Module 8: 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 $-$.

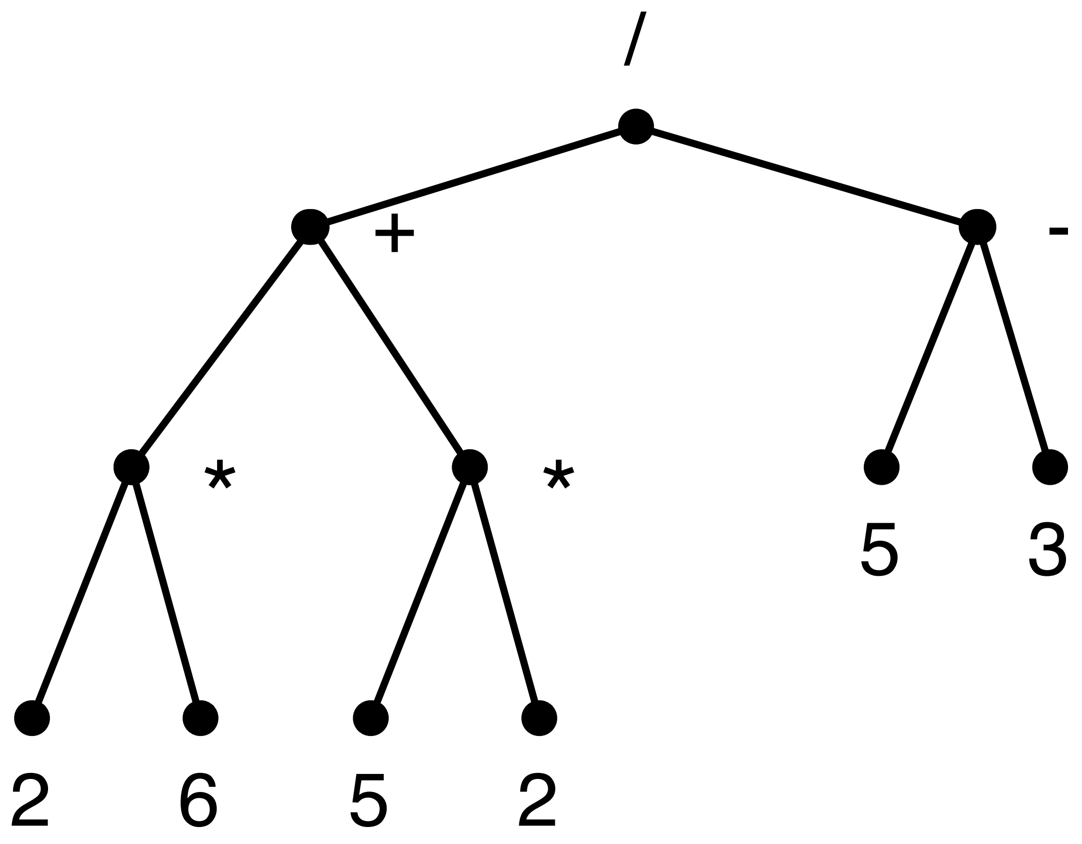
$((2 * 6) + (5 * 2))/(5 - 3)$ can be defined in terms of *two* smaller binary arithmetic expressions, $(2 * 6) + (5 * 2)$ and $5 - 3$.

