Exam 2 Review

Topics for Exam 2

Topics	Reading
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
Greedy Algorithms - Coin Change - Minimum Spanning Tree - Dijkstra's algorithm - Knapsack - Scheduling	6
Divide-and-Conquer - Mergesort and quicksort - Median - Closest pair	7

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;
}
```

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];
}
```

Θ(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;
   }
}
```

Θ(N)

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

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

Θ(N) in worst case

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

Θ(1)

A new merge algorithm

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

Θ(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;
}
```

Θ(1)

Total operations: Θ(N + nlog 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

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

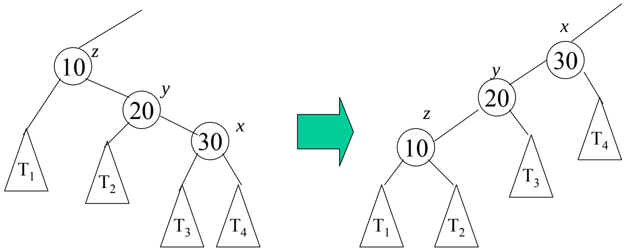
- Know the splaying steps after insertion, deletion and search

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

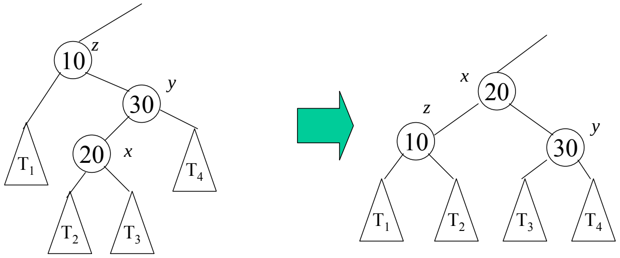
Splay *x*: zig-zig

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



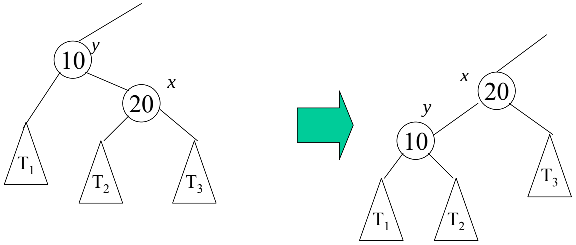
Splay *x*: zig-zag

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

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

Greedy algorithms

- Know the paradigm
  - Template of a greedy algorithm
  - Be able to design a greedy algorithm
- Understand the following algorithms
  - Coin change
  - MST (Prim's algorithm and Kruscal's algorithm)
  - Dijkstra's algorithm (single source shortest path)
  - Knapsack
  - Scheduling for shortest total response time

General characteristics of greedy algorithms

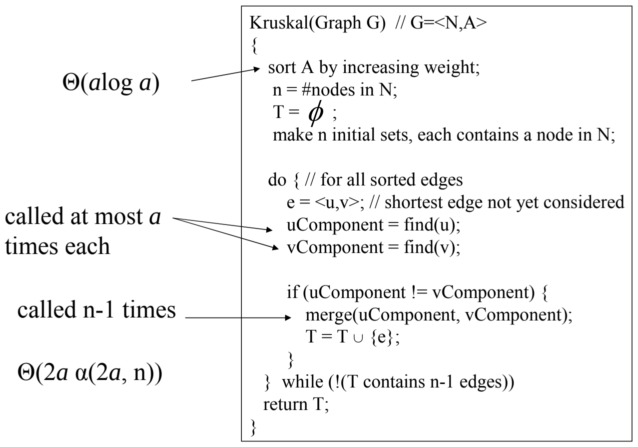
```
makeChange(int n)
{
  C = a set of available coins;
  S=∅;  // chosen coins
  s=0;
  while (C != ∅ && s != n) {
    x = a coin in C with the largest value
        such that s+x<=n;
    if no such x exists
          return “no solution found”;
    else {
       C = C \ {x};
       S = S⋃{x};
       s = s+x;
    }
  }
}
```

```
greedy(SET C)
{
  // C is set of candidates
  S= ∅;  // S is a partial solution

  while (C != ∅) {
    x = select(C);
    if  (feasible(S⋃{x}))
        S = S  ⋃  {x};
    if (solution(S))
       return S;
  }
  return “no solutions”;
}
```

