Homework # 3

1. 4.5-4 (p. 97) / 4.3-4 (p. 75)
2. 10.1-2 (p. 235) / 10.1-2 (p. 203)
3. 10-1 (p. 249) / 10-1 (p. 217)
4. 11.2-3 (p. 84) / 11.2-3 (p. 229)

Due March 25, 2010
Outline

- Review
- Binary search trees (cont.)
- Red-black trees
- Augmenting data structures
- Dynamic programming
Review: Dynamic Sets

• Sets that grow, shrink, or otherwise change with time are called **dynamic sets**.

• Each element in the set is represented by a data object.

• Usually one of the fields of an object is called a **key**, and plays a central role in the manipulation of the set data.

• Operations on a dynamic set generally fall into two categories,
  1. queries that return information about the set,
  2. modifying operations that change the set.
Review: Dynamic Sets

• Dynamic sets support *queries* such as:
  • $Search(S, k)$, $Minimum(S)$, $Maximum(S)$,
    $Successor(S, x)$, $Predecessor(S, x)$

• Dynamic sets support *modifying operations* like:
  • $Insert(S, x)$, $Delete(S, x)$
Review: Elementary Data Structures

- **Stacks**: push (insert), pop (delete)
- **Queues**: enqueue (insert), dequeue (delete)
- **Linked lists**: insert, delete, search
- **Rooted trees**
Review: Hashing Tables

• Motivation: symbol tables
  – A compiler uses a symbol table to relate symbols to associated data
    • Symbols: variable names, procedure names, etc.
    • Associated data: memory location, call graph, etc.
  – For a symbol table (also called a dictionary), we care about search, insertion, and deletion
  – We typically don’t care about sorted order
Review: Hash Tables

• More formally:
  – Given a table $T$ and a record $x$, with key (= symbol) and satellite data, we need to support:
    • Insert $(T, x)$
    • Delete $(T, x)$
    • Search$(T, x)$
  – We want these to be fast, but don’t care about sorting the records

• The structure we will use is a hash table
  – Supports all the above in $O(1)$ expected time!
Review: Chaining

- Chaining puts elements that hash to the same slot in a linked list:
Review: Analysis of Chaining

• Assume *simple uniform hashing*: each key in table is equally likely to be hashed to any slot.

• Given $n$ keys and $m$ slots in the table, the *load factor* $\alpha = n/m =$ average # keys per slot.

• The cost of searching $= O(1 + \alpha)$
  – In other words, we can make the expected cost of searching constant if we make $\alpha$ constant.

• *If the number of keys $n$ is proportional to the number of slots in the table, then* $\alpha = O(1)$. 
Review: Binary Search Trees

• *Binary Search Trees* (BSTs) are binary trees with binary search tree property:

  \[ \text{key}[\text{leftSubtree}(x)] \leq \text{key}[x] \leq \text{key}[\text{rightSubtree}(x)] \]

• In addition to satellite data, elements in a BST have:
  – *key*: an identifying field inducing a total ordering
  – *left*: pointer to a left child (may be NULL)
  – *right*: pointer to a right child (may be NULL)
  – *p*: pointer to a parent node (NULL for root)
Outline

• Review
• Binary search trees (cont.)
• Red-black trees
• Augmenting data structures
• Dynamic programming
More BST Operations: Successor and Predecessor

• For deletion, we will need a Successor() operation.
• What are the general rules for finding the successor of node $x$?
  
  • Two cases:
    – $x$ 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 algorithm

TreeWalk(x)
  TreeWalk(left[x]);
  print(x);
  TreeWalk(right[x]);
BST Operations: Delete

• Deletion is a bit tricky
• 3 cases:
  – (0). x has no children:
    • Remove x
  – (1). x has one child:
    • Splice out x
  – (2). x has two children:
    • Swap x with its successor y
    • Perform case (0) or (1) to delete y
BST Operations: Delete

• Why will case 2 always go to case 0 or case 1?
• A: because when x has 2 children, its successor is the minimum in its right subtree
• Could we swap x with predecessor instead of successor?
• A: yes.
Figure 12.4 Deleting a node $z$ from a binary search tree. Which node is actually removed depends on how many children $z$ has; this node is shown lightly shaded. (a) If $z$ has no children, we just remove it. (b) If $z$ has only one child, we splice out $z$. (c) If $z$ has two children, we splice out its successor $y$, which has at most one child, and then replace $z$’s key and satellite data with $y$’s key and satellite data.
Binary search trees: conclusion

• Search trees are data structures that support many dynamic-set operations.

