CSE250 Week 8: Sorted Data Structures (ch. 13 and into 15)

But first, an excursion into design differences between Scala and C++ (with Java and Python in-between) regarding which elements of data structures should be "**first-class**" objects.

Iterators and Ranges

When this course is taught in C++, the conceptual heart overlaps greatly with the domain of the ANSI C++ Standard Template Library (STL). A glance at Wikipedia's article on STL sufficies to see why:

> The **Standard Template Library** (**STL**) is a software library originally designed by Alexander Stepanov for the C++ programming language that influenced many parts of the C++ Standard Library. It provides four components called *algorithms*, *containers*, *functions*, and *iterators*.[1]

....

> The STL was created as the first library of generic algorithms and data structures for C++, with four ideas in mind: generic programming, abstractness without loss of efficiency, the Von Neumann computation model,[2] and value semantics.

Now value semantics has already been a major topic in this course with Scala. Generic programmimg is what I started exemplifying by having the `++=` method coded in the trait `ISR.scala` using the API elements only, so that the same code will work with any data structure class that implements the API. Abstractness without loss of efficieincy is a general goal. The Von Neumann Model means **random access**, in particular to **fingered** points of entry in data---and that is Scala's main bone of contention with STL.

A lot of the STL's ingredients have similar names in Scala:

> **Containers** [edit]
>
> The STL contains sequence containers and associative containers. The containers are objects that store data. The standard sequence containers include `vector`, `deque`, and `list`. The standard associative containers are `set`, `multiset`, `map`, `multimap`, `hash_set`, `hash_map`, `hash_multiset` and `hash_multimap`. There are also *container adaptors* `queue`, `priority_queue`, and `stack`, that are containers with specific interface, using other containers as implementation.

Associative container means that lookup is *by key* rather than by an already-known *index*. The ISR interface tries to have it both ways by using movable **iterators** in place of fixed indices. But this needs iterators to be objects in their own right. They are the key mediator of STL:

Iterators [edit]

The STL implements five different types of iterators. These are *input iterators* (that can only be used to read a sequence of values), *output iterators* (that can only be used to write a sequence of values), *forward iterators* (that can be read, written to, and move forward), *bidirectional iterators* (that are like forward iterators, but can also move backwards) and *random-access iterators* (that can move freely any number of steps in one operation).

A fundamental STL concept is a *range* which is a pair of iterators that designate the beginning and end of the computation, and most of the library's algorithmic templates that operate on data structures have interfaces that use ranges.[6]

To say that iterators are **first-class** means that they can be stored in variables, operated on, passed as arguments, and (most particularly) tested for equality. One distinguishing point of Scala is that functions are first-class: we've already been freely passing them as arguments either by name or in anonymous "rocket notation" form. However:

- Scala does not natively provide first-class iterators.
- Instead, an `Iterator` stands only for the data that it can produce.
- Moreover, `Range` is treated more as a first-class object.

Sorted data structures naturally give ranges: $[a, b]$ stands for all data items whose key k satisfies $a \leq k \leq b$, where $\leq$ is the key-comparison function. The range of the whole data structure is standardly `[begin ... end)`, with the round paren indicating that the right-hand side is exclusive. The whole range in Scala is more implicit: it is whatever `next()` and `hasNext` give you from the moment the iterator is created.

Iterators on Different Objects

In all languages we know, one can "wrap" the primitive notion of an index into an array into something that obeys the API of an interator. This leads to the easiest illustration of the most common bug when using iterators in C++:

- Using iterators on different objects, say to merge one into the other, but forgetting which object two iterators are on when comparing them.
- Simply put, if `itr1` is on index m of object 1 and `itr2` is on index m of object 2, and you test them for equality, you may think they are in the same place when they are not.
- Standard advice: have the iterator class maintain a reference to its host object as a field, and include that field when testing for equality.
- Still bug-prone.
- Scala averts this bug by making comparing iterators on different objects a **compile-time error!** Same with Nodes in different graph objects.
- But this requires special coding when you really want to do things with different objects...

