CSE 100: B+ TREE, 2-3 TREE, NP-COMPLETENESS
Analyzing find in B-trees

• Since the B-tree is perfectly height-balanced, the worst case time cost for find is $O(\log N)$

• Best case: If every internal node is completely filled
  • $N \leq m^H - 1$
  • $H \geq \log_m(N+1)$  

• Worst case ($d = \lceil m/2 \rceil$)
  • $N \geq 2^d (H-1) - 1$
  • $H \leq \log_d((N+1)/2)$

$H = 1$ for root node
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 ceil(L/2) 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 ceil(M/2) 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).
Designing a B+ tree

• Suppose each key takes K bytes, and each pointer to a child node takes P bytes
  • then each internal node must be able to hold $M*P + (M-1)*K$ bytes

• Suppose each data record takes R bytes
  • then each leaf node must be able to hold $R * L$ bytes

• If a disk block contains B bytes, then you should have M and L as large as possible and still satisfy these inequalities ( assume $R \leq B$; if not, you can use multiple blocks for each leaf ):
  • $B \geq M*P + (M-1)*K$, and so: $M \leq \frac{(B+K)}{(P+K)}$
  • $B \geq R*L$, and so: $L \leq \frac{B}{R}$
Designing B+ tree

• Suppose each data record takes 6 bytes and each disk block is 36 bytes. What should be the value of L in the B+ tree?
  
  A. 1  
  B. 4  
  C. 5  
  D. 6
Designing B+ tree

• Suppose both the key and the child pointer use 4 bytes and each disk block is 36 bytes. What should be the value of M in the B+ tree?
  
  A. 1
  B. 4
  C. 5
  D. 6
Designing a B+-tree, cont’d

• The branching factor in a B+-tree is at least $M/2$ (except possibly at the root), and there are at least $L/2$ records in each leaf node.
• So, if you are storing $N$ data records, there will be at most $2N/L$ leaf nodes.
• There are at most $(2N/L) / (M/2)$ nodes at the level above the leaves; etc.
• Therefore: the height of the B-tree with $N$ data records is at most

$$\log_{M/2} (2N/L) = \log_{M/2} N - \log_{M/2} L + \log_{M/2} 2$$

• In a typical application, with 32-bit int keys, 32-bit disk block pointers, 1024-byte disk blocks, and 256-byte data records, we would have:
  • $M = 128, L = 4$
  • And the height of the B+-tree storing $N$ data records would be at most

$$\log_{64} N$$

• How does all this relate to the performance of operations on the B+-tree?
Analyzing find in B+-trees: more detail

- Assume a two-level memory structure: reading a disk block takes $T_d$ time units; accessing a variable in memory takes $T_m$ time units
- The height of the tree is at most $\log_{M/2} N$; this is the maximum number of disk blocks that need to be accessed when doing a find
  - so the time taken with disk accesses during a find is at most $T_d \times \log_{M/2} N$
- The number of items in an internal node is at most $M$; assuming they are sorted, it takes at most $\log_2 M$ steps to search within a node, using binary search
  - so the time taken doing search within internal nodes during a find is no more than $T_m \times \log_2 M \times \log_{M/2} N$
  - Which about the same as $T_m \times \log_2 N$ since $\log_{M/2} N = \log_2 N / (\log_2 M/2)$
- The number of items in a leaf node is at most $L$; assuming they are sorted, it takes at most $\log_2 L$ steps to search within a node, using binary search
  - so the time taken searching within a leaf node during a find is at most $T_m \times \log_2 L$
Analyzing find in B+-trees: still more detail

• Putting all that together, the time to find a key in a B-tree with parameters $M, L,$ and memory and disk access times $T_m$ and $T_d$ is at most
  
  $T_d \cdot \log_{M/2} N + T_m \cdot \log_2 M \cdot \log_{M/2} N + T_m \cdot \log_2 L$

• This formula shows the breakdown of find time cost, distinguishing between time due to disk access (the first term) and time due to memory access (the last 2 terms)

• This more complete analysis can help understand issues of designing large database systems using B+-tree design

• For example, from the formula we can see that disk accesses (the first term) will dominate the total runtime cost for large $N$ if

  \[
  \frac{T_d}{T_m} \gg \log_2 M
  \]

• This will almost certainly be true, for typical inexpensive disks and memory and reasonable choices of $M$

• And so the biggest performance improvements will come from faster disk systems, or by storing as many top levels of the tree as possible in memory instead of on disk
2-3 Tree

• Every internal node has either two or three children

• We say that T is a 2-3 tree if and only if one of the following statements hold
  • T is empty
  • T is a 2-node r with data element a. If r has left child L and right child R, then
    • L and R are non-empty 2-3 trees of the same height,
    • a is greater than each element in L, and
    • a is less than or equal to each data element in R.
  • T is a 3-node r with data elements a and b, where a < b. If r has left child L, middle child M, and right child R, then
    • L, M, and R are non-empty 2-3 trees of equal height,
    • a is greater than each data element in L and less than or equal to each data element in M, and
    • b is greater than each data element in M and less than or equal to each data element in R.
Insert in a 2-3 tree

