arg1 arg2))
;; A Binary arithmetic expression Internal Node (BINode)
;;    is a (make-binode (anyof '* '+ '/ '-) BinExp BinExp)

;; A Binary arithmetic expression (BinExp) is one of:
;; * a Num
;; * a BINode
;; Examples
5
(make-binode '* 2 6)
(make-binode '+ 2 (make-binode '- 5 3))
```

A more complex example:

```
(make-binode '/
            (make-binode '+ (make-binode '* 2 6)
                            (make-binode '* 5 2))
            (make-binode '- 5 3))
```

Note that the definitions of binode and BinExp depend on each other. This is an example of a set of mutually recursive definitions.

Template for binary arithmetic expressions

The only new idea in forming the template is the application of the recursive function to *each* piece that satisfies the data definition.

```
;; binexp-template: BinExp → Any
(define (binexp-template ex)
  (cond [(number? ex) ... ex ...]
        [else (... (binode-op ex) ...

                   (binexp-template (binode-arg1 ex)) ...
                   (binexp-template (binode-arg2 ex)) ...)]))
```

Evaluating a binary arithmetic expression

Exercise: Develop a function eval that consumes a BinExp and produces its simplified numeric value. Use the template.

```
;; (eval ex) produces the simplified value of ex
;; eval: BinExp → Num
;; Requires: no division by 0
;; Examples:
(check-expect (eval 5) 5)
(check-expect (eval (make-binode '+ 2 (make-binode '- 5 3))) 4)

(define (eval ex) ... )
```

Our next example is also a binary tree (at most two children for each node), but differs from expression trees in several important ways:

- Internal nodes can have one child or two.
- The order of children always matters.
- The tree structure does not come from the data - we order it.

We use trees to try to provide a more efficient implementation of dictionaries.

Dictionaries revisited

Recall from Module 5 that a dictionary stores a set of (key, value) pairs, with at most one occurrence of any key.

It supports lookup, add, and remove operations.

We implemented a dictionary as an association list of two-element lists.

This implementation had the problem that a search could require looking through the entire list, which will be inefficient for large dictionaries.

Binary search trees

The new implementation uses a tree structure known as a **binary search tree**.

The (key,value) pairs are stored at nodes of a tree.

There is more than one possible tree.

The placement of pairs in nodes may make it possible to improve the running time compared to association lists.

We will start with a basic binary tree definition first.

```
(define-struct node (key val left right))
;; A Node is a (make-node Nat Str BT BT)

;; A binary tree (BT) is one of:
;; * empty
;; * a Node
```

What is the template?

Note: empty still refers to the empty list empty. We will use the constant empty in our definitions. However, remember that we are working with tress, not lists. In calculated values, empty may appear.

A BT template

```
;; bt-template: BT → Any
(define (bt-template t)
  (cond
    [(empty? t) . . . ]
    [else (. . . (node-key t) . . .
                 (node-val t) . . .
                 (bt-template (node-left t)) . . .
                 (bt-template (node-right t)) . . . )]))
```

Counting leaves in a BT

How many leaves are in a BT t?

- An empty tree has 0 leaves.
- A tree containing only a leaf has 1.
- Otherwise, add together the number of leaves in the left subtree and the number of leaves in the right subtree.

Solution

```
;; (count-leaves t) produces number of leaves in t
;; count-leaves: BT → Nat
(define (count-leaves t)
  (cond
    [(empty? t) 0]
    [(and (empty? (node-left t))
          (empty? (node-right t))) 1]
    [else (+ (count-leaves (node-left t))
             (count-leaves (node-right t)))]))
```

Counting values in a BT