Priority Queues

This is a kind of associative lookup in which the data item's key is a numerical **priority** and the item's body describes the associated task. Instead of finding the item with a given key, you want to find the item with **maximum** key. (Sometimes the highest priority item is represented as having the minimum key, so that sortedness is lowest-to-highest, but we'll follow the text.) Here are the desired operations:

1. Find a highest-priority item (called `peek` or `findMax`)
2. Not just find but `pop` it. In some data structures, this is a general `remove`.
3. `enqueue` a new task with priority (can also call this `add` or `append`) In some data structures, this requires an `insert` in the middle.
4. Test `isEmpty`, or more generally, compute the current `size` of the priority queue.

Now let's analyze how the choice of container data structure affects the timing for these operations.

I. Unsorted doubly-linked list

1. Finding the max always requires searching the entire list of n items, for $\Theta(n)$ time in all cases.
2. Once you find it, you can splice it out of the linked list in $O(1)$ additional time. This is an advantage of linked lists over arrays, where splicing out an element can require re-jiggering the whole array (or leaving "holes" in it).
3. Can `append` or `prepend` in $O(1)$ time. You'd like to prepend if the new task has relatively high priority, but still need to search entire list to check if it is highest.
4. Testing for the list being just the end-sentinel is $O(1)$ time. Or you can maintain `size` on-the-fly, at the cost of an extra instruction per `insert` or `remove` to update the `_size` field.

II. Sorted doubly-linked list (text, section 13.1.2)

1. Now the max item is always in front, for $O(1)$ time.
2. And removing it is a literal `popFront`, in $O(1)$ time. The ideal situation?
3. Alas, to preserve the **sortedness invariant** when inserting a new item, you need to find the place where it shouled go (`findPlace`).
   (a) In worst case, this requires searching all n items.
   (b) If the priorities are uniformly random, will find on average after comparing $n/2$ items. Still $\Theta(n)$.
   (c) But if new items tend to have lower priorities---such as if priorities increase with time---then the average enqueue time can be considerably less than $n/2$ comparisons.
4. Testing for empty still $O(1)$ time.

Thus we have a "lump-in-the-carpet" situation between 1 and 3: we can make either one $O(1)$ time but

