CS115 - Module 8 - Binary trees

Cameron Morland

Fall 2017

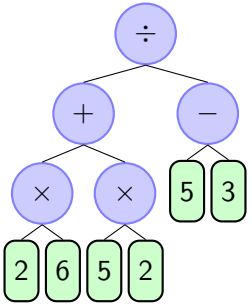
Reminder: if you have not already, ensure you:

- Read *How to Design Programs*, Section 14.

Operators such as $+$, $-$, $\times$, and $\div$ take two arguments, so we call them "binary operators". We have worked with these ourselves for years, but now we can get our computers to work with them for us. Consider:

$$((2 \times 6) + (5 \times 2)) \div (5 - 3)$$

We can split this expression into two expressions, and then recursively split them!



How can we represent this tree which stores a binary arithmetic expression?

Important features:

- Every **leaf** is a number
- Every **internal node** is an operator, and it operates on its **children**.

To represent the operators, we can use `Sym`:

```
;; an Operator is (anyof '+ '- '* '/)
```

One good approach for describing the tree: use a structure for each internal node.

```
(define-struct binode (op arg1 arg2))
;; a binary arithmetic expression internal node (BINode)
;; is a (make-binode Operator BinExp BinExp)

;; A binary arithmetic expression (BinExp) is either:
;;   a Num or
;;   a BINode
```

`(make-binode '* 7 6)` $\Leftrightarrow$ 7×6

`(make-binode '* 7 (make-binode '+ 2 4))` $\Leftrightarrow$ $7 \times (2 + 4)$

Evaluating binary arithmetic expressions

```
(make-binode '* 7 (make-binode '+ 2 4))
```

Evaluation works just like in tracing: evaluate arguments, recursively. Then apply the appropriate function to the two arguments. A number evaluates to itself.

Exercise: Complete `eval-binexp` so it can handle `'+` and `'*`.

```
;; (eval-binexp expr) return the value of expr.
;; eval-binexp: BinExp -> Num
;; Examples:
(check-expect (eval-binexp (make-binode '* 7 6)) 42)
(check-expect (eval-binexp (make-binode '* 7 (make-binode '+ 4 2))) 42)
```

Exercise: For completeness, extend `eval-binexp` so it also handles `'-` and `'/`.

A template for certain binary trees

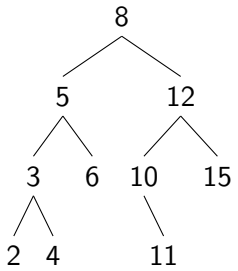
In order to develop a template which works generally, I will use a generic tree definition:

```
(define-struct bintree (label child1 child2))
;; a BinTree is a (make-bintree Any Any Any)
;; Requires: child1 and child2 are each either
;;   a BinTree or
;;   something else.
```

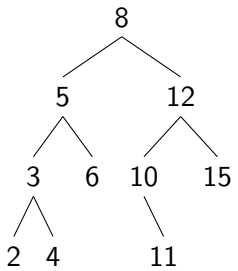
Following this, many binary tree functions can be created starting from the following template:

```
(define (my-bintree-fun T)
  (cond [(... T) ...] ; Some base case.
        [else (... (bintree-label T) ...
                   ... (my-bintree-fun (bintree-child1 T)) ...
                   ... (my-bintree-fun (bintree-child2 T)) ... )]))
```

I can store information in a tree very efficiently.
Suppose we design as follows: a tree stores a value as its label. It stores all smaller values in a tree, which is its left child. It stores all larger values in a tree, which is its right child.



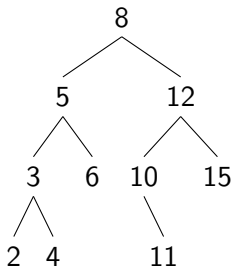

```
(define-struct snode (key left right))
;; a SNode is a (make-snode Num SSTree SSTree)

;; a simple search tree (SSTree) is either
;; * '() or
;; * a SNode, where keys in left are less than key, and
     in right greater.

(define tree12
  (make-snode 12
              (make-snode 10
                          '()
                          (make-snode 11 '() '()))
              (make-snode 15 '() '())))
```




```
(define-struct snode (key left right))
;; a SNode is a (make-snode Num SSTree SSTree)

;; a simple search tree (SSTree) is either
;; * '() or
;; * a SNode, where keys in left are less than key, and
    in right greater.

(define tree12
  (make-snode 12
              (make-snode 10
                          '()
                          (make-snode 11 '() '()))
              (make-snode 15 '() '())))
```

Exercise: Complete `count-leaves`.