- We will indicate leaf nodes with rectangles, and interior nodes with ellipses; and for simplicity we will just consider integer keys, with no associated data records
- Starting with an empty tree, insert 22; then 41

Insert 22: 22,

Insert 41: 22, 41

- Now insert 17...
Insert in a 2-3 tree, cont’d

- Inserting the third key would overflow the root/leaf! This is not a legal 2-3 tree:
  
  Insert 17: 

  17, 22, 41

- This node must be split into two. Since this is the root that is being split, a new root is formed: the tree grows in height
Insert in a 2-3 tree, still cont’d

• Suppose insertions have continued until this tree is produced:

```
  10, 15
   20,
     50,
    /   \
  /     \
10, 15
```

• What happens if 5 is inserted?
Now insert 65….
Insert in a 2-3 tree, still cont’d
2-3 tree converted to red-black tree
(2-3-4) B-trees
(2-3-4) B-trees == RBTs!
(2-3-4) B-trees == RBTs!

So which are asymptotically faster? RB-trees or 2-3-4 (B) trees?
A. RB trees
B. B-trees
C. They are the same
A note about graph algorithm time costs

• The graph algorithms we will study have fast algorithms:
  • Find shortest path in unweighted graphs
    • Solved by basic breadth-first search: \(O(|V|+|E|)\) worst case
  • Find shortest path in weighted graphs
    • Solved by Dijkstra’s algorithm: \(O(|E| \log|V|)\) worst case
  • Find minimum-cost spanning tree in weighted graphs
    • Solved by Prim’s or Kruskal’s algorithm: \(O(|E| \log|V|)\) worst case
• The “greedy” algorithms used for solving these problems have polynomial time cost functions in the worst case
  • since \(|E|\leq|V|^2\), Dijkstra’s, Prim’s and Kruskal’s algorithms are \(O(|V|^3)\)
• As a result, these problems can be solved in a reasonable amount of time, even for large graphs; they are considered to be ‘tractable’ problems
• However, many graph problems do not have any known polynomial time solutions...!
Intractable graph problems

- For many interesting graph problems, the best known algorithms to solve them have exponential time costs $O(2^{|V|})$

- In the worst case, these intractable problems simply cannot be solved exactly, except for quite small graphs (e.g. fewer than 100 vertices, even on the world’s fastest computers); the best known algorithms for these problems take too long to run

- For these problems, simple greedy best-first algorithms do not work... Essentially the best approach known for solving them exactly is basically to try all the possibilities, and there can be exponentially many possibilities to try

- These intractable graph problems are often members of the class called “NP-complete” problems, which includes many non-graph problems as well...
NP-complete problems

- A problem that can be exactly solved in time that is a polynomial function of the size of the problem is in the class “P” (for Polynomial time).

- A problem whose solution can be checked for correctness in time that is a polynomial function of the size of the problem is in the class “NP” (for Nondeterministic Polynomial time).
  - A “nondeterministic” computer could guess the solution to the problem, and then check if it is a solution in polynomial time, and never give a wrong answer.
  - Note that the class P is contained in NP.

- A problem that is in NP, and is as hard as any problem in NP (an algorithm for it is also essentially an algorithm for any NP problem) is NP-complete.

- For all the NP-complete problems, the best known algorithms take exponential time in the worst case...
  - ...However, nobody has yet been able to prove that there are no polynomial time algorithms for them! If you find one, you will be instantly very very famous.

- What are some of the NP-complete graph problems...
Examples of intractable graph problems

Here are a few examples of the many graph problems that are NP-complete, and so seem to require $O(2^{|V|})$ time worst-case:

- “Hamiltonian circuit”: Given a graph, say whether the graph has a cycle that includes all the vertices of the graph exactly once.
- “Travelling salesman”: Given a weighted graph, find the Hamiltonian circuit that has the smallest total cost.
- “Longest path”: Given a graph and two vertices s and d, find the longest path from s to d that doesn’t contain any cycles. (But note that “shortest path” is solvable in polynomial time!)
- “Shortest total path length spanning tree”: Given a graph, find the spanning tree that has the smallest total path lengths between all pairs of vertices
- “Steiner tree”: Given a graph $(V,E)$ and a subset $S$ of $V$, find the minimum-cost spanning tree that spans every vertex in $S$ (and may also span some other vertices) (but note that if $S=V$, the problem is solvable in polynomial time!)
The problem with intractable problems

- If a problem is NP-complete, the best known algorithms to solve it requires exponentially many steps in the worst case.
- Simple greedy algorithms do not work for these problems.
  - Backtracking, or some other way of looking at, and checking, possible alternatives is usually required...
  - ... and there are exponentially many alternatives to check!
- For example:
  - The problem has $N$ boolean variables, and you need to check the $2^N$ possible different assignments of truth values to them.
  - The problem has $N$ items, and you need to check each of the $2^N$ different subsets of those items.
- Because of the exponential time costs of the best known solutions to these problems, you have to either...
  - restrict yourself to small instances of the problems, or
  - try to find approximate algorithms that are fast, but not always exactly correct.