CSC263 Assignment 1

Juanxi Li

May 19, 2018

Sources Consulted

- Lecture 1 Notes - Introduction, Complexity Review
- `https://medium.com/@ssbothwell/counting-inversions-with-merge-sort-4d9910dc95f0`
- Delete a node from Binary Search Tree: `https://www.youtube.com/watch?v=gcULXE7ViZw`
- Young tableaus: `https://ita.skanev.com/06/problems/03.html`
- `https://kukuruku.co/post/young-tableaux-in-the-tasks-of-searching-and-sorting/`
- `http://codinggeeks.blogspot.com/2010/04/young-tableau.html`

Problems

1. Inversions
   (a) (3,4), (3,5) and (4,5) where A[3] = 7, A[4] = 5, and A[5] = 1.
   (b) The completely reverse-sorted descending-order array with items {n, n-1, n-2, ... , 2, 1} has the most inversions. It has $1+2+3+...+(n-1)+n$ inversions, or $\frac{n(n-1)}{2}$. This is also the worst-case runtime of insertion sort.
   (c) Runtime of insertion sort, above the best case θ(n), is proportional to # of inversions.

   Pseudo-code of insertion sort from lecture 1 uses an outer loop j which traverses the array from 2 to n. This loop runs θ(n) times.

   Inner loop **i** begins at **j-1** and traverses backwards, swapping elements as long as A[i] > A[j] (or, in other words, if an inversion exists). Therefore the # of inversions in the array = # of swaps that must be made by the inner loop during insertion sort.
   (d) Solution is mergesort, which runs in $\theta(n\log n)$, with an added line of code to increment an inversion counter during the merge step. This line runs in constant time $\theta(1)$; therefore the entire algorithm still runs in $\theta(n\log n)$.

   Recursively, total inversions = inversions encountered in sorting left half + inversions encountered in sorting right half + inversions between the two halves.

   During `merge()`, the left and right halves are already sorted, and their inversions will already have been recursively counted. So additional inversions can only exist *between* the two halves. When `merge()` copies a value from the right half back to the main array, it is necessarily an inversion relative to any numbers that remain in the left half. So this is the amount by which we increase the counter.

   Derived from pseudo-code in Lecture 1 slides:

```
mergeSortInversions(A,p,r) {
    inv = 0; // variable to track inversions
    if (p < r) {
        q = floor((p+r)/2)
        mergeSortInversions(A,p,q)
        mergeSortInversions(A, q+1, r)
        merge(A,p,q,r,inv)
    }
    return inv;
}

merge(A, p, q, r, inv) {
    n1 = q - p + 1
    n2 = r - q
    copy A[p,q] to L[1...n1]
    copy A[q+1,r] to R[1...n2]
    L[n1+1] = R[n1+1] = +Inf
    i = j = 1
    for (k=p to r) {
        if (L[i] < R[j]) {
            A[k] = L[i]
            i++
        } else {
            // a value R[j] was taken from the right
            // side in the process of merging;
            // R[j] must be an inversion relative to
            // everything remaining on the left side
            A[k] = R[j]
            inv += L.length - i
            j++
        }
    }
}
```

2. Extract second largest

   (a) This is the same thing as finding max in array Q and deleting the larger of its two children. Here is a modified version of `extractMax()` from lecture which accompishes this.

```
extractSecondLargest(Q) {
  int secondMax;
  int maxVal;

  // max is at 1, its children are at positions 2 and 3

  if (Q[2] == NULL) {       // second largest doesn't exist
    return NULL;
  }
  else if (Q[3] == NULL) { // Q only has two items
    secondMax = 2;
  }
  else {                    // compare Q2 and Q3
    if (Q[2] >= Q[3]) { secondMax = 2; }
    else              { secondMax = 3; }
  }

  maxVal = Q[secondMax];

  // swap w last, and delete
  Q[secondMax] = Q[Q.heapsize];
  Q.heapsize = Q.heapsize - 1;
  maxHeapify(Q, secondMax);

  return maxVal;
}
```

(b)
- **Why element is second largest:** because in our implicit definition of the heap as an array, the maximum element is always stored in position 1, and its children are stored in positions 2 and 3.

  We compare nodes 2 and 3, returning the larger value. This is the second-largest value, behind max.

- **Why maintains heap property:** Shape is maintained by swapping the second-largest with the last item and removing the last item. Heap property is maintained by calling `maxHeapify()` on the position previously occupied by the second-largest, which bubbles that value down until heap property is restored.