```
;; count-values: BT Str → Nat
(define (count-values abst v)
  (cond [(empty? abst) 0]
        [else (+ (cond [(equal? v (node-val abst)) 1]
                       [else 0])
                 (count-values (node-left abst) v)
                 (count-values (node-right abst) v))]))
```

Searching a BT

How would you determine if a specific key appears in a BT?

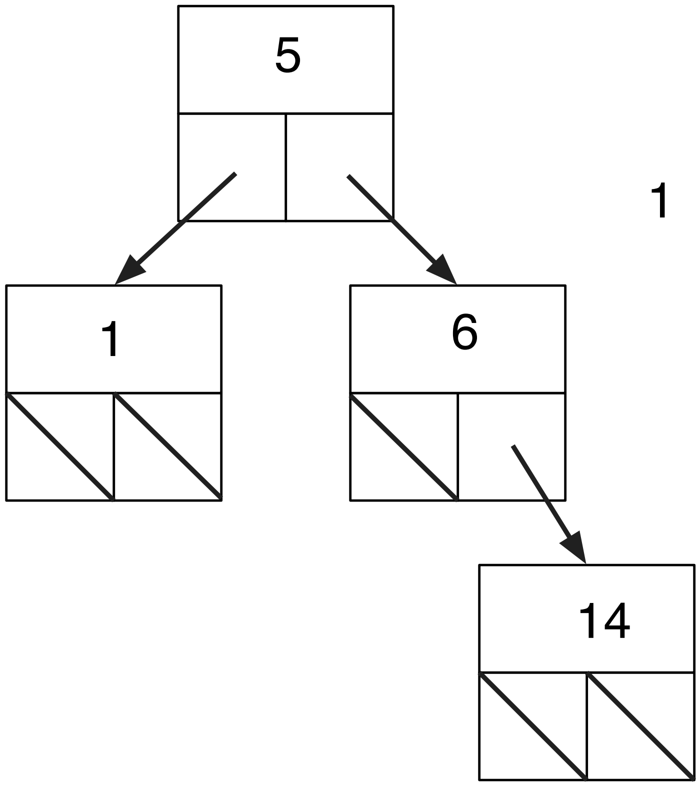
So far, we just have a binary tree. Add conditions to define a binary search tree.

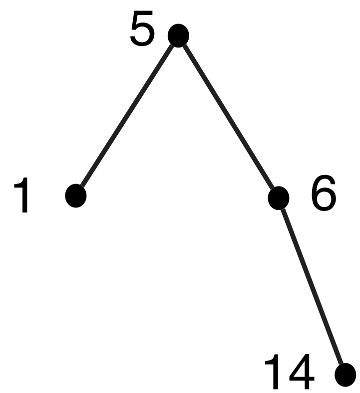
```
(define-struct node (key val left right))
;; A Node is a (make-Node Nat Str BST BST)

;; A binary search tree (BST) is either
;; * empty, or
;; * a Node
;; which satisfies the ordering property recursively:
;; * every key in left is less than key
;; * every key in right is greater than key
```

A BST example

```
(make-node 5 "Tony"
  (make-node 1 "Qiang" empty empty)
  (make-node 6 "Judy"
              empty
              (make-node 14 "Wole"
                          empty
                          empty)))
```

Drawing BSTs





(Note: the value field is not usually represented in diagrams since the keys determine the shape of the tree.)

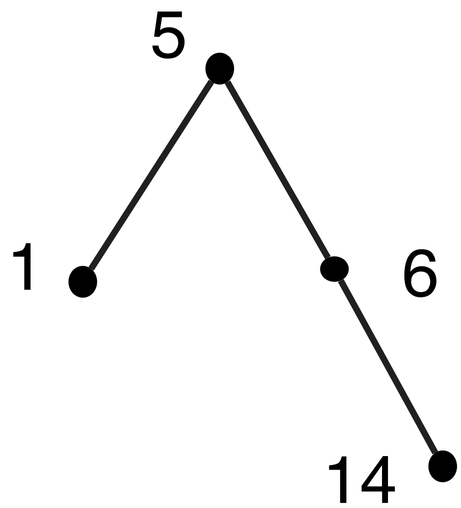
We have made several minor changes from HtDP:

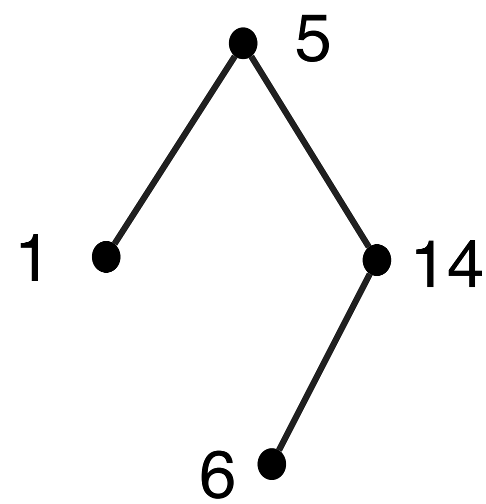
- We use empty instead of false for the base case. We will usually use the constant empty rather than false in our examples.
- We use key instead of ssn, and val instead of name.

The value can be any Racket value.

We can generalize to other key types with a comparison operation (e.g. strings, using string<?).

There can be several different BSTs holding a particular set of (key, value) pairs.





A BST template