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

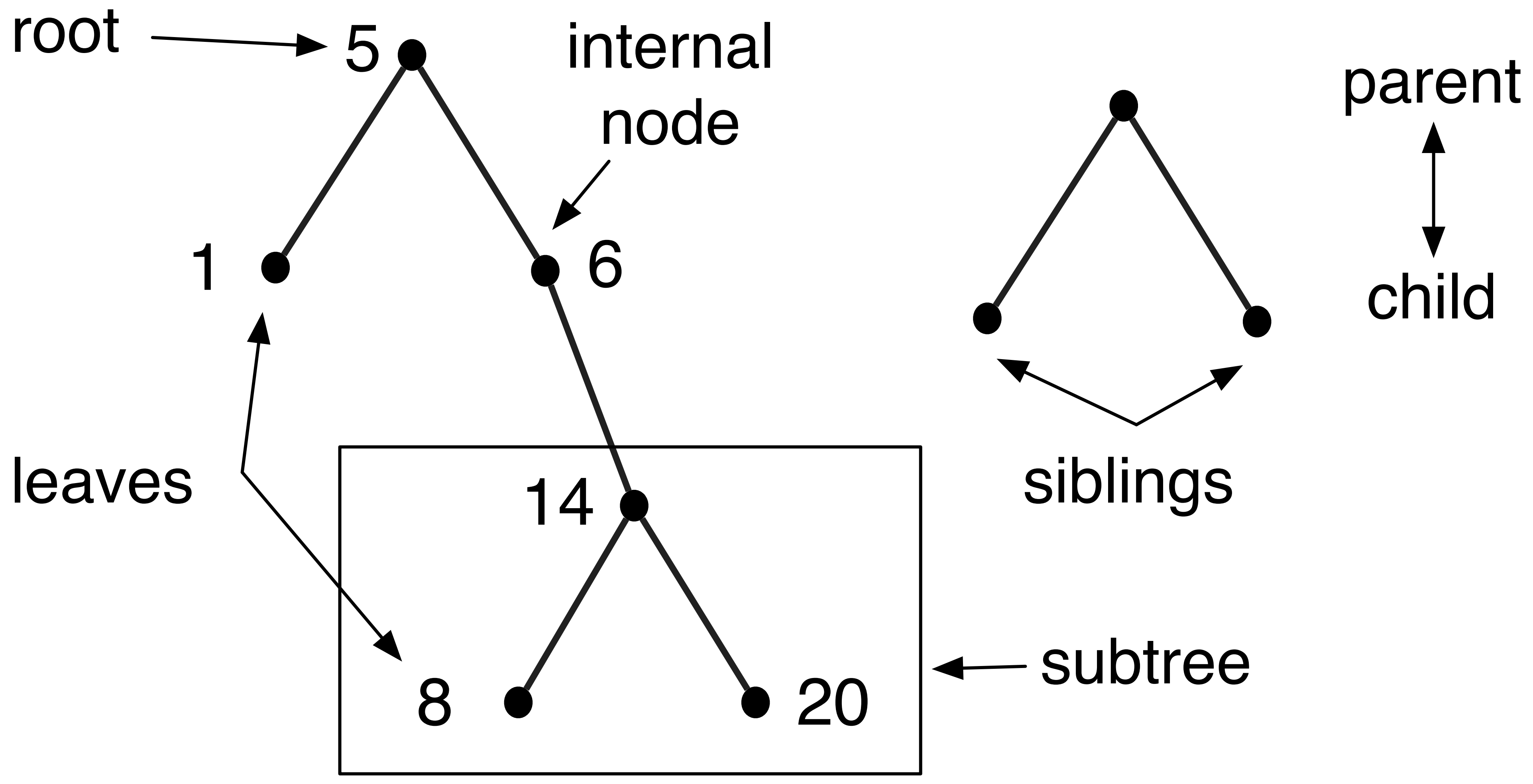
The smallest expressions are numbers.

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 (matters or not)
- 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 bae (fn arg1 arg2))
```

A binary arithmetic expression (**binexp**) is either

- a number or
- a structure (make-bae f a1 a2), where
  - ⋆ f is a symbol in the set '∗, '+, '/, '−,
  - ⋆ a1 is a *binexp*, and
  - ⋆ a2 is a *binexp*.

Note that the base case has a different type.

Examples of binary arithmetic expressions:

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

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.

