Lecture 7

Optimal Sorting
&
AVL Trees
Polyphase Merge Sort

An optimal - namely $O(n \log_2 n)$ - sort is the polyphase merge-sort which was originally used to sort data held secondary storage media such as magnetic tapes or discs.

**Strategy**
Order within the data is exploited and each distribute and merge phase approximately halves the disorder.

The algorithm uses $3 \times n$ units of space where $n$ is the number of keys to be sorted – so space has been traded for efficiency.

**Array Version**
We cauch the algorithm in terms of sorting an array $a$ into, as per usual, increasing order. Two auxiliary arrays $b$ and $c$ are needed.

**NB** When just 2 extra arrays are used the name of the sort is a 2-way polyphase merge-sort.
Run Concept

A run is a sequence of adjacent elements in an array that are in increasing order by index.

1, 2, 3, 4, 5  single run of length 5!

5, 4, 3, 2, 1  five runs - each of length 1!

Pair-Wise Merging of Runs

4, 7, 11, 2 ....  \{ 1, 4, 7, 11, 13  2, 5 ...

1, 13, 5 ...

Polyphase Merge-Sort Overview

Runs in a are alternately distributed between b and c respectively. Odd numbered runs go into b and evenly numbered runs go into c. Pair-wise merging of runs then takes place back into a. Above is repeated until there is just 1 run in a.
Merge-Sort Code

In the code below we assume that there are three arrays; a, and two auxiliaries b, and c.

read in a ;
sorted = false ;
while not sorted do
    while not end of a do
        while on run in a do
            copy latest element into b ;
        end while
        while on run in a do
            copy latest element into c
        end while
    end while
    if no runs in c
    then  sorted = true ;
    else  pair-wise merge runs from b and c back into a ;
end while
Polyphase Merge-Sort Ex.

1 2 3 4 5
a : 1, 3, 12 2, 19 13 4,8, 9 6,10

\[\text{distribute} \rightarrow b: 1,3,12 13 6,10\]
\[c: 2,19 4,8,9\]

\[\text{merge} \rightarrow a: 1,2,3,12,13,19 4,6,8,9,10\]

\[\text{distribute} \rightarrow b: 1,2,3,12,13,19\]
\[c: 4,6,8,9,10\]

\[\text{merge} \rightarrow a: 1,2,3,4,6,8,9,10,12,13,19\]

\[\text{distribute} \rightarrow b: 1,2,3,4,6,8,9,10,12,13,19\]
\[c: \text{Nothing goes into c therefore sorted!}\]

**NB** Decreases identify where one run ends and the next one starts. Odd/Even just means the number of the run.
Polyphase Merge-Sort Analysis

• Since the maximum number of distribute and merge phases is $O(\log_2 n)$ and linear work is done on each phase - the total complexity is $O(n \log_2 n)$

We next look at another balanced tree scheme called AVL-trees.

It should be noted that when balanced trees are traversed to yield keys in increasing order they also give $O(n \log_2 n)$ sorts.

**NB**
We qualify these mergesorts with the term polyphase as the term without this qualification is used in another context.
BSSTs Revisited

A BSST is an ordered tree such that nodes have either two or zero siblings. Keys are stored in the structure, subject to the search rule:

For any non-leaf node \( n \) containing a key \( k \):
- Its left sub-tree contains only keys less than \( k \).
- Its right sub-tree contains only keys greater than \( k \).

```
       n: k
       /   \    ....
     /     \     
  left  >k    right  <k
```
AVL-Trees

Over the next couple of slides we look at special Binary Sequence Search trees that have a balance factor of -1, 0 or +1 at each node. They are known as AVL trees.

Motivation

Remember with ordinary BSSTs we call trees of the following type balanced.

and trees of this type unbalanced.
Balanced Trees

With balanced tree structures we expect search insert and delete operations to be logarithmic in complexity. We have already seen one example of a balanced tree structure namely the B-tree.

