Midterm Review

Topics For Midterm

Topics	Reading
Introduction	1.1-1.3
Induction and Loop invariants	1.4-1.7, GT 1.3
Elementary Algorithmics	Chapter 2
Asymptotic Notation	Chapter 3
Algorithm Analysis - Analyzing control structures - Worst-case and Average-case - Amortized analysis	4.1-4.6
Solving Recurrences	4.7
Heap and Heap Sort	5.1-5.7
Binomial Heaps	5.8
Binary search tree, Splay Trees	GT 3.1, 3.4
Disjoint Set	5.9

Induction Proof

- Mastering
  - First and second principles of induction
  - Given a mathematical equation, know how to prove it by induction
    - Example: prove by induction that $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$
- Exposure
  - Constructive induction

Invariant

To prove some statement S about a loop is correct, define in terms of a series of smaller statement $S_0, S_1 \ldots, S_k$ where:

- The initial claim, S_0, is true before the loop begins.
- If S_{i-1} is true before iteration i begins, then one can show that S_i will be true after iteration i is over or at the beginning of loop $i+1$.
- The final statement, S_k, implies the statement S that we wish to justify as being true.

This is essentially an induction proof. The proof is for a loop iterating from 1 to k. It's trivial to expand this argument to other loop bounds.

Loop Invariant: Example

- Prove the following loop find the max(a[0], …, a[n-1])

```
int max(int a[n])
{
   int max = a[0];
   int i;

   for (i=1; i<=n-1; i++)
     if (max < a[i])
         max = a[i];

    return max;
}
```

Elementary Algorithmics

- Given a problem
  - What's an instance
  - Instance size
- What does efficiency mean?
  - Time

Average and worst-case analysis

- How to compare two algorithms
  - Worst case, average, best-case
- Worst case
  - Appropriate for an algorithm whose response time is critical
- Average
  - For an algorithm which is to be used many times on many different instances
  - Harder to analyze, need to know the distribution of the instances
- Best case

Elementary Operation

- An elementary operation is one whose execution time can be bounded above by a constant depending only on the particular implementation—the machine, the programming language, etc.

- Example
  - X = Sum{A[i] | 1 <= i <= n}
  - Fibonacci sequence, addition may not be an elementary operation

Asymptotic Notation

- What does "the order of" mean
- Big O, Ω, and Θ notations
- Properties of asymptotic notation
- Limit rule
- Duality rule
- Smooth and b-smooth

Asymptotic notations

- Know the definitions of big O, Ω, and Θ notations
  - Example: what does $O(n^2)$ mean?
- Know how to prove whether a function is in big O, Ω, or Θ based on definition
  - Example
    - Prove that if $f(n) \in O(g(n))$ then $g(n) \in \Omega(f(n))$

Maximum, Duality and Limit rules

- Know to prove asymptotic relationship using the rules
  - Example
    - Show that $O((n+1)^2) = O(n^2)$

The Maximum rule

- Let $f, g : N \to R^{\geq 0}$,

  then $O(f(n) + g(n)) = O(\max(f(n), g(n)))$

- Examples
  - $O(12n^3\text{-}5n\text{+}n\log n\text{+}36) = O(n^3)$
- The maximum rule let us ignore lower-order terms

The Duality Rule

$$t(n) \in \Omega(f(n))$$

iff

$$f(n) \in O(t(n))$$

Example: $\sqrt{n} \in \Omega(\log n)$

We can apply, similarly, the limit rule, the maximum rule, and the threshold rule for Ω using the duality rule

The Limit Rule

- Let $f, g : N \to R^{\geq 0}$, then

1. If $\lim_{n\to\infty} \frac{f(n)}{g(n)} \in R^+$ then $f(n) \in O(g(n))$ and $g(n) \in O(f(n))$
2. If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ then $f(n) \in O(g(n))$ and $g(n) \notin O(f(n))$
3. If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty$ then $f(n) \notin O(g(n))$ and $g(n) \in O(f(n))$