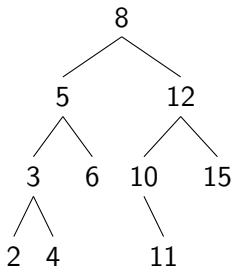
```
;; (count-leaves tree) return the number of leaves in tree.
;; count-leaves: SSTree -> Nat
;; Example:
(check-expect (count-leaves tree12) 2)
```




```
(define-struct snode (key left right))
;; a SNode is a (make-snode Num SSTree SSTree)

;; a simple search tree (SSTree) is either
;; * '() or
;; * a SNode, where keys in left are less than key, and
    in right greater.

(define tree12
  (make-snode 12
              (make-snode 10
                          '()
                          (make-snode 11 '() '()))
              (make-snode 15 '() '())))
```

Exercise: Complete `tree-sum`.

```
;; (tree-sum tree) return the total of all values in tree.
;; tree-sum: SSTree -> Num
;; Example:
(check-expect (tree-sum tree12) 48)
```




```
(define-struct snode (key left right))
;; a SNode is a (make-snode Num SSTree SSTree)

;; a simple search tree (SSTree) is either
;; * '() or
;; * a SNode, where keys in left are less than key, and
    in right greater.

(define tree12
  (make-snode 12
              (make-snode 10
                          '()
                          (make-snode 11 '() '()))
              (make-snode 15 '() '())))
```

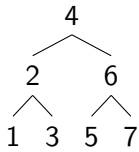
Exercise: Complete `tree-search`. Clever bit: only search `left` or `right`, not both.

```
;; (tree-search tree item) return true if item is in tree.
;; tree-search: SSTree Num -> Bool
;; Example:
(check-expect (tree-search tree12 10) true)
(check-expect (tree-search tree12 7) false)
```

Potential issues with binary search trees

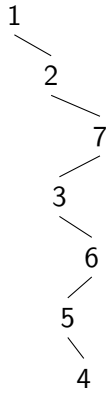
A binary search tree will be fast to search only if it is **balanced**, meaning both children are of approximately the same size.

A completely balanced tree:



It is thus important to keep such trees "reasonably well" balanced. This is an interesting but complex topic! We are not going to discuss how to keep trees balanced here, but it's something to think about.

A completely unbalanced tree:



Extending binary search trees to create better dictionaries

Now we can find items in a binary tree faster than if we simply stored them in a list.
Earlier as dictionaries we stored a list of key-value pairs:

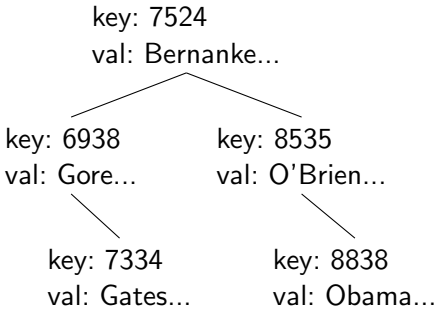
```
(define student-dict
  (list (list 6938 (make-student "Al Gore" 'government))
        (list 7334 (make-student "Bill Gates" 'appliedmath))
        (list 7524 (make-student "Ben Bernanke" 'economics))
```

By extending our tree data structure a tiny bit, we can store a values (`val`) as well as a `key`, to make a fast dictionary!