• Basic operations on a binary search tree take time propositional to the height of the tree.
  – The *height of a node* in a tree is the number of edges on the longest simple downward path from the node to a leaf.
  – The *height of a tree* is the height of its root.

• Up next: guaranteeing a $O(\log n)$ height tree
Outline

• Review
• Binary search trees (cont.)
• Red-black trees
• Augmenting data structures
• Dynamic programming
Red-Black Trees

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

• First: describe the properties of red-black trees
  Then: prove that these guarantee \( h = O(lg \, n) \)
  Finally: describe operations on red-black trees ■
Red-Black Properties

• The \textit{red-black properties}:
  1. Every node is either red or black
  2. Every leaf (NULL pointer) is black
     \quad \text{– Note: this means every “real” node has 2 children}
  3. If a node is red, both children are black
     \quad \text{– 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 ■
Figure 13.1 A red-black tree with black nodes darkened and red nodes shaded. Every node in a red-black tree is either red or black, the children of a red node are both black, and every simple path from a node to a descendant leaf contains the same number of black nodes. (a) Every leaf, shown as a NIL, is black. Each non-NIL node is marked with its black-height; NIL’s have black-height 0. (b) The same red-black tree but with each NIL replaced by the single sentinel nil[T], which is always black, and with black-heights omitted. The root’s parent is also the sentinel. (c) The same red-black tree but with leaves and the root’s parent omitted entirely. We shall use this drawing style in the remainder of this chapter.
Height of Red-Black Trees

• We call the number of black nodes on any path from, but not including, a node x to a leaf the black-height of the node, denoted $bh(x)$.

• **What is the minimum black-height of a node with height $h$?**

  **A:** a height-$h$ node has black-height $\geq h/2$

• **Theorem:** A red-black tree with $n$ internal nodes has height $h \leq 2 \log(n + 1)$ ■
RB Trees: Proving Height Bound

• Prove: $n$-node RB tree has height $h \leq 2 \lg(n+1)$

• Claim: A subtree rooted at a node $x$ contains at least $2^{\bh(x)} - 1$ internal nodes
  – Proof by induction on height $h$
  – **Base step**: $x$ has height 0 (i.e., NULL leaf node)
    • What is $\bh(x)$?
RB Trees: Proving Height Bound

- Prove: $n$-node RB tree has height $h \leq 2 \log(n+1)$
- Claim: A subtree rooted at a node $x$ contains at least $2^{bh(x)} - 1$ internal nodes
  - Proof by induction on height $h$
  - **Base step:** $x$ has height 0 (i.e., NULL leaf node)
    
    **What is $bh(x)$?**
    
    A: 0
    
    So…subtree contains $2^{bh(x)} - 1$
    
    $= 2^0 - 1$
    
    $= 0$ internal nodes (TRUE)
RB Trees: Proving Height Bound

- Inductive proof that subtree at node $x$ contains at least $2^{bh(x)} - 1$ internal nodes

  - **Inductive step**: $x$ has positive height and 2 children
    - Each child has black-height of $bh(x)$ or $bh(x)-1$ (Why?)
    - The height of a child = (height of $x$) - 1
    - So the subtrees rooted at each child contain at least $2^{bh(x)} - 1 - 1$ internal nodes

  - Thus subtree at $x$ contains
    $$(2^{bh(x)} - 1 - 1) + (2^{bh(x)} - 1 - 1) + 1$$
    $$= 2 \cdot 2^{bh(x)-1} - 1 = 2^{bh(x)} - 1 \text{ nodes}$$
RB Trees: Proving Height Bound

- Thus at the root of the red-black tree:
  \[ n \geq 2^{bh(root)} - 1 \]
  \[ n \geq 2^{h/2} - 1 \]
  \[ \log(n+1) \geq h/2 \]
  \[ h \leq 2 \log(n + 1) \]

Thus \( h = O(\log n) \) ■
RB Trees: Worst-Case Time

• So we’ve proved that a red-black tree has $O(\log n)$ height

• Corollary: These operations take $O(\log n)$ time:
  – Minimum(), Maximum()
  – Successor(), Predecessor()
  – Search()

• Insert() and Delete():
  – Will also take $O(\log n)$ time
  – But will need special care since they modify tree ■
Red-Black Trees: An Example

- **Color this tree:**

Red-black properties:
1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black
Red-Black Trees: The Problem With Insertion

- Insert 8
  - Where does it go?

