15-122: Principles of Imperative Computation

Recitation 11 Manu Garg, Josh Zimmerman, Arjun Hans, Nivedita Chopra

Discussing the code for deletion from a heap

When we delete the minimum element from the heap, we notice that while we're performing the operation, the ordering invariant of the heap is violated i.e. the intermediate heaps don't satisfy the ordering invariant.

To prove that the ordering invariant is satisfied at the end of the delete operation, we need to come up with an invariant that will hold of the intermeditate heaps and that is strong enough to prove the ordering invariant for the final heap.

Turns out an invariant that holds at each step is that the heap invariant is satisfied at all points, except at the current element, which may be larger than its children. We express this invariant as the `is_heap_except_down` function. Below is the invariant along with some library code that might be useful as reference.

```
1  struct heap_header {
2    int limit;  /* limit = capacity+1 */
3    int next;   /* 1 <= next && next <= limit */
4    elem[] data; /* \length(data) == limit */
5  };
6  typedef struct heap_header* heap;
7
8  /* Just checks the basic invariants described above, none
9   * of the ordering invariants. */
10 bool is_safe_heap(struct heap_header* H) {
11   if (!(H != NULL)) return false;
12   if (!(1 <= H->next && H->next <= H->limit)) return false;
13   //@assert \length(H->data) == H->limit;
14   return true;
15 }
16
17 /* Returns true if the priority of the node at index at i is greater than
18  * the priority of the node at index j */
19 bool ok_above(struct heap_header* H, int i, int j)
20 //@requires is_safe_heap(H);
21 {
22   return !higher_priority(H->data[j], H->data[i]);
23 }
24
25 bool is_heap(struct heap_header* H) {
26   if (!is_safe_heap(H)) return false;
27   for (int i = 2; i < H->next; i++)
28     //@loop_invariant 2 <= i;
29     if (!(ok_above(H, i/2, i))) return false;
30   return true;
31 }
32
33 /* H is a valid heap, except possibly at n,
34  * looking down in the tree */
35 /* If 2*n >= H->next then is_heap_except_down(H, n) == is_heap(H) */
36 bool is_heap_except_down(heap H, int n) {
37   if (!is_safe_heap(H)) return false;
```

```
 /* check parent <= node for all nodes except root (i = 1) */
 /* and children of n (i/2 = n) */
 for (int i = 2; i < H->next; i++)
  //@loop_invariant 2 <= i;
  {
   if (i/2 != n && !(ok_above(H, i/2, i)))
    return false;
   /* for children of node n, check grandparent */
   if (i/2 == n && (i/2)/2 >= 1 && !(ok_above(H, (i/2)/2, i)))
    return false;
  }
 return true;
}
```

Using this `is_heap_except_down` function as an invariant, we can write the `sift_down` and `pq_delmin` function as follows:

```
void sift_down(heap H, int i)
//@requires 1 <= i && i < H->next;
//@requires is_heap_except_down(H, i);
//@ensures is_heap(H);
{ int n = H->next;
 int left = 2*i;
 int right = left+1;
 while (left < n)
  //@loop_invariant 1 <= i && i < n;
  //@loop_invariant left == 2*i && right == 2*i+1;
  //@loop_invariant is_heap_except_down(H, i);
  {
   if (ok_above(H, i, left)
      && (right >= n || ok_above(H, i, right)))
    return;
   if (right >= n || ok_above(H, left, right)) {
    swap(H->data, i, left);
    i = left;
   } else {
    //@assert right < n && ok_above(H, right, left);
    swap(H->data, i, right);
    i = right;
   }
   left = 2*i;
   right = left+1;
  }
 //@assert i < n && 2*i >= n;
 //@assert is_heap_except_down(H, i);
 return;
}

elem pq_delmin(heap H)
//@requires is_heap(H) && !pq_empty(H);
//@ensures is_heap(H);
{
 int n = H->next;
 elem min = H->data[1];
 H->data[1] = H->data[n-1];
 H->next = n-1;
```

```
40   if (H->next > 1) {
41     /* H is no longer a heap! */
42     sift_down(H, 1);
43   }
44   return min;
45 }
```

