CPSC 331: DATA STRUCTURES, ALGORITHMS, AND THEIR ANALYSIS

GO UEZONO

April 8, 2024

CONTENTS

LIST OF FIGURES

LIST OF TABLES

3.2 Red and Black Trees

3.2.1 *Properties*

A **red-black tree** is a concrete implementation of a **self-balancing binary-search tree** (reference here) that automatically maintains balance. Giving each node their respective color ensures that no path is more than twice as long as any other, thus is able to maintain approximate balance.

1. Every node is red/black
2. Root must be black
3. Leaves (*null*) are black
   - *null* vertices contain no values, while other (interior) do
4. If a node is red, then both its children are black
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes

The following **lemma** shows why red-black trees make good search trees:

Lemma 1. *A red-black tree with n internal nodes has height at most $2\log(n+1)$*

Proof. Start by showing subtree rooted at any ndoe x x contains at least a $2^{bh(x)}-1$ internal nodes. We prove this by **mathematical induction** on the height of x.

CLAIM: If height of $x = 0$, then the leaf must be *T.null*, and the subtree rooted at x contains at least $2^{bh(x)} - 1 = 2^0 - 1 = 0$ internal nodes.

INDUCTIVE STEP:
- Consider a node x that has positive height and is an internal node with two children.
- Each *child* has a black-height of either $bh(x)$ or $bh(x)-1$ (depending on whether it is red or black respectively).
- Since height of a *child* of x is less than the height of x itself, we can apply the **I.H** to conlude that:
  - Each child has at least $2^{bh(x)-1}-1$ internal nodes.

Thus, subtree rooted at x contains at least

$$(2^{bh(x)-1}-1)+(2^{bh(x)-1}-1)+1$$

internal nodes, which proves the claim.

To complete the proof, let h be the height of thr tree. According to property 4 (reference above Properties), at least half the nodes from the root to a leaf (not including the root) must be black.

Consequently, the bh of the root must be at least $h/2$; thus,

$$n \geqslant 2^{h/2}-1$$

Moving 1 to the left side and taking log on both sides yields:

$$\log(n+1) \geqslant h/2$$

or

$$h \leqslant 2\log(n+1)$$

□

3.2.2 *Rotational Properties*

Search operations *TREE-INSERT* and *TREE-DELETE* take $O(\log n)$ time. Since modifications are done to the tree, we must change the color of some of the nodes.

Definition (Rotation). Local operation that preserves the binary-tree property.

- **Left Rotation:** assume that its right child y is not *null*
- **Right Rotation:** assume that its left child y is not *null*
  - x can be any node on the tree whose respective child is not *null*
  - Left/Right rotations "pivots" around the link from x to y
  - Makes y the new root, x as y's left(right) child, y's left(right) child as x's right(left) child
- Both L/R rotates run in $O(1)$ time
- Only pointers are changed, all attributes in a node remain the same

Algorithm 1: Left-Rotate(T, x)

```
1  x = y.right ;                                                    // set y
2  x.right = y.left ;      // Turn y's left subtree into x's right subtree
3  if y.left ≠ T.null then
4  |   y.left.p = x;
5  y.p = x.p;
6  if x.p == T.null then
7  |   T.root = y;
8  else if x == x.p.left then
9  |   x.p.left = y;
10 else
11 |   x.p.right = y;
12 y.left = x;
13 x.p = y;
```

3.2.3 *Insertion*

Inserting a node can be done in $O(\log n)$ time. Below is a pseudo-code that shows how insertion *RB-INSERT* works:

Algorithm 2: RB-INSERT(T, z)

Data: z node to insert,

```
1  y = T.null;
2  x = T.root;
3  while x ≠ T.null do
4  |   y = x;
5  |   if z.key < x.key then
6  |   |   x = x.left;
7  |   else
8  |   |   x = x.right;
9  |   end
10 end
11 z.p = y;
12 if y == T.null then
13 |   T.root = z;
14 else if z.key < y.key then
15 |   y.left = z;
16 else
17 |   y.right = z;
18 z.left = T.null;
19 z.right = T.null;
20 z.color = T.RED;
21 RB-INSERT(T, z);
```

To ensure we preserve the red-black properties,

3.3 Heaps

3.3.1 *Priority Queues*

- Priority queues are **NOT** FIFO
- These queues are interested in removing items (dequeue) with the **highest priority**
- Assume that higher priority value (HPV) entails higher priority
  - Not true in general (UNIX OS; smaller PV = higher priority)
- Similar operations as the standard Queue ADT:

```
// Queue ADT
public interface QueueADT<T> {
   public void enqueue(T item);
   public T dequeue(); // different implementation
   public boolean isEmpty();
   public boolean isFull();
}
```

PRIORITY QUEUE IMPLEMENTATIONS

- **Lists**
  - **Sorted** list by PV with array implementation
    - *Ascending order*: Remove **last** item
    - *Dequeue*: $O(1)$, n^{th} element of the array to be removed
    - *Enqueue*: $O(n)$, needs to be sorted after enqueue
  - **Sorted** list by PV with linked-list implementation
    - *Descending order*: Remove **first** item
    - *Ascending order*: Circular list implementation
    - *Dequeue*: $O(1)$
    - *Enqueue*: $O(n)$
  - **Unsorted** list
    - *Dequeue*: $O(n)$
    - *Enqueue*: $O(1)$
- **BST**
  - *Dequeue*: $O(\log(n))$ average-case, $O(n)$ worst-case
  - *Enqueue*: $O(\log(n))$ average-case, $O(n)$ worst-case
- **Heaps**
  - *Dequeue*: $O(\log(n))$ worst-case
  - *Enqueue*: $O(\log(n))$ worst-case