```
(define (my-binexp-fun ex)
  (cond
    [(number? ex) . . . ]
    [(bae? ex) . . .  (bae-fn ex) . . .
                 . . .  (my-binexp-fun (bae-arg1 ex)) . . .
                 . . .  (my-binexp-fun (bae-arg2 ex)) . . . ]))
```

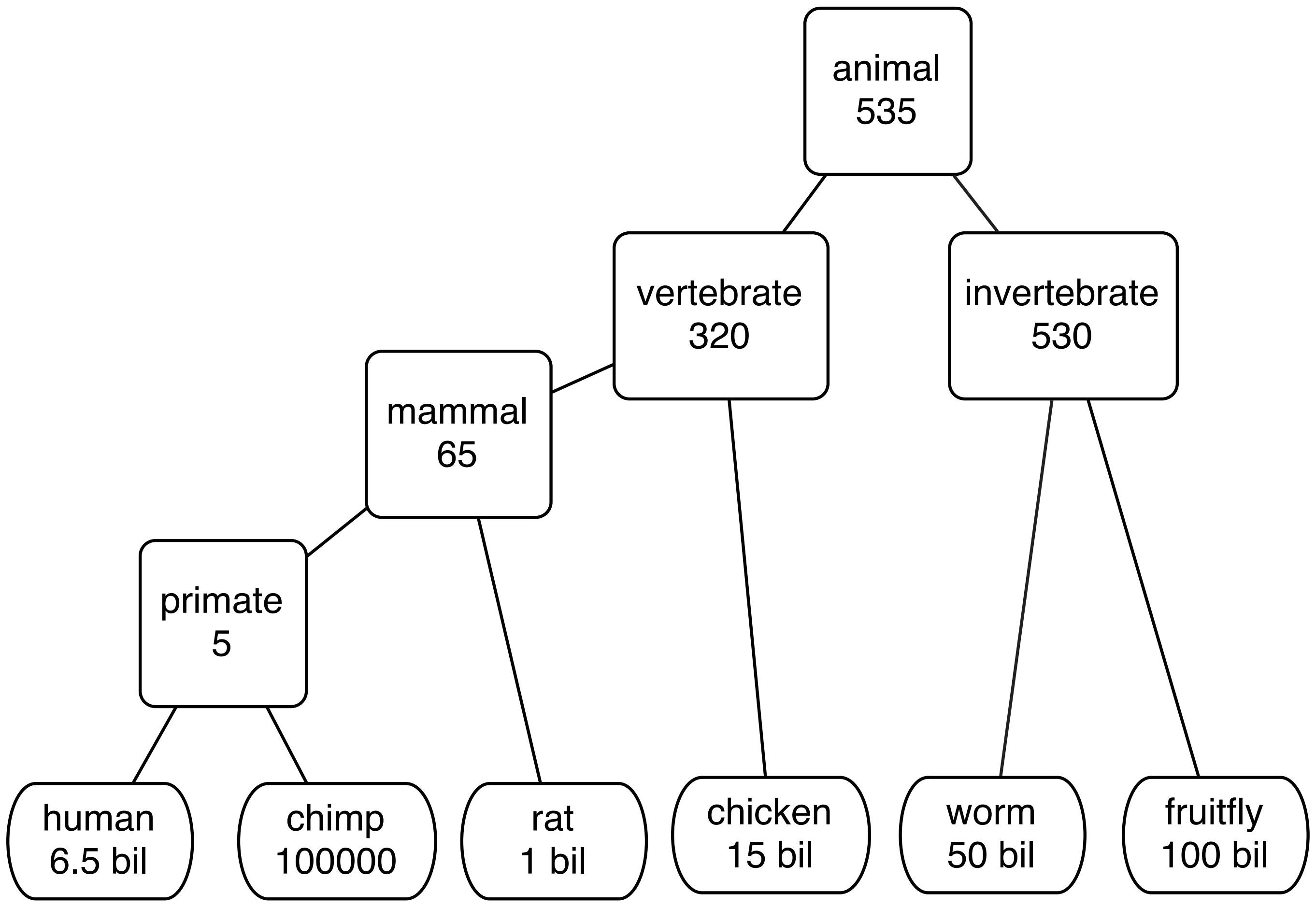
Evaluation of expressions

