Lab 6: Sorting and Searching
12:00 PM, Oct 14, 2018

Contents

1 List Dictionaries 1
2 Flexible Merge Sort 2
3 Sorting Pairs 2
4 Binary Search Trees 3
5 Searching the Tree 5
6 Timing 6

1 List Dictionaries

Professor Klein needs a way to look up a CS17 student’s name by their Banner ID, and has requested HTA Cody’s help. Cody has an idea; we can make a list of two-element lists where the first element is a number representing the student’s Banner ID (with the B removed) and the second element is a symbol representing the student’s name! Unfortunately, Cody has made plans to go to Boston and eat a pie, and doesn’t have time to write the code to do this. He needs your help!

We will call such a two-element list a key-value pair. In order to make the code more readable and more maintainable, you will first write procedures to construct and access these lists using appropriate names.

Task: Write three utility procedures to work with key-value pairs:

- a procedure key-value-pair that takes a key and a value and returns a key-value pair,
- a procedure get-key that takes in a key-value pair and returns the key, and
- a procedure get-value that takes in a key-value pair and returns the value.

You don’t need to follow the design recipe on these very simple procedures. These are written using familiar list-processing operations car, cdr, and cons.

Example:

(key-value-pair 10000000 (quote Cody))
=> (10000000 Cody)
(get-key (quote (10000000 Cody)))
=> 10000000
Task: Write a procedure list-search. This procedure should take in two arguments. The first argument, \texttt{db}, should be a list of key-value pairs. We'll call this a \textit{list-dict}, short for \textit{list-based dictionary}. The second argument, \texttt{id}, should be a number. This procedure should find a key-value pair in \texttt{db} where the key is \texttt{id}, and should return the corresponding value. You may assume that no two key-value pairs have the same key, and that \texttt{id} is in one of the key-value pairs. Example:

\begin{verbatim}
(get-value (quote ((10000000 Cody)))
=> Cody

(list-search (quote ((10000000 Cody))) 10000000)
=> Cody

(list-search (quote ((10000000 Cody) (11111111 Rishi))) 11111111)
=> Rishi
\end{verbatim}

You've reached a checkpoint! Please sign up to get a lab TA to review your work.

2 Flexible Merge Sort

In lecture, we've recently covered the MergeSort algorithm. Our implementation of Merge Sort takes in a list of numbers and returns a list of those numbers sorted from least to greatest. The code can be seen in lecture notes for October 12.

Now, this is, of course, an excellent sorting procedure. But what if we want the list sorted in descending order, from largest to smallest? Or perhaps we want it sorted by absolute value. Or maybe we want to sort polygons by area, or employee records by salary, or .... This implementation of \texttt{mergesort} will not provide this flexibility, and we don't want to write a new implementation every time we want to sort something a different way.

The solution: \textit{comparators}. For our purposes, a comparator is a procedure that takes two arguments, and returns \texttt{#true} if the first should come before the second, and \texttt{#false} otherwise.

Examples of comparators:

\begin{verbatim}
;; input: two integers, \texttt{x} and \texttt{y}
;; output: boolean representing if \texttt{x} is greater than \texttt{y}
(greater-than-comparator 2 1)
=> #true

(greater-than-comparator 1 2)
=> #false

;; input: two integers, \texttt{x} and \texttt{y}
;; output: the boolean representing if the absolute value of \texttt{x} is less than the absolute value of \texttt{y}
(abs-val-comparator -5 4)
=> #false

(abs-val-comparator -3 4)
=> #true
\end{verbatim}
Instead of hardcoding our procedure to use < to compare list elements, we can have the procedure take in a comparator as a second argument.

**Task:** Modify the provided implementation of `mergesort` so the first argument is a list of arbitrary data objects and the second argument is a comparator for those data objects; the output is the list consisting of data objects in the original list, but sorted according to the comparator.

| You’ve reached a checkpoint! Please sign up to get a lab TA to review your work.

## 3 Sorting Pairs

The lookup procedure from Section 1 works great! Unfortunately, it is somewhat slow. Cody’s currently on a train to Boston and can’t code right now, but he’s had another brilliant idea; if we sort the list so the Banner IDs are strictly increasing, then it should be much easier to find the given ID in the list and return the name!

Now, you could write a completely new procedure that sorts these dicts, but that seems like a lot of work. What if there were a way to use some code you’ve already written to sort these list-dicts...

