COMP50001: Algorithm Design & Analysis

Sheet 7 (Week 8)

Exercise 7.1

There is an algorithm related to quick sort called quick select: given an unsorted array, it returns the kth smallest element, in $\mathcal{O}(n)$ time.

Just as with quick sort the algorithm partitions the input according to a pivot. Unlike quick sort, the algorithm does not recurse on both partitions: given that the kth element is to be found, if the returned pivot index is bigger than k, only the left partition is needed; otherwise, if it is smaller only the right partition is needed; otherwise they are equal and the pivot is the element required.

Implement $qselect :: Ord\ a \Rightarrow Int \to [a] \to Maybe\ a$, a *persistent* version of quick select that works directly on lists. Also implement $qselect'$ with the same signature, but that makes use of $aqselect :: Ord\ a \Rightarrow Int \to STArray\ s\ Int\ a \to Int \to Int \to ST\ s\ (Maybe\ a)$ to perform an in-place quick select, and comment on the performance difference between the two implementations.

Exercise 7.2

Often it is useful to give efficient but mutation-heavy algorithms a pure interface. Using the functions *runSTArray* and *thaw*, implement a pure interface to the *qsort* presented in lectures:

$$qsortp :: Ord\ a \Rightarrow Array\ Int\ a \to Array\ Int\ a$$

Explain if it is possible to implement a similar pure interface to the functions $index :: Int \to Array\ Int\ a \to a$ and $update :: a \to Int \to Array\ Int\ a \to Array\ Int\ a$ which maintains their $\mathcal{O}(1)$ running time.

Exercise 7.3

Consider the following type which represents sets of strings of as:

$$\textbf{data}\ Trie\ a = Bool \prec [(a, Trie\ a)]$$

1. Calculate the time complexity of *member*, in terms of both the size of the trie and the length of the input list:

$$\begin{aligned}
&member :: Eq\ a \Rightarrow [a] \to Trie\ a \to Bool \\
&member\ [\,]\quad\ (e \prec _) = e \\
&member\ (x : xs)\ (_ \prec ys) = maybe\ False\ (member\ xs)\ (lookup\ x\ ys)
\end{aligned}$$

2. If a is a finite type like *Word4*, the time complexity of *member* changes. Calculate its time complexity when $a \equiv Word4$.

3. Implement $insert :: Eq\ a \Rightarrow [a] \to Trie\ a \to Trie\ a$.

4. Assume you have a perfect hashing function on a, called $hashList :: Hashable\ a \Rightarrow a \to [Bool]$. Implement $nubT :: Hashable\ a \Rightarrow [a] \to [a]$ using this hash function and the trie.

The following is a slow, but correct, implementation of selection.

```
slowselect :: Ord a ⇒ Int → [a] → Maybe a
slowselect i xs
  | i < length xs = Just (sort xs !! i)
  | otherwise     = Nothing
```

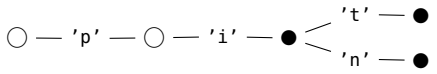
```
swap :: STArray s Int a → Int → Int → ST s ()
swap axs i j = do
  x ← readArray axs i
  y ← readArray axs j
  writeArray axs i y
  writeArray axs j x
```

```
apartition :: Ord a
           ⇒ STArray s Int a
           → Int
           → Int
           → ST s Int
apartition axs p q = do
  x ← readArray axs p
  let loop i j
    | i > j = do swap axs p j
                 return j
    | otherwise = do u ← readArray axs i
                     if u < x
                       then do loop (i + 1) j
                       else do swap axs i j
                               loop i (j − 1)
  loop (p + 1) q
```

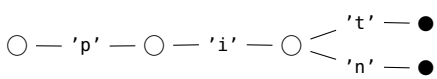
```
runSTArray :: (∀s.ST s (STArray s i a)) → Array i a
thaw       :: Ix i ⇒ Array i a → ST s (STArray s i a)
```

```
fromList :: Eq a ⇒ [[a]] → Trie a
fromList = foldr insert (False ≺ [])
```

```
fromList ["pi","pin","pit"] =
  False ≺
    [('p', False ≺
      [('i', True ≺
        [('n', True ≺ [])
        ,('t', True ≺ [])])])]
```




```
fromList ["pin","pit"] =
  False ≺
    [('p', False ≺
      [('i', False ≺
        [('n', True ≺ [])
        ,('t', True ≺ [])])])]
```



Word4 is the type of 4-bit words, i.e. unsigned integers ranging in value from 0 to 15.

