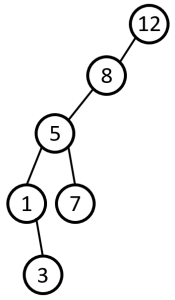
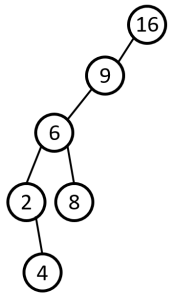
CPSC 223 - Fall 2017 - Exam #2

Problem 0 (1 point): Write your name on the *back* of this exam package. What CFL team finished second in the Eastern Conference and lost to Saskatchewan in the East Semifinal?

Problem 1 (6 points): Show the end result of adding the following values in the given order into a plain (unbalanced) binary search tree. 50 90 80 10 40 99

Problem 2 (12 points): Show the first splay tree after searching for 1 in it and the second after searching for 3 in it.

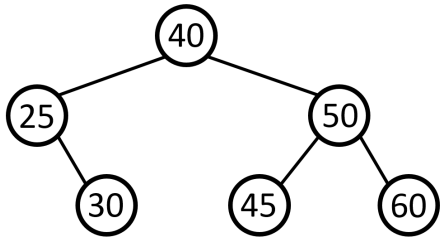




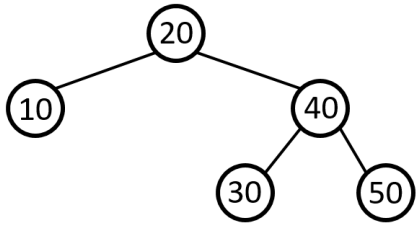
Problem 3 (18 points):

(a) For each of the following AVL trees, show the AVL tree that results from adding 33.

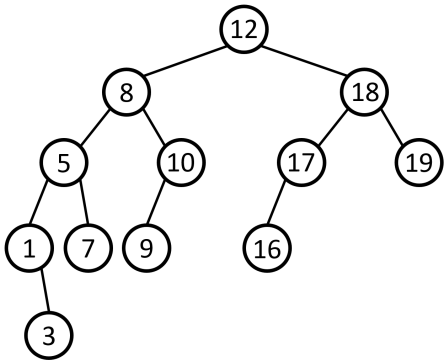
(i)



(ii)

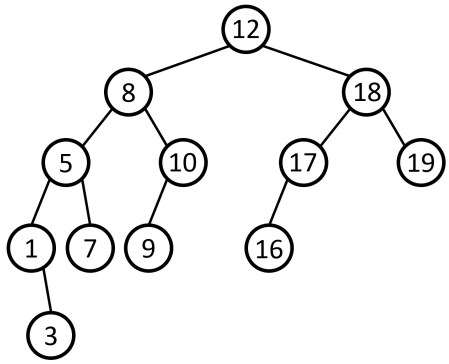


(b) Show the AVL tree that results from deleting 19 from the following AVL tree.

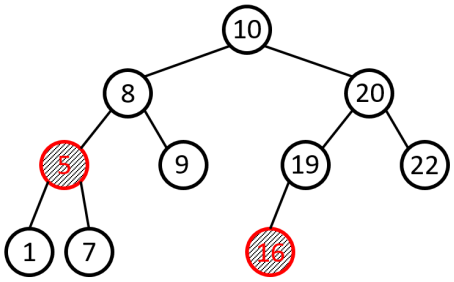


Problem 4 (12 points):

(a) Without changing the structure of the following tree, color some of the nodes red in order to make it a Red-Black tree. List the values of the nodes you colored red.



(b) Show the Red-Black tree that results from adding 18 to the following Red-Black tree (the cross-hatched nodes are red).



Problem 5 (9 points): Prof. Nels J. Greman suggests the following implementation of `smap_remove` for a hash table using open addressing with linear probing. It attempts to avoid having to mark slots as deleted by finding an entry to fill the slot vacated by the removed entry. His idea is to find the last entry in the contiguous block of used entries that contains the one to be removed and move that one to fill the vacated slot. His idea does not work. Given the following values of the hash function, which of the following sequences of operations exposes the error (that is, which sequences give at least one incorrect result for one of the operations in the sequence)? Assume that the size of the hash table is 8 during the entirety of each sequence. Any number of the sequences may expose the error.

```
// hash("BOM") = 1    hash("BWI") = 5    hash("CTU") = 5
// hash("DCA") = 5    hash("KIX") = 0    hash("SCL") = 0

void smap_remove(smap *m, const char *key) {
  if (m != NULL) {
      int index = smap_table_find_key(m->table, key, m->hash, m->capacity);
      if (m->table[index].occupied)
        {
          // find entry before next open slot
          int end = index;
          while (m->table[(end + 1) % m->capacity].occupied) {
              end = (end + 1) % m->capacity;
          }

          // move that entry to replace removed one
          free(m->table[index].key);
          m->table[index].key = m->table[end].key;
          m->table[index].value = m->table[end].value;
          m->table[end].occupied = false;
          m->size--;
        }
  }
}
```

(a)

```
int dummy = 2;
smap *m = smap_create(hash);
smap_put(m, "BOM", &dummy);
smap_put(m, "BWI", &dummy);
smap_remove(m, "BOM");
bool bwi = smap_contains_key(m, "BWI");
smap_destroy(m);
```