```
;; bst-template: BST → Any
(define (bst-template t)
  (cond
    [(empty? t) . . . ]
    [else (. . . (node-key t) . . .
                 (node-val t) . . .
                 (bst-template (node-left t)) . . .
                 (bst-template (node-right t)) . . . )]))
```

Note that this template is identical to the BT template.

We can write code that exploits the ordering property of BST. If we produce a new binary tree by adding 1 to every key of an existing BST, the result still has the ordering property and so is a BST.

```
;; increment: BST → BST
(define (increment t)
  (cond
    [(empty? t) empty]
    [else (make-node (add1 (node-key t))
                     (node-val t)
                     (increment (node-left t))
                     (increment (node-right t)))]))
```

Retrieving all keys in a BST

The following function creates a list of all keys in a BST.

```
(define (all-keys t)
  (cond [(empty? t) empty]
        [else
          (append (all-keys (node-left t))
                  (list (node-key t))
                  (all-keys (node-right t)))]))
```

What is special about the order of the produced list for a BST?

Making use of the ordering property

The search-bt, count-values, increment, and all-keys functions can be used on plain BT trees, without the ordering property.

But, when increment is applied to a BST, we get a new BST. When all-keys is applied to a BST, the list is in increasing order.

For some problems, like searching and adding, the ordering property allows us to avoid one of the recursive calls entirely.

We will write the code for searching, and briefly sketch adding, leaving you to write the Racket code.

Searching in a BST

How do we search for a key n in a BST?

We reason using the data definition of bst.

If the BST is empty, then n is not in the BST.

If the BST is of the form (make-node k v l r), and k equals n, then we have found it.

Otherwise it might be in either of the trees l, r, and we can determine which one by comparing the values of k and n.

If k $>$ n, then n cannot be in r, and we only need to recursively search in l.

If k $<$ n, then n cannot be in l, and we only need to recursively search in r.

Either way, we save one recursive call.

```
;; (search-bst n t) produces the value associated with n in t,
;;    or false if n is not in t.
;; search-bst: Nat BST → (anyof Str false)
(define (search-bst n t)
  (cond
    [(empty? t) false]
    [(= n (node-key t)) (node-val t)]
    [(< n (node-key t)) (search-bst n (node-left t))]
    [else (search-bst n (node-right t))]))
```

Creating a BST

How do we create a BST from an association list?

We reason using the data definition of a list.

If the list is empty, the BST is empty.

If the list is of the form (cons (list k v) lst), we add the pair (k, v) to the BST created from the list lst.

Essentially, this is (foldr bst-insert empty lst).

Adding to a BST

How do we add a pair (k, v) to a BST bstree?

If bstree is empty, then the result is a BST with only one node.

Otherwise bstree is of the form (make-node n w l r).

If k = n, we form the tree with k, v, l, and r (we replace the old value, w, by v).

If $k < n$, then the pair must be added to l, and if $k > n$, then the pair must be added to r. Again, we need only make one recursive call.

Binary search trees in practice

If the BST has all left subtrees empty, it looks and behaves like a sorted association list, and the advantage is lost.

Some later CS courses will cover ways to keep a BST "balanced" so that "most" nodes have nonempty left and right children.

By that time, if you take those courses, you will better understand how to analyze the efficiency of algorithms and operations on data structures.

Goals of this module

You should be familiar with tree terminology.

You should understand the data definitions for binary arithmetic expressions, binary trees, and binary search trees, understand how the templates are derived from those definitions, and how to use the templates to write functions that consume those types of data.

You should understand the definition of a binary search tree, and how it can be generalized to hold additional data.

You should be able to write functions which consume binary search trees, including those sketched (but not developed fully) in lecture.

You should be able to develop and use templates for other binary trees, not necessarily presented in lecture.