A *perfect* hashing function is one that maps elements without any collisions.

5. Consider another hashing function which returns the bits of the hash in four-bit chunks: $hashChunks :: Hashable\ a \Rightarrow a \to [Word4]$. Calculate the time complexity of $insert \circ hashChunks$ in terms of the cardinality of a.

6. Calculate the time complexity when *hashChunks* returns n-bit chunks.

Exercise 7.4

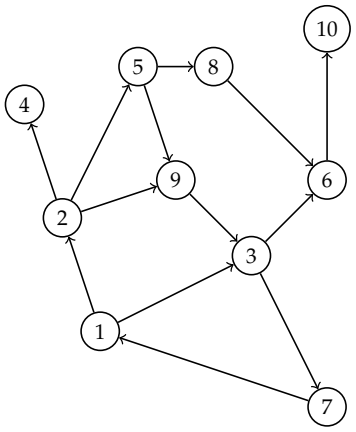
Consider the following representation of graphs:

type $Graph\ a = a \to [a]$

Implement the following two versions of depth-first-search, where $dfs\ g\ r$ returns a list of all the nodes in the graph g visited in depth-first-search order starting from a root r.

$dfs\ :: Ord \quad a \Rightarrow Graph\ a \to a \to [a]$
$dfs' :: Hashable\ a \Rightarrow Graph\ a \to a \to [a]$

dfs should have complexity $\mathcal{O}(n \log n)$, and *dfs'* should be $\mathcal{O}(n)$. You may use data structures implemented in other assignments to implement this solution.



$graph :: Graph\ Int$
$graph\ 1 = [2,3]$
$graph\ 2 = [4,5,9]$
$graph\ 3 = [6,7]$
$graph\ 4 = [\,]$
$graph\ 5 = [8,9]$
$graph\ 6 = [10]$
$graph\ 7 = [1]$
$graph\ 8 = [6]$
$graph\ 9 = [3]$
$graph\ 10 = [\,]$

A valid result for $dfs\ \ graph\ \ 3$ is $[3,6,10,7,1,2,4,5,8,9]$.

Exercise 7.5

Given a uniformly-distributed hash function and n buckets, calculate the chance of a collision happening after inserting m elements.

Exercise 7.6

A bloom filter is a *probabilistic* data structure which can act as a set, with some small probability of false positives in the set. It works by storing a bit array; to insert a value into the set, take multiple hash functions, run each of them on the value, and set the corresponding bits in the bit array.

Implement $nubB :: [a \to Int] \to [a] \to [a]$ which uses a bloom filter of size 256. The first argument is the list of hash functions that should be used.

Exercise 7.7

Cuckoo hashing is an efficient way to handle hash collisions in a hash table. It works by storing two hash tables, with two different hash functions. When inserting, if an element already is in the bucket being inserted into, it is kicked out and inserted into the other hash table. This procedure repeats until an empty bucket is found or the whole hash table has been traversed, at which point two new tables (double the size) are allocated.

Implement a set as a mutable hash table with *insert* and *member* functions using cuckoo hashing.

data $HashSet\ s\ a = ...$
$newHashSet :: ST\ s\ (HashSet\ s\ a)$
$insert\ \ :: Hashable\ a \Rightarrow a \to HashSet\ s\ a \to ST\ s\ ()$
$member :: Hashable\ a \Rightarrow a \to HashSet\ s\ a \to ST\ s\ Bool$