csce350 — Data Structures and Algorithms
Fall 2019 — Lecture Notes: Transform and Conquer

This document contains slides from the lecture, formatted to be suitable for printing or individual reading, and with occasional supplemental explanations added. It is intended as a supplement to, rather than a replacement for, the lectures themselves — you should not expect the notes to be self-contained or complete on their own.

1 Introduction

6

Transform and conquer algorithms are based on changing the problem instance into a form that can be solved more readily.

Three flavors:

- Transform into a simpler instance of the same problem.
- Transform into different representation of the same instance.
- Transform into an instance of a different problem.

2 Example: Element Uniqueness

Recall this algorithm:

```
ElementsUnique(A[0, ..., n − 1])
  for i ← 0, ..., n − 2 do
    for j ← i + 1, ..., n − 1 do
      if A[i] = A[j] then
        return false
      end if
    end for
  end for
  return true
```

3 Presorting

We can do better by transforming to a simpler instance of the same problem:

```
PresortElementsUnique(A[0, ..., n − 1])
  Sort(A[0, ..., n − 1])
  for i ← 0, ..., n − 2 do
    if A[i] = A[j] then
      return false
    end if
  end for
  return true
```

4 Example: Array Search

Recall the simple algorithm we saw:

```
SEQUENTIALSEARCH(A[0, ..., n − 1], K)
  i ← 0
  while i ≤ n and A[i] ≠ K do
    i ← i + 1
  end while
  if i < n then
    return i
  else
    return −1
  end if
```

5 Be careful!

```
PRESORTSEARCH(A[0, ..., n − 1], K)
  SORT(A)
  return BINARYSEARCH(A, K)
```

Is this a good idea?

6 AVL Trees

6.3

Remember **binary search trees (BST)**.

- One **key** stored at each node.
- Each node's node's key is...
  - greater than the keys in its left subtree
  - less than the keys in its right subtree

7 BSTs: Simple operations

Search: Compare to the root key. If needed, recur on the left or right subtree.

Insert: Just like search, then create a new leaf there.

8 BSTs: What can go wrong?

9 Solution: Change the shape of the tree

Solution: After inserting, modify the tree to keep balance.

One method, called **AVL trees**, guarantees that *at each node, the height of the left subtree and the height of the right subtree differ by at most 1.*

An AVL tree is a special kind of BST — all of the usual rules for BSTs apply.

The **balance factor** of a node is (left subtree height) − (right subtree height).

10 Rotations

We can use two operations called **right rotation** and **left rotation** to change the shape of a BST.

11 Rotation details

```
ROTATER(a)
  b ← a.left
  T2 ← b.right

  if a.parent.left = a then
    a.parent.left ← b
  else
    a.parent.right ← b
  end if
  b.parent ← a.parent

  b.right ← a
  a.parent ← b

  a.left ← T2
  T2.parent ← a
```

12 When and where to rotate

In AVL trees, we can restore balance after inserting using at most **2** rotations.

- **Insert** normally.
- **Trace** back up the tree toward the root, updating balance factors.
- At the first node T with a balance factor of $+2$ or -2, **rebalance**.

```
REBALANCE(T)
  if T.bf = −2 then
    if T.right.bf = +1 then
      ROTATER(T.right)
    end if
    ROTATEL(T)
  else
    if T.left.bf = −1 then
      ROTATEL(T.left)
    end if
    ROTATER(T)
  end if
```

13 AVL tree example

Insert: 5, 6, 8, 3, 2, 4, 7

14 AVL tree example

Insert: 5, 6, 8, 3, 2, 4, 7

15 2-3 Trees

Instead of relying on rotations, we can also **store multiple keys at each node.**

In a **2-3 tree**, there are two kinds of nodes:

- **2-nodes** have one key and zero or two children.
- **3-nodes** have two keys and zero or three children.

The keys are organized similarly to a binary search tree, but with a 'between' case for 3-nodes.

16 2-3 Trees: Insert

To insert a new key into a 2-3 tree:

- **Search** normally to find the correct leaf.
- **Add** the new key to that leaf.
- If we have more than three keys, **split** into two 2-nodes and **push** one key up to the level above.

17 2-3 tree example

Insert: 5, 6, 8, 3, 2, 4, 7, 9

18 2-3 tree example

Insert: 5, 6, 8, 3, 2, 4, 7, 9

19 Heaps

A **heap** is a data structure for implementing **priority queues**.

- INSERT a key.
- EXTRACT the key with the highest priority.

Definition A heap is a binary tree with one key stored at each node, such that:

- The tree is **essentially complete**. That is, every level is full, except possibly the bottom level, which may be missing some of its rightmost nodes.
- The key at each node is **greater than** the keys at its children.

20 Representing a heap

The special shape of a heap makes it possible to store it in an array, instead of allocating memory for each node.

- Root node: $A[0]$
- Left child of $A[i]$: $A[2i + 1]$
- Right child of $A[i]$: $A[2i + 2]$
- Parent of $A[i]$: $A[\lfloor (i - 1)/2 \rfloor]$

21 Heap insertion

- **Attach** the new key as a new rightmost leaf.
- **Compare** the new key to its parent, and 'swap up' if necessary.
- **Repeat** until the heap condition is restored.

22 Heap deletion

- **Replace** the root element with the right most leaf.
- **Compare** the new root to both of its children, and 'swap down' with the larger child if necessary.
- **Repeat** until the heap condition is restored.

23 HeapSort

Because a heap can efficiently give us the largest remaining element, it can be directly used to sort.

- **Construct** a heap from the array elements.
- **Extract** the largest elements from the heap, one at a time.