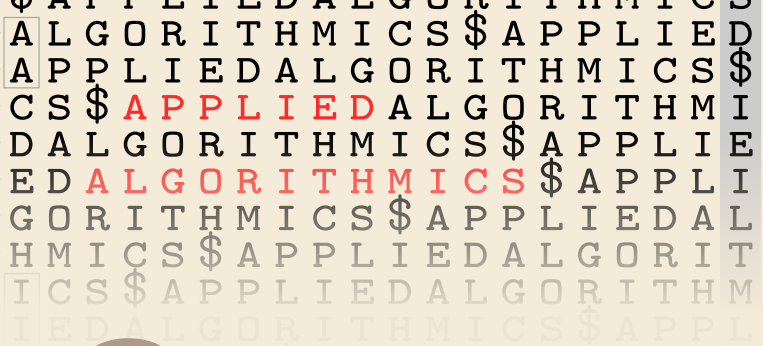


2 Fundamental Data Structures

04 February 2020

Sebastian Wild

version 2021-02-15 12:23 H

Outline

2 Fundamental Data Structures

2.1 Stacks & Queues

Abstract Data Types

abstract data type (ADT)

- list of supported operatic
- **what** should happen
- **not:** how to do it
- **not:** how to store data

≈ Java `interface`
(with Javadoc comments)

Why separate?

- Can swap out implement
- ⇝ reusable code!
- (Often) better abstractions
- Prove generic lower bounds (⇝ Unit 3)

Stacks

Stack ADT

- `top()`
  Return the topmost item on the stack
  Does not modify the stack.
- `push(`x`)`
  Add x onto the top of the stack.
- `pop()`
  Remove the topmost item from the stack
  (and return it).
- `isEmpty()`
  Returns `true` iff stack is empty.
- `create()`
  Create and return an new empty stack.

Linked-list implementation for Stack

Invariants:

- maintain `top` pointer to topmost element
- each element points to the element below it (or `null` if bottommost)

Linked stacks:

- require $\Theta(n)$ space when n elements on stack
- All operations take $O(1)$ time

Array-based implementation for Stack

Can we avoid extra space for pointers?

⇝ array-based implementation

Invariants:

- maintain array `S` of elements, from bottommost to topmost
- maintain index `top` of position of topmost element in `S`.

What to do if stack is full upon `pop`?

Array stacks:

- require *fixed capacity* C (known at creation time)!
- require $\Theta(C)$ space for a capacity of C elements
- all operations take $O(1)$ time

2.2 Resizable Arrays

Digression – Arrays as ADT

Arrays can also be seen as an ADT!

Array operations:

- `create(`n`)` *Java*: `A = new int[`n`];`
  Create a new array with n cells, with positions $0, 1, \ldots, n-1$
- `get(`i`)` *Java*: `A[`i`]`
  Return the content of cell i
- `set(`i`,`x`)` *Java*: `A[`i`] =` x`;`
  Set the content of cell i to x.

⇝ Arrays have fixed size (supplied at creation).

Usually directly implemented by compiler + operating system / virtual machine.

Difference to others ADTs: Implementation usually fixed to "a contiguous chunk of memory".

Doubling trick

Can we have unbounded stacks based on arrays? Yes!

Invariants:

- maintain array `S` of elements, from bottommost to topmost
- maintain index `top` of position of topmost element in `S`
- maintain capacity C = `S.length` so that $\frac{1}{4}C \leq n \leq C$

$\rightsquigarrow$ can always push more elements!

How to maintain the last invariant?

- before `push`
  If $n = C$, allocate new array of size $2n$, copy all elements.
- after `pop`
  If $n < \frac{1}{4}C$, allocate new array of size $2n$, copy all elements.

$\rightsquigarrow$ ***"Resizing Arrays"***

an implementation technique, not an ADT!

Amortized Analysis

- Any individual operation `push` / `pop` can be expensive!
  $\Theta(n)$ time to copy all elements to new array.
- **But:** An one expensive operation of cost T means $\Omega(T)$ next operations are cheap!

distance to boundary

Formally: consider "credits/potential" $\Phi = \min\{n - \frac{1}{4}C, C - n\} \in [0, \frac{1}{2}n]$

- amortized cost of an operation = actual cost $- 2 \cdot$ change in Φ
  - cheap `push`/`pop`: actual cost 1 array access, consumes ≤ 1 credits $\rightsquigarrow$ amortized cost ≤ 3
  - copying `push`: actual cost $2n + 1$ array accesses, creates n credits $\rightsquigarrow$ amortized cost 1
  - copying `pop`: actual cost $n + 1$ array accesses, creates $\frac{1}{2}n$ credits $\rightsquigarrow$ amortized cost 1

$\rightsquigarrow$ **sequence** of m operations: total actual cost $\leq$ total amortized cost + initial credits

here: $\leq 3n + \leq 2 \cdot \frac{1}{2}n = 4n$

Queues

Operations:

- `enqueue(`x`)`
  Add x at the end of the queue.
- `dequeue()`
  Remove item at the front of the queue and return it.

Implementations similar to stacks.

Bags

What do Stack and Queue have in common?

