Common Lisp and Introduction to Functional Programming
Lecture 9: Functional Data Structures 2/2

Yuri Zhykin

Apr 14, 2021

Functional Data Structures

- **Persistent** data structures are data structures that allow **multiple versions** of the data structure to exist at the same time.
- **Ephemeral** data structures are data structures that allow only a **single version** to exist at a time.
- In functional programming languages **all** data structures are persistent.

Structure Sharing

- Every operation that **updates** a persistent data structures, **creates a new version** of the data structure with the contents that correspond to the updated data.
- **Persistent** data structures are inefficient when fully copied.
- Efficiency can be achieved by utilizing **structure sharing**.
- Common Lisp's list is a persistent data structure.
- More complex structure-sharing data structures can be built on lists.

Stacks 1/2

- **Stack** is an a collection of elements with two operations:
  - **push** - add an element to the collection,
  - **pop** - remove **the most recently added element** from the collection (**LIFO**, **Last In, First Out**).
- Common Lisp has built-in **destructive** operations `push` and `pop` for using lists as stacks:

```
CL-USER> (setq s '(1 2 3))
(1 2 3)
CL-USER> (push 0 s)
(0 1 2 3)
CL-USER> (pop s)
0
CL-USER> s
(1 2 3)
```

Stacks 2/2

- Functional stack can be trivially implemented as follows:

```
;; Simply construct a new list with a new element at its head.
(defun stack-push (e s)
  (cons e s))

;; Return both the top stack value and the new version of stack.
(defun stack-pop (s)
  (values (car s) (cdr s)))
```

- Example:

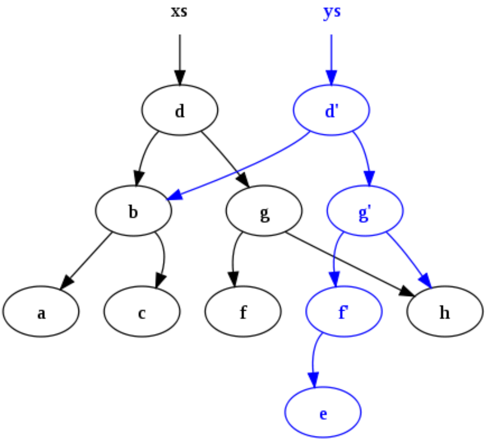
```
CL-USER> (let ((s0 nil))
           (format t "s0: ~a~%" s0)
           (let ((s1 (stack-push 1337 s0)))
             (format t "s1: ~a~%" s1)
             (multiple-value-bind (v s2) (stack-pop s1)
               (format t "s2: ~a~% v: ~a~%" s2 v))))
s0: nil
s1: (1337)
s2: nil
 v: 1337
```

Dictionaries

- **Dictionary** or **finite map** is a collections of key-value associations:
  - **insert** - add a key-value association,
  - **delete** - remove an association between a key and its value,
  - **lookup** - find an associated value for a given key.
- Dictionaries can be represented with lists (e.g. built-in **plist** and **alist** structures).
- Sequential representation is inefficient:
  - **insert** is $O(1)$, but
  - **delete** is $O(n)$ and
  - **lookup** is $O(n)$ as well.

Trees

- **Structure-sharing dictionary** can be implemented with the **tree** data structure.
- Example: operation that inserts new node e, must created new nodes d', g' and f', but can share nodes b and all its children.



Binary Search Trees 1/5

- **Binary search tree** (**BST**) is a **rooted binary tree** whose internal nodes each store a key
  - each node contains a **key** and, *optionally*, an associated **value**,
  - each node has two subtrees (children), denoted **left** and **right**,
  - **binary search property**: for every node its key is greater than all the keys in the node's left subtree and less than those in its right subtree.
- *On average*, binary tree allows for significantly more efficient operations (still $O(n)$ in the worst case):
  - **insert** is $O(\log n)$,
  - **delete** is $O(\log n)$,
  - **lookup** is $O(\log n)$.

Binary Search Trees 2/5

- Binary search trees can be built on top of lists using the following **abstraction**:

```
(defun make-tree ()
  nil)

(defun make-node (key value left right)
  (list key value left right))

(defun node-key (node)
  (first node))

(defun node-value (node)
  (second node))

(defun node-left-child (node)
  (third node))

(defun node-right-child (node)
  (fourth node))
```

Binary Search Trees 3/5

- **Insert** operation works as follows:

```
(defun tree-insert (node key value)
  (cond ((null node)
         (make-node key value nil nil))
        ((string= key (node-key node))
         (make-node key
                    value
                    (node-left-child node)
                    (node-right-child node)))
        ((string< key (node-key node))
         (make-node (node-key node)
                    (node-value node)
                    (tree-insert (node-left-child node) key value)
                    (node-right-child node)))
        (t ;; (string> key (node-key node))
         (make-node (node-key node)
                    (node-value node)
                    (node-left-child node)
                    (tree-insert (node-right-child node) key value)))))
```

- **Lookup** operation is straightforward:

```
(defun tree-lookup (node key)
  (cond ((null node)
         (values nil nil))
        ((string= key (node-key node))
         (values (node-value node) t))
        ((string< key (node-key node))
         (tree-lookup (node-left-child node) key))
        (t ;; (string> key (node-key node))
         (tree-lookup (node-right-child node) key))))
```

Binary Search Trees 5/5

- **Delete** operation is slightly more complicated because the **binary search property** must be preserved, so the tree must be rebalanced.
- If the values being inserted are **ordered**, the tree becomes **degenerate** and will provide worst case $O(n)$ performance.
- Self-balancing structures like **red-black trees** can be used to mitigate that.