When there are \( n \) keys in the structure the B-tree complexities are:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search for a key</td>
<td>( O(\log_2 n) )</td>
</tr>
<tr>
<td>Insert a key</td>
<td>( O(\log_2 n) )</td>
</tr>
<tr>
<td>Delete a key</td>
<td>( O(\log_2 n) )</td>
</tr>
</tbody>
</table>

AVL-Tree Efficiencies

AVL trees (named after their inventors, Adelson-Velski and Landis) also turn out to have the above algorithmic efficiency for each of the three main data dictionary operations.
Height of a Tree

The height or (depth) of a binary tree is defined as the length of a longest path from the root to a leaf.

**NB** Such a path is not necessarily unique.

**Example**

A height 4 tree.

![Tree Diagram]

**NB** Some authors don't count the last leg, so they would define height as 3 above.
Balance Factor

The balance factor at a node is the depth of its left subtree minus the depth of its right subtree.

Example

```
0  38
 -1 20 +1 72
  0 29  0 56
 0 22  0 60
```

NB

Some authors subtract the left depth from the right depth, so with that definition all the above balance factors would be negated.
Algorithmics Lecture 7

**AVL Tree Property**

In an AVL tree the only permitted balance factors are -1, 0 and +1.

![AVL Tree Diagram]

The above tree is an AVL-tree because the absolute value of its balance factors are no more than 1.

**NB** "absolute" is a fancy way of saying ignore signs.
Rotation Concept

A rotation is an operation that preserves a tree's search properties.

Think of a "pulley". Node 03 and 07 are moved to the left and the link to 05 is adjusted as it swings leftwards.
Right Rotation

Here we look at the right-handed version of the previous operation.

A "right" rotation at node 03.
AVL-Tree Insertion Strategy

The AVL tree-search algorithm for the leaf-point of insertion is determined by the usual BSST search algorithm – starting from the root.

• Imbalances are addressed by left or right rotations.

• The Node at which balancing needs to take place may be quite high up the structure.

• Imbalance doesn't always occur - in which case insertion is straightforward.

**NB**

When balancing is required, only one or two rotations are necessary. We examine the two basic cases over the following slides.
This can be read as … As a result of inserting a new key in the left subtree of a its balance factor is now +1.
Synopsis
Node a was originally left-heavy and now it has been made even more left-heavy with a balance factor of +2. The insertion is in the left subtree of b.
Single Rotation Case

+2
a

+1
b

A

B

C

A

B

C

0

0

a

b

Algorithmics Lecture 7
Synopsis
Node a was originally left-heavy and now it has been made even more left-heavy with a balance factor of +2. This time the insertion is right of b. We only show the B case, insertion below C is similar.
Double Rotation Case

+2  a

-1  b

+1  c

D

A

B

C

0  c

0  b

-1  a

D

A

B

C
AVL-Tree Insertion Analysis

- Imbalance can only occur for nodes that are on the path from the inserted node to the root.

- When balancing, since the tree height of the node to be balanced has effectively been "restored" no further nodes require any balancing.

- Worst-case trees have height $O(\log_2 n)$ so the insertion algorithm takes logarithmic time.
Further Discussion

• AVL-Deletion can also be performed in logarithmic time as can searching, so all the basic data dictionary operations can be implemented logarithmically.

• There are similar operations to the AVL rotations for B-trees - to make them more resilient to splitting! These improved B-trees where rotations as well as splits are used, are known as B*-trees.

NB A B-tree split can be avoided when a sister B-tree node has space.
Index

10 Balance Factor
11 AVL Tree Property
12 Rotation Concept
13 Right Rotation
14 AVL-Tree Insertion Strategy
15 Schematic
16 Single Rotation Scenario
17 Single Rotation Case
18 Double Rotation Scenario
19 Double Rotation Case
Index

20  AVL-Tree Insertion Analysis

21  Further Discussion