# Midterm II Exam

15-122 Principles of Imperative Computation
Frank Pfenning

November 13, 2012

Name: **Sample Solution** Andrew ID: **fp** Section:

## Instructions

- This exam is closed-book with one sheet of notes permitted.
- You have 80 minutes to complete the exam.
- There are 6 problems on 12 pages.
- Read each problem carefully before attempting to solve it.
- Do not spend too much time on any one problem.
- Consider if you might want to skip a problem on a first pass and return to it later.

| | Search in Heaps | PQAD's | Implementing PQAD's | Amortized Analysis | BST's | Safety | |
|---|---|---|---|---|---|---|---|
| | Prob 1 | Prob 2 | Prob 3 | Prob 4 | Prob 5 | Prob 6 | Total |
| Score | **10** | **20** | **10** | **20** | **15** | **25** | **100** |
| Max | 10 | 20 | 10 | 20 | 15 | 25 | 100 |
| Grader | | | | | | | |

# 1 Search in Heaps (10 pts)

In this problem and the next we will discuss *min heaps*, as they are used to implement priority queues.

A heap satisfies an ordering invariant that we can exploit to some extent when searching for an element of a specified priority. Assume the function `priority(H,i)` will return the priority of the element stored in the heap array at index $i$, under the precondition that `1 <= i && i < H->next`.

**Task 1** (8 pts). Complete the following program, where `pq_in(H,p)` is supposed to return `true` if there is an element with priority $p$ in $H$, and `false` if there is no such element. If you cannot think of a solution that fits into the function's structure mapped out below, you may revise it for partial credit by clearly writing it to the right of the given code.

```
bool find(heap H, int i, int p)
//@requires is_heap(H);
//@requires 1 <= i;
{ if (!(i < H->next)) {
    /* not in tree */
    return false;
  } else if (priority(H,i) == p) {
    /* found */
    return true;
  } else if (priority(H,i) > p) {
    /* exploit heap ordering invariant */
    return false;
  } else {
    /* make recursive call(s) */
    return find(H, 2*i, p)     /* left child */
        || find(H, 2*i+1, p);  /* right child */
  }
}
bool pq_in(heap H, int p)
//@requires is_heap(H);
{
  return find(H, 1, p);
}
```

**Task 2** (2 pts). What is the worst-case asymptotic complexity of `pq_in(H, p)` in terms of the number $n$ of elements in $H$? Use big-O notation.

$O(n)$

## 2 Priority Queues with Arbitrary Deletion (PQAD's) (20 pts)

A *Priority Queue with Arbitrary Deletion* (PQAD) is like a priority queue, with a restriction and an additional operation:

1. Every element we ever add to the PQAD has a unique priority. This can be ensured by incorporating timestamps into priorities. We are not concerned with how this is accomplished.

2. We have a new operation

   ```
   void delete(pqad R, int p);
   ```

   which deletes the unique element from the PQAD $R$ that has priority $p$. This operation has as a precondition that $R$ actually contains an element with priority $p$.

As we have seen in the previous problem, it is expensive to find an element in a heap, not even considering what it takes to delete it from the data structure while maintaining both shape and ordering invariants.

The idea is to implement a PQAD with *two priority queues*, one `Ins` holding elements as they are inserted to the PQAD, another `Del` with elements that have been deleted from the PQAD. At any time the elements that are in `Ins` but not in `Del` are the *actual elements* in the PQAD.

Let `min(P)` stand for the priority of the minimal element in a priority queue. We maintain the following invariants:

(1) `Ins` and `Del` are always valid priority queues.

(2) The elements in `Ins` are a superset of those in `Del`.

(3) `min(Ins)` $<$ `min(Del)` if `Ins` and `Del` are both non-empty.
If either or both are empty, (3) is deemed to be satisfied.

The interesting invariant here is (3). Invariants (1) and (2) together imply

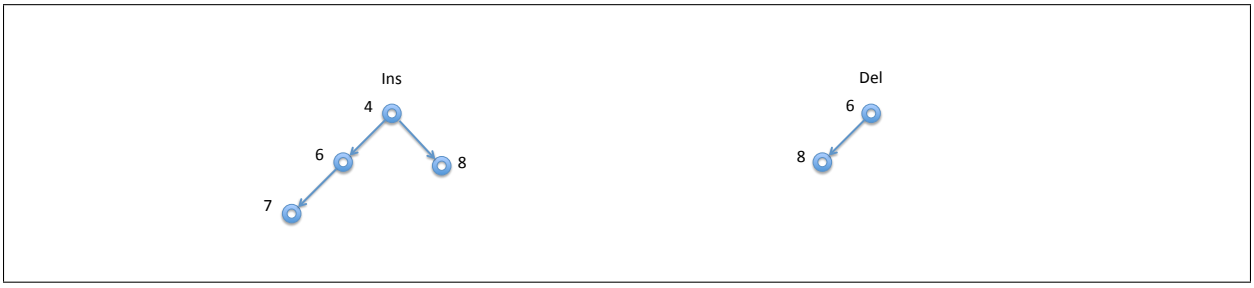
`min(Ins)` $\leq$ `min(Del)` if `Ins` and `Del` are both non-empty.