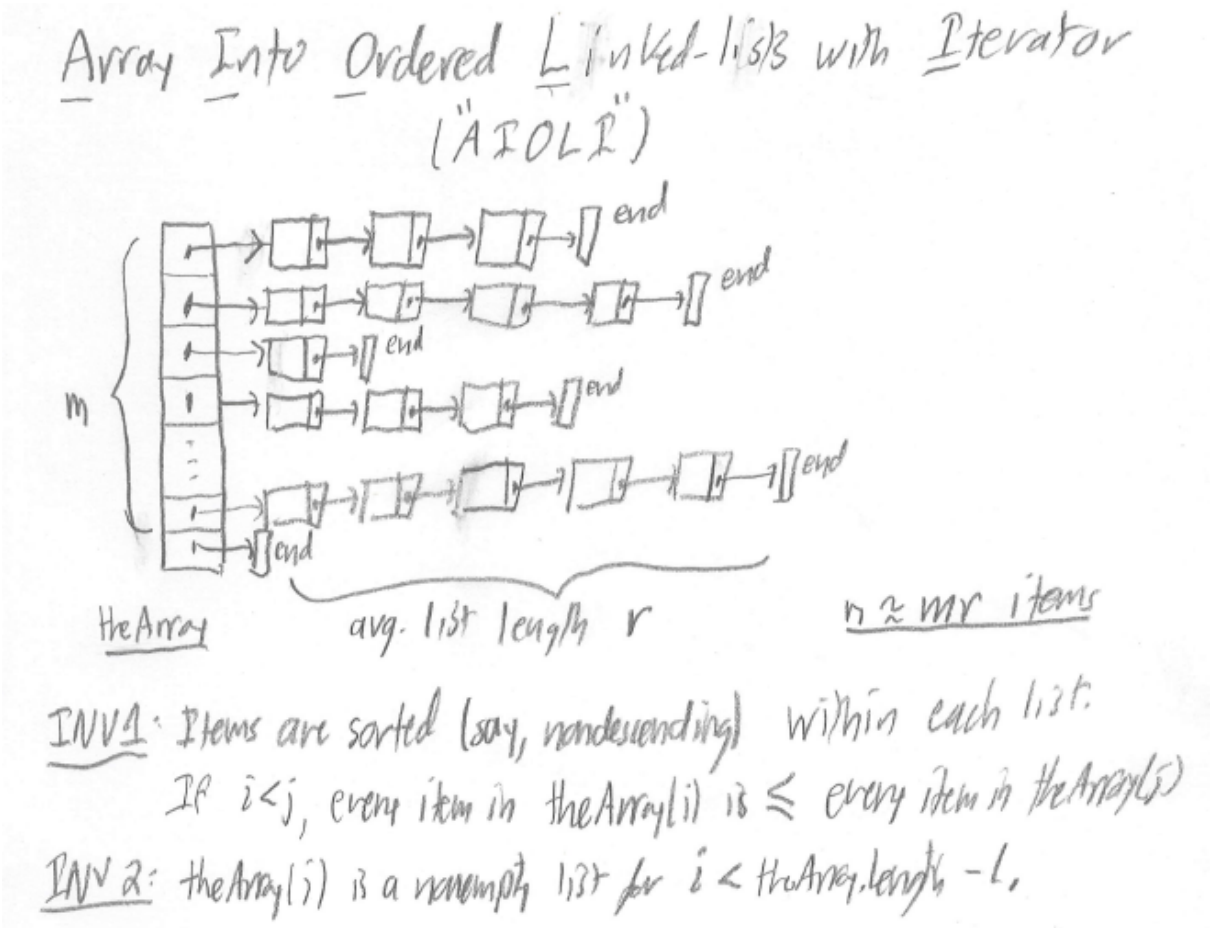
then the other becomes "$O(n)$ best possible" at least on average.

III. Arrays (section 13.1.3)

Unsorted arrays don't help the search. The one time-saving idea is that for removal, one can fill a "hole" in $O(1)$ time by moving the last item into its place.

This sets up the idea of various compromises involving multi-level data structures.

IV. Array of Linked Lists ("AIOLI" code in `/.../cse250/DataStructures/`)



1. The highest (or least, when priority is represented as minimum) element is `theArray(0).begin` which is identified in $O(1)$ time.
2. Popping it from its list is $O(1)$ time. But if the first list then becomes empty, then we need to remove the first array element too---in order to preserve the invariant that each list is nonempty (which we will rely on for binary search in part 3). OK, for `ArrayBuffer`, maybe that is $O(1)$

time, perhaps *amortized*. But if it is $O(m)$ time, "bummer..."

3. To find the place to insert a new item, use **binary search** on the array to find the row i where the new item should go, followed by linear search on list `theArray(i)`. This takes $\Theta(\log m) + \Theta(r)$ time. The actual insertion into the linked list is $O(1)$ time---that is why we are using the lists, not just the array.
4. INV 2, by avoiding "holes", ensures that testing `isEmpty` is always $O(1)$ time.

By maintaining the data structure with a longish array and short lists, i.e.