- **Why worst-case run-time is O(logn):** Locating the second-largest item and swapping with last item occurs in constant time. max() is found in $\theta(1)$ since it's just the first item, and we make at most one comparision between the children at Q[2] and Q[3] if they exist. The swap takes constant time.

  `maxHeapify()` bubbles down at most h times where h is height of tree, which is $log(n)$ for binary heap.

3. Heap Delete

   (a) This will be very similar to the pseudocode from Q2, but we delete item i instead of item 2 or 3. I assume the function should return the item.

```
heapDelete(A, i) {
  int iVal = A[i];

  // swap w last, and delete
  A[i] = A[A.heapsize];
  A.heapsize = A.heapsize - 1;
  maxHeapify(A, i);

  return iVal;
}
```

   (b)
   - **Why works:** Swapping ith node with last element, and then deleting last element (by shortening the array) maintains heap shape. Calling `maxHeapify()` on the subtree rooted at i ensures that the new ith element bubbles down until the heap property is restored. Outside of this subtree, no other part of the heap is changed.
   - **Why O(logn):** Location of node to be removed is already known and supplied via integer i. Swap takes constant time. After swap, `maxHeapify()` bubbles element i down max h_i times, where h_i is height of subtree rooted at i. Worst case, i is the root and h_i = height of entire tree, which is $log(n)$ for complete binary tree.

4. d-ary heap

   Answers are given for **root position at $i = 0$**, as this was the easiest for actually testing code.

   (a) For any node in d-ary array with index i:
   - parent: $\lfloor \frac{i-1}{d} \rfloor$
   - children: $\{ni + 1, ni + 2, ..., ni + d\}$

   (b) (n is # of items; i.e. item n exists at index n-1)

   height: $\lceil log_d(d-1) + log_d(n) - 1 \rceil = \theta(log_d(n))$

(c) I thought the best way to understand this question would be to write a working implementation in C (not all functions shown):

```c
#define DARY 3 // <- just as example, i.e. ternary heap

void maxHeapify(int *arr, int i, int *heapsize) {
  // maxHeapify the heap rooted at i
  // each node has DARY children
  int largest = i;
  int dary = DARY;
  for (int k = 1; k <= dary; k++) {
    int child = dary*i + k;
    if (arr[child] > arr[largest]) {
      largest = child;
    }
  }

  if (largest != i) {
    // swap with largest,
    // then maxHeapify the tree rooted at largest
    exchange(arr, i, largest);
    maxHeapify(arr, largest, heapsize);
  }
}

int extractMax(int *arr, int *heapsize) {
  if (*heapsize < 1) {
    printf("ERROR: heap is empty. Nothing to return.\n");
    return INT_MIN; // int placeholder for -inf
  }
  else {
    int max = arr[0];
    // swap root with last item, get rid of last item
    int last = arr[*heapsize-1];
    arr[0] = last;
    *heapsize = *heapsize - 1;
    maxHeapify(arr, 0, heapsize);
    return max;
  }
}
```

`extractMax()` **runtime analysis:**

- `int max` is found in constant time (always the root).
- `maxHeapify()` is recursively called, starting with the root position; worst case, the value there needs to bubble all the way to the bottom of the tree, or h times. Per question b, height of a d-ary tree is $\theta(log_d(n))$.
- Therefore worst-case runtime of `extractMax()` for d-ary tree is $O(log_d(n))$.

(d) Implementation of increaseKey():

```
void increaseKey(int *arr, int i, int key) {
  if (key < arr[i]){ printf("ERROR: %d is less than current arr[i] %d\n", key, i);
     }
  else {
    arr[i] = key;
    while (i > 0) {
      int j = (i-1) / DARY; // parent
      if (arr[j] >= arr[i]) {
        break; // parent is greater, all is well
      } else {
        // parent is smaller; swap them
        exchange(arr, i, j);
        i = j;
      }
    }
  }
}
```

`increaseKey()` runtime analysis:

- The `while` loop runs as long as the current node (at position i) being checked has a parent, and while `arr[i]` is greater than its parent. The max number of times this can happen is the # of edges between `i` and the root. Worst case, `i` is at the bottom of the tree, meaning the loop can occur maximum h times, where h is the height of the tree.
- Per question b, height of a d-ary tree is $\theta(log_d(n))$.
- Therefore worst-case runtime of `increaseKey()` for d-ary tree is $O(log_d(n))$.