Proof: use the definition of limit and big-O

The Limit Rule

- Let $f, g : N \to R^{\geq 0}$, then

1. If $\lim_{n\to\infty} \frac{f(n)}{g(n)} \in R^+$ then $f(n) \in \Theta(g(n))$
2. If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ then $f(n) \in O(g(n))$ but $f(n) \notin \Theta(g(n))$
3. If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty$ then $f(n) \in \Omega(g(n))$ but $f(n) \notin \Theta(g(n))$

Semantics of big-O and Ω

- When we say an algorithm takes worst-case time *t(n)* ∈ O(*f(n)*), then there exist a real constant *c* such that *c*f(n)* is an upper bound for any instances of size of sufficiently large *n*
- When we say an algorithm takes worst-case time *t(n)* ∈ Ω(*f(n)*), then there exist a real constant *d* such that there exists at least one instance of size *n* whose execution time >= *d*f(n)*, for any sufficiently large *n*
- Example
  - Is it possible an algorithm takes worst-case time O(*n*) and Ω(*nlog n*)?

Practice Problems

```
anAlgorithm( int n)
{
  // if (x) is an elementary
  // operation
  if (x) {
    some work done
    by n^2 elementary
    operations;
  } else {
    some work done
    by n^3 elementary
    operations;
  }
}
```

- True or false
  - The algorithm takes time in $O(n^2)$ F
  - The algorithm takes time in $\Omega(n^2)$ T
  - The algorithm takes time in $O(n^3)$ T
  - The algorithm takes time in $\Omega(n^3)$ F
  - The algorithm takes time in $\Theta(n^3)$ F
  - The algorithm takes time in $\Theta(n^2)$ F
  - The algorithm takes worst case time in $O(n^3)$ T
  - The algorithm takes worst case time in $\Omega(n^3)$ T
  - The algorithm takes worst case time in $\Theta(n^3)$ T
  - The algorithm takes best case time in $\Omega(n^3)$ F

Smooth

- Know the definition of smooth and how to prove if a function is smooth or not
  - Example: what does b-smooth mean?
  - Prove that n^2 is smooth

- A function $f : N \to R^{\geq 0}$ is *eventually nondecreasing* if there exists an integer threshold n_0 such that $f(n) \leq f(n+1)$ for all $n \geq n_0$
- Function f is *b-smooth* (b is an integer >1) if it is eventually nondecreasing and it satisfies condition $f(bn) \in O(f(n))$
- A function is *smooth* if it is b-smooth for every integer b>=2
- **Theorem**: If a function is b-smooth for any b>=2, it is smooth

Exposure: Smoothness rule

- Let $f : N \to R^{\geq 0}$ be a smooth function and let $t : N \to R^{\geq 0}$ be an eventually nondecreasing function. Then $t(n) \in \Theta(f(n))$ whenever $t(n) \in \Theta(f(n) \mid n \text{ is power of } b)$

- The rule holds for O and Ω

Analysis of Algorithms

- Mastering
  - Analyzing control structures
    - Sequencing
    - For loops
    - While and repeat loops
    - Recursive calls
  - Finding and using a barometer
  - Average case analysis

- Exposure
  - Amortized analysis

Control structures: sequences

- P is an algorithm that consists of two fragments, P1 and P2

```
P
{
   P1;
   P2;
}
```

- P1 takes time t1 and P2 takes times t2
- The sequencing rule asserts P takes time $t = t1 + t2 \in \Theta(\max(t1,t2))$.