1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black
Red-Black Trees:
The Problem With Insertion

• Insert 8 (Cont.)
  – Where does it go?
  – What color should it be?

1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black
Red-Black Trees: The Problem With Insertion

- Insert 8 (Cont.)
  - Where does it go?
  - What color should it be?

1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black
Red-Black Trees:
The Problem With Insertion

• Insert 11
  – *Where does it go?*

1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black
Red-Black Trees: The Problem With Insertion

• Insert 11 (Cont.)
  – *Where does it go?*
  – *What color?*

1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black
Red-Black Trees: The Problem With Insertion

• Insert 11 (Cont.)
  – *Where does it go?*
  – *What color?*
    • Can’t be red! (#3)

1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black
Red-Black Trees: The Problem With Insertion

- Insert 11 (Cont.)
  - Where does it go?
  - What color?
    - Can’t be red! (#3)
    - Can’t be black! (#4)

1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black
Red-Black Trees: The Problem With Insertion

• Insert 11 (Cont.)
  – *Where does it go?*
  – *What color?*

* Solution: recolor the tree

1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black
Red-Black Trees: The Problem With Insertion

• Insert 11 (Cont.)
  – Where does it go?
  – What color?
  • Solution: recolor the tree

1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black
Red-Black Trees: The Problem With Insertion

• Insert 10
  – Where does it go?

1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black
Red-Black Trees: The Problem With Insertion

- Insert 10 (Cont.)
  - Where does it go?
  - What color?

1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black
Red-Black Trees: The Problem With Insertion

- Insert 10 (Cont.)
  - Where does it go?
  - What color?
  - A: no color! Tree is too imbalanced
  - Must change tree structure to allow re-coloring
  - Goal: restructure tree in $O(\log n)$ time
RB Trees: Rotation

• Our basic operation for changing tree structure is called rotation:

\[
\begin{align*}
\text{rightRotate}(y) & \quad \text{leftRotate}(x)
\end{align*}
\]

• Does rotation preserve inorder key ordering?
• What would the code for \texttt{rightRotate()} actually do?
RB Trees: Rotation

• **Answer:** A lot of pointer manipulation
  – $x$ keeps its left child
  – $y$ keeps its right child
  – $x$’s right child becomes $y$’s left child
  – $x$’s and $y$’s parents change

• **What is the running time?**
Figure 13.3  An example of how the procedure \textsc{Left-Rotate}(T, x) modifies a binary search tree. Inorder tree walks of the input tree and the modified tree produce the same listing of key values.
Red-Black Trees: Insertion

• Insertion: the basic idea
  – Insert $x$ into tree, color $x$ red
  – Only r-b property 3 might be violated (if $p[x]$ red)
    • If so, move violation up tree until a place is found where it can be fixed (by recoloring nodes and performing rotations)
  – Total time will be $O(\log n)$
Red-Black Trees: Insertion

• There are actually six cases to consider, but three of them are symmetric to the other three, depending on whether $x$'s parent $p[x]$ is a left child or a right child of $x$'s grandparent $p[p[x]]$.

• We consider the situation in which $p[x]$ is a left child.
  – Case 1: the color of $x$'s parent's sibling, or "uncle" $y$ is red
  – Case 2: the color of $x$'s uncle $y$ is black $x$ is a right child of $p[x]$.  
  – Case 3: the color of $x$'s uncle $y$ is black $x$ is a left child of $p[x]$.

