Lecture 17

- Simple vs. more sophisticated algorithm cost analysis
- The cost of accessing memory
- B-trees
- B-tree performance analysis
- B-tree find, insert, and delete operations
- B-tree example: a 2-3 tree

Reading: Weiss, Ch. 4 section 7

Less simple cost analysis

- In ordinary algorithmic analysis, you look at the time or space cost of doing a single operation
- Ordinary, simple algorithmic analysis does not give a useful picture in some cases
    - For example, operations on self-adjusting structures can have poor worst-case performance for a single operation, but the total cost of a sequence of operations is very good
        - Amortized cost analysis is a more sophisticated analysis which may be more appropritate in this case
- In ordinary algorithmic analysis, you also assume that any “direct” memory access takes the same amount of time, no matter what
- This also does not give a useful picture in some cases, and we will need a more sophisticated form of analysis

Memory accesses

- Suppose you are accessing elements of an array:

```
    if ( a[i] < a[j] ) {
```

- ... or suppose you are dereferencing pointers:

```
    temp->next->next = elem->prev->prev;
```

- ... or in general reading or writing the values of variables:

```
    disc = x*x / (4 * a * c);
```

- In simple algorithmic analysis, each of these variable accesses is assumed to have the same, constant, time cost
- However, in reality this assumption may not hold
- Accessing a variable may in fact have very different time costs, depending where in the memory hierarchy that variable happens to be stored

The memory hierarchy

- In a typical computer, there is a lot of memory
- This memory is of different types: CPU registers, level 1 and level 2 cache, main memory (RAM), hard disk, etc.
- This memory is organized in a hierarchy
- As you move down the hierarchy, memory is
  - cheaper,
  - slower,
  - and there is more of it
- Differences in memory speeds can be very dramatic, and so it can be very important for algorithmic analysis to take memory speed into account

Typical memory hierarchy: a picture

$$\begin{tikzcd}
\boxed{\text{CPU}} \arrow[d, shift right] \\
\text{CPU registers} \arrow[u, shift right] \arrow[d, shift right] \\
\text{cache} \arrow[u, shift right] \arrow[d, shift right] \\
\text{main memory} \arrow[u, shift right] \arrow[d, shift right] \\
\text{disk} \arrow[u, shift right]
\end{tikzcd}$$

AMOUNT OF STORAGE (approx!)	CPU	TIME TO ACCESS (approx!)
several kilobytes	CPU registers	1 nanosecond
several megabytes	cache	10 nanoseconds
several gigabytes	main memory	100 nanoseconds
several terabytes	disk	10 milliseconds

Consequences of the memory hierarchy

- Accessing a variable can be fast or slow, depending on various factors
- If a variable is in slow memory, acessing it will be slow
- However, when it is accessed, the operating system will typically move that variable to faster memory (“cache” or “buffer” it), along with some nearby variables
  - The idea is: if a variable is accessed once in a program, it (and nearby variables) is likely to be accessed again
- So it is possible for one access of a variable to be slow, and the next access to be faster; possibly orders of magnitude faster

```
x = z[i];   // if z[i] is on disk this takes a long time
z[i] = 3;    // now z[i] is in cache, so this is very fast!
z[i+1] = 9;   // nearby variables also moved, so this is fast
```

- The biggest speed difference is between disk access and semiconductor memory access, so that’s what we will pay most attention to

Accesssing data on disk

- Data on disk is organized into *blocks*
- Typical block size: 1 kilobyte (1024 bytes), 4 kilobytes (4096 bytes), or more
- Because of the physical properties of disk drives (head seek time, platter rotation time), it is approximately as fast to read an entire disk block at once as it is to read any part of the block
- So, if you access a variable stored on disk, the operating system will read the entire disk block containing that variable into semiconductor memory
- While in semiconductor memory, accessing any item in that block will be fast
- Because disk accesses are many (thousands!) of times slower than semiconductor memory accesses, if a datastructure is going to reside on disk, it is important that it can be used with very few disk accesses
- The most commonly used data structure for large disk databases is a B-tree, which can be designed to use disk accesses very efficiently