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

;; A binary search tree (BST) is either
;; * '() or
;; * (make-node Nat Str BST BST)...
;; which satisfies the ordering property recursively:
;; * every key in left is less than key
;; * every key in right is greater than key
```

BST Dictionaries

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

(define-struct student (name programme))

(define student-bst
  (make-node 7524
    (make-student "Ben Bernanke" 'economics)
    (make-node 6938
      (make-student "Al Gore" 'government)
      '()
      (make-node 7334
        (make-student "Bill Gates" 'appliedmath)
        '()
        '()))
    (make-node 8535
      (make-student "Conan O'Brien" 'history)
      '()
      (make-node 8838
        (make-student "Barack Obama" 'law)
        '() '()))))
```



For compactness, we usually draw only the `key` on our trees.
The `val` could be any type.

Dictionary lookup

Our lookup code is almost exactly the same as `tree-search` in a `SSTree`. The are two small differences: the structure has an extra field; and when we find the `key`, we return the corresponding `val`, instead of `true`.

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

;; (dict-search dict item) return correct val if item is in dict.
;; dict-search: BST Num -> Any
;; Example:
(check-expect (dict-search student-bst 6938)
              (make-student "Al Gore" 'government))
(check-expect (dict-search student-bst 9805) false)

(define (dict-search dict item)
  (cond [(empty? dict) false]
        [(= item (node-key dict)) (node-val dict)]
        [(< item (node-key dict)) (dict-search (node-left dict) item)]
        [(> item (node-key dict)) (dict-search (node-right dict) item)]
        ))
```

Creating a BST

It's easier to create a list of key-value pairs than a BST. So let's discuss how to convert a list to a BST.

First consider: how can we add one key-value pair to a BST?

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

;; A binary search tree (BST) is either
;; * '() or
;; * (make-node Nat Str BST BST)...
;; which satisfies the ordering property recursively:
;; * every key in left is less than key
;; * every key in right is greater than key
```

Creating a BST

- If the BST is empty, return a single node which represents the key-value pair. (Base case.)
- Otherwise, add the key to the left or right child, as appropriate.

How to do this? We need to use the template for a function that consumes and returns a `make-node`, and modify it a little.

```
;; (mynode-template tree) do something to tree, and return it.
;; mynode-template: BST -> BST
(define (mynode-template tree)
  (make-node (node-key tree)
             (node-val tree)
             (node-left tree)
             (node-right tree)))
```

Most of this we don't change; we make a copy of the same node. Recursively add to the `(node-left tree)` or `(node-right tree)`.

Creating a BST

Exercise: Complete `dict-add`.

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

;; A binary search tree (BST) is either
;; * '() or
;; * (make-node Nat Str BST BST)...

;; (dict-add newkey newval tree) return tree with newkey-newval added.
;; dict-add: Num Any BST -> BST
;; Examples:
(check-expect (dict-add 4 "four" '()) (make-node 4 "four" '() '()))
(check-expect
 (dict-add 6 "six" (dict-add 2 "two" (dict-add 4 "four" '())))
 (make-node 4 "four"
            (make-node 2 "two" '() '())
            (make-node 6 "six" '() '())))
```

Creating a BST

These ideas will let us add one item to a tree. How can we extend that to add a list of items? Idea: add the first item of the list to the tree created by adding the rest of the list to the tree.

Exercise: Complete `expand-bst`.

```
;; (expand-bst L tree) add all items in L to tree, adding the last first.
;; expand-bst: (listof (list Num Any)) BST -> BST
;; Example:
(check-expect
 (expand-bst (list (list 4 "four")) '())
 (make-node 4 "four" '() '()))
(check-expect
 (expand-bst (list (list 2 "two") (list 6 "six") (list 4 "four")) '())
 (make-node 4 "four"
            (make-node 2 "two" '() '()) (make-node 6 "six" '() '())))
```

Module summary

Become familiar with the vocabulary of trees: parents, children, and siblings; the root and leaves; subtrees.
Be able to write programs which consume or return binary trees.
Be able to implement binary search trees, including those which store additional data in the nodes, and those which do not.
Be able to develop and use templates for other binary trees, not necessarily presented in lecture.

Before we begin the next module, please

- Read *How to Design Programs*, Intermezzo 3 (Section 18); Sections 19-23.