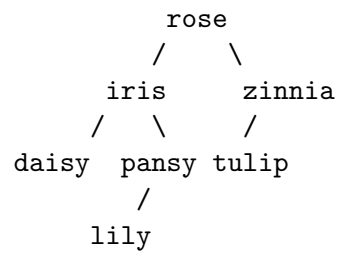
CSE250 Prelim II Answer Key Spring 2022

Closed book, closed-notes-except-for-1-sheet, closed neighbors, 48 minutes. This question paper has THREE problems totaling 67 pts., subdivided as shown. Please show all work on these sheets.

(1) (4+4+3+4+4 = 19 pts.) Consider this binary search tree ordered by alphabetical order:



(a) Which of the following orders of inserting words could have produced this tree?

(i) `iris, pansy, lily, rose, tulip, zinnia, daisy`

(ii) `rose, tulip, zinnia, iris, daisy, pansy, lily`

(iii) `rose, pansy, lily, iris, daisy, zinnia, tulip`

(iv) `rose, zinnia, iris, daisy, tulip, pansy, lily`

(b) Give the *postorder* transversal of this tree.

(c) Show the tree that results from inserting `phlox`

(d) Show the result of removing `iris` in the original tree above (not your tree from part c).

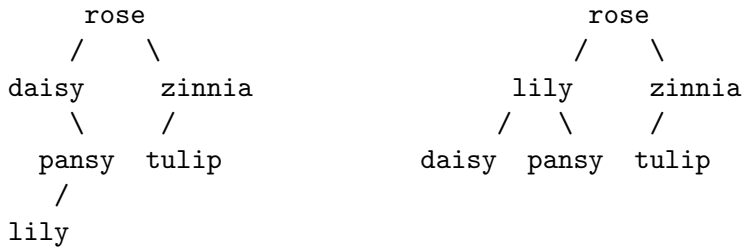
(e) Consider the original tree as an ordinary undirected graph, and regard `daisy` as the start node of a breadth-first search. What is the last node to be visited by that search? (Show work.)

Answers: In (a), order (iv) is the only consistent one: `rose` must be the root, and neither `tulip` nor `pansy` can be the next node since it would have to be a child of `rose`.

(b) The (left-to-right) postorder transversal is `daisy lily pansy iris tulip zinnia rose`.

(c) The tree is as above except that `phloz` becomes the right child of `pansy`.

(d) If we make the inorder predecessor of `iris` the victim" then it is the first ; with my re-use of the `inorderSuccessor` code, it is the second tree.



(e) Breadth-first search from `daisy` expands first (1) `iris`, then (2) `pansy` and `rose` in either order, then (3) `lily` and `zinnia` (in either order depending on the choice of first neighbor in (2)), and finally (4), `tulip`. So `tulip` is the last node visited.

(2) ($10 \times 3 = 30$ pts.)

For each of the following operations in a data structure holding n items, say whether it is:

(a) Guaranteed $O(1)$ time.

(b) Usually $O(1)$ time—now this includes amortized $O(1)$ time.

(c) Guaranteed $O(\log n)$ time.

(d) Usually $O(\log n)$ time but seldom $O(1)$ time.

(e) Guaranteed $o(n)$ time, but seldom $O(\log n)$ time.

(f) Guaranteed $O(n)$ time, but generally not better—i.e., seldom or never $o(n)$ time.

The data structure operations have standard names used in this course and are prefixed by the classes that would have them—without the `[A]` part for the generic type of a data element `item`—and `itr` stands for an iterator. Only the standard `hasNext` and `next()` iterator operations inherited by Scala from Java are used in this question. (Problem (3) on this exam allows you to use others in the `Iter` class.) The operations `find` and `remove` are as in class notes; the text uses `get` and `-=` in related ways. Justifications are not required but could help for partial credit.

1. `Stack.push(item)` *Answer:* (a) Guaranteed $O(1)$ time. (It is true that an array implementation of a `Stack` could have a capacity overflow needing $O(n)$ refresh, but a linked-list implementation does not—and it is part of the definition of `List` that operations at the head are $O(1)$ time. The `Stack` ADT in lecture specified $O(1)$ time for all operations, likewise `Queue` and "`Staque`." Partial credit was given for (b) with explanation.)

