Final Exam

• Location
  • YORK 2622

• Time
  • Thursday, December 10, 2015, 3:00pm – 6:00pm
Final Exam...

- 4 Parts
- Part 1: Basic knowledge of data structures and C++
  - 20% to 40% of final score
  - Multiple choice
- Part 2: Application, Comparison and Implementation of the data structures we have covered
  - 20% to 40% of final score
  - Short answers
- Part 3: Simulating algorithms and run time analysis
  - 20% to 40% of final score
  - Short answers
- Part 4: C++ and programming assignments
Topics

A. B trees
B. Red-Black trees
C. Multiway tries and ternary tries
D. Hashing
E. Skip lists
Properties of an m-order B trees

1. The root has at least 2 sub-trees, unless it is a leaf
2. All leaves are at the same level
3. Each node (leaf as well as non leaf) holds k-1 keys where \( \text{ceil}(m/2) \leq k \leq m \)
4. Each non-leaf node additionally holds k pointers to subtrees where \( \text{ceil}(m/2) \leq k \leq m \).
(b) Which of the following are legal 2,3 trees (B tree of order 3)? For a tree that is not a valid 2,3 tree, state a reason why.
Insert the value 42 into the following BTree.
B+-tree with parameter M, L

- There are several variants of the basic B tree idea
  - B+ trees is one of them
- In a B+ tree, data records are stored only in the leaves
  - A leaf always holds between $\lceil L/2 \rceil$ and L data records (inclusive)
  - All the leaves are at the same level
- Internal nodes store key values which guide searching in the tree
  - An internal node always has between $\lceil M/2 \rceil$ and M children (inclusive)
    - (except the root, which, if it is not a leaf, can have between 2 and M children)
  - An internal node holds one fewer keys than it has children:
    - the leftmost child has no key stored for it
    - every other child has a key stored which is equal to the smallest key in the subtree rooted at that child
A B+-tree with M=4, L=3

- Every internal node must have 2, 3, or 4 children
- Every leaf is at the same level (level 2 in the tree shown), and every leaf must have 2 or 3 data records (keys only shown)
Red-Black Trees

1. Nodes are either red or black
2. Root is always black
3. If a node is red, all its children must be black
4. For every node X, every path from X to a null reference must contain the same number of black nodes
How can we make the above tree a valid red-black tree

Insert 50. Then insert 66. Draw the resulting red-black tree.
Red-black tree bounds on height

Red-black tree with $N$ black nodes.

1. Merge all red-nodes.
2. All leaves are at the same level.

No of nodes at level 0 = 1
No of nodes at level 1 $\geq$ 2
No of nodes at level $i$ $\geq$ $2^i$
Total no of nodes = $N_{\text{black}}$ $\geq$ $1 + 2 + \ldots + 2^i + \ldots + 2^{h-1}$
$\geq 2^h - 1$

$h \leq \log_2(N_{\text{black}} + 1)$

Height of original RBT $\leq 2 \times h \leq 2 \times \log_2(N_{\text{black}} + 1)$
Red-black tree bounds on height

Red-black tree with N black nodes.

1. Merge all red-nodes.
2. All leaves are at the same level.

No of nodes at level 0 = 1
No of nodes at level 1 <= 4
No of nodes at level i <= 4^i
Total no of nodes = N_{black} <= 1 + 4 + … + 4^i + … + 4^{h-1}
<= (4^h -1)/3
h >= \log_4(3N_{black} + 1)

Height of original RBT >= h >= \log_4(3N_{black} + 1)
Red-black tree bounds on height

Red-black tree with total N nodes.

1. Height is lowest when the tree is a complete binary tree.

No of nodes at level 0 = 1
No of nodes at level 1 <= 2
No of nodes at level i <= 2^i
Total no of nodes = N <= 1 + 2 + … + 2^i + … + 2^{h-1} <= (2^h - 1)

h >= log_2(N + 1)

Height of original RBT >= h >= log_2(N + 1)
Multi-way tries: Efficient finding of keys by their sequence

Build the trie which holds the following number keys:

Assuming your trie could potentially hold any decimal number, how many children does each node (potentially) have?
A. 2    B. 8    C. 10    D. Other
List all the words (strings) you can find in this TST

Are the following in the tree? (A=yes, B=no)
• get
• if
• gif
• its
• gacar
• tsem
Open addressing vs. separate chaining

- Linear probing and random hashing are appropriate if the keys are kept as entries in the hashtable itself...
  - doing that is called "open addressing"
  - it is also called "closed hashing"

- Another idea: Entries in the hashtable are just pointers to the head of a linked list ("chain"); elements of the linked list contain the keys...
  - this is called "separate chaining"
  - it is also called "open hashing"

- Collision resolution becomes easy with separate chaining: just insert a key in its linked list if it is not already there
  - (It is possible to use fancier data structures than linked lists for this; but linked lists work very well in the average case)
Write the sequence of vertices visited when running DFS on the following graph. Assume the link to the vertex with the minimum edge weight is chosen when multiple choices are available.

Write the sequence of vertices visited when running BFS on the following graph. Assume the link to the vertex with the minimum edge weight is chosen when multiple choices are available.
Which of the following is/are balanced trees?

And thus can become AVL trees by adding the balance factors.

D. A&C
E. A&B&C

Annotate the trees with balance factors.
Insert 50. Then insert 66. Draw the resulting AVL tree.
## Data structure Comparison

<table>
<thead>
<tr>
<th></th>
<th>Insert</th>
<th>Find</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Avg</td>
<td>Worst</td>
</tr>
<tr>
<td>Sorted array</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sorted Linked list</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Queue</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Skip list</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BST</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AVL/RBT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Min-heap</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hash table</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B-trees</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Data structure Comparison

Which of the following pairs of data structures, which of the pair is the better choice for.

• Inserting a list of *sorted* elements (worst case):
  A. AVL tree
  B. Binary search tree
  C. They are about equal

• Ease of implementation (assume it’s not built in):
  A. Skip list
  B. Red-Black tree (RBT)
  C. about equal

• In-order traversal of elements:
  A. Hashtable
  B. Binary search tree
  C. They are about equal
Data structure Comparison

Which of the following pairs of data structures, which of the pair is the better choice for.

• Smallest average-case (Big-O) time to find an element:
  A. Hashtable
  B. AVL tree
  C. They are about equal

• Fastest *actual* time to find an element from secondary storage (NOT big-O)
  A. RBT
  B. AVL tree
  C. B-trees
  D. They are all about equal

• Requires less space:
  A. Multi-way trie
  B. Ternary tree
  C. They are about equal
Good luck with the final!