CS 303
Design and Analysis of Algorithms

Review For Final Exam
Dong Xu
(Based on class note of David Luebke)

Final Exam
• 8am-10am, Monday, May 10
• Close book
• Bring your calculator
• 40% of your final score
• Office hours during final
  ■ Dong (109 EBW) : 1:30pm-4pm, Friday, May 7
  ■ Ashwin (302 EBN) :
    ○ 11am-1pm, Thursday, May 6
    ○ 11am-3pm, Wednesday, May 12 (an opportunity to verify the grading of final and quiz scores).

Final Exam Tips
• 7 problems (6 with 15 pts, 1 with 10 pts)
• Review your quizzes and homework
• 1 hard problem in dynamic programming
• Coverage
  ■ 12.1, 12.2, 12.3
  ■ 13.1, 13.2, 13.3
  ■ 14.1, 14.2
  ■ 15.1, 15.2, 15.3
  ■ 16.1, 16.2

Review Topics
• Binary search tree
• Red-black tree
• Augmenting data structure
• Dynamic programming
• Greedy algorithm

Review: Binary Search Trees
• BST property:
  key[leaf(x)] ≤ key[x] ≤ key[right(x)]
• Example:

Review: Inorder Tree Walk
• An inorder walk prints the set in sorted order:
  TreeWalk(x)
  TreeWalk(left[x]);
  print(x);
  TreeWalk(right[x]);
• Easy to show by induction on the BST property
Review: BST Search

TreeSearch(x, k)
    if (x = NULL or k = key[x])
        return x;
    if (k < key[x])
        return TreeSearch(left[x], k);
    else
        return TreeSearch(right[x], k);

Review: BST Insert

● Adds an element x to the tree so that the binary search tree property continues to hold
● The basic algorithm
  ▪ Like the search procedure above
  ▪ Insert x in place of NULL
  ▪ Use a “trailing pointer” to keep track of where you came from (like inserting into singly linked list)
  ▪ Like search, takes time $O(h)$, $h$ = tree height

Review: Sorting With BSTs

● Basic algorithm:
  ▪ Insert elements of unsorted array from 1..n
  ▪ Do an inorder tree walk to print in sorted order
● Running time:
  ▪ Best case: $\Omega(n \log n)$ (it’s a comparison sort)
  ▪ Worst case: $O(n^2)$
  ▪ Average case: $O(n \log n)$ (it’s a quick sort!)

Review: More BST Operations

● Minimum:
  ▪ Find leftmost node in tree
● Successor:
  ▪ x has a right subtree: successor is minimum node in right subtree
  ▪ x has no right subtree: successor is first ancestor of x whose left child is also ancestor of x
    ▪ Intuition: As long as you move to the left up the tree, you’re visiting smaller nodes.
● Predecessor: similar to successor

Review: More BST Operations

● Delete:
  ▪ x has no children:
    ○ Remove x
  ▪ x has one child:
    ○ Splice out x
  ▪ x has two children:
    ○ Swap x with successor
    ○ Perform case 1 or 2 to delete it

Example: delete K or H or B
Review: Red-Black Trees

- **Red-black trees:**
  - Binary search trees augmented with node color
  - Operations designed to guarantee that the height \( h = O(lg \ n) \)

Red-Black Properties

- **The red-black properties:**
  1. Every node is either red or black
  2. Every leaf (NULL pointer) is black
     - Note: this means every "real" node has 2 children
  3. If a node is red, both children are black
     - Note: can’t have 2 consecutive reds on a path
  4. Every path from node to descendent leaf contains the same number of black nodes
  5. The root is always black

  - **black-height:** # black nodes on path to leaf
    - Lets us prove RB tree has height \( h \leq 2 lg(n+1) \)

Operations On RB Trees

- Since height is \( O(lg \ n) \), we can show that all BST operations take \( O(lg \ n) \) time
- Problem: BST Insert() and Delete() modify the tree and could destroy red-black properties
- Solution: restructure the tree in \( O(lg \ n) \) time
  - You should understand the basic approach of these operations
  - Key operation: rotation

