CS136 - Midterm Review Questions

David R. Cheriton School of Computer Science
University of Waterloo

February 28, 2013

1 Linked Lists

Recall the interface given for linked lists.

```
struct node {
  int first;
  struct node *rest;
};

struct node *cons(int item, struct node *lst);
// PRE: lst is either NULL or a valid pointer to a list (node)
// POST: returns a new node with first as item, and rest as lst

void print_list(struct node *lst);
// PRE: lst is either NULL or a valid pointer to a list (node)
// POST: prints out the list

void destroy_list(struct node *lst);
// PRE: lst is either NULL or a valid pointer to a list (node)
// POST: deallocates (frees) every node in the list
```

- Implement the function **bool** has_cycle(**struct** node *lst) that consumes a list and returns **true** if and only if the list has a cycle. You may not use recursion and you may not mutate the existing list. Furthermore, your function must not allocate any new nodes on the heap. The following snippet illustrates a few examples.

```
struct node *tail = cons(1, NULL);
struct node *lst = cons(4, cons(3, cons(2, tail)));
tail->rest = lst->rest; // create a cycle in our list

bool result = has_cycle(lst); // will return true
tail->rest = NULL; // fix the cycle
result = has_cycle(lst); // will return false

destroy_list(lst);
```

- Implement the function **void** two_sum(**struct** node *lst, **int** k) that prints all pairs of integers x and y in the list, such that $x+y=k$. Each pair should be printed on a seperate line in the form (x, y). If no such pairs exist, no printing is required. For example, the following snippet would produce the output $(1, 4)$ and $(2, 3)$ on seperate lines.

```c
int main() {
    struct node *lst = cons(1, cons(2, cons(3, cons(4, NULL))));
    two_sum(lst, 5); // produces the output specified above
    destroy_list(lst);
    return 0;
}
```

- Implement the function **struct** node *append(**struct** node *lst1, **struct** node *lst2) that appends lst1 and lst2 into a single list. Your function should not allocate any new nodes on the heap, and should run in $O(m)$ time, where m is the length of lst1.

  Similarly, implement the function **struct** node *split(**struct** node *lst, **int** k) which consumes a list of length n and an integer k. Your function will then mutate the list such that lst becomes a list of length k, and then returns a pointer to the remaining portion of the list. Note that the list returned will be of length $n-k$. For convinience, you may assume that the list is of length 2 or greater. You may also assume that $1 \leq k < n$. Your function should run in $O(k)$ time.

  Hint: You may find it helpful to test these two functions together.

2 Big-O Notation

Recall the definition for Big-O notation. Let $f(n)$ and $g(n)$ be positive functions. Then we say that $f(n) \in O(g(n))$ if there exists constants $c \geq 1$ and $n_0 \geq 1$ such that $f(n) \leq c * g(n)$, $\forall n \geq n_0$.

For example, we would say that $2n + 1 \in O(n)$ since $2n + 1 \leq 2n + n \leq 3n$, $\forall n \geq 1$. Similarly, we can show that $7n^2 - nlog(n) - 1 \notin O(n)$, since no pairs of constants (c, n_0) can be chosen to satisfy the required inequality.

- Prove or disprove the following statements using the formal definition of Big-Oh. In other words, if the statement is true, then provide adequate values for c and n_0 that satisfy the required inequality. If the statement is false, then prove that for all values of c and n_0, $\exists n \geq n_0$ such that $f(n) > c * g(n)$.
  - $631n + 136 \in O(n)$?
  - $3^n \in O(2^n)$?
  - $2n\log(n) + 5n - 3 \in O(n^2)$?
- Analyze the following pieces of code using the method of your choice.

– Example in C

```c
void foo(int n) {
    for(int i = 0; i < n; i = i + 1) {
        int j = i;
        while(j > 1) { j = j / 2; }
    }
}
```

– Another Example in C

```c
void foo(int n){
   int i = 0;
   while(i <= n){
      for(int j = 5 * n; j > 0; j = j - 5){
         printf("%d\n", i + j);
      }
      printf("\n");
      i = i + 5;
   }
}
```

– An Example in Racket (Note that this question is challenging)

```racket
(define (power-set set)
   (cond
      [(empty? set) (list empty)]
      [else
       (define power-set-of-rest (power-set (rest set)))
       (append power-set-of-rest
                  (map (lambda (subset) (cons (first set) subset))
                           power-set-of-rest))]))
```

3 Stack Frames and Recursion in C

Recall the implementation of Fibonacci numbers from the lecture slides.

```c
int fib(int n) {
    if (n == 0) return 0;
    else if (n == 1) return 1;
    else
        return fib(n -1) + fib(n -2);
}
```

- Draw the sequence of changes in the stack that would result from a call to fib(3) as follows:

```c
int main() {
    int i = 3;
    int x = fib(i);
    return 0;
}
```

- How many bytes does one fib stack frame use? Justify your answer.
- Re-write this recursive function into an equivilant iterative function. Your function should run in O(n) time.

4 Abstraction and Interaction in Racket

Consider the following interface for the stack ADT.

```
;; new-stack!: -> Stack!
;; PRE: true
;; POST: an empty stack

;; stack!-empty?: Any -> Bool
;; PRE: true
;; POST: produces #t if stack is empty, #f otherwise

;; push!: Stack! Any -> Void
;; PRE: true
;; POST: updates stk with item on the top of the stk

;; pop!: Stack! -> Void
;; PRE: stk is non-empty
;; POST: updates the stk with the top item removed

;; top: Stack! -> Any
;; PRE: stk is non-empty
;; POST: returns the value at the top of the stk

;; stack-print: Stack! -> Void
;; PRE: true
;; POST: prints the stk from top down
```

- Implement the following functions for a mutable queue using the stack interface defined above (Hint: Use two stacks). Briefly describe how you could prevent a client program from manipulating your queue ADT in ways non-intended by the interface.

```
;; new-queue!: -> Queue!
;; PRE: true
;; POST: produces a new (empty) Queue!

;; queue!-empty?: Queue! -> Bool
;; PRE: True
;; POST: Produces True if sequence is empty, False otherwise
;; One parameter, a queue Q = (q1,q2, ...,qn)

;; enqueue! : Queue! Any -> Void
;; PRE: True
;  POST: Modifies Q so that now Q = (e,q1,q2, ...,qn)
;; Two parameters, an item e and a queue Q = (q1,q2, ...,qn)

;; head : Queue! -> Any
```

```
;; PRE: n >= 1
;; POST: Produces value qn.
;; One parameter, a queue Q = (q1,q2, ...,qn)

;; dequeue! : Queue! -> Void
;; PRE: n > 1
;; POST: Modifies Q so that now Q = (q1, ...,qn-1)
;; One parameter, a queue Q = (q1,q2, ...,qn)
```

- Using the queue ADT defined above, implement a Racket program that behaves according to the following interface:

```
;; queue-ui creates a new queue, runs until EOF and accepts the following
;; commands:
;;     e itm - enqueues an item
;;     d - dequeues an item
;;     h - produces the head the of the queue
;;     e? - checks if the current queue is empty
;;     q - quits the program
```