• Figure 13.4
Red-Black Trees: Insertion

• We will not cover the details in class.
  – You should read section 13.3 on your own ■
Red-Black Trees: Deletion

• We will not cover RB delete in class either.
  – **You should read section 13.4 on your own**
  – Read for the overall picture, not the details
Balancing a binary search tree

- **1962** Adel’son-Vel’skii and Landis **AVL tree** (problem 13-3)
- **1970** Hopcroft **2-3 trees** (B-trees Chapter 18)
- **1972** Bayer **Red-black tree**
- **1983** Sleator and Tarjan **Splay trees** (amortized cost)
- **1990** Pugh **SkipLists** (probabilistic)
Outline

• Review
• Binary search trees (cont.)
• Red-black trees
• **Augmenting data structures**
• Dynamic programming
Augmenting Data Structures

• This course is about design and analysis of algorithms.
• So far, we’ve only looked at two design techniques:
  - *induction* (incremental approach)
  - *divide and conquer*
• Next up: augmenting data structures
Augmenting Data Structures

• Often, it will suffice to augment a textbook data structure by storing additional information in it.
  – You can then program new operations for the data structure to support the desired application.
  – Augmenting a data structure is not always straightforward, however, since the added information must be updated and maintained by the ordinary operations on the data structure.
Methodology For Augmenting Data Structures

1. Choose underlying data structure
2. Determine additional information to maintain
3. Verify that information can be maintained for operations that modify the structure
4. Develop new operations
Example: interval trees

- Maintain a dynamic set of intervals such that it is efficient to find out what events occurred during a given interval.
Interval

• The problem: maintain a set of intervals
  – E.g., time intervals for a scheduling program:

\[ i = [7,10]; \text{low}[i] = 7; \text{high}[i] = 10 \]
Interval

• The problem: maintain a set of intervals
  – E.g., time intervals for a scheduling program:

  \[ i = [7,10]; \text{low}[i] = 7; \text{high}[i] = 10 \]

  \[ \begin{array}{ccc}
  4 & \rightarrow & 8 \\
  5 & \rightarrow & 11 \\
  7 & \rightarrow & 10 \\
  10 & \rightarrow & 11 \\
  15 & \rightarrow & 18 \\
  17 & \rightarrow & 19 \\
  21 & \rightarrow & 23 \\
  \end{array} \]

  – Query: find an interval in the set that overlaps a given query interval

  • \([14,16] \rightarrow [15,18]\)
  • \([16,19] \rightarrow [15,18] \text{ or } [17,19]\)
  • \([12,14] \rightarrow \text{NULL}\)
Interval Trees

• We need a data structure to efficiently implement the following operations:
  - insert intervals
  - delete intervals
  - search for overlap
• Can we use hash table? Why?
• If efficiency is not an issue, then we can simply use linked lists.
Interval Trees

• All of the data structures we have studied so far cannot be used directly.

• Issues:
  - What is the key of an interval? (I,D)
  - How to decide if two intervals overlap? (S)
Figure 14.3  The interval trichotomy for two closed intervals \(i\) and \(i'\). (a) If \(i\) and \(i'\) overlap, there are four situations; in each, \(\text{low}[i] \leq \text{high}[i']\) and \(\text{low}[i'] \leq \text{high}[i]\). (b) The intervals do not overlap, and \(\text{high}[i] < \text{low}[i']\). (c) The intervals do not overlap, and \(\text{high}[i'] < \text{low}[i]\).
Interval Trees

• Following the methodology:
  – Pick underlying data structure
  – Decide what additional information to store
  – Figure out how to maintain the information
  – Develop the desired new operations
Interval Trees

• Following the methodology:
  – *Pick underlying data structure*
    • Red-black trees will store intervals, keyed on $low[i]$
  – Decide what additional information to store
  – Figure out how to maintain the information
  – Develop the desired new operations
Interval Trees

• Following the methodology:
  – Pick underlying data structure
    • Red-black trees will store intervals, keyed on low[i]
  – Decide what additional information to store
    • We will store max, the maximum endpoint in the subtree rooted at i (Figure 14.4)
  – Figure out how to maintain the information
  – Develop the desired new operations
Figure 14.4  An interval tree. (a) A set of 10 intervals, shown sorted bottom to top by left endpoint. (b) The interval tree that represents them. An inorder tree walk of the tree lists the nodes in sorted order by left endpoint.
Interval Trees

• Following the methodology:
  – Pick underlying data structure
    • Red-black trees will store intervals, keyed on low\[i\]
  – Decide what additional information to store
    • Store the maximum endpoint in the subtree rooted at \(i\)
  – **Figure out how to maintain the information**
    • *How would we maintain max field?*
  – Develop the desired new operations
• What are the new max values for the subtrees?
• What are the new max values for the subtrees?
• A: Unchanged
• What are the new max values for x and y?
• What are the new max values for the subtrees?
  • A: Unchanged
• What are the new max values for x and y?
  • A: root value unchanged, recompute other
Interval Trees

