Logistics

Reading
Wikipedia: Persistent Data Structures

Goals: Persistent
- Stacks
- Queues
- Search Trees
- Hash Tables

Lab09
- Functors - useful for A4
- Memoization - useful generally

Assignment 4
- Implement a persistent tree
- Used for Sets and Key-Val Maps
- Review Binary Search Trees
- Just a normal BST
- Due Sun 11/11
Exercise: Properties of Sets and Maps

1. As seen in our module discussion, do sets/maps produced by Set.Make and Map.Make ever change?
2. What vocab word is associated with such things?
3. Why not just use association lists (key-value pair lists) rather than these more elaborate mechanisms?
4. What do you expect in terms of performance and implementation details for OCaml’s sets and maps?
**Answers: Properties of Sets and Maps**

1. Do sets and maps produced by `Set.Make` and `Map.Make` ever change?

2. What vocab word is associated with such things? The sets/maps don’t change: they are **persistent** or **immutable**. Functions like add create new versions with elements added. These new versions must be let-bound.

3. Why not just use association lists (key-value pair lists) rather than these more elaborate mechanisms?

   Association lists have linear performance, \(O(N)\) where \(N\) is the number of bindings. As \(N\) increases, worst-case search time increases proportionally.

4. What do you expect in terms of performance and implementation details for OCaml’s sets and maps?

   OCaml’s sets and maps are based on balanced search trees. This gives them \(O(\log N)\) logarithmic performance. The code implementing `Set.Make` is linked below.

Why Persistent Data?

**immutable:** *(adjective)* unchanging over time or unable to be changed.

**persistent:** *(adjective)* Def 2: continuing to exist or endure over a prolonged period.

Often easier to reason about than mutating/stateful data;

- Get undo/redo for free - generally can track history of edits
- Easier to apply formal proofs to immutable data
- Requires experience to understand why mutation creates difficulty - must incorporate **moment in time** into analysis
- **Concurrent** code in which multiple execution paths can execute in any order is particularly difficult to reason about
- Locks/Mutexes (CSCI 4061) are often used but persistent data often don’t need these
Basic strategies with Persistent/Immutable Structures

The following ideas are typically used when implementing a persistent/purely functional data structures

1. Create new pieces rather than changing existing ones
2. Retain pointers to old parts as they won't change: new versions share as much as they can with old versions.
3. Avoid large, contiguous chunks of overly data as they will need to be copied.
4. Avoid copying the entire structure whenever possible.
5. Strive for equivalent complexity to mutable counterparts acknowledging this is not always possible.

These tend to push persistent DSs in the direction of linked structures like lists and trees with small nodes strung together.

---

"Persistent", "Immutable", and "Purely Functional" data structures are usually all the same thing.
Exercise: Start Simple: The Stack

Stacks (Last In, First Out [LIFOs]) have 3 central operations

- **push stack x** return a copy of stack with x at the top
- **pop stack** return a copy of stack with the top element removed
- **top stack** return the element at the top of the stack

`pop` and `top` error out (exceptions) if the stack is empty.

Implement a persistent stack by defining these three operations

- Use any built-in types in OCaml that are relevant
- OR define your own types as needed
## Answers: Start Simple: The Stack

### Code

```ocaml
(* stack.ml: basic persistent stack based on built-in linked lists *)

let empty = [];;

let push stack x = x :: stack;;

let pop stack =
  match stack with
  | [] -> failwith "empty stack"
  | top::rest -> rest;

let top stack =
  match stack with
  | [] -> failwith "empty stack"
  | top::rest -> top;
```

### Use

```ocaml
# #mod_use "stack.ml";;
module Stack :
  sig
    val empty : 'a list
    val push : 'a list -> 'a -> 'a list
    val pop : 'a list -> 'a list
    val top : 'a list -> 'a
  end

# let stack0 = Stack.empty;;
val stack0 : 'a list []

# let stack1 = Stack.push stack0 "a";;
val stack1 : string list = ["a"]

# let stack2 = Stack.push stack1 "b";;
val stack2 : string list = ["b"; "a"]

# let stack3 = Stack.push stack2 "c";;
val stack3 : string list = ["c";"b";"a"]

# let stack4 = Stack.pop stack3;;
val stack4 : string list = ["b"; "a"]
```

---

* Stack usage example:
Important Points on Stacks

- Based implementation on linked nodes.
- Mutable stacks have $O(1)$ push/pop/top operations, run in constant time and space.
- Immutable version provided runs in constant time/space: not losing (much) speed.