B-trees

- B-trees are search trees
- B-trees are not binary trees!
  - Depending on the design, a B-tree can have a very large branching factor (i.e. a node can have many children)
- B-trees are balanced search trees
  - In B-tree, all leaves are at the same level
  - This requires allowing nodes in the tree to have different numbers of children, in a range from a minimum to a maximum possible
- A node in a B-tree are designed to fit in one disk block
  - When a node is accessed, its entire contents are read into main memory
  - But because of the large branching factor, the tree is not deep, and only a few disk accesses are necessary to reach the leaves
- There are several variants of the basic B tree idea... we will describe B+ trees

B tree properties

- A B-tree design depends on parameters M, L which are selected to maximize performance, depending on the size of data records stored and the size of disk blocks

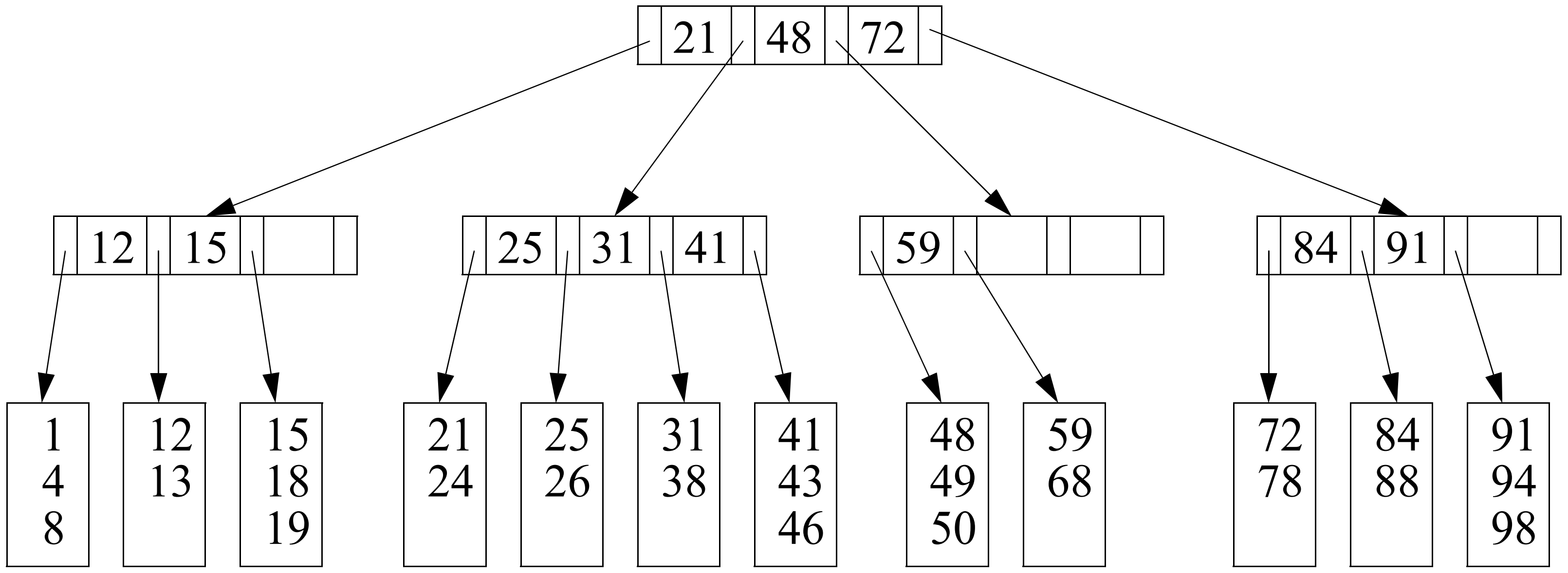
- A B-tree is designed to hold data records; each data record has a unique key (e.g. an integer) plus additional data. Keys are ordered, and guide search in the tree

- 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

- Here is an example of 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)
- In a typical actual application, M is considerably larger than 4... How to pick M, L?

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 $R<=B$; if not, you can use multiple blocks for each leaf):

  - $B >= M*P + (M-1)*K$, and so: $M <= (B+K)/(P+K)$

  - $B >= R*L$, and so: $L <= B/R$

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?