If an operation would ever make `min(Ins)` and `min(Del)` equal, we can restore the stronger invariant by repeatedly deleting minimal and equal elements from the two priority queues until (3) holds again.

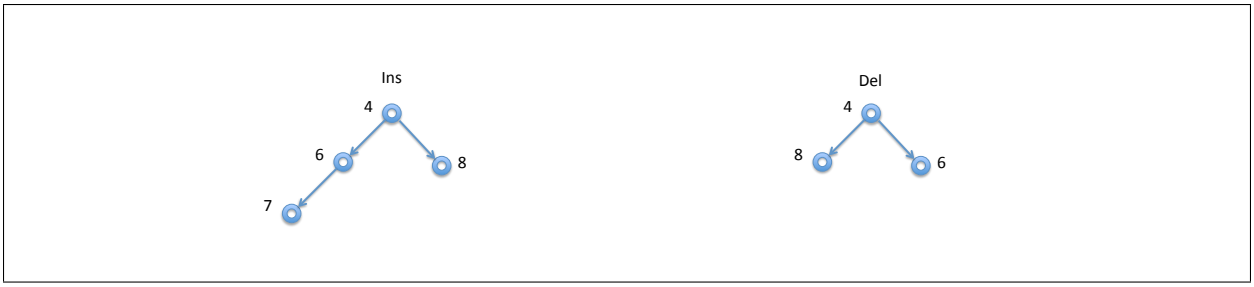
We now walk through an example of the operation of a PQAD. For this purpose of these tasks, identify elements with their priority. Draw all heaps in the form of trees, not arrays.

**Task 1** (5 pts). On the left below, draw the state of the heap `Ins` after inserting elements 6, 7, 8, 4 into an empty PQAD, in this order. The heap `Del` will still be empty, and the *actual elements* in the PQAD will be 4, 6, 7, 8. [Hint: check your work by making sure that at least the minimal element is at the top.]

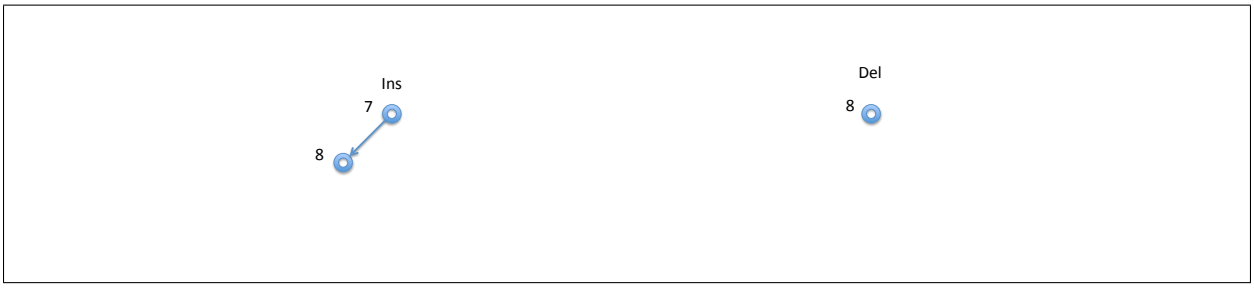
**Task 2** (5 pts). On the right below, draw the state of the heap `Del` after deleting elements 6 and 8 from the PQAD. This just means that they are inserted into the heap `Del`. At this point the *actual elements* are 4 and 7.




**Task 3** (5 pts). Next we delete 4 from the PQAD. We do this in two stages. In the first stage we insert 4 into the `Del` heap. Show the state of both heaps below. Be aware that this state will violate invariant (3).



**Task 4** (5 pts). Now successively delete matching minimal elements from the priority queues `Ins` and `Del` until invariant (3) is restored. Show the state of both heaps `Ins` and `Del` once the invariant has been restored.