(e) Implementation of `insert()`:

```
define MAX 10; // <- to help malloc space for array
void insert(int *arr, int key, int *heapsize) {
  if (*heapsize == MAX) {
    printf("ERROR: Heap at max size %d\n", *heapsize);
  } else {
    *heapsize = *heapsize + 1; // expand queue by 1
    arr[*heapsize-1] = INT_MIN; // set new, last item to large negative value
    increaseKey(arr, *heapsize-1, key); // update its priority to "key" and bubble
         up to right place
  }
}
```

`insert()` runtime analysis:

- `insert()` makes a single call to `increaseKey()` after adding the new value to the end of the array. All other lines run in constant time.
- Per question d above, `increaseKey()` worst-case runtime is $O(log_d(n))$, so `insert()` also runs in $O(log_d(n))$.

5. (a) reverse Young Tableau:

15	14	8	4
12	9	5	$-\infty$
3	$-\infty$	$-\infty$	$-\infty$
1	$-\infty$	$-\infty$	$-\infty$

(b) **The definition of an empty reverse Young tableau** is that all elements are nonexistent = $-\infty$.

A reverse Young Tableau requires that in an $m \times n$ tableau:

$$Y[a,b] \geq Y[a+i, b+j]$$

for
$a = 1, 2, ..., m$
$b = 1, 2, ..., n$
$i = 1, 2, ...m - a$
$j = 1, 2, ...n - b$

There exist no values smaller than -∞; therefore the only tableau that satisfies this constraint for $Y[1,1]$ = -∞ is if all other elements are also -∞.

The definition of a full reverse Young tableau is that no elements are $-\infty$.

If $Y[m,n] > -\infty$, all elements $Y[a,b]$ for $a \leq m$ and $b \leq n$ must be $\geq Y[m,n]$, or $> -\infty$. Therefore no elements are empty and the tableau must be full.

(c) Again, will put here a working implementation in C. Not all function definitions are shown.

```c
int extractMax(tableau *t) {
  int max = t->elements[0];
  if(max == -INT_MAX){
    printf("ERROR: tableau is empty. No max to return\n");
  } else {
    int last = t->m*t->n-1;
    // swap max and last item (smallest)
    exchange(t, 0, last);
    // empty out the last position
    t->elements[last] = -INT_MAX;
    // recursively bubble down the element at 0th position
    cell topLeftCell = cellAt(1,1);
    bubbleDown(t, topLeftCell);
  }
  return max;
}

void bubbleDown(tableau *t, cell c) {
// Compare down, compare right.
// If smaller than either/both, exchange w larger item
// Then call itself on the newly exchanged cells

  cell pos = c; // track position in the tableau
  int c_v = get(t,c); // get c's value

  char look; // which directions can we look?
  cell d = lookDown(c);
  cell r = lookRight(c);
```

```
if(isIn(t,d)) {
  if(isIn(t,r)) { look = 'b'; } // can look both ways
  else { look = 'd'; }  // look down
} else {
  if(isIn(t,r)){ look = 'r'; } // look right
}

char exch; // which exchange case are we doing?
int d_v;
int r_v;

switch(look) {
  case 'b': // look both ways
    d_v = get(t, d);
    r_v = get(t, r);

    if (c_v >= d_v && c_v >= r_v) { ; } // all is ok
    else if (c_v < d_v && c_v >= r_v) { exch = 'd'; }
    else if (c_v < r_v && c_v >= d_v) { exch = 'r'; }
    else { // smaller than both; swap with bigger one
      if (r_v > d_v) { exch = 'r'; }
      else { exch = 'd'; }
    }
    break;
  case 'r': // check r
    r_v = get(t, r);
    if (c_v < r_v) { exch = 'r'; }
    break;
  case 'd': //check d
    d_v = get(t, d);
    if (c_v < d_v) { exch = 'd'; }
    break;
}

// do exchange
switch(exch) {
  case 'd':
    exchange(t, getIndex(t, c), getIndex(t, d));
    pos = d;
    break;
  case 'r':
    exchange(t, getIndex(t, c), getIndex(t, r));
    pos = r;
    break;
  default:
    break; // no exchange
}

// if exchanged, recursively call bubbleDown on the two exchanged positions
if (0 == cellsEqual(t, pos, c)) {
  bubbleDown(t, pos);
  bubbleDown(t, c); }
```

Runtime analysis of `extractMax()`:

- Finding max happens in constant time (first item in the array).
- `bubbleDown()` from any given cell can occur a maximum of $m-1$ times towards the right, and $n-1$ times downwards.
- Therefore worst-case runtime of extractMax() is $O(m+n)$.