```
(define (eval ex)
  (cond [(number? ex) ex]
        [(bae? ex) (cond [(symbol=? (bae-fn ex) '*)
                          (* (eval (bae-arg1 ex)) (eval (bae-arg2 ex)))]
                         [(symbol=? (bae-fn ex) '+)
                          (+ (eval (bae-arg1 ex)) (eval (bae-arg2 ex)))]
                         [(symbol=? (bae-fn ex) '/)
                          (/ (eval (bae-arg1 ex)) (eval (bae-arg2 ex)))]
                         [(symbol=? (bae-fn ex) '-)
                          (- (eval (bae-arg1 ex)) (eval (bae-arg2 ex)))])]))
```

Evolution trees

- a data structure recording information about the evolution of species
- contains two kinds of information:
  - modern species (e.g. humans) with population count
  - evolutionary history via **evolution events**, with an estimate of how long ago the event occurred

An “evolution event” leads to a splitting of one species into two distinct species (for example, through physical separation).


animal
535
vertebrate
320
invertebrate
530
mammal
65
primate
5
human
6.5 bil
chimp
100000
rat
1 bil
chicken
15 bil
worm
50 bil
fruitfly
100 bil

Representing evolution trees

Internal nodes each have exactly two children.

Leaves have names and populations of modern species.

Internal nodes have names and dates of evolution events.

The order of children does not matter.

The structure of the tree is dictated by a hypothesis about evolution.

```
(define-struct t-modern (name pop))
(define-struct t-ancient (name age left right))
```

taxon is either

- a structure (make-t-modern n p), where
  - ⋆ n is a string, and
  - ⋆ p is a nat (current population)
- or a structure (make-t-ancient n a l r), where
  - ⋆ n is a string,
  - ⋆ a is a nat (in millions of years),
  - ⋆ l is a *taxon*, and
  - ⋆ r is a *taxon*.

```
(define human (make-t-modern "human" 6.5e9))
(define chimp (make-t-modern "chimpanzee" 1.0e5))
(define rat (make-t-modern "rat" 1.0e9))
(define chicken (make-t-modern "chicken" 1.5e10))
(define worm
  (make-t-modern "worm" 5.0e10))
(define fruit-fly
  (make-t-modern "fruit fly" 1.0e11))
```

```
(define primate (make-t-ancient "primate" 5 human chimp))
(define mammal (make-t-ancient "mammal" 65 primate rat))
(define vertebrate
   (make-t-ancient "vertebrate" 320 mammal chicken))
(define invertebrate
   (make-t-ancient "invertebrate" 530 worm fruit-fly))
(define animal (make-t-ancient "animal" 535 vertebrate invertebrate))
```

The template for computation on taxons is derived from the data definition.

```
;; my-taxon-fun: taxon → any
(define (my-taxon-fun t)
  (cond
    [(t-modern? t) . . . ]
    [(t-ancient? t) . . . ]))
```

We expand this template by using the structure templates for t-modern and t-ancient.

Then we put in recursive calls, as we did for the list template.

```
;; my-taxon-fun: taxon → any
(define (my-taxon-fun t)
  (cond
    [(t-modern? t) . . .
         . . . (t-modern-name t) . . .
         . . . (t-modern-pop t) . . . ]
    [(t-ancient? t) . . .
         . . . (t-ancient-name t) . . .
         . . . (t-ancient-age t) . . .
         . . . (t-ancient-left t) . . .
         . . . (t-ancient-right t) . . . ]))
```

```
;; my-taxon-fun: taxon → any
(define (my-taxon-fun t)
  (cond
    [(t-modern? t) . . .
        . . . (t-modern-name t) . . .
        . . . (t-modern-pop t) . . .]
    [(t-ancient? t) . . .
        . . . (t-ancient-name t) . . .
        . . . (t-ancient-age t) . . .
        . . . (my-taxon-fun (t-ancient-left t)) . . .
        . . . (my-taxon-fun (t-ancient-right t)) . . .]))
```

A function on taxons

This function counts the number of modern descendant species of a taxon. A taxon is its own descendant.