As luck would have it, there is a way!

**Task:** Using your `mergesort` from the first task, write a procedure, `dict-sort`, that takes in a list-dict and returns a list with all of the sublists in that list-dict sorted by the Banner ID. The procedure should be only one line, and should just call `mergesort` with an appropriate comparator. The comparator should be specified with a lambda expression inside `dict-sort`. It should use the utility key-value-pair procedures you wrote in Section 1. Example:

```
(dict-sort (quote ((11111111 Rishi) (10000000 Cody))))
=> ((10000000 Cody) (1111111 Rishi))
```

| You’ve reached a checkpoint! Please sign up to get a lab TA to review your work.

## 4 Binary Search Trees

Great job on the sorting! Unfortunately, Cody was fairly short-sighted on assigning that task, since it still takes linear time to look up a name in a sorted list-dict. He’s really sorry about that. (He was thinking about Semester II, when we’ll start using a data structure called an *array*.)

However, he’s pretty sure we can leverage this sorting idea into something that will allow for quick look-up. His idea is that, using the sorted list-dict, we can make a *Binary Search Tree* containing all of the same information. He’d do it himself, but he’s got a whole bunch of pie he has to eat.

A Binary Search Tree (BST) is a rooted tree consisting of nodes and leaves. Each node has a key, a value, and two other Binary Search Trees, called its left subtree and its right subtree. A leaf contains no information, and is used to indicate that the bottom of the tree has been reached. The tree is sorted by its keys, to make it easy to look up values. Every key in a node’s left subtree should be less than the node’s key, and every key in a node’s right subtree should be greater than the node’s key. The first node in the tree is called the root. The following diagram is an visual example of a BST (Binary Search Tree). Empty circles are leaves and circles with numbers and letters are nodes.
For our purposes, we’ll be representing a node by a three-element list. The first element of the list will be a key-value pair (see Section 1). The second element will be the node’s left subtree, and the third element will be the node’s right subtree. A leaf will be represented by the empty list. Consider the tree below:

This tree would be represented by the list `((2 B) ((1 A) () ()) ((3 C) () ()`).

First, to make the process easier, write a couple of utility procedures for dealing with BSTs.

**Task:** Write utility procedures to work with BSTs:

- a procedure `binary-search-tree` that takes a key-value pair, a left subtree (possibly empty) an a right subtree (possibly empty), and returns a new BST, where the root stores the given key-value pair;

- a procedure `get-key-value-pair` that takes a BST and returns the key-value-pair associated with the root;

- a procedure `get-left-subtree` that takes a BST and returns its left subtree;

- a procedure `get-right-subtree` that takes a BST and returns its right subtree.

You don’t need to follow the design recipe on these very simple procedures. These are written using familiar list-processing operations `car`, `cdr`, and `cons`.

Examples:
Now, we’re going to write code to construct a binary search tree from a list of key-value pairs. Recall Lecture on October 12: it was advantageous that the rooted tree of invocations of \texttt{mergesort} consist of a small number of levels. For similar reasons, we want a BST to be \textit{balanced}. That is, for each node, we want that node’s left subtree to have approximately the same number of nodes as its right subtree. The strategy for achieving balance is similar to that used in \texttt{mergesort}: use \texttt{take} and \texttt{drop} to extract appropriate sublists of the list of key-value pairs.

Recall that we also need all the keys in the left subtree to be less than the key at the root, and we need all the keys in the right subtree to be greater than the key at the root. To make this work, you should start with a list of key-value pairs that is already sorted by key.

\textbf{Task:} Write a procedure \texttt{make-tree} that takes in a sorted list-dict and returns a BST of the items in the list-dict.

\textbf{Hint:} You may find \texttt{take} and \texttt{drop} from Homework 4 useful for this task!

Example:

```
(make-tree (quote ()))
=> ()
(make-tree (quote ((10000000 Cody)))))
=> ((10000000 Cody) () ()))
(make-tree (quote ((11111111 Rishi) (12222222 Shawna) (13333333 Jacob))))
=> ((12222222 Shawna) ((11111111 Rishi) () ())) ((13333333 Jacob) () ()))
```

You’ve reached a checkpoint! Please sign up to get a lab TA to review your work.

5 Searching the Tree

Gorgeous trees! Binary Search Trees have an interesting property that allows us to look up values very quickly. Let’s say the algorithm is trying to find the value associated with 6 in the tree (I know this isn’t a valid Banner ID, but bear with me for a second).

