CPSC 223 - Fall 2018 - Exam #2

Write your name and NetID on only this page of this exam package in the boxes provided and **write your answers on the front of the provided sheets no closer than $\frac{1}{2}$ inch from the edges of the pages**.

Name

NetID

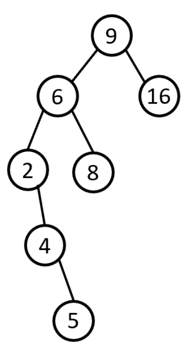
Problem 0 (1 point):

(a) What is the opposite of "to embiggen"?

(b) You learned a lot about C, data structures, and programming techniques this semester. You are happy. Draw a picture of your happy face.

Problem 1 (6 points): Show the end result of adding the following values in the given order into a plain (unbalanced) binary search tree. 30 50 40 10 20 25

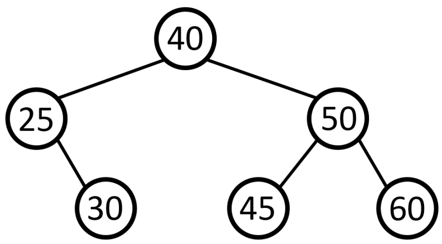
Problem 2 (4 points): Show the result of searching for 5 in the following splay tree.



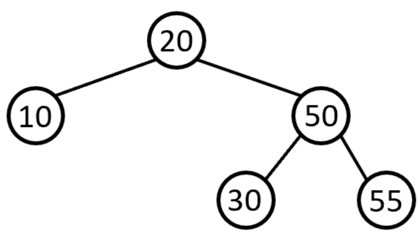
Problem 3 (10 points):

(a) For each of the following AVL trees, show the AVL tree that results from adding 47.

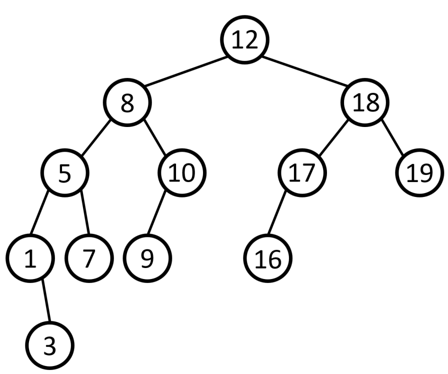
(i)



(ii)

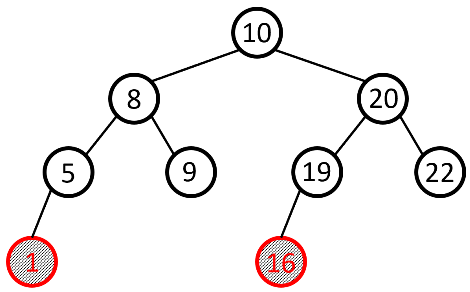


(b) Show the AVL tree that results from deleting 9 from the following AVL tree.

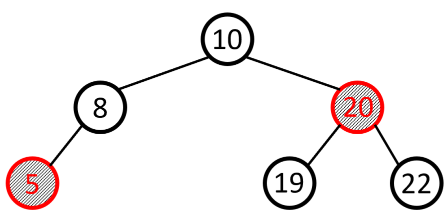


Problem 4 (12 points):

(a) Show the Red-Black tree that results from adding 3 to the following Red-Black tree (the cross-hatched nodes are red).