For loops

```
for (i=0; i<m; i++) {
   P(i);
}
```

- Case 1: P(i) takes time *t* independent of i and n, then the loop takes time *O(mt)* if m>0.
- Case 2: P(i) takes time *t(i)*, the loop takes time $\sum_{i=0}^{m-1} t(i)$

Example: analyzing the following nests

```
for (i=0; i<n; i++) {
   for (j=0; j<n; j++)
      constant work
}
```

```
for (i=1; i<n; i++) {
   for (j=0; j<i; j++)
      constant work
}
```

```
for (i=1; i<n; i++) {
   for (j=0; j<i*i; j++)
      constant work
}
```

```
for (i=1; i<n; i++) {
   for (j=0; j<i; j++)
      constant work

   for (k=0; k<i*i; k++)
      constant work
}
```

"while" and "repeat" loops

- The bounds may not be explicit as in the for loops
- Careful about the inner loops
  - Is it a function of the variables in outer loops?
- Analyze the following two algorithms

```
int example1(int n)
{
  while (n>0) {
      work in constant;
      n = n/3;
  }
}
```

```
int example2(int n)
{
  while (n>0) {
     for (i=0; i<n; i++) {
        work in constant;
     }
     n = n/3;
  }
}
```

Recursive calls

Typically we can come out a recurrence equation to mimics the control flow.

```
double fibRecursive(int n)
{
  double ret;
  if (n<2)
    ret = (double)n;
  else
    ret = fibRecursive(n-1)+fibRecursive(n-2);
  return ret;
}
```

$$T(n) = \begin{cases} a & \text{if } n = 0 \text{ or } 1 \\ T(n-1)+T(n-2)+h(n) & \text{otherwise} \end{cases}$$

Using a Barometer

- A ***barometer*** instruction is one that is executed at least as often as any other instruction in the algorithm
- We can then count the number of times that the barometer instruction get executed
  - Provided that the time taken by each instruction is bounded by a constant, the time taken by the entire algorithm is in the exact order of the number of times the barometer instruction is executed

Average Case Analysis

- We need to know instance distribution
  - Given the instance distribution, know how to calculate the average cost

$$\text{average cost} = \sum_{i=1}^{m} p_i c_i$$

- Sometimes we make ideal assumption that all instances of any given sizes are equally distributed

Solving Recurrence

- Know how to solve a homogeneous or inhomogeneous recurrence
- Know how to use the simplified version of the Master theorem

Heaps

- Know the definition
  - What is the heap property?
- Given a node, know how to calculate its parent and children
- Know how percolate, sift-down, make heap, and heap sort work
  - Can write and analyze these algorithms
  - Given an example heap, demonstrate how these algorithms work
  - Design a new similar heap related algorithm

Some important properties of heaps

- Given a node *T[i]*
  - It's parent is *T[i/2]*, if *i>1*.
  - It's left child is *T[2*i]*, if *2*i<=n*.
  - It's right child is *T[2*i+1]*, if *2*i+1<=n*.
- The height of a heap containing n nodes is $\lfloor \lg n \rfloor$

Heap Algorithms and Efficiency

```
Class Heap {
  int T[];
  int n;

  public void alterHeap(int i, int v); // O(lg n)
  public void siftDown(int i);         // O(lg n)
  public void percolate(int i);        // O(lg n)
  public int findMax();                // Θ(1)
  public int deleteMax();              // O(lg n)
  public void insert(int v);           // O(lg n)
  public void makeHeap();              // O(n)
  public void heapSort();              // O(nlogn)
}
```

Cost analysis for makeHeap() not required.

Binomial Heaps

- Know the definition of Binomial Trees and Binomial Heaps
- Understand the following algorithms

  (Can write and analyze these algorithms.
  Given an example binomial heap, demonstrate how these algorithms work.
  Design a new similar binomial heap related algorithm)
  - Merge two equal size binomial trees
  - Merge two binomial heaps
  - findMax()
  - deleteMax()
  - Insert()

Merge two equal size binomial trees

```
BinomialTree mergeBinomialTrees(B1, B2){
    // B1, B2 are the same size
    if (B1.root().key > B2.root().key) {
        B.copy(B1);
        B.setChild(B1.rank(), B2);
        B.setRank(B1.rank()+1);
    } else {
        // link in the other way
        …
    }
}
```

It takes a time in O(1).