B-tree operations: find

- The find or lookup operation for a key in a B-tree is a straightforward generalization from binary search tree find
    - Read in to main memory from disk the keys in the root node
    - Do a search among these keys to find which child to branch to (note the keys in an internal node are sorted, so you can use binary search there)
    - When you visit an internal node, read in to main memory from disk the keys stored there and continue
    - When you reach a leaf, read in to main memory from disk the key/data pairs there and do a search among them for the desired key (binary search can work here also)
- So, the number of disk accesses required for a find is equal to the height of the tree, assuming the entire tree is stored on disk. But semiconductor memory accesses are required too... how do they compare to each other?

Analyzing find in B-trees

- Since the B-tree is perfectly height-balanced, the worst case time cost for find is O(logN)
- However, this simple, qualitative big-O analysis is not very informative! It doesn't say anything about choices of M, L, etc.
- Let's take a more detailed look

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 $T_d * \log_{M/2} N$
- 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 $T_m * \log_2 M * \log_{M/2} N$
- 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 $T_m * \log_2 L$

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
  - $\boxed{T_d * \log_{M/2} N} \quad + \quad \boxed{T_m * \log_2 M * \log_{M/2} N \quad + \quad T_m * \log_2 L}$
- 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

B-tree operations: insert and delete

- The find operation in a B-tree is pretty straightforward
- Insert and delete are more involved; they modify the tree, and this must be done in a way to keep the B-tree properties invariant
- In a B+ tree with parameters M,L these must be invariant:
  - 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
  - Keys in internal nodes are kept in an array, in sorted order

Insert in B-trees

- When given a key-data pair to insert in the tree, do the following:
    - Descend the tree, to find the leaf node that should contain the key
    - If there is room in the leaf (fewer than L data records are there already), put the new item there; done!
    - Otherwise, the leaf must be split into two leaves, one with floor((L+1)/2.) data items and one with ceil((L+1)/2.) data items
    - This will require modifying the parent of this split leaf:
        - the parent will gain a new child pointer, and a new key value
    - If there is room in the parent for these (fewer than M child pointers were there already), make these modifications. Done!
    - Otherwise, this internal node must be split into two internal nodes, one with floor((M+1)/2.) children and one with ceil((M+1)/2.) children
    - ...Etc.! It is possible for this process to pass all the way up the tree, splitting the root
        - This requires creating a new root, with 2 children.
        - This is the only way the height of the B-tree can increase

Delete in B-trees

- When given a key value to delete from the tree, do the following:
    - Descend the tree, to find the leaf node that contains the key
    - If the leaf has more than ceil(L/2.) data records, just remove the record with matching key. (May have to change key values in ancestor nodes too.) Done!
    - Otherwise, if one of this leaf's siblings has more than ceil(L/2.) data records, borrow a data record (which one?) from it, replacing the deleted record. (May have to change key values in ancestor nodes too.) Done!
    - Otherwise, delete the data record with matching key, and give all the leaf's remaining data records to one of its siblings (which will have room for them... why?)
    - This will require modifying the parent of this eliminated leaf: the parent will lose a child pointer
    - If at least ceil(M/2.) child pointers remain in the parent, done!
    - Otherwise, this internal node must try to borrow children from its siblings, or else give up all its children to one of them and disappear
    - Etc.! It is possible for this process to pass all the way up the tree, eliminating the root. This is the only way the height of the B-tree can decrease

Examples using a 2-3 tree

- A B-tree with M=L=3 is called a 2-3 tree
- This name comes from the fact that ceil(M/2.) = 2, so each internal node must have either 2 or 3 children
- Also, every leaf must have either 2 or 3 data records (except the root when it is a leaf could have 1 data record)
- 2-3 trees have M,L too small to be useful in disk database applications, but they are sometimes implemented as an in-memory balanced tree data structure with good performance, comparable to AVL or red-black trees
- We will use a 2-3 tree as an example for insertions in B-trees

