Data Structures Toolbox

Key Points

- ADTs vs data structures
- common categories of ADTs
  - common container ADTs – characteristics, properties, operations, applications
- two (three) main kinds of data structures
  - characteristics and tradeoffs
  - array, linked list, and binary tree operations
- basic implementations of containers
  - to understand available library implementations, their running times, and their suitability for particular applications
  - to be able to build your own
- strategies for improving implementations

ADTs vs Data Structures

- an *abstract data type* is defined by its operations (and concept)
  - an algorithm's needs determine which standard ADT is appropriate, if any, and the operations needed for a custom ADT
- *concrete data structures* are used to realize the implementation of an ADT
  - generally have choices, with different time/space tradeoffs
  - changing the data structure used to implement a given ADT does not change the correctness of the algorithm, but may have a big influence on time/space requirements
    - choice of implementation data structure goes hand-in-hand with the design of the algorithm

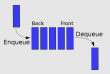
Fundamental ADTs

Some categories of standard ADTs –

- *containers* provide storage and retrieval of elements independent of value
  - ordering of elements depends on the structure of the container rather than the elements themselves
  - elements can be of any type
- *dictionaries* provide access to elements by value
  - lookup according to an element's key
  - elements can be of any type; the key type must support equality comparison
- *priority queues* provide access to elements in order by content
  - ordered by priority associated with elements
  - elements can be of any type; priority must be comparable (so there is an ordering)

ADTs – Common Containers

typical operations

Vector / List / Sequence	linear order, access by rank (index) or position 	rank-based operations • add(x), add(r,x) – add x at the end / with rank r • get(r) – get element with rank r • remove(r) – remove (and return) elt with rank r • replace(r,x) – replace elt at rank r with x position-based operations • first, last() – get first/last position • before(p), after(p) – get position before/after p • addBefore(p,x), addAfter(p,x) – insert x after/before position p • get(p) – get element at position p • remove(p) – remove (and return) elt at pos p • replace(p,x) – replace elt at pos p with x bridge operations • atRank(r) – get pos at rank r • rankOf(p) – get rank of pos p
Stack 	linear order, access only at one end • LIFO – insert and remove at the same end	• push(x) – insert x at the top of the stack • top() – return top item (without removal) • pop() – remove and return the top item on the stack
Queue variations • **Deque** – insert/remove at either end	linear order, access only at both ends • FIFO – insert at one end, remove from the other	• enqueue(x) – insert x at the back of the queue • peek() – return front item (without removal) • dequeue() – remove and return the front item in the queue 

ADTs for Algorithm Design

The kind of access to elements imposed by different types of containers can be exploited to achieve algorithmic goals.

ADT	some applications of the ADT
Vector / List / Sequence	general-purpose container round-robin scheduling, taking turns
Stack	match most recent thing, proper nesting, reversing DFS – go deep before backing up has ties to recursive procedures – supports iterative implementation of recursive ideas
Queue	FIFO order minimizes waiting time BFS – spread out in levels round-robin scheduling, taking turns

ADTs – Map/Dictionary and Set

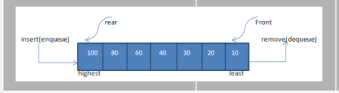
typical operations

- searching and lookup (access by value)

Map / Dictionary variations • **OrderedDictionary** – also supports min/max, predecessor(k)/ successor(k) based on an ordering of the keys	lookup (no duplicate keys)	• find(k) – find elt with key k if it exists • insert(k,v) – add elt v with key k • delete(k) – remove elt, key with key k (may return elt)
Set	membership (no duplicate elements)	• add(x) – add elt x if not already present • remove(x) – remove elt x • contains(x) – return whether x is present

ADTs – PriorityQueue

- sorting/ordering elements in a dynamic environment when the elements are not all known in advance

PriorityQueue 	maintain removal order when there are out-of-order additions	• insert(x,p) – insert elt x with priority p • findMin() or findMax() – find elt with min/max priority • deleteMin() or deleteMax() – remove (and return) elt with min/max key note: a PQ is typically either a min-PQ or a max-PQ – it does not support both min and max operations simultaneously

Choosing Between Collections ADTs

Use a queue when –

- you want things out in the same order you put them in

Use a priority queue when –

- you want to remove things in sorted order but you don't necessarily have all of the things at the beginning

Use a stack when –

- you want things out in the reverse of the order you put them in
- you want to access the most recent thing added

Use a list when –

- stacks and queues don't serve your needs
- need to insert/remove/access at any position

Use a dictionary when –

- you want to associate values with keys and do efficient lookup

Use a set when –

- you want to ask questions (only) about membership