RB Trees: Rotation

- Our basic operation for changing tree structure:

  - Rotation preserves inorder key ordering
  - Rotation takes \( O(1) \) time (just swaps pointers)

Review: Dynamic Order Statistics

- We’ve seen algorithms for finding the \( i \)th element of an unordered set in \( O(n) \) time
- **OS-Trees:** a structure to support finding the \( i \)th element of a dynamic set in \( O(lg \ n) \) time
  - Support standard dynamic set operations (\( \text{Insert}() \), \( \text{Delete}() \), \( \text{Min}() \), \( \text{Max}() \), \( \text{Succ}() \), \( \text{Pred}() \))
  - Also support these order statistic operations:
    - \text{void \text{OS-Select}}(\text{root}, \text{ i});
    - \text{int \text{OS-Rank}}(\text{x});

Review: Order Statistic Trees

- OS Trees augment red-black trees:
  - Associate a \textit{size} field with each node in the tree
  - \textit{x->size} records the size of subtree rooted at \textit{x}, including \textit{x} itself:
Review: OS-Select

Example: show OS-Select(root, 5):

```
OS-Select(x, i)
{
    r = x->left->size + 1;
    if (i == r)
        return x;
    else if (i < r)
        return OS-Select(x->left, i);
    else
        return OS-Select(x->right, i-r);
}
```

Note: use a sentinel NIL element at the leaves with size = 0 to simplify code, avoid testing for NULL.
Review: Determining The Rank Of An Element

Idea: rank of right child $x$ is one more than its parent’s rank, plus the size of $x$’s left subtree

$\text{OS-Rank}(T, x)$

```c
r = x->left->size + 1;
y = x;
while (y != T->root)
    if (y == y->p->right)
        r = r + y->p->left->size + 1;
y = y->p;
return r;
```

Example 1: find rank of element with key $H$

$\text{OS-Rank}(T, x)$

```c
r = x->left->size + 1;
y = x;
while (y != T->root)
    if (y == y->p->right)
        r = r + y->p->left->size + 1;
y = y->p;
return r;
```

Example 1: find rank of element with key $H$

```c
r = x->left->size + 1;
y = x;
while (y != T->root)
    if (y == y->p->right)
        r = r + y->p->left->size + 1;
y = y->p;
return r;
```

Example 1: find rank of element with key $H$

```c
r = x->left->size + 1;
y = x;
while (y != T->root)
    if (y == y->p->right)
        r = r + y->p->left->size + 1;
y = y->p;
return r;
```

Example 1: find rank of element with key $H$

```c
r = x->left->size + 1;
y = x;
while (y != T->root)
    if (y == y->p->right)
        r = r + y->p->left->size + 1;
y = y->p;
return r;
```

Review: Maintaining Subtree Sizes

- So by keeping subtree sizes, order statistic operations can be done in $O(\log n)$ time
- Next: maintain sizes during Insert() and Delete() operations
  - Insert(): Increment size fields of nodes traversed during search down the tree
  - Delete(): Decrement sizes along a path from the deleted node to the root
  - Both: Update sizes correctly during rotations
Review: Maintaining Subtree Sizes

- Note that rotation invalidates only x and y
- Can recalculate their sizes in constant time
- Thm 15.1: can compute any property in \(O(\log n)\) time that depends only on node, left child, and right child

Review: Dynamic Programming

- Summary of the basic idea:
  - Optimal substructure: optimal solution to problem consists of optimal solutions to subproblems
  - Overlapping subproblems: few subproblems in total, many recurring instances of each
  - Solve bottom-up, building a table of solved subproblems that are used to solve larger ones

Matrix Chain-Products

- Matrix Chain-Product:
  - Compute \(A = A_0 * A_1 * \ldots * A_{n-1}\)
  - \(A_i \) is \(d_i \times d_{i+1}\)
  - Problem: How to parenthesize?
- Matrix Chain-Product Alg.:
  - Try all possible ways to parenthesize \(A = A_0 * A_1 * \ldots * A_{n-1}\)
  - Calculate number of ops for each one
  - Pick the one that is best

A “Recursive” Approach

- Define subproblems:
  - Find the best parenthesization of \(A_0 * A_1 * \ldots * A_k\).
  - Let \(N_{i,j}\) denote the number of operations done by this subproblem.
  - The optimal solution for the whole problem is \(N_{0,n-1}\).