(d) Again, C implementation of insertion:

```
void insert(tableau *t, int value) {
  int bottomRight = t->elements[t->m*t->n-1];
  // always put at bottom-right most position and bubble up
  if(bottomRight != -INT_MAX) {
  // tableau is full
    printf("ERROR: tableau full, cannot insert\n");
  }
  else {
    t->elements[t->m*t->n-1] = value;
    cell bottomRightCell = cellAt(t->m, t->n);
    bubbleUp(t, bottomRightCell);
  }
}

// bubbleUp() code is the mirror image of bubbleDown() in the opposite directions
    (up and to the left) - won't include again here
```

Runtime analysis of `insert()`:

- Checking bottom right for non-fullness and adding an element happens in constant time.
- `bubbleUp()` code is equivalent to `bubbleDown()` code in the opposite direction. Any given cell can bubble up a maximum of $m-1$ times towards the left, and $n-1$ times upwards.
- Therefore worst-case runtime of insert() is $O(m+n)$.

(e) C implementation of search:

```
int search(tableau *t, cell c, int value) {
  // start top right
  // if too big, everything to its left is also too big, so check the next row
  // if too small, everything below it is by def too small, so move left
  // repeat until found or have eliminated entire tableau
  cell current = c;
  int cur_val = get(t, c);

  if (cur_val == value) {
    int index = getIndex(t, current);
    printf("found at getIndex() = %d\n", index);
    return index;
  } else if (cur_val > value) {
    // go down a row
    current.j += 1;
  } else {
    // go left a column
    current.i -=1;
  }
  // as long as new cell is valid, recurse on search
  if (isIn(t, current)) {
    return search(t, current, value);
  }
  // if all of the above fails to return anything
  return -1;
}

int searchFromStart(tableau *t, int value) {
  cell startCell = cellAt(t->m, 1);
  return search(t, startCell, value);
}
```

Runtime analysis of `search()`:

- Making comparisons of current value against search value occur in constant time.
- `search()` recurs every time we move down or move left and still find a valid cell inside the tableau. By definition if we start from the top right cell, this can only happen (m-1) + (n-1) times.
- Therefore worst-case runtime of search() is $O(m+n)$.

(f) This is pretty straightforward: we run `extractMax()` over and over and output the values in order until the Young tableau is empty. This outputs a reverse sorted list; we can flip it if needed by reading this list backwards (which will add $O(n^2)$ steps).

```
void sort(tableau * t) {
int dim = (t->m) * (t->n);
for (int i = 0; i < dim; i++) {
  printf("%d ",extractMax(t)); // or add this to another array etc
  }
}
```

Runtime analysis of `sort()`:

- Per question c, each call to `extractMax()` runs in $O(m+n)$, or $O(2n)$ for $m = n$.
- We run `extractMax()` once for each item in the tableau, which is $n \times m = n^2$ for $n = m$.
- Therefore worst-case runtime of sort() is $O(n \times n^2) = O(n^3)$.

6. The solution is basically to use an in-order graph walk to count the number of items and push the in-order values into a stack. We then return item $n/2$ from that stack.

```
inOrderWalkCount(Node x, int counter, Stack stack) {
  if(x != NULL ) {

  // initially called with x = root
  Node left = x.left;
  Node right = x.right;

  inOrderWalkCount(left, counter, stack);
  stack.push(x.value);
  counter += 1;
  inOrderWalkCount(right, counter, stack);
  }
}

int findMedian(Node x) {
  // returns median of tree rooted at x
  // so call this on root
  int counter = 0;
  Stack stack = new Stack; // stack of ints

  inOrderWalkCount(x, counter, stack);
  int medianIndex;
  if (counter % 2 == 0) { // even
    medianIndex = counter / 2;
  }
  else {
    medianIndex = (counter + 1 / 2); // use ceiling, per assignment definition
  }

  // pop items from stack until just before the median
  for (int i = 1; i < medianIndex; i++) {
    stack.pop();
  }

  return stack.pop();
```

Runtime analysis of `findMedian()`:

- `inOrderWalkCount()` runs in $O(n)$ time (since we simply traverse every item in the tree).
- `stack.pop()` runs $n/2$ times.
- Therefore worst-case runtime of `inOrderWalkCount()` is $O(n + n/2) = O(n)$.