What’s that other linear data structure like stacks, the FIFO one?
Exercise: Queue Implementation

Queues are First In, First Out (FIFO)

- What operations does a queue support?
- What are the big-O runtimes for these and what are their space complexities?
- How do they change the queue’s data?
- Recall a typical implementation for queues built on linked lists
- Draw pictures of how these operations work
- Identify barriers to creating a persistent version
**Answers: Queue Implementation**

**Operations**

- **enqueue(x)** add x at the back
- **dequeue()** remove front element
- **front()** return front element

All these are $O(1)$ time/space operations.
Ops change pointers at both ends of the queue creating a major barrier to a direct persistent version.
Queue Possibilities

- dequeue q is fine, point at a different node
- enqueue q x changes the last null pointer to a new node
- Could copy ALL queue nodes on enqueue
- Drawback: $O(1)$ operation becomes $O(N)$

Alternative: Two Stacks Strategy

- Keep two lists of incoming and outgoing elements
- Both are stacks which are persistent
- enqueue adds to incoming
- dequeue removes from outgoing
- Reverse and transfer incoming to outgoing when needed
Queues via Two-Stack in Pictures

- enqueue q x "pushes" into incoming stack
- dequeue q "pops" off outgoing stack
- Empty outgoing during dequeue means reverse incoming

1. Three enqueues: A is added first, "last" in incoming

2. Start of dequeue(): reverse incoming to outgoing

3. Finish dequeue(): A removed

4. enqueue(D)

5. enqueue(E)

6. dequeue() twice

7. Start of dequeue(): reverse incoming to outgoing

8. Finish dequeue(): D removed
Demo Use

```ocaml
# let q0 = Queue.empty;;
val q0 : 'a Queue.queue = {Queue.incoming = []; outgoing = []}

# let q1 = Queue.enqueue q0 "A";;
val q1 : string Queue.queue = {Queue.incoming = ["A"]; outgoing = []}

# let q2 = Queue.enqueue q1 "B";;
val q2 : string Queue.queue = {Queue.incoming = ["B"; "A"]; outgoing = []}

# let q3 = Queue.enqueue q2 "C";;
val q3 : string Queue.queue = {Queue.incoming = ["C"; "B"; "A"]; outgoing = []}

# let q4 = Queue.dequeue q3;;
val q4 : string Queue.queue = {Queue.incoming = []; outgoing = ["B"; "C"]}

# let (f, q5) = Queue.front q4;;
val f : string = "B"
val q5 : string Queue.queue = {Queue.incoming = []; outgoing = ["B"; "C"]}
```
Exercise: Queues via Two-Stack in Code

Complete the dequeue function below according to the template given

type 'a queue = { (* immutable fields *)
  incoming : 'a list; (* incoming for enqueue *)
  outgoing : 'a list; (* outgoing for dequeue *)
};;

(* An empty queue record *)
let empty = {incoming=[]; outgoing=[]};;

(* Return a queue with x pushed into the incoming stack *)
let enqueue q x =
  {q with incoming=(x::q.incoming)}
;;

let dequeue q = ... ;;
(* Return a queue with one element removed; pops an element from outgoing stack if not empty; otherwise reverses incoming list, pops an element and then assigns remainder to outgoing leaving incoming empty. raises and exception on both incoming/outgoing empty. *)

IMPLEMENT ME
Answers: Queues via Two-Stack in Code

(* Return a queue with one element removed; pops an element from outgoing stack if not empty; otherwise reverses incoming list, pops an element and then assigns remainder to outgoing leaving incoming empty. raises and exception on both incoming/outgoing empty. *)

let dequeue q =
  match q with
  | {outgoing=[]; incoming=[]} -> failwith "empty queue"
  | {outgoing=front::rest} -> {q with outgoing=rest}
  | _ -> (* outgoing empty, incoming present *)
    let revin = List.rev q.incoming in
    {incoming=[]; outgoing=(List.tl revin)}

(* Returns a pair of the front element of a queue and a potentially re-arranged queue. Re-arrangement is done like in dequeue if the outgoing stack is empty. Exception on both incoming/outgoing empty. *)

let front q =
  match q with
  | {outgoing=[]; incoming=[]} -> failwith "empty queue"
  | {outgoing=front::rest} -> (front,q)
  | _ -> (* outgoing empty, incoming present *)
    let revin = List.rev q.incoming in
    let front = List.hd revin in
    let newq = {incoming=[]; outgoing=revin} in
    (front,newq)

;;
Exercise: Complexity of Two-Stack Queues

Standard Big-O runtime analysis looks at worst-case

► What is the worst case runtime of dequeue for standard mutating queues?