Merge two binomial heaps

```
mergeBinomialHeaps(H1, H2)
{
   while (simultaneously following the links in H1 and H2) {
      if there are three rank i trees{
         merge two of them and set it as carry-on;
         add the remainder to H;
      } else if there are two rank i trees {
         merge the two trees;
         set it as carry on;
      } else if there is one rank i tree{
         add it to H;
      }
   }
   add the carry-on if exists to H.
}
```

Assume the result binomial heap contains n nodes. The construction can be done in $\lfloor \lg n \rfloor + 1$ stages. Time in O(log n)

findMax()

- Return the node pointed by the *max* pointer.

deleteMax()

```
deleteMax(H)
{
    take the max binomial tree B out (H/B);
    remove the root of B;
    join the subtrees into a new binomial heap H2;
    merge H/B and H2;
}
```

Cost: O(log n)

insert

```
insert(v, H)
{
   make a 1 node binomial tree B0;
   Build a binomial heap H0 that contains B0;
   merge H0 and H;
}
```

Binary search tree

- Know the definition
- Know how search(), insert(), delete() work

Binary search tree

- Definition:
  - A binary tree,
  - Where each internal node *v* stores an element *e*
  - The left subtree of *v* are <= e
  - The right subtree of *v* are >=e
- Assume all external nodes are empty
- The in-order traversal of binary search tree visits elements in non-decreasing order

Search A Binary Search Tree

```
Node binaryTreeSearch(Key k, Node v)
// Parameters: k, key to search
//                    v, the root of the subtree to search
// return a node when found match key
// otherwise, return an external node
{
   if (v is an external node)
      return v;
   if (k == key(v))
      return v;
   else if (k < key(v))
      binaryTreeSearch(k, v.leftChild());
   else
      binaryTreeSearch(k, v.rightChild());
}
```

Cost? Best case? Worst Case?

Insertion in a Binary Search Tree

- To insert element *e* with key *k*.
- Let *w* be the node returned by binaryTreeSearch()
  1. If *w* is an external node, replace it by an internal node with the key *k* and element *e*.
  2. If *w* is an internal node, continue to search its right subtree (or left subtree) until find an external node. Then apply case 1.

Removal in a Binary Search Tree

- To remove a node with key *k*, Let *w* be the node returned by binaryTreeSearch(k, root)
  1. If *w* is an external node, done!
  2. If *w* is an internal node
     a) One of *w*'s children is an external node, *z*. Remove *w* and *z*, and replace *w* by *z*'s sibling
     b) Both children of node *w* are internal nodes
        - Find internal node *y* that follows *w* in an inorder traversal
        - Replace *w*'s content by *y*'s.
        - Remove *y* using case (a).

Splay Trees

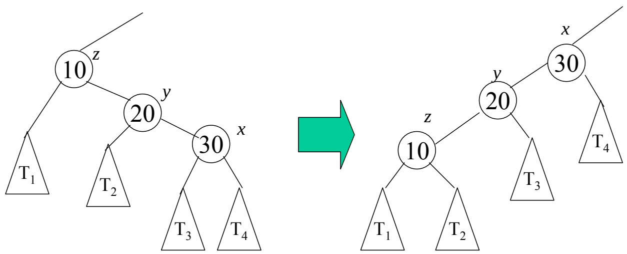
- Know the splaying steps after insertion, deletion and search

Splay Trees

- Apply *splaying* after every access to keep the search tree balanced in an amortized sense
- Splaying
  - Splay *x* by moving *x* to the root through a sequence of restructurings
  - One specific operation depends on the relative positions of *x*, its parent *y*, and its grandparent *z*
    - Zig-Zig
    - Zig-Zag
    - Zig