2. `SinglyLinkedList.find(item)` *Answer:* (f) $O(n)$ time and seldom better.

3. `SortedArray.find(item)` *Answer:* (c) Guaranteed $O(\log n)$ time by binary search.

4. `BST.find(item)` *Answer:* (d) Usually $O(\log n)$ time—but can be more if tree is unbalanced.

5. `hasNext` for a `BST` iterator *Answer:* (a) Guaranteed $O(1)$ time—it just polls its node for `null` or `endSentinel`.

6. `next()` for a `BST` iterator *Answer:* (b) Usually $O(1)$ time, in fact amortized $O(1)$ time.

7. `SortedArray.remove(ind)` (f) $O(n)$ time and seldom better, because a remove anywhere but at (or near, maybe) the end of an array forces it to reallocate. (There are compromise implementations of arrays along lines of "BALBOA" that would give answer (e); "AIOLI" could even give (d) but is wonkier in practice.)

8. `DoublyLinkedList.remove(itr)` *Answer:* (a) Guaranteed $O(1)$ time—the point of linked lists is to support additions and removals at known lcoations.

9. Searching for the longest word in an $m \times m$ matrix, where $m = \sqrt{n}$. *Answer:* (f) $O(n)$ time and seldom better.

10. Searching for the longest word in an $m \times m$ matrix ($m = \sqrt{n}$) where there is an extra column such that for each row, the entry in that row is the length of the longest word in that row. *Answer:* (e) $o(n)$ time, indeed guaranteed $O(m) = O(\sqrt{n})$ because you can first peruse the extra column with m lengths to get the longest one, and then spend m more steps to find the word in the row that gives that longest length.

(3) (19 pts. total)

Suppose `sds` is an instance of a data structure of n items sorted by `item.key` in which different items having the same key value may be present. Such items will be consecutive in the iteration, and we may suppose that the iterator `itr` returned by `sds.find(item)` always has the first such item via `itr()`. Call an item `item` *special* if the next key `k'` different from `item.key` satisfies a given relation `R(item,k')`. Sketch in Scala-like pseudocode a routine `isSpecial(item)` to determine whether a given item is special.

For a concrete case, R(item,k') can be the relation that the word `k'` is a synonym of the original word in the dictionary entry `item`. Then `item` being "special" means that the next different word in alphabetical order is listed as a synonym. You are welcome to write either `R(...)` or `(...).synonyms.contains(...)` in your code sketch. A helpful case of this in `Fallows1898fx.txt` is that the adjective form of `clean` lists the next word `clear` as a synonym, while `clean` also has a verb form (which does not list `clear`). (And `clear` has its own verb and adjective forms, but you need not worry about that as only the key `k'` of the second word is involved. Fallows has several other special entries, but the first entry `aback` listing `abaft` as a synonym does not count because `abaft` does not appear as an entry—the next word in the dictionary is `abandon`.)

Answer: The design of the question avoided multiple pitfalls, which are discussed at length in extra notes after this key and may be worked into lecture or recitation coverage. Answer code:

```
def isSpecial(item: A): Boolean = {    //A can be SynonymEntry
   val itr = find(item)      //or you could say sds.find(item) as client code
   if (itr.hasNext) {
      val item = itr.next()  //could just write itr() here.
      while (itr.hasNext && keyComp(item, itr()) == 0) {
         itr.next()          //could move this up inside keyComp(...) call
      }                      //and/or test (itr.hasNext && item.key == itr().key)
      if (itr.hasNext) {     //now on next different word
         return (item.synonyms.contains(itr().key))  //or return R(item,itr().key)
      }
   } //control here means item not found or item is last (word in dictionary)
    return false
}
```

The main small mistake was forgetting that `itr` could go off the end inside the body—this happens exactly when `item` is the last entry in the dictionary. The main large mistake was forgetting the need to iterate over duplicate-key items. Presuming that `find(item)` finds `item`, this code is correct: Once we find where `item` is, we advance `itr` until its key no longer compares equal to `item.key`. Then it is on the next different word (or off the end of the dictionary in case `item` is the last item, which we test for). We do not have to worry about different speech forms of the next word since only its simple key matters, so we just test whether the word (which is `itr().key`) is on the synonyms list of the original item and return the Boolean result.