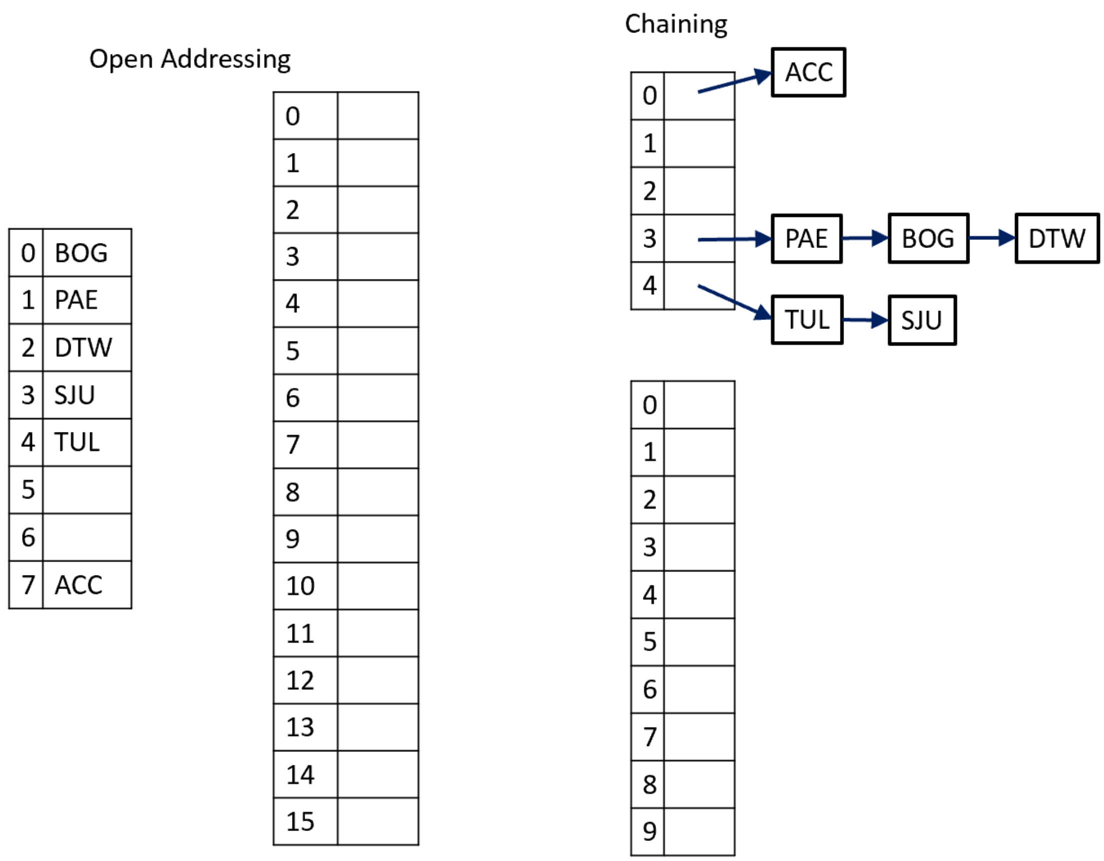


(b) Show the Red-Black tree that results from deleting 22 from the following Red-Black tree.



Problem 5 (12 points): Below is an illustration of a set of keys first stored in a hash table using open addressing and then in a hash table using chaining. For each implementation, draw the diagram that results after doubling the number of slots in the hash table, adding the old keys to the new hash table, and then adding keys EAT and UST, assuming the hash values are as shown in the table below.

Key	Value	Key	Value
ACC	15	PAE	8
BOG	23	SJU	64
DTW	18	TUL	34
EAT	20	UST	31



Problem 6 (15 points): Consider the following subset of an ADT for a list of 3-letter strings.

```
// returns the size of the list
int apl_size(const apl *l);

// returns a pointer to the string at the given index in the list
const char *apl_get(const apl *l, int i);

// creates a new iterator at the beginning of the list
apl_it *apl_start(const apl *l);

// returns a pointer to the string at the current position of the iterator and
// advances the iterator; returns NULL when past the end of the list
const char *apl_it_get(apl_it *i);

// destroys an iterator
void apl_it_destroy(apl_it *i);
```

The following function determines whether two lists are equal: whether they are the same length and each string on the first list compares equal to the string at the same index on the second list.

```
bool lists_equal(const apl* l1, const apl* l2)
{
  if (l1 == NULL || l2 == NULL || apl_size(l1) != apl_size(l2)) return false;
  int i = 0;
  while (i < apl_size(l1) && strcmp(apl_get(l1, i), apl_get(l2, i)) == 0)
      i++;
  return i == apl_size(l1);
}
```

(a) What is the worst-case asymptotic (big-O) running time of `lists_equal` when the list is implemented with a doubly-linked list? (For this and subsequent parts, assume that `malloc` and `free` run in $O(1)$ time.)

(b) What is the worst-case asymptotic running time when the list is implemented with an array?

(c) Rewrite `lists_equal` so it uses an iterator and runs in worst-case $O(n)$ time for both list implementations. There should be no memory leaks. You needn't rewrite the header or the first line.

Problem 7 (20 points): Consider the following subset of an ADT for a map from length-3 strings to arrays of integers and a function that determines if two maps contain the same set of keys.

```
bool smap_contains_key(const smap *m, const char *key);
void smap_for_each(smap *m, void (*f)(const char *, int *, void *, void *), void *arg1, void *arg2);

void check_membership(const char *key, int *value, void *arg1, void *arg2) {
  smap *m2 = arg1;
  int *count = arg2;

  if (smap_contains_key(m2, key))
      (*count)++;
}

bool same_keys(smap *m1, smap *m2) {
  if (m1 == NULL || m2 == NULL || smap_size(m1) != smap_size(m2)) return false;

  int match_count = 0;
  smap_for_each(m1, check_membership, m2, &match_count);
  return match_count == smap_size(m1);
}
```

(a) (i) What is the worst-case asymptotic (big-O) running time of `same_keys` when the maps are both implemented using hash tables, in terms of n, the total number of keys in the map, and m, the number of slots in the hash table?