Queues 1/5

- **Queue** is a collection of entities that can be modified by the addition of entities at one end of the sequence and the removal of entities from the other end of the sequence.
- By convention, the end of the sequence where elements are added is called the **back** or **tail** of the queue.
- The end where elements are removed is called the **front** or **head** of the queue.
- Queue data structure has two main operations:
  - **enqueue** - add an element to the queue,
  - **dequeue** - take an element from the queue in the **FIFO** (**First In, First Out**) order.

Queues 2/5

- Using list as a queue is inefficient, because only one operation can be implemented efficiently:
  - **enqueue** is $O(1)$, then **dequeue** $O(n)$ or
  - **dequeue** is $O(1)$, then **enqueue** $O(n)$.
- Efficient queue can be constructed by splitting the queue list into two parts:
  - first part, **front**, is sorted **from the least recent to the most recent element**,
  - second part, **back**, is sorted **from the most recent to the least recent element**,
  - **enqueue** operation adds a new element to the **head** of the **front** list,
  - **dequeue** operation takes the **head** of the **back** list as the result.

- The following queue

$$Q = (1, 2, 3, 4, 5)$$

is represented internally like this

```
       +---+   +---+   +---+   +---+
front: | 1 |-->| 2 |-->| 3 |-->|NIL|
       +---+   +---+   +---+   +---+
       +---+   +---+   +---+
back:  | 5 |-->| 4 |-->|NIL|
       +---+   +---+   +---+
```

- Adding an element 6 to the queue results in

```
       +---+   +---+   +---+   +---+
front: | 1 |-->| 2 |-->| 3 |-->|NIL|
       +---+   +---+   +---+   +---+
       +---+   +---+   +---+   +---+
back:  | 6 |-->| 5 |-->| 4 |-->|NIL|
       +---+   +---+   +---+   +---+
```

- Taking an element from the queue results in

```
       +---+   +---+   +---+
front: | 2 |-->| 3 |-->|NIL|
       +---+   +---+   +---+
       +---+   +---+   +---+   +---+
back:  | 6 |-->| 5 |-->| 4 |-->|NIL|
       +---+   +---+   +---+   +---+
```

Queues 4/5

- If the **front** of the queue is empty, but the **back** is not, we need to move the element from the **back** to the **front**:
  - before

```
       +---+
front: |NIL|
       +---+
       +---+   +---+   +---+   +---+
back:  | 6 |-->| 5 |-->| 4 |-->|NIL|
       +---+   +---+   +---+   +---+
```

  - after

```
       +---+   +---+   +---+   +---+
front: | 4 |-->| 5 |-->| 6 |-->|NIL|
       +---+   +---+   +---+   +---+
       +---+
back:  |NIL|
       +---+
```

Queues 5/5

- The queue implementation above satisfies the requirement
  - **enqueue** runs in $O(1)$,
  - **dequeue** runs in $O(1)$, $O(n)$ in the worst case.
- The operation of moving the elements from the **back** to the **front** causes the worst case scenario for **dequeue** operation, but it only happens when the **front** is empty.
- The $O(n)$ cost of moving the element from the **back** to the **front** is **amortized** across multiple $O(1)$ operations.

Amortized Analysis 1/2

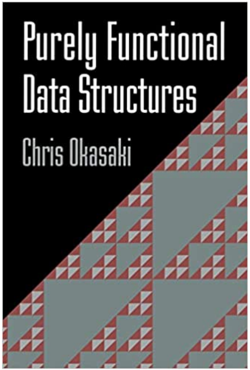
- **Amortized analysis** is an approach to complexity analysis that considers the possible sequences of operations instead of isolated operations.
- First formally introduced by Robert Tarjan in his 1985 paper *Amortized Computational Complexity*.
- The basic idea is that a **worst-case operation** can alter the state in such a way that the **worst case cannot occur again for a long time**, thus "amortizing" its cost.

Amortized Analysis 2/2

- Three main ways to perform amortized analysis:
  - the **aggregate method** (determine the upper bound $T(n)$ on the total cost of a sequence of n operations, then calculate the amortized cost to be $T(n)/n$),
  - the **accounting method**, (operations have a cost higher than the actual cost, accumulating a saved “credit” that is used to “pay” for the later operations),
  - the **potential method** (operation has a cost and a change in potential, which is some function of the state of the data structure).

Useful Resources

- **Purely Functional Data Structures** by Chris Okasaki



The End

Thank you!