objective() function is implicit

Kruskal's algorithm -- efficiency



$\Theta(a\log a)$

called at most *a* times each

called n-1 times

$\Theta(2a\ \alpha(2a, n))$

```
Kruskal(Graph G)  // G=<N,A>
{
  sort A by increasing weight;
   n = #nodes in N;
   T = ϕ ;
   make n initial sets, each contains a node in N;

  do { // for all sorted edges
     e = <u,v>; // shortest edge not yet considered
     uComponent = find(u);
     vComponent = find(v);

     if (uComponent != vComponent) {
        merge(uComponent, vComponent);
        T = T ∪ {e};
     }
  }  while (!(T contains n-1 edges))
  return T;
}
```

Prim's algorithm

```
Prim(int L[][])
{
   T = ∅;
   for (i=2; i<=n; i++) {
      nearest[i] = 1;                    Θ(n)
      mindist[i] = L[i,1];
   }

   for (i=2; i<n; i++) {                 Θ(n^2)
      min = ∞;
      for (j=2; j<=n; j++) {
         if  (mindist[j] >=0 && mindist < min) {
            min = mindist[j];
             k = j;
         }
      }
```

```
      T = T ∪ <nearest[k], k>;
      mindist[k] = -1;

      for (j=2; j<=n; j++) {
         if (L[j,k] < mindist[j]) {
            mindist[j] =L[j,k];
            nearest[j] = k;
         }
      }
   }
}
```

Dijkstra's algorithm

C: candidate set
S : partial solution set
L[i][j]: weight of edge <i,j>
D[i]: length of the special path for the source to node i.
P[i]: the previous node of i along its shortest path.

```
Dijkstra(Weight L[][])
{
   /* initialization */
   C = {i | 2 <= i <= n};   // S = {1}
   for (i=2; i<=n; i++) {
      D[i] = L[1,i];
      P[i] = 1;
   }

   for (i=1; i<=n-2; i++) { // repeat n-2 times
      v = some element of C minimizing D[v];
      C = C – {v};  // S = S ∪ {v}
      for (each w ) {
         if ( D[v]+L[v,w]  < D[w]) {
            D[w] = D[v]+L[v,w];
            P[w] = v;
         }
      }
   }
}
```

Analysis of Dijkstra's algorithm: using heap

a: #edges
n: #nodes

```
                              Dijkstra(Weight L[][])
                              {
                                 /* initialization */
                                 C = {i | 2 <= i <= n};   // S = {1}
Θ(n)                             for (i=2; i<=n; i++) {
                                    D[i] = L[1,i];
                                    P[i] = 1;
                                 }
Θ(n)                             buildHeap(D); // inverted heap

                                 for (i=1; i<=n-2; i++) { // repeat n-2 times
Θ(1)                                v = findMin(D[]);
O(log n)                            deleteMin(v);         //C = C – {v}; S = S ∪ {v}
At most a times if use              for (each w ) {
adjacent list                          if ( D[v]+L[v,w]  < D[w]) {
                                          D[w] = D[v]+L[v,w];
O(log n)                                  percolate(v);
                                          P[w] = v;
                                    }
Total: O((a+n) log n)            }
       =O(a log n)            }
```

The knapsack problem

- Given
  - n objects numbered from 1 to n. Object i has a positive weight w_i and a positive value v_i
  - a knapsack that can carry a weight not exceeding W
- Problem
  - Fill the knapsack in a way that maximize the value of the included objects, while respecting the capacity constraints
  - In this version, we assume that the objects can be broken into small pieces

A greedy algorithm

```
Knapsack(w[], v[], W)
{
   for (i=1; i<=n; i++)
      x[i] =0;
   weight = 0;

   while (weight < W) {
      i = select the best remaining object;
      if (weight + w[i] < W)
         x[i] = 1;
      else
         x[i] = (W-weight)/w[i];
   }
   return x;
}
```