## 3 Implementing PQAD's (10 pts)

We now implement PQAD's in C0. We use the following representation:

```
struct pqad_header {
  pq Ins;
  pq Del;
};
typedef struct pqad_header* pqad;
```

Here is a reminder of the relevant functions from the interface to priority queues, with the new function `pq_in` from Problem 1.

```
int elem_priority(elem e);      /* provided by client */

bool pq_empty(pq P);            /* is P empty? */
void pq_insert(pq P, elem e);   /* insert e into P */
elem pq_min(pq P);              /* return minimum, P not empty */
elem pq_delmin(pq P);           /* delete minimum, P not empty */
```

You may also use the helper function `min` on a priority queue:

```
int min(pq P) {
  return elem_priority(pq_min(P));
}
```

The following functions are for use in contracts:

```
bool is_pqad(R);                /* check invariants (1), (2), and (3) */
bool almost_pqad(R);            /* check invariants (1) and (2) only */
bool pq_in(pq P, int p);        /* does P have an element with priority p */
```

**Task 1** (10 pts). Implement a function `restore` by filling in the missing code. Your loop invariant(s) should be strong enough that, together with the negated loop guard, they imply the postcondition.

```
void restore(pqad R)
//@requires almost_pqad(R);
//@ensures is_pqad(R);
{
  while (!pq_empty(R->Del)
         && min(R->Ins) == min(R->Del))
    //@loop_invariant almost_pqad(R);
    {
       pq_delmin(R->Ins);
       pq_delmin(R->Del);
    }
  return;
}
```

## 4 Amortized Analysis (20 pts)

Let's walk through the amortized analysis of the PQAD data structure. We count complexity exactly in terms of the number of *swap* operations that may have to be performed during the sift-up or sift-down phase of `pq_insert` or `pq_delete`. We assume that the underlying implementation of priority queues is as min heaps, as done in lecture.

**Task 1** (3 pts). Insertion of an element into a PQAD always takes place by insertion into the priority queue `Ins`. In the worst case, how many $swap$ operations do we have to perform while inserting? Express your answer in terms of $k$, the number of elements in `Ins` before the insertion. You may use the notation $\mathsf{ilog}(x) = \lfloor \log_2(x) \rfloor$. We are looking for an exact bound, not a big-O approximation.

| |
|---|
| $\mathsf{ilog}(k+1)$ |

**Task 2** (3 pts). Deleting an arbitrary element from a PQAD taks place in two stages. In the first stage we insert it into `Del`. In the worst case, how many $swap$ oerations do we have to perform for this stage? Express your answer in terms of $d$, the number of elements in `Del` before we insert into it. Again, we are looking for an exact solution, not a big-O approximation.

| |
|---|
| $\mathsf{ilog}(d+1)$ |

**Task 3** (4 pts). However, this may violate invariant (3) in case the element we delete is the minimal one in `Ins`. So we have to call `restore` in order to restore the invariant. In the worst case, how many total *delmin* operations do we have to perform on `Ins` and `Del`? Express your exact answer in terms of `k` and `d`.

| |
|---|
| $k + d$, where $k = d$ is the worst case. |

**Task 4** (4 pts). Give a good worst-case bound on the number of total number of *swap* operations that `restore` has to perform in terms of $k$ and $d$, the number of elements in `Ins` and `Del`, respectively? While the answer does not have to be exact, it should be precise enough to justify your answer in Task 5. Do not use a big-O approximation.

$$\mathsf{ilog}(d) + \cdots + \mathsf{ilog}(1) + \mathsf{ilog}(k) + \cdots + \mathsf{ilog}(1)$$

**Task 5** (6 pts). Finally, we consider a sequence of $n$ *insert* and *delete* operations, starting from an empty PQAD. How many tokens do we need to set aside during each operation in order to demonstrate that deletion has worst-case amortized cost of $O(\mathsf{log}(n))$ *swap* operations? Express the number of tokens in terms of $k$ (the number of elements in `Ins`) and $d$ (the number of elements in `Del`) during each insertion:

- For `pqad_insert(R,e)`, if there are $k$ elements in `Ins` and $d$ elements in `Del`,

  reserve $\mathsf{ilog}(k)$ tokens.

- For `pqad_delete(R,e)`, if there are $k$ elements in `Ins` and $d$ elements in `Del`,

  reserve $\mathsf{ilog}(d)$ tokens.

## 5 Binary Search Trees (15 pts)

Deleting elements from a binary search tree is in some ways more complicated than insertion. The difficulty is how to change the structure of the tree while maintaining the order invariant. In this problem we are not concerned with any shape invariant.

We define the *immediate successor node* of a node as the unique node in the tree with the next highest key in the tree. For example, in the binary search tree below, the immediate successor of 3 is 5, and the immediate successor of 9 is 11.

To delete the element with key $k$ from the binary search tree $B$ as follows:

(1) Find the node $T$ whose element has key $k$.

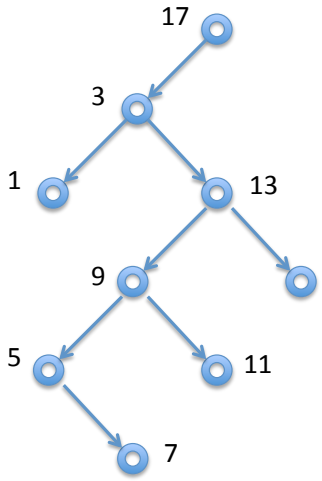
(2) If $T$ has no children, simply delete $T$ from the tree.