Binary search trees — a quick recap

- Binary search trees are an implementation of *associative arrays* in a tree structure.
- We maintain the invariant (an *ordering* invariant). For every node in the tree:
  (a) All the elements in the left subtree have keys smaller than that of the node
  (b) All elements in the right subtree have keys greater than than that of the node
- The asymptotic complexity of insertion/lookup in a BST is $O(\log n)$. While this is slower than the $O(1)$ insertion/lookup time with hash tables, there are some good reasons why we may want to use BSTs instead:
  (a) Hash tables only have $O(1)$ insertion/lookup time if the hashing function has a good key distribution. Otherwise, if most of the elements hash to a small number of chains, insertion/lookup will slow down to linear time.
  (b) Also, hash tables require us to allocate large backbone arrays; a number of these may be empty throughout the hash table's usage, which wastes memory.

  There are some applications in which hash tables are preferable and some in which BSTs are preferable, so it's a matter of evaluating the specific case and deciding which data structure works better for it.
- One important thing to note is that BSTs are not always balanced, and so we won't necessarily always have $O(\log(n))$ runtime for the common operations. This is largely dependent on the order in which the keys are inserted.

Checkpoint 0

Insert the keys 1,2 and 3 in the order 2, 1, 3 and draw the tree formed. Then insert the keys in the order 1,2,3. What do you notice?

Playing with binary search trees!

Using the visualization at `http://www.cs.usfca.edu/~galles/visualization/BST.html`, try inserting these keys in order:

$$2, 4, 1, 3, 5$$

Then try deleting 2 and then 4 from the tree.

Discussing the code for insertion into the tree

Now let's consider the code for `tree_insert`. It's a recursive function.

```
tree* tree_insert(tree* T, elem e)
//@requires is_ordtree(T);
//@requires e != NULL;
//@ensures is_ordtree(\result);
{
    if (T == NULL) {
        /* create new node and return it */
        T = alloc(struct tree_node);
        T->data = e;
        T->left = NULL;
        T->right = NULL;
        return T;
    }
    int r = key_compare(elem_key(e), elem_key(T->data));
    if (r == 0) {
        T->data = e;        /* modify in place */
    } else if (r < 0) {
        T->left = tree_insert(T->left, e);
    } else {
        //@assert r > 0;
        T->right = tree_insert(T->right, e);
    }
    return T;
}
```

Note that in the cases where `r < 0` and `r > 0` we assign the right and left sub-trees to the result of calling the function recursively. Also, we need to return `T` in the case where we enter the case where `T == NULL` (an empty tree), since we have no way of modifying the tree that was passed in as an argument.

We can understand `tree_insert` as follows. Suppose that the element we want to insert is e

(1) If we reach an empty tree (`T == NULL`), then we just create a new node, set the data to e and return the node.

(2) Otherwise, we compare the key e_k of e to the key n_k of the node n we're currently at in the tree. We then have the following three cases:

   (i) If k_e and k_n are equal, then we just write over the data in the node n with e.

   (ii) If $k_e < k_n$, then element e belongs in the left-subtree of n. We therefore call `tree_insert` on the subtree `T->left` and insert e somewhere in `T->left`

   (iii) Otherwise, $k_e < k_n$, so the element e belongs in the right-subtree of n. We therefore call `tree_insert` on the subtree `T->right` and insert e somewhere in `T->right`.

Checkpoint 1

Recursion on binary search trees can be a tricky concept to master. Here are some practice problems to help you. Note that some of these problems only rely upon the structure of a BST, while others rely upon the BST ordering invariant. Your solution should use the struct definitions and the client-side functions we defined in class (given below for your reference).

Your solutions should be *short* (around 10 lines at most). You may also need a helper function to access the internals of the tree (as we did in `bst_insert` with `tree_insert`.

```
key elem_key(elem e);

int key_compare(key k1, key k2);

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

struct bst_header {
  tree* root;
};
typedef struct bst_header* bst;
```

(a) Write a function `elem bst_max(bst B)` that returns the element with the maximum key in a given BST.

(b) Write a function `int count_leaves(bst B)` that counts the number of leaves in a given BST.