The key is which object to select

Scheduling

- Minimizing time in the system
  - Know the problem
  - Know the proof
  - Know the algorithm
- Scheduling with deadlines (not required)

Minimizing time in the system

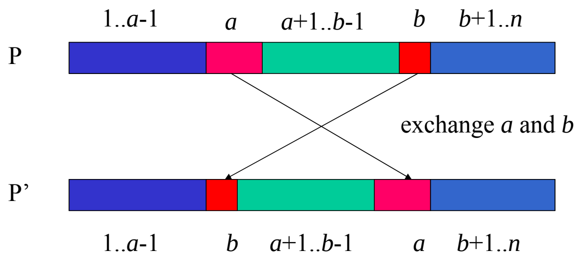
- Assume we submit *n* jobs into a system at the same time.
  - The service time of job *i*, t_i, know in advance
- Problem
  - Design an algorithm that minimizing the average response time
  - This is equivalent to
    - Minimizing the total response time
    - T = $\sum_{i=1}^{n}$ (response time of job j)

A greedy algorithm

- Algorithm
  1. Sort the jobs by their service times
  2. Repeat
     1. Serve the job with minimal service time among the remaining jobs
- Analysis
  - Step 1: O(n log n)
  - Step 2: Θ(n)
  - Total: O(n log n)

Optimality of the greedy scheduling algorithm

- Theorem 6.6.1. The greedy algorithm is optimal



Compares schedules P and P’, job *a* at P’ leaves at the same time as job *b* in P. Jobs *b* and *a*+1 to *b*-1 in P’ leaves earlier then the corresponding jobs in P.

Optimality of the greedy scheduling algorithm

$$\begin{aligned} T(P) &= s_1 + (s_1 + s_2) + \ldots (s_1 + s_2 + \ldots s_n) \\ &= ns_1 + (n-1)s_2 + \ldots + 1s_n \\ &= \sum_{k=1}^{n} (n-k+1)s_k \end{aligned}$$

$$T(P) = (n-a+1)s_a + (n-b+1)s_b + \sum_{k=1,k\neq a,b}^{n} (n-k+1)s_k$$

$$T(P') = (n-a+1)s_b + (n-b+1)s_a + \sum_{k=1,k\neq a,b}^{n} (n-k+1)s_k$$

$$T(P) - T(P') = (n-a+1)(s_a - s_b) + (n-b+1)(s_b - s_a) = (b-a)(s_a - s_b) > 0$$

Divide and Conquer

- Given a problem, know how to design a D&C algorithm
- Know how to analyze a D&C algorithm
  - You need to remember the simple version of the Master Theorem.
- Know the following algorithms
  - Merge sort
  - Quick sort
  - Find median
  - Closest pair

Running-time analysis

- Assume that the *l* sub-instances have roughly the same size n/*b* for some constant *b*
- Let g(n) be the time required by DC on instances of size n, excluding the times need for the recursive calls. We have
  - $t(n) = l \cdot t(n/b) + g(n)$
- If $g(n) \in \Theta(n^k)$ for an integer k, we have

$$t(n) \in \begin{cases} \Theta(n^k) & \text{if } l < b^k \\ \Theta(n^k \log n) & \text{if } l = b^k \\ \Theta(n^{\log_b l}) & \text{if } l > b^k \end{cases}$$

Merge sort

```
mergeSort(int T[n])
{
   if (n is sufficiently small)
      insertionSort(T);
   else {
      int U[n/2], V[(n+1)/2];
      copy T[1..n/2] to U[1..n/2];
      copy T[n/2+1,..,n] to V[1.., (n+1)/2];
      mergeSort(U[n/2]);
      mergeSort(V[(n+1)/2);
      merge(U,V,T);
   }
}
```

Merge two sorted arrays

```
Merge(U[m], V[n], T[m+n])
// merge sorted arrays U and V into T
{
   u = 0;   // cursor for U
   v = 0;   // cursor for V
   U[m] = V[n] = +∞; // sentinels
   for (t=0; t<m+n; t++) {  // t is cursor for T
      if (U[u] < V[v]) {
          T[t] = U[u];
          u++;
      } else {
         T[t] = V[v];
         v++;
      }
   }
}
```

What to do if we do not use the two sentinels?