- $r \sim \log n$
- $m \sim n / \log n$,

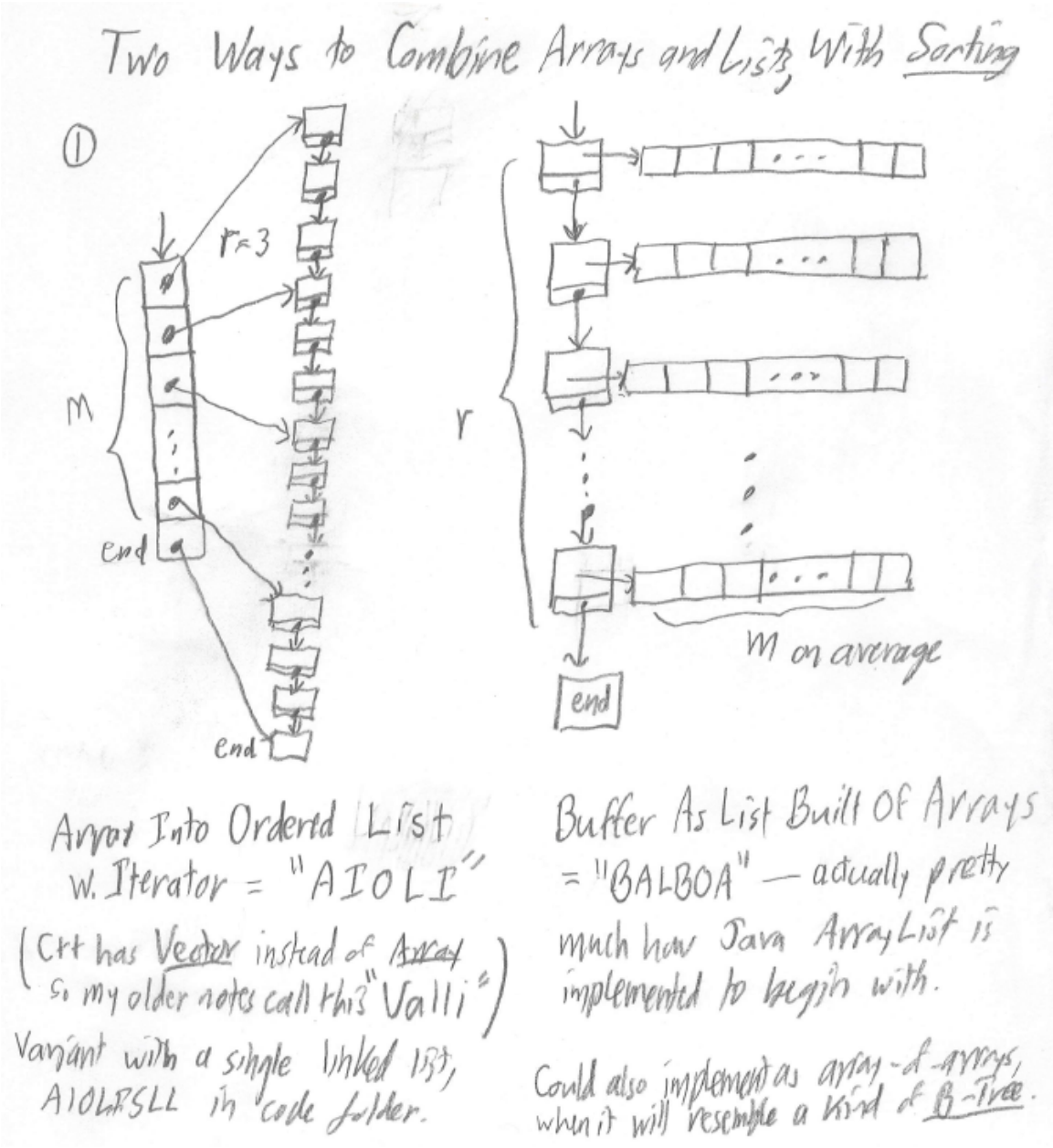
we can achieve time $O(\log n)$ for each operation.

That is, unless removal really takes $O(m)$ time best-possible, as could happen in the more general setting where we can remove any element, not just the highest one. Then the best **tradeoff** becomes $r = m = \sqrt{n}$, giving $O(\sqrt{n})$ time for all.

A different data structure called a **heap**, which we will be ready for after Chapter 16, guarantees $\Theta(\log n)$ time with much less fuss-and-bother. But it works only for the priority queue, not general associative lookup, for which we will use **binary search trees** instead.

V. Linked List of Arrays

The following also shows a variant of "AIOLI" where the linked lists are all joined together. This makes iteration thru the list quicker but makes the overall code trickier.



Here the order-m time shows up on insertion, and that does more to lock us into an $O\left(\sqrt{n}\right)$ tradeoff.

[Show code.]