(ii) What is the expected asymptotic running time in terms of n and m, assuming the hash function is good?

(b) What is the worst-case asymptotic (big-O) running time of `same_keys` in terms of n when the maps are implemented using the following kinds of binary search tree? (`smap_for_each` iterates through the keys in sorted order in these cases.)

(i) both are unbalanced binary search trees

(ii) both are AVL trees

(iii) both are red-black trees

(iv) both are splay trees

(c) On the next page, write pseudocode for an implementation of `same_keys` that improves the asymptotic worst-case running time for all the above implementations and explain briefly what that running time is. (Your worst-case improvement may come at the expense of worse expected running time in some cases.)

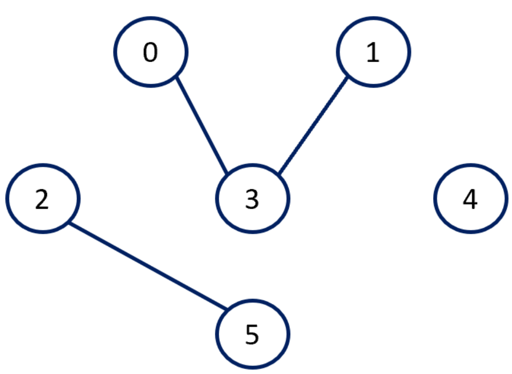
Pseudocode for `same_keys` goes here.

Problem 8 (12 points): Consider the following partial implementation of an undirected graph using adjacency lists.

```
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
} lugraph;

typedef struct
{
  int *status;    // UNSEEN, PROCESSING, or DONE (see enum below)
  int num;        // the number of connected components
  int *size;      // the size of each connected component
  int **vertices; // the vertices in each connected component
} lu_search;

enum {UNSEEN, PROCESSING, DONE};

// creates an lu_search struct with all vertices initialized to UNSEEN,
// number and sizes of all connected components initialized to 0,
// and room for the largest possible number of connected components
// and the number of vertices in each of those components
lu_search *lu_search_create(int n);
```

On the following page, complete the code to find all the connected components of an undirected graph using depth-first search. Your completed code should run in $O(n + m)$ time on an graph represented with an adjacency matrix. A connected component is a maximal non-empty set of vertices so that for every pair of vertices in the set there is at least one path between the pair. For example, in the graph below the connected components are $\{0, 1, 3\}$, $\{2, 5\}$, and $\{4\}$, and so the `lu_search` struct returned from `lugraph_connected_components` should be set so that the `num` field is 3, the `size` field is the array {`3, 2, 1`}, and the `vertices` field is the ragged 2-D array { {`0, 1, 3`}, {`2, 5`}, {`4`} } (the vertices in each row may be in any order, and the rows may be in any order as long as the sizes are in the corresponding order too).

(Hint: think about where in the DFS algorithm you will be finished with one connected component and where you should add a vertex to the current connected component.)



```
lu_search *lugraph_connected_components(const luraph *g)
{
  lu_search *s = lu_search_create(g->n);

  for (int i = 0; i < g->n; i++)
    {
      if (s->status[i] == UNSEEN)
        {
          lugraph_dfs_visit(g, s, i);



        }
    }
  return s;
}

void lugraph_dfs_visit(const lugraph* g, lu_search *s, int from)
{




  s->status[from] = PROCESSING;

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






          lugraph_dfs_visit(g, s, to);
        }
    }

  // record current vertex finished
  s->status[from] = DONE;




}
```

Problem 9 (8 points): Below is a recursive implementation of a function to compute the nth Catalan number.

```
// computes the nth catalan number for nonnegative integers n
long catalan_rec(int n)
{
  if (n < 0) return 0; // check that n is nonnegative
  if (n == 0) return 1;
  long sum = 0;
  for (int i = 0; i < n; i++)  {
      sum += catalan_rec(i) * catalan_rec(n - 1 - i);
  }
  return sum;
}
```

(a) Explain briefly why the recursive solution is inefficient.

(b) Rewrite the implementation using dynamic programming.

```
long catalan_dp(int n)
{
  // create a memo to hold all the values



  // initialize the value corresponding to the base case



  // loop over all the other entries in the memo...

  for (                                    )
    {
      // ...computing the value of that entry






      // record that value


    }

  // return the entry we're interested in


}
```