Quick Sort

- Choose an element from the array to be sorted as a pivot
- Partition the array on either side of the pivot such that those no smaller than the pivot are to its right and those no greater are to its left
- Recursive calls on both sides

The algorithm

```
quickSort(T, i, j)
{
   l = pivot(T, i, j); // l is the position of the pivot at end
   if (i<l-1)
      quickSort(T, i, l-1);
   if  (l+1<j)
      quickSort(T, l+1, j);
}
```

Pivot I

```
int pivot(T, i, j)
{
  // choose T[i] as the pivot
  p = T[i];
  l = i;   //  left cursor
  r = j+1;   // right cursor
  do {
    l++;
  } while (T[l]<=p and l < r)

  do {
    r--;
  } while (T[r] > p);
```

```
  while (l<r) {
     swap(T[l], T[r]);
     do {
        l++;
     } while (T[l] <= p);
      do {
        r--;
      } while (T[r] >  p)
   }
   swap(T[i], T[r]);
   return r;
}
```

T[i] is at the bound after the algorithm

Analysis

- Worst case: the array is sorted, $\Omega(n^2)$
- Best case
  - $T(n) = 2T(n/2) + \Theta(n)$, $T(n) \in \Theta(n \log n)$
- Average case (not required)

Selection using pseudomedian

```
selection(T[1..n], s)
{
   l = 1; r = n;
   while (true) {
      x = pseudomedian(T[l..r]);
      p = pivot(T[l..r], x);
      if (s<p) r = p-1;
      else if (s>p) l = p+1;
      else return p;
   }
}
```

```
pseudomedian(T[1..n])
{
   if (n <= 5)
     return adhocmedian(T);
   z = ⌊n/5⌋;
   for (i=1; i<=z; i++)
      Z[i] = adhocmedian(T[5i-4..5i]);
    return selection(Z, ⌈z/2⌉);
}
```

We assume the elements are distinct.
You need to know the time complexity of this algorithm

Closest Pair

- Problem
  - Given n points on a two-dimension space, find the closest pair
- A simple algorithm
  - Calculate the distance for all possible pairs, find a smallest one
    - Total $\binom{n}{2}$ pairs
    - Cost: $\Theta(n^2)$
- A better algorithm
  - Divide-and-conquer

Algorithm

```
double closestPair(Points p)
{
   n = p.size();
   mergeSort(p);  // by x-coordinate
   return recursiveClosestPair(p, 1, n)
}
```

```
double recursiveClosestPair(p, i, j)
{
   if ( j-i < 3) {
     return adhocClosest(p, i, j);
     sort p[i..j] by y-coordinate;
   }

   k = (i+j)/2;
   deltaL = recursiveClosestPair(p, i, k);
   deltaR = recursiveClosestPair(p, k+1, j);
   delta = min(deltaL, deltaR);
   return findClosestInStrip(p, i, j, delta);
}
```

Note: p[i..j] are sorted by x-coordinate before getting in recursiveCloestPair();
sorted by y-coordinate after it returns;

---

findClosestInStrip

m = j-i+1

```
findClosestInStrip(p, i, j, delta)
{
   k = (i+j)/2;        l = p[k].x;
   // p[i..k] sorted by y-coordinate
   // p[k+1..j] sorted by y-coordinate
   merge(p, i, k, j);  // p[i..j] sorted by y-coordinate

   t = 0;
   for (k=i; k<=j; k++) {
     if (p[k].x > l – delta
       && p[k].x < l+delta)  // in the strip
        v[++t] = p[k];
   }
   for (k=1; k<t; k++) {
         for (s=k+1; s<=min(t, k+7); s++)
         delta = min(delta, dist(v[k], v[s]));
   }
   return delta
}
```

Cost?

Θ(m)

Θ(m)

O(m)

Total: Θ(m)