TTIT33 Algorithms and Optimization – DALG Lecture 6

Content:

- Balanced Search Trees
  - AVL Trees [GT 10.2, LD 7.1]
  - Multi-way search trees [GT 10.4, LD 7.2, CL 18.1-2]
- Priority Queues – Heaps [GT 8.1.3, LD 8.1, CL 6.1-5]

AVL Trees

AVL = Adelson-Vel’nikov and Landis, 1962

The idea: Keep a balance information at each node

The balance b(v) of a node v:
- **LeftHeight(v)**: if v has no left child
  1 + Height(L(v)) otherwise
- **RightHeight(v)**: if v has no right child
  1 + Height(R(v)) otherwise
- b(v) = LeftHeight(v) - RightHeight(v)

A BST is an AVL-tree if b(v) ∈ {-1, 0, 1} for every node v.

Worst AVL Tree

A worst AVL tree of height k:
- as few nodes as possible to achieve height k

Construction: Node balance always ±1, subtrees always worst AVL trees

Fibonacci numbers f_k = f_k-1 + f_k-2: 0, 1, 1, 2, 3, 5, 8, 13, ...

We can prove:
- f_k = f_{k-1} + f_{k-2} (by induction)

AVL Tree Height Theorem:
- k < 1.44f_{log_2n}

Hence: LookUp in O(log_2n) steps
- Example: insert 44, 17, 78, 30, 88, 62

AVL-Tree Insertion – Rebalance...

Insertion may destroy the balance.

Insertion with rebalancing:
1. search downwards the tree, the lowest ancestor of the insertion node with balance ≠ 0
2. insert
3. rotate at critical node if necessary

AVL-Tree Insertion... (Goodrich/Tamassia)

How to insert an item with key 50?
- Perform a LookUp(50)
- While searching for 50, keep track of last passed node with balance ≠ 0
- If not found, insert at the leaf where search ended
- Recompute balance on the way back
- Check critical node! If balance ≠ {-1, 1}, rebalance!
AVL-Tree Insertion & Re-balance...

Now try an item with key 15...
- Perform a LookUp(15)
- While searching for 15, keep track of last passed node with balance ≠ 0
  ➔ critical node
- If not found, insert at the leaf where search ended
- Recompute balance on the way back
- Check critical node!
  If balance ∉ [-1..1], rebalance!

Time to re-balance...
- Label the critical node and its 2 descendants on the path to "15" as $a$, $b$, $c$, such that $a < b < c$, in an in-order traversal
- Re-structure the nodes $a$, $b$, and $c$ so that $b$ has $a$ and $c$ as children!
- Update balance!

Removal of a node...
Same thing, but in reverse!
1. Perform a LookUp and Remove as in an ordinary binary tree
2. Update the balance on the way back to the root
3. If too unbalanced: Re-structure! ...but:
   - Label the critical node, the child on the deepest side, and its descendants on the deepest side as $a$, $b$, $c$, such that $a < b < c$, in an in-order traversal
   - Re-structure as previous
   - Go to #2 and continue update and check towards the root (we may have to re-balance more than once!)

Tri-node restructuring = rotations....
Other authors use left and right rotations:
- Single left rotation:
  - left part of the subtree ($a$ and $j$) is lowered
- We have "rotated (up) $b$ over $a$"...

Double rotations...
Two rotations are needed when the nodes to re-balance are placed in a zig-zag pattern...
1. Rotate up $b$ over $a$
2. Rotate up $b$ over $c$

Note: Labeling of $a$, $b$ and $c$ same as before!
New approach: relax some condition...

- AVL-Tree: binary tree, accepts a small unbalance

Recall:
- Full binary tree: nonempty; degree is either 0 or 2 for each node
- Perfect binary tree: full, all leaves have the same depth

Can we build and maintain a perfect tree (if we skip "binary")??

\[ \text{we would always know the worst search time exactly!} \]

(2,3) trees, (2,4) trees, or \((a,b)\)-trees...

