Cosc 201
Algorithms and Data Structures
Lecture 8 (19/3/2024)
MergeSort completed.
Stacks and queues and heaps

Brendan McCane
brendan.mccane@otago.ac.nz
And
Michael Albert

UNIVERSITY OF OTAGO
Te Whare Wānanga o Otāgo
NEW ZEALAND

Variations on mergesort

- To avoid the recursion (why?) its possible to take a bottom-up approach to mergesort.
- In its basic form that means “first merge all pairs”, then “pairs of pairs” then “pairs of fours”, until you’re done. There’s some messiness at the end of the array.

```
public static void mergeSort(int[] a) {
  int blockSize = 1;
  while (blockSize < a.length) {
    int lo = 0;
    while (lo + blockSize < a.length) {
      int hi = lo + 2*blockSize;
      if (hi > a.length) hi = a.length;
      merge(a, lo, lo + blockSize, hi);
      lo = hi;
    }
    blockSize *= 2;
  }
}
```

Time complexity from the bottom up

- Let n be the number of elements in the array, `a`.
- The outer while loop (controlled by `blockSize`, or b) is executed $log(n)$ times since its upper bound is n and b is doubled each time.
- In the inner loop, the update on `lo` is to add $2b$ so it is executed $n/(2b)$ times.
- The inner loop merges two arrays of size b, so each instance does $\Theta(b)$ work.
- That gives an upper bound on the work done in one instance of the outer loop of the form:

$$(n/(2b)) \times (A \times b) = (A/2) \times n$$

  for some constant A, and there is a matching lower bound.
- Thus, the work done in one instance of the outer loop is $\Theta(n)$
- And so, the total complexity is $\Theta(n \log n)$.
- The bottom-up version does *exactly* the same thing as the top-down version, just in an apparently different order, so this analysis applies to the top-down version as well.

Can we do better?

- If there are already long runs of data that are increasing (or decreasing) it may be more efficient to merge complete runs.
- Suppose we're merging X followed by Y:
  - All the elements of X smaller than the least element of Y are already in their correct places as are all the elements of Y larger than the greatest element of X.
  - The smaller of the two remaining parts can be copied and this frees up enough space to do the remaining merge in place.
  - This saves some memory and sometimes some time.
- Putting all of this (and a bit more) together gets you to Timsort the *de facto* standard sorting algorithm in many languages including Java.
- In case you think sorting is old hat, *Timsort* was introduced in 2002 in Python, whereas quicksort was invented in 1959 and mergesort in 1945.
- In 2022, Powersort replaced Timsort in Python.
- In *Timsort* the runs are arranged in a stack and occasionally merged.

Dynamic linear data types

Let's start with a bit of COMP162 review (Lectures 13 to 16 roughly)

- A *dynamic linear data type* (interface) is an abstraction of a collection of data, organised "in a line" which supports addition of new elements and the removal of (some) old elements.
- The examples considered in COMP162 were stacks and queues.
- They differ in the removal operation
  - In a stack, the removal operation (called *pop*) produces the last of the remaining elements added (Last In First Out - LIFO). In other words, addition and removal occur at the same end of the line.
  - In a queue, the removal operation (variety of names) produces the first remaining element added (First In First Out - FIFO). In other words, addition and removal occur at opposite ends of the line.

Stack and queue representation

- To represent either a stack or a queue we can use an array or a (linked) list.
- The latter seems more natural.
  - For a stack we need maintain only a single reference to the top of the stack and links downward.
  - For a queue we need references both to the front and the back, and links point backwards.
- And yet, the default Java implementation `java.util.ArrayDeque` that supports both abstractions uses an array.
- The reasons are somewhat technical, but mainly have to do with memory management particularly in structures that are being hit with many operations of both types.
- If you need to represent a stack or queue in Java you should use `ArrayDeque` unless there is a **very** good reason to do otherwise.

Priority queues

- The priority queue is another dynamic linear data type that supports addition and removal of entries.
- Each entry has an associated priority (no surprise there) which is (for our purposes) a positive integer.
- The removal operation removes and returns the element in the priority queue of greatest priority.
- Various simple implementations are possible:
  - Store the items and their priorities in an array. Add at the end ($\Theta(1)$), remove by finding the maximum and exchanging with the end element ($\Theta(n)$)
  - Store the items and their priorities in an array (or list) in sorted order. Now removal is $\Theta(1)$ but addition is $O(n)$ since we may need to make space for the new element by moving larger elements along.
- Have a look at the code in the 201 code repository.
- It feels like it should be possible to do better probably by balancing the addition and removal costs somehow.

The idea of the heap

What do we want to be able to do?

- Access and remove the element of greatest priority.
- Add new elements.

The difficulty is to ensure that both operations can be performed without having to move $\Theta(n)$ items (the problem with the sorted version) or examine $\Theta(n)$ items (the problem with the unsorted version).

In the third union-find implementation we saw the idea of using a ranked data structure to store at least 2^k things, while keeping a rank bounded by k. Perhaps that can help?