——(This endeth the exam answer key, but a data structures course never ends...)——

Extra notes: I was expecting to get a question during the exam along lines of: What if `find(item)` doesn't find the exact item? This is actually the case with the example of `clean` in Fallows: the second-listed form is the one with `clear` as a synonym, but by the stipulation "the iterator `itr` returned by `sds.find(item)` always has the first such item," the first form will be returned. This does not matter to the above code: the list of synonyms we need is already given in `item` (if it were a dummy item the answer would be automatically `false`) and the while-loop only cares that `itr` arrives somewhere in the range of equal-key items. (Maybe some exam papers will

note this point; two people maybe came close to querying it in the last minutes of the exam. I have not started grading yet.)

On some related tasks, however, you could get hung up by this. Suppose "special" needed the different word to be the next *entry*, not the next word. This would still be true of `clean`—but there is no way you could tell with the above operators. You can't compare `item == itr()` to see where the exact item is because that is reference equality. In fact, the built-in Scala `equals` works in this case. So if you really did need `find(item)` to give `itr` on the same exact item, you could rely on the "first such item" guarantee to iterate up to it:

```
var itr = lookup.find(item)
while (itr.hasNext && (!(itr().equals(item))) && keyComp(item, itr()) == 0) {
   itr.next()
} //now itr is on the exact item, presuming it exists
```

Whether `equals` is guaranteed to work in general is a touchy subject, seemingly as much in Scala as in C++. The main question is whether it compares only the top-level fields or works recursively. In our case we're OK because comparing two Scala `List` objects (the lists of synonyms) automatically tests `equals` on their respective items, because a `List` is a *value object*. In C++ this happens so long as you use value fields—not pointers or references. In Java you get hung up because every object identifier (i.e., apart from the primitive types) is a reference. To do this with both full generality and full safety in Scala, we should do either of the following:

- Make the user pass in `itemEquals` function as well as the `keyComp` function.
- Require the generic type `A` to implement a `trait` with an `itemEquals` function.

I shied away from the former making heavier syntax—having to do the `class ... { Outer =>` business was already a clunker. I had started to go the route of the latter for the key comparison too, but I did want to show the idea of passing in a function. Also because your Assignment 6 did not need an item-equality function, I decided I could leave the equality issue "on the back-burner" for a time like now—with hash tables using `itemMatch`.

Another way to cope, which I considered exploring for coding assignments, could be changing the key-comparator on the fly. Here the idea (in the Assignment 6 context) would be to temporarily change `lookup.keyComp` to a key comparator that would include the part-of-speech field in the key. This would have required making `keyComp` a `var` field:

```
//in definitions at the top:
class SynonymBox(var keyComp: (SynonymEntry,SynonymEntry) => Int)
         extends Cardbox[SynonymEntry](keyComp) { ... }
...
//in the client main:
   def keyComp2(x: SynonymEntry, y: SynonymEntry): Int = x.key.compareTo(y.key)
   val lookup = new SynonymBox(keyComp2)
...
//used later on:
      while (worditr.hasNext && lookup.keyComp(wordAsItem,worditr()) == 0) {
         ...
```

It actually does make sense to say that the key-comparison function pertains to the specific database by having you write `lookup.keyComp`. But I opted for the simpler idea of making `SynonymBox` be defined once with the key comparison fixed, so that the only adaptation would be comparing the

word keys without the part-of-speech tag. Plus: extending the key comparison to include the part-of-speech would have required me to describe the topic of "two-level sorting" which is not supported in the text. The related topics of Bucket Sort (example: how I sort collected exams by first making piles for each letter and then insertion-sorting each pile) and radix sort have been in my course before but seem not to be mentioned in our text.)