```
;; ndesc-species: taxon → nat[>=1]
(define (ndesc-species t)
  (cond
    [(t-modern? t) 1]
    [(t-ancient? t) (+ (ndesc-species (t-ancient-left t))
                       (ndesc-species (t-ancient-right t)))]))
(check-expect (ndesc-species animal) 6)
(check-expect (ndesc-species human) 1)
```

Counting evolution events

```
;; recent-events: taxon nat → nat
;; Produces the number of evolution events taking place within
;; the last n million years starting from species t.
;; Examples: (recent-events worm 500) ⇒ 0
;; (recent-events animal 500) ⇒ 3
;; (recent-events animal 530) ⇒ 4
(define (recent-events t n) . . .
```

For a more complicated computation, consider trying to compute the list of names from a taxon to one of descendants.

The function ancestors consumes a taxon t and a string tname. If t is the ancestor of a taxon with name tname, it produces the list of names from the name of t to tname. Otherwise, it produces false.

What cases should the examples cover?

- The consumed taxon can be either a t-modern or a t-ancient.
- The function can produce either false or a list of strings.

```
;; ancestors: taxon string → (union (listof string) false)
(define (ancestors t tname)
  (cond
    [(t-modern? t)
     (cond
       [(equal? (t-modern-name t) tname) (list tname)]
       [else false])]
    [(t-ancient? t) ;; continued on next slide
```

```
(define (ancestors t tname)
  (cond
    [(t-modern? t) ;; details on previous slide
    [(t-ancient? t)
     (cond
       [(equal? (t-ancient-name t) tname) (list tname)]
       [(cons? (ancestors (t-ancient-left t) tname))
        (cons (t-ancient-name t) (ancestors (t-ancient-left t) tname))]
       [(cons? (ancestors (t-ancient-right t) tname))
        (cons (t-ancient-name t) (ancestors (t-ancient-right t) tname))]
       [else false])]))
```

When we try filling in the recursive case, we notice that we have to use the results of the recursive call more than once.

To avoid repeating the computation, we can pass the results of the recursive calls (plus whatever other information is needed) to a helper function.

```
(define (ancestors t tname)
   (cond
      [(t-modern? t)
       (cond [(equal? (t-modern-name t) tname) (list tname)]
                 [else false])]
      [(t-ancient? t)
       (cond
          [(equal? (t-ancient-name t) tname) (list tname)]
          [else (extend-list (t-ancient-name t)
                                 (ancestors (t-ancient-left t) tname)
                                 (ancestors (t-ancient-right t) tname))])]))
```

The function extend-list