3.3.2 *Properties*

A heap is a **complete** binary tree that satisfies the Heap Property:

- Each node of a tree corresponds to an element of an array
  - Always stored **contiguously**
    - If there are blanks, they are on the rightside of the array
    - Otherwise, no blanks inbetween indices
- It is of height h and contains n nodes

Proof. Height h □

HEAP PROPERTIES

- **Min Heap Property**
  - Every node has a value $\leqslant$ than the value of its children
  - Root of any subtree has the *minimum* value in the subtree

  GIVE EXAMPLE
- **Max Heap Property**
  - Every node has a value $\geqslant$ than the value of its children
  - Root of any subtree has the *maximum* value in the subtree

  GIVE EXAMPLE

MAXHEAP CLASS Similar to the Queue ADT, differences in the enqueue and dequeue functions to maintain **heap** property

```
public class MaxHeap<T> {
   private T[] queue;
   private int size;

   public MaxHeap(Class<T> clazz, int maxSize) {
      queue = (T[]) Array.newInstance(clazz, maxSize);
      size = 0;
   }
   public boolean isEmpty() {
      return (size == 0);
   }
   public boolean isFull() {
      return (size == queue.length);
   }
}
// enqueue()/dequeue() functions shown later
```

Definition. *Enqueue*

- Must keep the tree **complete**
- Must keep the **max heap** property
- **Complexity of *Enqueue***
  - *Worst-case*: added node percolates from leaf to root
  - Since the tree is **complete**, height is $O(\log(n))$
  - Hence, **enqueue** is $O(\log(n))$

Enqueue:

```
public void enqueue(T item) {
   queue[size] = item;

   // Fix heap
   int loc = size;
   int parent = (loc - 1)/2;
   while (loc > 0 && queue[loc].compareTo(queue[parent]) > 0) {
      swap(loc, parent);
      loc = parent;
      parent = (loc - 1)/2;
   }
   size++;
}
```

Correctness: **Loop Invariant** for Enqueue

Proof. □

Definition. *Dequeue*

- Must keep the tree **complete**
- Must keep the **max heap** property
- **Complexity of *Dequeue***
  - Deleted root "hole" always sinks to a leaf
  - Since the tree is complete, height is $O(\log(n))$
  - Hence, **dequeue** is $O(\log(n))$

Dequeue:

```
public T dequeue() {
   T max = queue[0];
   queue[0] = queue[size - 1];
   sink(0);
   queue[size - 1] = null;
   size--;
   return max;
}
```

Correctness: **Bound Function** for Dequeue

Proof. Prove that sink() maintains heap property □

3.4 Hash Table

3.5 Graphs

3.6 Dijkstra's Algorithm

4 SORTING ALGORITHMS

4.1 Bubble Sort

Understand algorithm, prove correctness, time complexity

4.2 Selection Sort

Understand algorithm, prove correctness, time complexity

4.3 Insertion Sort

Understand algorithm, prove correctness, time complexity

4.4 Heap Sort

Understand algorithm, prove correctness, time complexity

5 ADVANCED SORTING ALGORITHMS

5.1 Merge Sort

Understand algorithm, prove correctness, time complexity

5.2 Quick Sort

Understand algorithm, prove correctness, time complexity

6 SEARCHING

6.1 Linear Search

Algorithm, prove correctness, time complexity

6.2 Binary Search

Algorithm, prove correctness, time complexity

7 GRAPH TRAVERSAL

7.1 Depth-First Search

Develop algorithm, prove correctness, analyze time complexity, applications

7.2 Breadth-First Search

Develop algorithm, prove correctness, analyze time complexity, applications

8 METHODS

Test math notation: $\cos \pi = -1$ and $\alpha\omega$
Test Algorithm

Algorithm 3: Left-Rotate(T, x)

input : Test
output: Test

```
1 i ← 1;
2 if condition then
3 |  then block
4 else
5 |  else block
6 end
```

1. 1st item in list
2. 2nd item
3. 3rd

8.1 Test sub

DESCRIPTION

2ND DESCRIPTION

8.2 Math subsection

$$\cos^3 \theta = \frac{1}{4}\cos\theta + \frac{3}{4}\cos 3\theta \tag{1}$$

Definition (Gauss). To a mathematician, it is obvious that $\int_{-\inf}^{+\inf} e^{-x^2}\,dx = \sqrt{pi}$.

Theorem 8.1 (Red and Black Trees). *Red trees are better than black trees.*

Proof. We have that $\log(1)^2 = 2\log(1)$. We also have that $\log(-1)^2 = \log(1) = 0$. Then, $2\log(-1) = 0$, from which the proof. □

9 RESULTS AND DISCUSSION

9.1 Subsection

Test subsec

9.2 Subsubsection

Test sub

WORD Definition

CONCEPT Explanation

IDEA Text

Test Test

- First
- Second
- Third