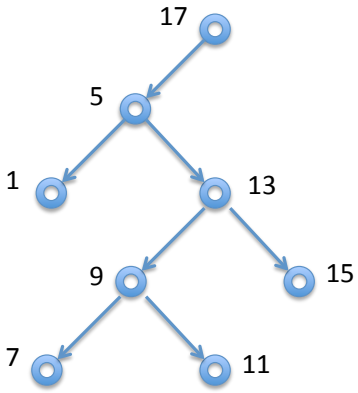
(3) If $T$ has one child, replace $T$ with that child.

(4) If the node has two children:

- Find the *immediate successor* node $S$ for $T$.
- The node $S$ can only have zero or one children.
- Delete $S$ (as in (2) or (3)) and put its element into $T$.

**Task 1** (5 pts). Illustrate the behavior of deletion by removing the element with key 3 from the tree shown on the left, drawing the resulting tree on the right.



Recall the definition of binary search tree nodes from lecture.

```
struct tree_node {
  elem data;
  struct tree_node* left;
  struct tree_node* right;
};
typedef struct tree_node tree;

bool is_ordtree(tree* T);  /* T is ordered binary tree */
```

**Task 2** (10 pts). Complete the following program to find the subtree whose key is the immediate successor node to the one in $T$ and return it. Your precondition should be strong enough to guarantee that the function can satisfy its postcondition and, at the same time, allow this function to be used as a part of deletion as described above. Your loop invariant(s) should be strong enough to imply safety of all pointer dereferences. In addition, together with the negated loop guard, it should imply the postcondition. It does not need to imply that the key at the result node is indeed the successor. **[Hint: Your code should not require any key comparisons.]**

```
tree* successor(tree* T)
//@requires is_ordtree(T) && T != NULL;
//@requires T->right != NULL;
//@ensures is_ordtree(\result) && \result != NULL;
{
  tree* S = T->right;
  while (S->left != NULL)
    //@loop_invariant is_ordtree(S) && S != NULL;
    {
      S = S->left;
    }
   return S;
}
```

## 6 Safety in C0 and C (25 pts)

Assume the following declarations, where "..." refers to an arbitrary unknown value.

**In C0**

```
int INT_MAX = 0x7FFFFFFF;

int x = ...;
int y = ...;
int z = ...;

int[] p = alloc_array(int, 5);
```

**In C**

```
#include <limit.h>

int x = ...;
int y = ...;
int z = ...;

int *p = xcalloc(5, sizeof(int));
```

**Task 1** (10 pts). For the following expressions, indicate all possible outcomes among `true`, `false`, *abort*, and *undefined*. If the behavior may be *undefined* then all other outcomes are automatically possible, so you don't need to list them explicitly. Assume that all initializations succeed without error. To get you started we have already filled in first row for you.

| Expression | In C0 | In C |
|---|---|---|
| `x-1 < x` | `false`, `true` | *undefined* |
| `x == INT_MAX \|\| x + 1 > x` | **true** | **true** |
| `x > INT_MAX - y \|\| x + y == y + x` | **true** | **undefined** |
| `x/(y/z) == (x/y)*z` | **true, false, abort** | **undefined** |
| `p[2] == p[3]` | **true** | **true** |
| `p[0] == p[5]` | **abort** | **undefined** |

The following C program is intended to compute and return an array containing the Fibonacci numbers $f_0, \ldots, f_n$, where $f_0 = 0$, $f_1 = 1$ and $f_n = f_{n-1} + f_{n-2}$ for $n \geq 2$.

Unfortunately, even if the contract is respected by the caller, there are several lines of code whose execution can exhibit *undefined behavior* in C and therefore do something completely arbitrary.

**Task 2** (5 pts). List exactly the lines in the program below whose behavior may be *undefined* in C, assuming that the caller adheres to the stated contract. We will deduct points for listed lines whose behavior is actually defined, as well as for missing lines whose behavior is undefined.

- Lines: **4, 5, 7, (9)**

**Task 3** (10 pts). In the space on the right, clearly write a proposed correction that fixes the program so it has the intended behavior. Your corrections should keep the general structure of the program intact, but they do not necessarily need to involve only the lines from the previous question. If the result would be undefined in ways that cannot be easily repaired, your function should return `NULL`. You may assume `contracts.h`, `xalloc.h` and other standard C libraries have been `#include`'d.

```
int *fib(int n) {
  REQUIRES(n >= 0);
  int *F = xcalloc(n+1, sizeof(int));
  F[0] = 0;
  if (n > 0) F[1] = 1;
  for (int i = 0; i <= n-2; i++) {
    if (F[i+1] <= INT_MAX - F[i]) {
      F[i+2] = F[i+1] + F[i];
    } else {
      free(F); return NULL;
    }
  }
  return F;
}
```