CS136 EXERCISE SOLUTIONS: MODULE 10

Linked lists

```
struct llnode *cons(int first, struct llnode *rest) {
  struct llnode *node = malloc(sizeof(struct llnode));
  node->data = first;
  node->next = rest;
  return node;
}
```

Linked lists: recursive traversal

```
void llnode_print(const struct llnode *node) {
  if (node == NULL) {
    printf("empty");
  } else {
    printf("(cons %d ", node->data);
    llnode_print(node->next);
    printf(")");
  }
}
```

```
struct llnode *square_each(const struct llnode *node) {
  if (node == NULL) {
    return NULL;
  } else {
    return cons(node->data * node->data,
                square_each(node->next));
  }
}
```

Functional vs Imperative approach

```
void square_each_mutate(struct llnode *node) {
  while (node != NULL) {
    node->data *= node->data;
    node = node->next;
  }
}
```

```c
void llnode_print(const struct llnode *node) {
  int bracket_count = 0;
  while (node != NULL) {
    printf("(cons %d ", node->data);
    node = node->next;
    bracket_count++;
  }
  printf("empty");
  while(bracket_count) {
    printf(")");
    bracket_count--;
  }
  printf("\n");
}
```

Node sharing: wrappers

```c
void list_insert_back(struct llist *ll, int item) {
  if (ll->front == NULL) {
    ll->front = cons(item, NULL);
  } else {
    struct llnode *node = ll->front;

    while (node->next != NULL) {
      node = node->next;
    }
    node->next = cons(item, NULL);
  }
}
```

Node sharing: wrappers

```c
struct llnode *llnode_cp(const struct llnode *node) {
  if (node == NULL) {
    return NULL;
  } else {
    return cons(node->data, llnode_cp(node->next));
  }
}
struct llist *list_cp(const struct llist *ll){
  struct llist *newlist = malloc(sizeof(struct llist));
  newlist->front = llnode_cp(ll->front);
  return newlist;
}
```

```c
int list_remove_index(struct llist *ll, int index) {
  assert(ll);
  if (index == 0) return list_remove_front(ll);
  struct llnode *node = ll->front;
  while (index > 1) {
    node = node->next;
    index--;
  }
  int val = node->next->data;
  struct llnode *oldnode = node->next;
  node->next = node->next->next;
  free(oldnode);
  return val;
}
```

Augmentation

```c
// to list_create, add:
llst->length = 0;

// to list_insert_back, list_insert_front, add:
llst->length++;

// to list_remove_back, list_remove_front, add:
llst->length--;

int list_length(const struct llist *lst) {
  return lst->length;
}
```

Trees

```c
void bstnode_destroy(struct bstnode *n) {
  if (n != NULL) {
    bstnode_destroy(n->left);
    bstnode_destroy(n->right);
    free(n);
  }
}
void bst_destroy(struct bst *b) {
  assert(b);
  bstnode_destroy(b->root);
  free(b);
}
```

```c
void bstnode_insert(struct bstnode *n, int item) {
  if (item < n->item) { // must insert to the left
    if (n->left == NULL) n->left = make_leaf(item);
    else bstnode_insert(n->left, item);
  } else if (item > n->item) { // must insert to the right
    if (n->right == NULL) n->right = make_leaf(item);
    else bstnode_insert(n->right, item);
  }
}

void bst_insert(struct bst *b, int item) {
  if (b->root == NULL) {
    b->root = make_leaf(item);
  } else {
    bstnode_insert(b->root, item);
  }
}
```

Array-based trees

```c
const int SENTINEL = -1;

int sum_tree(int *arr, int len) {
  int total = 0;
  for (int i = 0; i < len; ++i) {
    if (*(arr + i) != SENTINEL) {
      total += *(arr + i);
    }
  }
  return total;
}

int main(void) {
  int tree[15] = {20, 10, 30, 8, 15, 26, 31, -1, -1, 12, -1, 24, 29, -1, 32};
  assert(sum_tree(tree, 15) == 237);
}
```