- Subproblem optimality: The optimal solution can be defined in terms of optimal subproblems
  - Assume the final multiply is at index \(i\): \((A_0 * \ldots * A_i) * (A_{i+1} * \ldots * A_{n-1})\).
  - Then the optimal solution \(N_{i,n-1}\) is the sum of two optimal subproblems, \(N_{i,j}\) and \(N_{i+1,n-1}\) plus the time for the last multiply.

A Characterizing Equation

- Let us consider all possible places for that final multiply:
  - Recall that \(A_i\) is a \(d_i \times d_{i+1}\) dimensional matrix.
  - So, a characterizing equation for \(N_{i,j}\) is the following:

\[
N_{i,j} = \min_{i \leq k < j} \left( N_{i,k} + N_{k+1,j} + d_i d_k d_{j+1} \right)
\]

- Note that subproblems are not independent—the subproblems overlap.
Greedy Algorithms

- A greedy algorithm always makes the choice that looks best at the moment
- Indicators:
  - Optimal substructure
  - Greedy choice property: a locally optimal choice leads to a globally optimal solution
- Dynamic programming can be overkill; greedy algorithms tend to be easier to code

Activity-Selection

- Formally:
  - Given a set \( S \) of \( n \) activities
  - \( s_i \) = start time of activity \( i \)
  - \( f_i \) = finish time of activity \( i \)
  - Find max-size subset \( A \) of compatible activities

Activity Selection: A Greedy Algorithm

- So actual algorithm is simple:
  - Sort the activities by finish time
  - Schedule the first activity
  - Then schedule the next activity in sorted list which starts after previous activity finishes
  - Repeat until no more activities
- Intuition is even more simple:
  - Always pick the activity with the nearest finish time available and reject the conflicts

Review: The Knapsack Problem

- More formally, the 0-1 knapsack problem:
  - The thief must choose among \( n \) items, where the \( i \)th item worth \( v_i \) dollars and weighs \( w_i \) pounds
  - Carrying at most \( W \) pounds, maximize value
    - Note: assume \( v_i, w_i \), and \( W \) are all integers
    - "0-1" b/c each item must be taken or left in entirety
- A variation, the fractional knapsack problem:
  - Thieves can take fractions of items
  - Think of items in 0-1 problem as gold ingots, in fractional problem as buckets of gold dust

Solving The Knapsack Problem

- The optimal solution to the fractional knapsack problem can be found with a greedy algorithm
  - How?
  - The optimal solution to the 0-1 problem cannot be found with the same greedy strategy
    - Greedy strategy: take in order of dollars/pound
    - Example: 3 items weighing 10, 20, and 30 pounds, knapsack can hold 50 pounds

  - Suppose item 2 is worth $100. Assign values to the other items so that the greedy strategy will fail
0-1 Knapsack problem: brute-force approach

- Since there are \( n \) items, there are \( 2^n \) possible combinations of items.
- We go through all combinations and find the one with the most total value and with total weight less or equal to \( W \).
- Running time will be \( O(2^n) \).
- Can be done with better efficiency.

---

0-1 Knapsack problem: a picture

<table>
<thead>
<tr>
<th>Items</th>
<th>Weight</th>
<th>Benefit value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( w_i )</td>
<td>( b_i )</td>
</tr>
<tr>
<td>This is a knapsack</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Max weight: ( W = 20 )</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>