Now for a real pitfall, one especially nasty for someone used to how things work in C++. It would arise if we used the idea of keeping `itr` where it is (on the exact matching item) and sending a separate "scout pointer" ahead to find the next different word. We would want to start the scout pointer on the next entry. For a non-nasty pitfall, it is wrong on several counts to define `var scout = itr.next()`. First, `next()` returns the current data item, not an iterator. Second, it has the side effect of advancing `itr`, which we want to keep where it is. Since there isn't a convenient function for getting the next step of an iterator as a pure value, the natural next idea is to define `var scout = itr` and move it ahead one step before comparing:

```
   if (itr.hasNext) {
      var scout = itr  //AOK in C++ but dangerous in Scala
      scout.next()     //OK since itr was in valid position by if (itr.hasNext) test
      while (scout.hasNext && keyComp(item, scout()) == 0) {
         scout.next()
      }   //now scout is on the next different word
      ... //rest of task similar to above
   }
```

The nasty pitfall is that `var scout = itr` is a reference copy, not a value copy. Thus `scout.next()` causes `itr.next()` to be implicitly executed too. When I first tried to do the task of finding all the special words in the Fallows text, I iterated `itr` from the beginning and did not have `item` given as a separate input—rather, `itr()` furnished the item. The reference identity of `itr` and `scout` makes that use of `itr()` throw an unexpected error when both prematurely get to the end. (There is no deduction for this pitfall on the exam.)

In C++ this is averted first by making the iterator a value type of itself (even though it is emulating a "smart pointer") so that the default behavior of assignment is to make a copy. (Well, by copying all top-level fields.) I had expected making `Iter` a `case class` would have this effect, but no. For full safety in C++, one would define both a *copy constructor* **and** a custom assignment operator, which together with a custom destructor are called the "Big Three." In Scala one cannot override the `=` operator. The syntax `update` allows assigning only to those fields read via `apply`, which in our `Iter` class is just the data `item: A` field of the iterator's `at` location. Nor does Scala provide a generic copy constructor—that is, one cannot automatically write

```
      var scout = new lookup.Iter(itr)
```

Instead, Scala wants you to write `var scout = itr.clone` like with `clone()` in Java. But even here there is a gotcha. You can't do this automatically even with Scala's built-in iterators, let alone the ISR ones:

```
/** File "CloneWars.scala" by KWR for CSE250, Spring 2022
    Does not compile.
    Yet another Scala iterator limitation
 */
object CloneWars extends App {
   val myList = List(1,2,3,4,5)
   val liter1 = myList.iterator
```

```
   val liter2 = liter1.clone
}


timberlake<ScalaSamples> scalac CloneWars.scala
CloneWars.scala:9: error: method clone in class Object cannot be accessed
                    as a member of Iterator[Int] from object CloneWars
 Access to protected method clone not permitted because
 prefix type Iterator[Int] does not conform to
 object CloneWars where the access takes place
   val liter2 = liter1.clone
```

I have not yet been able to *grok* this error message myself. But within the `Iter` inner-class of any implementation, one *can* put a one-line method like

```
override def clone = new Iter(at)   //in BSTISR.scala or SortedDLL.scala
```

and then it all magically works. This has to be done individually and differently for the other repo classes, and this is useful enough to warrant a repo update. But IMPHO this really should not be needed: There is no natural reason for iterators not to be copied by value, which in C++ happens the way you want automatically (it does not copy the underlying data, just the `at` field and/or similar uppermost location fields). Even with `case class`, the rule in Scala is that it only emulates a value copy when the data is hierarchically immutable so that reference copy is tantamount to value copy. Making immutable data needs jumping through more hoops than I'd like for CSE250 (it's grist for CSE305, I say), and the text in section 16.3.2 blithely drops in a very advanced piece of syntax `<%` between classes. [On the flip side: in C++ one can purposely make a reference to an iterator, and this was one way of handling the communication between the movie and user data structures on the final project. But I have not had time to make sure this won't break in Scala.]

Zooming out, the takeaway in all of this is that Scala has fundamentally different architecture for handling the basic interface to client data. There are more built-in tools and behaviors but less flexibility in other ways. The Scala code can be wonderfully economical: my C++ **BST** implementation has 516 lines compared to only 287 right now in Scala. But it's different—and different from the purer world of its other ancestor, the functional ML programming language.