Choosing Data Structures

"Building algorithms around data structures such as dictionaries and priority queues leads to both clean structure and good performance." [Skiena, ADM]

- first design the ADT – identify how your collection is accessed and what operations are needed
- then choose an implementation that delivers the necessary performance
- isolate the implementation of the data structure from the rest of the code
  - in Java, this means writing a class to implement the ADT with methods for the ADT operations

Choosing Implementations

Consider the characteristics of your task –

- dictionaries
  - how many items? is the size known in advance?
    - if small, simplicity of implementation is most important
    - if very large, running out of memory is an issue
  - what are the relative frequencies of insert, delete, find operations?
    - static (no modifications after construction) and semi-dynamic structures (insertion but no deletion) can be simpler than fully dynamic
  - is the access pattern for keys uniform and random?
    - in some data structures, non-uniform distributions lead to worst-case performance while others can take advantage of temporal locality
  - do individual operations need to be fast, or just minimize the total amount of work of the whole program?
    - focus on achieving good worst case performance vs amortized or expected performance

Implementation Choices for Dictionaries

- for small data sets, unsorted arrays are simple and have better cache performance than linked lists
- for moderate-to-large data sets, hashtables are likely best
- for very large data sets where there isn't enough room in memory, use B-trees
- *self-organizing lists* (move-to-front heuristic) are often better than sorted or unsorted lists in practice
  - many applications have uneven access frequencies and locality of reference
- sorted arrays OK if not too many insertions or deletions
- the inability to use binary search makes sorted linked lists often not worth it
- for balanced search trees, the best choice is likely the one with the best implementation
- *skip lists* are easier than balanced search trees to implement and analyze

Implementation Choices for Dictionaries

- creating good hashtables
  - open addressing has better cache performance, but overall performance decreases quickly with higher load factors
  - with open addressing, *N* should be 30-50% larger than the maximum number of elements expected at once
    - *N* should be prime
  - use a good hash function + an efficient implementation

$$H(S) = \alpha^{|S|} + \sum_{i=0}^{|S|-1} \alpha^{|S|-(i+1)} \times char(s_i) \pmod{m}$$

    - gather stats on the distribution of keys to see how well the hash function performs

Choosing Implementations

Consider the characteristics of your task –

- priority queues
  - max size? is it known in advance?
    - preallocating the necessary space saves having to grow a container
  - are the key values limited?
  - what operations are needed?
    - if no insertion after construction, no need for PQ – just sort
    - other operations: searching for or removing arbitrary elements vs only the min/max
  - can priorities of elements already in the PQ be changed?
    - implies needing to retrieve elements by key, not just the min/max ones

Implementation Choices for Priority Queues

- sorted array or list when there aren't any insertions
  - very efficient for identifying and removing the smallest element
- binary heaps when the max number of elements is known
  - fixed array size can be mitigated with dynamic arrays
- *bounded-height priority queues* when there is a small, discrete range of keys
- BSTs when other dictionary operations are needed, or when there is an unbounded key range and the max PQ size isn't known in advance
- *Fibonacci and pairing heaps* improve the efficiency of decrease key operations
  - effective for large computations if implemented and used well

Designing Your Own Data Structures

- know what kinds of structures lead to what kinds of running times
  - can use that knowledge to guide/constrain thinking
  - O(n log n) or better is typically required in practice for large data sets
- strategies for rolling your own data structures
  - store more information for faster access – as long as it can be kept up-to-date efficiently
  - add additional properties to speed desired operations – as long as they can be maintained efficiently
- knowledge of complexity is useful
  - for NP-complete problems, look for heuristics rather than optimal solutions

Designing Data Structures

"...in practice, it is more important to avoid using a bad data structure than to identify the single best option available."

[Skiena, ADM]

- ask "do we need to do better?" before "can we do better?"

Design a data structure to efficiently support –

- insert(k,e) – insert element with key k
- findMin() – find element with smallest key
- removeMin() – remove element with smallest key
- decreaseKey(e,k) – decrease element e's key to k