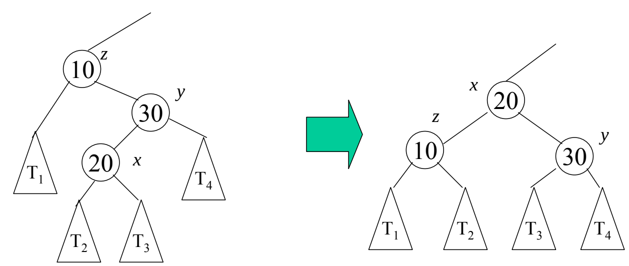
Splay *x*: zig-zig

The node *x* and its parent *y* are both left or right children



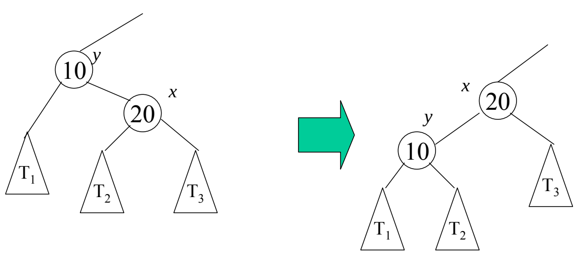
Splay *x*: zig-zag

One of *x* and *y* is a left child and the other is a right child.



Splay *x*: zig

The node x does not have a grandparent (or the grandparent is not of our concern)



When to Splay

- When searching for key *k*, splay the found internal node or the parent of the external node when search fails
- When inserting a key *k*, splay the newly created internal node
- When deleting a key *k*, splay the parent of the node that gets removed (See slide: Removal in a Binary Search Tree).

Properties of Splay Trees

- Linear depth when inserting keys in increasing order
  - What's the worst case cost for search, insertion, and deletion respectively?
- Consider a sequence of m operations on a splay tree, each a search, insertion, or deletion, starting from an empty tree with zero keys, also let n_i be the number of keys in the tree after operation i, and n be the total number of insertions. The total running time for performing the sequence of operations is

$$O(m + \sum_{i=1}^{m} \log n_i) = O(m \log n)$$

Properties of Splay Trees

- Consider a sequence of m operations on a splay tree, each a search, insertion, or deletion, starting from an empty tree with zero keys, also let *f(i)* be the number of times the item i is accessed in the splay tree, that is, its *frequency*, and let n be total number of items. Assuming that each item is accessed at least once, then the total running time for performing the sequence of operations is

$$O(m + \sum_{i=1}^{m} f(i) \log(m / f(i)))$$

Disjoint set structures

- Know the definition
- Given set[], know how to draw the sets in trees
- Know how the following algorithms work
  - find1() and merge1()
  - find2() and merge2()
  - find3() and merge3()

Representation 1: $\Theta(n^2)$

- Use the smallest member of each set as label
- Declare an array *set[1..n]* where *set[i]* is the label of object *i*.

```
find1(x)
{
   return set[x];
}
```

$\Theta(1)$

```
Merge1(a,b)
{
   i = min(a,b);
   j = max(a,b);
   for (k=1; k<=N; k++) {
      if (set[k] == j)
         set[k] = i;
   }
}
```

$\Theta(N)$

Rooted tree: $\Theta(n^2)$

```
find2(x)
{
  r = x;
  while (set[r] != r)
    r = set[r];
  return r;
}
```

$\Theta(N)$ in worst case

```
merge2(a, b)
{
   if (a<b)
     set[b] = a;
   else
     set[a] = b;
}
```

$\Theta(1)$

A new merge algorithm

```
find2(x)
{
  r = x;
  while (set[r] != r)
    r = set[r];
  return r;
}
```

$\Theta(\log N)$ in worst case

```
merge3(a,b)
{
   if (height(a) == height(b)) {
     height(a) = height(a) + 1;
     set[b] = a;
  } else if (height(a) < height(b))
     set[a] = b;
  else
     set[b] = a;
}
```

$\Theta(1)$

Total operations: $\Theta(N + n\log N)$

A further improvement

- Squash the path when doing find(), so the next find() will be likely quicker (path compression).
  - first pass to find the root
  - second pass change the pointers along the path to the root and make them all point to the root

```
find3(x)
{
    r = x;
    while (set[r] <> r)
       r = set[r];

    i = x;
    while (i <> r) {
       j = set[i];
      set[i] = r;
       i = j;
    }
    return r;
}
```

Cost analysis not required