```
(define (extend-list aname l r)
  (cond
    [(cons? l) (cons aname l)]
    [(cons? r) (cons aname r)]
    [else false]))
```

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

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

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

Dictionaries revisited

Recall from Module 6 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 can make it possible to improve the running time compared to association lists.

(define-struct node (key val left right))

A binary search tree (**bst**) is either

- empty or
- a structure (make-node k v l r), where
  - ⋆ k is a number,
  - ⋆ v is a string,
  - ⋆ l is a *bst*, and
  - ⋆ r is a *bst*.

So far, we just have a binary tree.

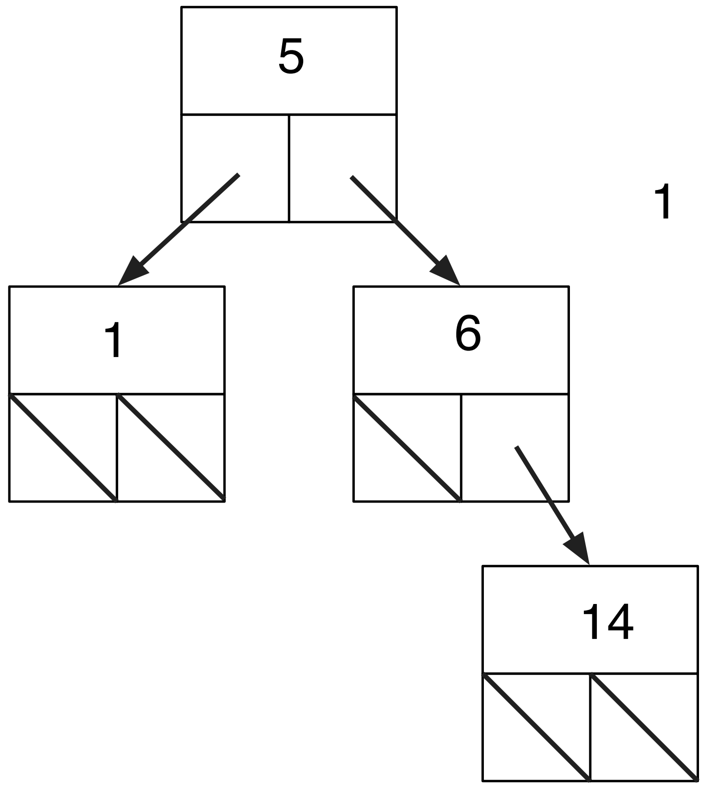
A binary search tree (make-node k v l r) satisfies two more conditions, which together form the **ordering property**:

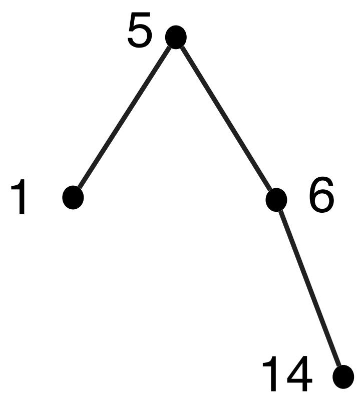
- Every key in l is less than k.
- Every key in r is greater than k.

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 represented.)

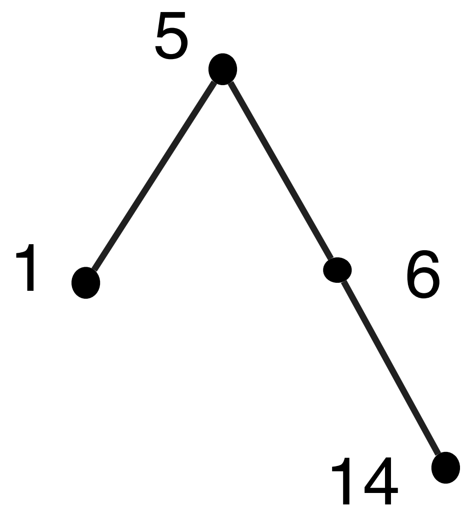
We have made several minor changes from HtDP:

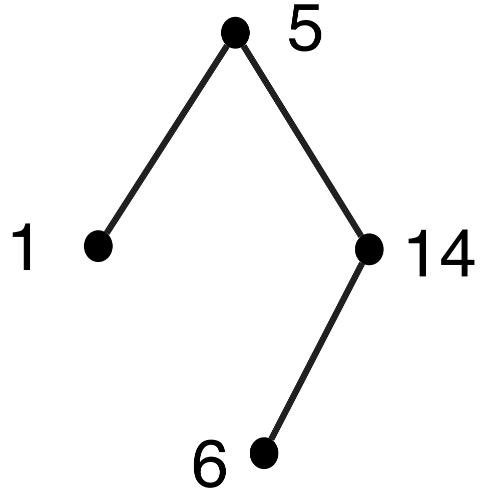
- We use empty instead of false for the base case.
- We use key instead of ssn, and val instead of name.

The value can be any Scheme value.

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

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





A BST template

```
;; my-bst-fun: bst → any
(define (my-bst-fun t)
  (cond
    [(empty? t) . . . ]
    [else . . . (node-key t) . . .
          . . . (node-val t) . . .
          . . . (my-bst-fun (node-left t)) . . .
          . . . (my-bst-fun (node-right t)) . . . ]))
```

Counting values in a BST

Some uses of the BST template do not make use of the ordering property (e.g. counting values equal to v).

```
;; count-values: bst string → 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))]))
```

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)))]))
```

Making use of the ordering property

Main advantage: for certain computations, one of the recursive calls in the template can always be avoided.

This is more efficient (sometimes considerably so).

In the following slides, we will demonstrate this advantage for searching and adding.

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

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: num bst → (union string false)
;; Produces value associated with n or false if n is not in t.
(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))]
    [(> n (node-key t)) (search-bst n (node-right t))]))
```

Creating a BST

How do we create a BST from a list of keys?

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.

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 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.

In later courses, you will see ways to keep a BST “balanced” so that “most” nodes have nonempty left and right children.

By that time 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, evolution 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.