(b)

```
int dummy = 2;
smap *m = smap_create(hash);
smap_put(m, "DCA", &dummy);
smap_put(m, "BWI", &dummy);
smap_put(m, "CTU", &dummy);
smap_put(m, "KIX", &dummy);
smap_put(m, "SCL", &dummy);
smap_remove(m, "KIX");
bool bwi = smap_contains_key(m, "BWI");
bool scl = smap_contains_key(m, "SCL");
smap_destroy(m);
```

(c)

```
int dummy = 2;
smap *m = smap_create(hash);
smap_put(m, "DCA", &dummy);
smap_put(m, "BWI", &dummy);
smap_put(m, "CTU", &dummy);
smap_put(m, "KIX", &dummy);
smap_put(m, "SCL", &dummy);
smap_remove(m, "BWI");
bool scl = smap_contains_key(m, "SCL");
smap_destroy(m);
```

Problem 6 (18 points): For each of the following, give your best (tightest) answer in asymptotic (big-O) notation. Except where noted, no justification is required.

(a) Suppose we added `find_min_int` to our balanced binary search tree implementation of `isset`, which returns the smallest integer in the set. What is the running time of `find_min_int`? (Assume that no changes are made to other functions.)

(b) Suppose we added a `count_consecutive_keys` operation to our hash table implementation of `ismap` (map from integer keys to strings), which counts the k such that both k and $k+1$ are present as keys in the map. What is the expected running time of `count_consecutive_keys` assuming that the hash function works well? What is the worst-case running time? (Again, assume no changes to other functions. Also, assume the load factor is $\Theta(1)$.) Explain your answers very briefly (your explanation may consist of pseudocode).

(c) Repeat (b), but with `ismap` implemented using a balanced binary search tree. No explanation is necessary.

(d) Given an adjacency matrix representation of a graph directed G, how long does it take to create the adjacency list representation? (Assume you have access to the members of the `struct`s used for both representations.) Does your answer change for an undirected graph?

(e) Suppose we had a special case of graphs and a corresponding implementation of a priority queue that could perform both `extract_min` and `decrease_priority` operations in $O(1)$ time and `build` in $O(n)$ time. Assuming the graph is represented using an adjacency list, what is the resulting running time for Dijkstra's algorithm?

Problem 7 (12 points): Show how to modify depth-first search so that it counts the number of distinct simple paths that begin at the starting vertex. For example, starting at vertex 0 in a directed graph with edges $(0,1),(0,3),(1,2),(1,3),(2,0),(2,3)$ would result in a count of 6 because the paths are 0, 01, 03, 012, 013, 0123. You should modify or add no more than eight lines of code to the following. Your change will result in an algorithm with a superpolynomial worst-case running time.

```
typedef struct {
  int n;          // the number of vertices
  int *list_size; // the size of each adjacency list
  int *list_cap;  // the capacity of each adjacency list
  int **adj;      // the adjacency lists
} ldigraph;

typedef struct {
  const ldigraph *g;
  int from;
  int *color;
} ldig_search;

int digraph_count_paths(const ldigraph *g, int from) {
  if (g == NULL || from < 0 || from >= g->n)
      return -1;

  ldig_search *s = ldig_search_create(g, from);
  if (s != NULL) {
      // start at from
      s->color[from] = GRAY;
      ldigraph_count_visit(g, s, from);
  }
  else
      return -1;

  ldig_search_destroy(s);
  return -1;
}

void ldigraph_count_visit(const ldigraph* g, ldig_search *s, int from) {
  // get vertices we can reach in one edge from from
  const int *neighbors = g->adj[from];

  // iterate over outgoing edges
  for (int i = 0; i < g->list_size[from]; i++) {
      int to = neighbors[i];
      if (s->color[to] == WHITE) {
          // found an edge to a new vertex -- explore it
          s->color[to] = GRAY;

          ldigraph_count_visit(g, s, to);
      }
  }

  // mark current vertex finished
  s->color[from] = BLACK;
}
```

Problem 8 (12 points): The following recursive function computes the probability of reaching a total of exactly n before rolling a 1 when repeatedly rolling a fair six-sided die. The recurrence is $p(0) = 1.0$ (you will always reach a total of zero), $p(1) = 0.0$ (you can't get a total of 1 without rolling a 1), and $p(n) = \frac{1}{6}\sum_{r=2}^{min(6,n)} p(n-r)$ for $n \geq 2$ (the probability of getting to, for example, 20 is the probability of getting to 18 and then rolling a 2 plus the probability of getting to 17 and then rolling a 3 plus the probability of getting to 16 and then rolling a 4 plus the probability of getting to 15 and then rolling a 5 plus the probability of getting to 14 and then rolling a 6).

```
double roll_rec(int n)
{
  if (n == 0)
      return 1.0;
  else if (n == 1)
      return 0.0;
  else
    {
      double sum = 0.0;
      for (int r = 2; r <= min(6, n); r++)
          sum += roll_rec(n - r);
      return sum / 6.0;
    }
}
```

The recursive solution is inefficient. Convert it to a dynamic programming solution that runs in $O(n)$ time.

```
double roll_dp(int n)
{
  // allocate an array to hold results of subproblems (on the stack is OK)


  // initialize elements of that array corresponding to base cases



  // loop over other elements in array

  for (int i =       ;            ;          )
    {
      // compute value of formula
      double sum = 0.0;
      for (int r = 2; r <= min(6, i); r++)
        {


        }
     // save result


   }
   // return element of the array that holds the answer to original problem

   return

}
```