• Following the methodology:
  – Pick underlying data structure
    • Red-black trees will store intervals, keyed on $i \rightarrow \text{low}$
  – Decide what additional information to store
    • Store the maximum endpoint in the subtree rooted at $i$
  – Figure out how to maintain the information
    • Insert: update max on way down, during rotations
    • Delete: similar
  – Develop the desired new operations
    • $\text{Interval-search}(T, i)$
Searching Interval Trees

```
INTERVAL-SEARCH(T, i)
1   x ← root[T]
2   while x ≠ nil[T] and i does not overlap int[x]
3       do if left[x] ≠ nil[T] and max[left[x]] ≥ low[i]
4           then x ← left[x]
5       else x ← right[x]
6   return x
```

- **What will be the running time?** $O(h)$
- **Proof of Correctness**
IntervalSearch() Example

- Example: search for interval overlapping [14,16]

**Algorithm:**

```
INTerval-Search(T, i)
1   x ← root[T]
2   while x ≠ nil[T] and i does not overlap int[x]
3       do if left[x] ≠ nil[T] and max[left[x]] ≥ low[i]
4           then x ← left[x]
5       else x ← right[x]
6   return x
```
IntervalSearch() Example

- Example: search for interval overlapping $[12, 14]$
Correctness of IntervalSearch()

- Key idea: need to check only 1 of node’s 2 children
  - Case 1: search goes right
    • Show that \( \exists \) overlap in right subtree, or no overlap at all
  - Case 2: search goes left
    • Show that \( \exists \) overlap in left subtree, or no overlap at all

```
INTERVAL-SEARCH(T, i)
1  x ← root[T]
2  while x ≠ nil[T] and i does not overlap int[x]
3     do if left[x] ≠ nil[T] and max[left[x]] ≥ low[i]
4         then x ← left[x]
5     else x ← right[x]
6  return x
```
Prove correctness by loop invariant
Prove correctness

• Invariant for the while loop of lines 2-5:
  If tree $T$ contains an interval that overlaps $i$, then there is such an interval in the subtree rooted at $x$.

• Use blackboard.

```
INTERVAL-SEARCH($T$, $i$)
1  $x \leftarrow \text{root}[T]$
2  while $x \neq \text{nil}[T]$ and $i$ does not overlap $\text{int}[x]$
3      do if $\text{left}[x] \neq \text{nil}[T]$ and $\text{max}[\text{left}[x]] \geq \text{low}[i]$
4          then $x \leftarrow \text{left}[x]$
5      else $x \leftarrow \text{right}[x]$
6  return $x$
```
Outline

- Review
- Binary search trees (cont.)
- Red-black trees
- Augmenting data structures
- **Dynamic programming**
Dynamic Programming

• Another strategy for designing algorithms is *dynamic programming*
  – A metatechnique, not an algorithm (like divide & conquer)
  – The word "Programming" in this context refers to a tabular method, not to writing computer code.

• Typically applied to optimization problems

• Use when problem breaks down into recurring small subproblems
Optimization problems

• The problems we have studied so far have *one correct solution*, which we wanted to compute:
  – There is only one sorted sequence containing a given set of numbers

• **Optimization problems** have *many solutions*:
  – Each solution has a *value* (cost, gain, …)
  – We want to compute an *optimal solution* (with minimal cost, maximal gain, …)
Dynamic Programming vs. Divide and Conquer

• Divide-and-Conquer
  – Partition the problem into *independent* subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem.

• Dynamic Programming
  – Applicable when the subproblems are *not* independent, that is, when subproblems share subsubproblems.
  – Solves every subsubproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time the subsubproblem is encountered.
Development of A Dynamic-Programming Algorithm

• Characterize the structure of an optimal solution
• Recursively define the value of an optimal solution
• Compute the value of an optimal solution in a bottom-up fashion
• Construct an optimal solution from computed information
Dynamic programming

• Example 1
  – Matrix-chain multiplication

• Example 2
  – Longest common subsequence