They are special cases of a ***Bag***!

Operations:

- `insert(`x`)`
  Add x to the items in the bag.
- `delAny()`
  Remove any one item from the bag and return it.
  (Not specified which; any choice is fine.)
- roughly similar to Java's `Collection`

Sometimes it is useful to state that order is irrelevant $\rightsquigarrow$ Bag
Implementation of Bag usually just a Stack or a Queue

2.3 Priority Queues

Priority Queue ADT

Now: elements in the bag have different *priorities*.

(Max-oriented) Priority Queue (MaxPQ):

- `insert(`x`,`p`)`
  Insert item x with priority p into PQ.
- `max()`
  Return item with largest priority.
  (Does not modify the PQ.)
- `delMax()`
  Remove the item with largest priority and return it.
- `changeKey(`x`,`p'`)`
  Update x's priority to p'.
  Sometimes restricted to *increasing* priority.
- `isEmpty()`

Fundamental building block in many applications.

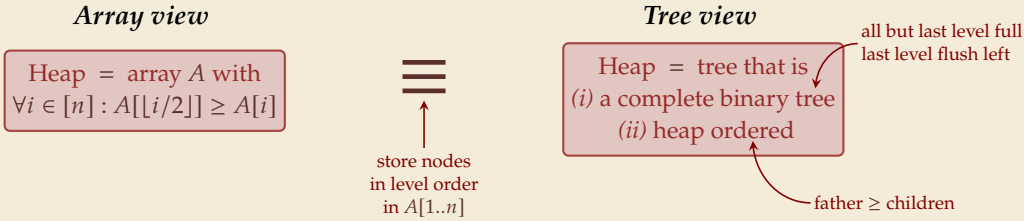
PQ implementations

Elementary implementations

- unordered list $\rightsquigarrow$ $\Theta(1)$ `insert`, but $\Theta(n)$ `delMax`
- sorted list $\rightsquigarrow$ $\Theta(1)$ `delMax`, but $\Theta(n)$ `insert`

Can we get something between these extremes? Like a "slightly sorted" list?

Yes! *Binary heaps.*



Binary heap example

Why heap-shaped trees?

Why complete binary tree shape?

- only one possible tree shape $\rightsquigarrow$ keep it simple!
- complete binary trees have minimal height among all binary trees
- simple formulas for moving from a node to parent or children:
  For a node at index k in A
  - parent at $\lfloor k/2 \rfloor$
  - left child at $2k$
  - right child at $2k + 1$

Why heap ordered?

- Maximum must be at root! $\rightsquigarrow$ `max()` is trivial!
- But: Sorted only along paths of the tree; leaves lots of leeway for fast inserts

how? . . . stay tuned

Insert

Delete Min

Heap construction

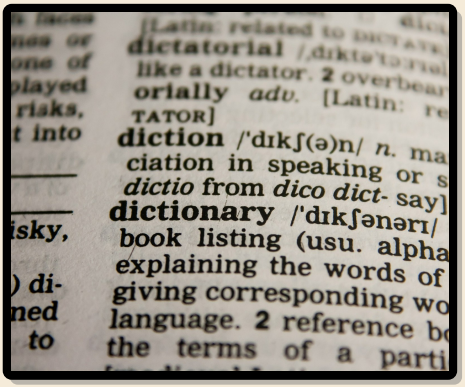
Analysis

2.4 Binary Search Trees

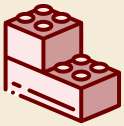
Symbol table ADT

Java: `java.util.Map<K,V>`

Symbol table / Dictionary / Map / Associative array / key-value store:



- `put(`k`,`v`)` *Python* dict: `d[`k`] = `v
  Put key-value pair (k, v) into table
- `get(`k`)` *Python* dict: `d[`k`]`
  Return value associated with key k
- `delete(`k`)`
  Remove key k (any associated value) form table
- `contains(`k`)`
  Returns whether the table has a value for key k
- `isEmpty(), size()`
- `create()`



Most fundamental building block in computer science.

(Every programming library has a symbol table implementation.)

Symbol tables vs mathematical functions

- similar interface
- but: mathematical functions are static (never change their mapping)
  (Different mapping is a *different* function)
- symbol table = *dynamic* mapping
  Function may change over time

Elementary implementations

Unordered (linked) list:

- 👍 Fast `put`
- 👎 $\Theta(n)$ time for `get`
- ⇝ Too slow to be useful

Sorted *linked* list:

- 👎 $\Theta(n)$ time for `put`
- 👎 $\Theta(n)$ time for `get`
- ⇝ Too slow to be useful

- ⇝ *Sorted order does not help us at all?!*

Binary search

Binary search trees

Binary search trees (BSTs) $\approx$ dynamic sorted array

- binary tree
  - Each node has left and right child
  - Either can be empty (`null`)
- Keys satisfy ***search-tree property***

  all keys in left subtree $\leq$ root key $\leq$ all keys in right subtree

BST example & find

BST insert

BST delete

Ordered symbol tables

Augmented BSTs

Analysis

Balanced BSTs

2.5 Summary

BSTs vs. Heaps