► What is the worst-case runtime of dequeue for two-stack persistent queues?

► How often does the worst-case runtime happen for two-stack queues?

```ocaml
let dequeue q =
  match q with
  | {outgoing=[]; incoming=[]} -> failwith "empty queue"
  | {outgoing=front::rest} -> {q with outgoing=rest}
  | _ -> (* outgoing empty, incoming present *)
    let revin = List.rev q.incoming in
    {incoming=[]; outgoing=(List.tl revin)}
;;
```
Answers: Complexity of Two-Stack Queues

- Standard mutating queues: dequeue worst-case runtime is $O(1)$, a couple pointer re-arrangements
- Two-stack persistent queues: dequeue Worst case runtime $O(N)$, reversal of incoming field, linear on the size of the queue
- **BUT** worst-case happens once and is followed by subsequent $O(1)$ dequeue complexities
- To understand the full complexity picture, worst-case analysis is insufficient
Amortized Analysis

*amortize:* (verb) *to gradually reduce or write off the cost*

- Amortized Analysis of data structures and algorithms is a more advanced technique to study runtime and space complexity.
- Standard analysis considers the cost of a single operations on a DS in isolation.
- Amortized Analysis considers the total cost of many operations on a DS.
- Amortized analysis accounts for:
  - A few expensive operations . . .
  - Offset by Many cheap operations.
## Simple Amortized Analysis of Two-Stack Queues

<table>
<thead>
<tr>
<th>In</th>
<th>Out</th>
<th>Operation</th>
<th>PerOp Work</th>
<th>Num of Ops</th>
<th>Total Op Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>empty</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>enqueue</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>enqueue</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>50</td>
<td>0</td>
<td>enqueue</td>
<td>1</td>
<td>48</td>
<td>48</td>
</tr>
<tr>
<td>0</td>
<td>49</td>
<td>dequeue</td>
<td>N</td>
<td>1</td>
<td>51</td>
</tr>
<tr>
<td>0</td>
<td>48</td>
<td>dequeue</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>47</td>
<td>dequeue</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>25</td>
<td>dequeue</td>
<td>1</td>
<td>22</td>
<td>22</td>
</tr>
<tr>
<td>10</td>
<td>25</td>
<td>enqueue</td>
<td>1</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>24</td>
<td>dequeue</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>dequeue</td>
<td>1</td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>dequeue</td>
<td>N</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>dequeue</td>
<td>1</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

| TOTALS | 121 | 179 |

- M = 121 operations
- About 1.5 M = 179 units of work
- Amortized cost per op is 1.5 units of work
Two-Stacks Queues Analysis

- Worst-case dequeue / front is $O(N)$
- Total cost of a sequence of $M$ enqueue / dequeue / front ops $O(M)$ giving Amortized $O(1)$ per operation
- While somewhat more expensive, immutable queue is in the same ballpark as mutable queue

Notes on Amortized Analysis

- Have shown only an informal version of Amortized Analysis
- Proper formal version would need to cover any sequence of operations and prove they all lead to linear scaling
- Covered in much more detail in an Advanced Algs/DSs class such as CSCI 5421
- Amortized Analysis is NOT unique to persistent data structures: widely applicable to many other weird and beautiful structures,
- Notably extensible arrays use Amortized Analysis to prove $O(1)$ runtime for add() (OCaml’s Buffer, Java’s ArrayList)
Persistent Search Trees

- Persistent trees utilize **path copying**
- Every add/remove creates a new root and path to the changed node
- Required as some part of the path changes
- Branches not on the change path **can be shared**
- Leads to $O(\text{height}(T))$ complexity, same as mutative version
- $O(\log N)$ for balanced trees
- Naive implementations create new paths even if the tree doesn’t change (A4)

```ocaml
ext let ys =
    Treeset.add xs "e" in
...```
Persistent Arrays and Hash Tables

- Typically use path copying on trees with "fat" nodes: more than two children
- Balanced tree with 4 children per node gives $O(\log_4 N)$ performance for operations: shorter trees with wider nodes
- **Not** $O(1)$ performance but for 32 children, $O(\log_{32} N)$ is reasonable: $\log_{32} 500\,000 \approx 3.8$, $\log_{32} 1\,000\,000 \approx 3.9$
- Many details involved but basis is same as persistent trees

Source: Understanding Clojure’s Persistent Vectors by Jean Niklas L’orange