Part 2

—

Simple Dictionaries

CSCI 3110 Code

Summer 2015

1 Introduction

Throughout the implementation of algorithms, we'll use dictionaries to maintain sets of items and associated information. In an imperative world, we'd almost always use hash tables to implement this. In a functional setting, we're left with deterministic constructions: association lists, search trees, and arrays. Which of these is most appropriate depends on the capabilities of the types we want to store in the dictionary, captured by different type classes. We do not care about dynamic updates of dictionaries. We want to turn a sequence of key-value pairs into a dictionary and we want to have a search function that allows us to report all values associated with a given key (in case there is more than one key-value pair in the input with the same key.) So every dictionary should support two operations:

$$\begin{aligned} &\mathit{makeDict} :: [k] \to [(k,v)] \to d \\ &\mathit{dictFind} \;\; :: d \to k \to [v] \end{aligned}$$

In some of the applications where we use this type of dictionary, we also care about not disturbing the order of elements with the same key. In particular, if the pairs with key k in the key-value list given to *makeDict* are $(k,v_1),(k,v_2),\ldots,(k,v_t)$ in this order, then *dictFind d k* should return the list $[v_1,v_2,\ldots,v_t]$. We take care to ensure this property in all our dictionary implementations here.

2 *Eq* Keys

If all we can do with the keys is to test them for equality, then an association list is the best possible dictionary type we can find:

$$\textbf{newtype}\ \mathit{EqDict}\ k\ v = \mathit{EqDict}\ [(k,v)]$$

$$\begin{aligned} &\mathit{makeEqDict} \;\; :: \mathit{Eq}\ k \Rightarrow [k] \to [(k,v)] \to \mathit{EqDict}\ k\ v \\ &\mathit{makeEqDict}\ \_ = \mathit{EqDict} \end{aligned}$$

```
eqDictFind                  :: Eq k ⇒ EqDict k v → k → [v]
eqDictFind (EqDict xs) k = map snd (filter (λ(k', _) → k ≡ k') xs)
```

In this case, building the dictionary takes constant time because all we do is wrap the given list in an *EqDict*. Every *eqDictFind* operation takes linear time because it needs to scan the entire list to find elements whose keys match the search key.

3 *Ord* Keys

If we have an ordering on elements of type v, we can use a search tree structure as our dictionary. We could use an (a, b)-tree, discussed later in class, to do this, but this would be overkill because, as agreed above, we do not care about dynamic updates of the dictionary. Thus, we implement a very simple binary search tree here:

```
data OrdDict k v  = EmptyOrdDict
                  | OrdDict {key           :: k
                            , vals          :: [v]
                            , leftChild, rightChild :: OrdDict k v
                            }

makeOrdDict       :: Ord k ⇒ [k] → [(k, v)] → OrdDict k v
makeOrdDict _ [ ] = EmptyOrdDict
makeOrdDict _ xs  = mergeDicts                                                    $
                    map (λxs → OrdDict (fst $ head xs) (map snd xs) EmptyOrdDict EmptyOrdDict) $
                    groupBy ((≡) on fst)                                          $
                    mergeSortBy (compare on fst) xs

  where mergeDicts [d] = d
        mergeDicts ds  = mergeDicts (mergePairs ds)

        mergePairs [ ]                = [ ]
        mergePairs [d]                = [d]
        mergePairs [d1, d2]           = [d2 {leftChild = d1}]
        mergePairs [d1, d2, d3]       = [d2 {leftChild = d1, rightChild = d3}]
        mergePairs (d1 : d2 : d3 : d4 : ds) = d2 {leftChild = d1, rightChild = d3} : d4 : mergePairs ds
```

makeOrdDict sorts the elements in the input list by their keys (using *mergeSortBy*) and groups them by their keys (using *groupBy*). Since *mergeSortBy* is stable, it does not alter the order of the elements with the same key, that is, the elements in each of the list produced by *groupBy* appear in the same order as in the input list. *makeOrdDict* then turns each such list into a singleton dictionary. Next it merges this sorted list of singleton dictionaries in a bottom-up fashion. It maintains the invariant that every other entry in the current list of dictionaries is a singleton dictionary. *mergeDicts* then builds bigger

dictionaries by making the predecessor and successor of such a singleton dictionary children of the singleton dictionary. *mergeSortBy* takes $O(n \lg n)$ time. *groupBy* takes linear time, as does turning all lists in the output of *groupBy* into singleton dictionaries. It is not hard to see that the input size of each recursive call of *mergeDicts* is half the input size of its parent invocation, so *mergeDicts* takes linear time. In total, *makeOrdDict* takes $O(n \lg n)$ time, and the height of tree produced by *mergeDicts* is $\lceil \lg n \rceil$.

ordDictFind now applies standard binary tree search to find the elements in the dictionary we are looking for:

```
ordDictFind      :: Ord k ⇒ OrdDict k v → k → [v]
ordDictFind d k = fd d
  where fd EmptyOrdDict      = [ ]
        fd (OrdDict k' vs l r) | k ≡ k'      = vs
                               | k < k'      = fd l
                               | otherwise = fd r
```

Since each recursive call of *fd* takes constant time and the height of the given dictionary is $\lceil \lg n \rceil$, *ordDictFind* takes $O(\lg n)$ time.

4 *Ix* Keys

If the keys are of an *Ix* type, we can use an array as our dictionary structure:

```
newtype IxDict k v = IxDict (Array k [v])

makeIxDict       :: Ix k ⇒ [k] → [(k, v)] → IxDict k v
makeIxDict ks xs = IxDict $ fmap reverse $ accumArray (flip (:)) [ ] (l, u) xs
  where l = minimum ks
        u = maximum ks
```

So *makeIxDict* produces the list *ks* of keys in the input and scans this list twice to find the minimum and maximum key. This is the index range of the array we need to allocate. The array antries are then constructed by collecting the values associated with the different keys using *accumArray*. In the interest of efficiency, we prepend the value v of every key-value pair (k, v) to the list of values associated with k so far. Thus, building this array takes constant time per list element, linear time in total, but the list of values associated with each keys stores these values in reverse order compared to the order in which they appear in the input. To fix this, we apply *reverse* to each arry element using *fmap reverse*, which ensures the values in each list are once again stored in the order they occur in the input. Reversing all these lists takes linear time in their total size, that is, linear time in the size of the input list to *makeIxDict*. In total, *makeIxDict* thus takes linear time.

Searching for a key in the dictionary now reduces to a simple array lookup and thus takes constant time:

```
ixDictFind                :: Ix k ⇒ IxDict k v → k → [v]
ixDictFind (IxDict a) k = a ! k
```