- Each node is either a leaf, or has \(c\) children where \(a \leq c \leq b\)

- LookUp works approximately as before

- Insert must check that a node does not overflow (then we split the node)

- Delete must check that a node does not become empty (then we transfer or merge nodes)

Delete in a \((2,3)\)-tree

Three cases:
1. No constraints are violated by removal
2. A leaf is removed (becomes empty)
   \[ \xrightarrow{\text{transfer}} \text{some other key to that leaf,} \]
   ...ok if we have a sibling with \(2+\) elements

\[ \text{No pivot ??} \quad \text{Too many children!} \]
Delete in a (2,3)-tree

3. An internal node becomes empty
   Root: replace with in-order pred. or succ.
   \( \Rightarrow \) repair inconsistencies with suitable merge and transfer operations...

```
Delete(20)
Replace...
...merge leaves
...merge nodes
```

Properties of a (2,3) tree

- Always a perfect tree
- A minimal tree of height \( h \) will have \( n = 2^{h+1} - 1 \) nodes (it's a full binary tree with full set of nodes at all levels)
- A maximal (2,3)-tree will have a branching factor of 3, thus
  \[
  n = \sum_{i=0}^{h} 3^i = \left(3^{h+1} - 1\right) / 2
  \]
  \( \Rightarrow \) \( k = 3^{h+1} - 1 \)
- Thus the height \( h = \left\lceil \log_3 k \right\rceil \)

B-Tree

- Used to keep an index over external data (e.g. content of a disc)
- It's only an \( (a,b) \)-tree where \( a = \left\lceil \frac{b}{2} \right\rceil \)
- We may now choose \( b \) so that \( b-1 \) references to children (other disc blocks) fit into a single disc block
- By defining \( a = \left\lceil \frac{b}{2} \right\rceil \) we will always fill up a disc block when two blocks are merged!

ADT Priority Queue:

- Linearly ordered set \( K \) of keys
- We store pairs \( < k, i > \) (as in Dictionary), multiple pairs with same key are allowed. The key denotes priority
- Items retrieved by priority: i.e. by the minimal key.

Operations on a Priority Queue \( P(Q) \):

- \( makeEmpty(P,Q) \)
- \( isEmpty(P,Q) \)
- \( insert(P,Q,i) \)
- \( findMin(P,Q) \)
- \( deleteMin(P,Q) \)
- \( decreaseKey(P,Q,i,k) \)

Implementing Priority Queues

Searching for a minimal element in a search tree (BST, AVL, 2-3,...)

Another idea: keep the minimal element as the root of the tree
\( \Rightarrow \) partially ordered tree

```
This is a complete binary tree!
```

Updates on a Heap Structure

- \( \text{DeleteMin} = \text{deletion of the root} \)
  - Replace root by last leaf
  - Restore partial order by swapping nodes downwards
    "down-heap bubbling"

- \( \text{Insert} \)
  - Insert new node after last leaf
  - Restore partial ordering by "up-heap bubbling"
HEAP Properties:
- `size()`, `minElement()`: \( O(1) \)
- `minElement()`, `minKey()`: \( O(1) \)
- `insertItem()`, `removeMin()`: \( O(\log n) \)

>> HeapSort \( O(n \log n) \) sorting algorithm

Recall vector representation of BST!
- A complete binary tree...
- Compact vector representation
- Bubble-up and bubble-down have fast implementations

Heap Sort
- Kinds of heaps:
  - `minKey` in the root \( \Rightarrow \) (Min)Heap
  - `maxKey` in the root \( \Rightarrow \) MaxHeap

Heap Sort: input \( D[0..n] \)
- **Heapify** \( D \Rightarrow \) Max Heap
  - worst case \( O(n \log n) \)
- **For** \( i \) from 0 to \( n-1 \)
  - **DeleteMax** \( \Rightarrow \) put in \( D[n-i] \)
  - (worst case \( O(\log n) \) for restoring heap \( D[0..i] \))