The algorithm first inspects the root of the tree. If the key of this node is 6, then— great, the algorithm returns the corresponding value! But what if the key isn’t 6? What if the key of this node is, for example, 8? Well, because of how a BST is laid out, we know that all of the keys less than or equal to 8 are found in the left child, and all of the keys greater than 8 are found in the right child. Since 6 is less than 8, we know the node we’re looking for is in the left child, and we can ignore all of the nodes in the right child entirely.
This allows us to, at each node, essentially ignore half of the remaining nodes. This is much faster than some silly linear search!

Now, if only we could search our trees...

**Task:** Write a procedure, `tree-search`, that takes in a BST, \( T \), and a key, \( k \), and returns the value associated with \( k \) in \( T \). Your procedure should only search the sections of the tree that \( k \) might be found in; you should be able to ignore most of the tree. You may assume that \( k \) is bound to a value in \( T \).

Examples:

```scheme
(tree-search '((5 homer) empty empty) 5)
=> homer
```

```scheme
(tree-search '((8 cody) ((3 shibei) ((1 koyena) empty empty) ((6 evan) empty empty)) ((10 homer) empty empty)) 6)
=> evan
```

You’ve reached a checkpoint! Please sign up to get a lab TA to review your work.

## 6 Timing

Fantastic work! Cody is so excited that you’ve made such an excellent BST implementation. However, now that he’s really thought about it, he’s not sure if searching in a BST will be any faster than searching in a list. So let’s see for sure!

We’ve provided a file, `big-dictionary.rkt`, that contains some very long list-dicts and lists of ids and a procedure, `lookup-all`, that takes in a list of ids, a dictionary (either a list-dict or a BST) containing those ids, and the corresponding search procedure (`list-search` or `tree-search`) to the dictionary, and returns the result of looking up each id in the dictionary.

**Task:** Copy `big-dictionary.rkt` from `/course/cs0170/src/lab06` to your lab directory, and add `(require "big-dictionary.rkt")` to the top of the file.

Provided in the file are several dictionaries, labelled by the number of entries in the dictionary. We provide `dictionary1000`, `dictionary2000`, `dictionary4000`, `dictionary8000`, `dictionary16000`, and `dictionary100000`. For each of these, we also provide an equivalent list of the ids found in the dictionary, called `ids1000`, `ids2000`, `ids4000`, `ids8000`, `ids16000`, and `ids100000`. Each dictionary/id list is a subset of all of the larger dictionaries/id lists.

Since we don’t care about the outputs for this problem, only the runtimes, we’re bring back our favorite procedure from the previous labs, `suppress-output`! The procedure is as follows:

```scheme
(define suppress-output (lambda (x) 0))
```

**Task:** Using `time` and `suppress-output`, record the amount of time it takes DrRacket to run `lookup-all` with the following inputs:

1. `ids1000 dictionary1000 list-search`
2. `ids1000 dictionary2000 list-search`
3. ids1000 dictionary4000 list-search
4. ids1000 dictionary8000 list-search
5. ids1000 dictionary16000 list-search

**Task:** Write a procedure, sort-and-make-tree, that takes in a list-dict, sorts it, and makes a tree out of it. This procedure should be one line, and call your dict-sort and make-tree procedures.

**Task:** Add the following definitions to your file:

lookup-all with the following inputs:

1. (define T1000 (sort-and-make-tree dictionary1000))
2. (define T2000 (sort-and-make-tree dictionary2000))
3. (define T4000 (sort-and-make-tree dictionary4000))
4. (define T8000 (sort-and-make-tree dictionary8000))
5. (define T16000 (sort-and-make-tree dictionary16000))
6. (define T100000 (sort-and-make-tree dictionary100000))

**Task:** Using time and suppress-output, record the amount of time it takes DrRacket to run lookup-all with the following inputs:

1. ids1000 T1000 tree-search
2. ids1000 T2000 tree-search
3. ids1000 T4000 tree-search
4. ids1000 T8000 tree-search
5. ids1000 T16000 tree-search
6. ids1000 T100000 tree-search

What do you notice about how the runtime grows with the size of dictionary for list-dicts and for BSTs?

You’ve reached a checkpoint! Please sign up to get a lab TA to review your work.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS 17 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/csci0170/feedback](http://cs.brown.edu/courses/csci0170/feedback).