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; then 16

Insert 22: $\boxed{22}$

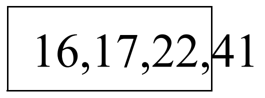
Insert 41: $\boxed{22,41}$

Insert 16: $\boxed{16,22,41}$

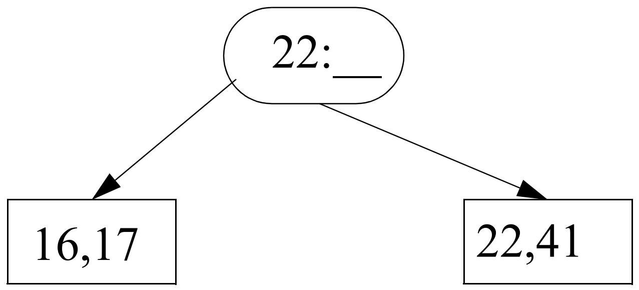
- Now insert 17...

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:

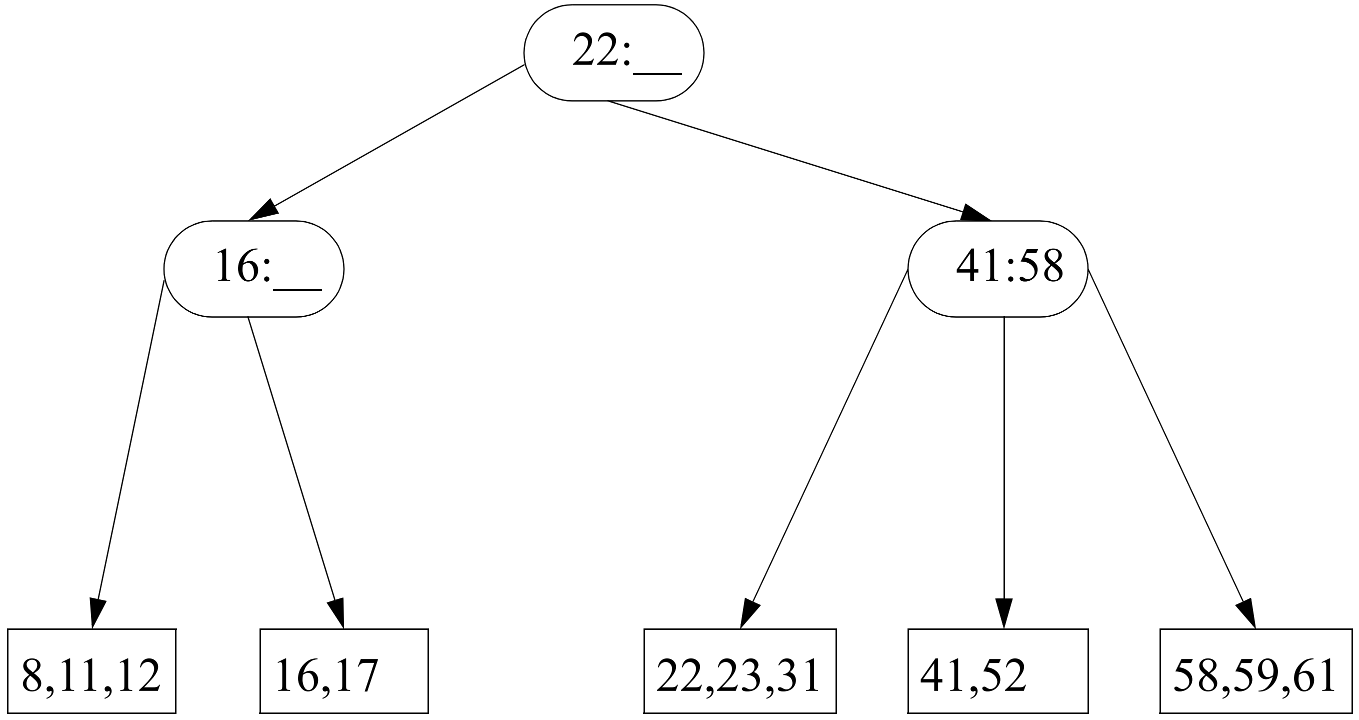


- This node must be split into two. Since this is the root that is being split, a new root is formed: the tree grows in height



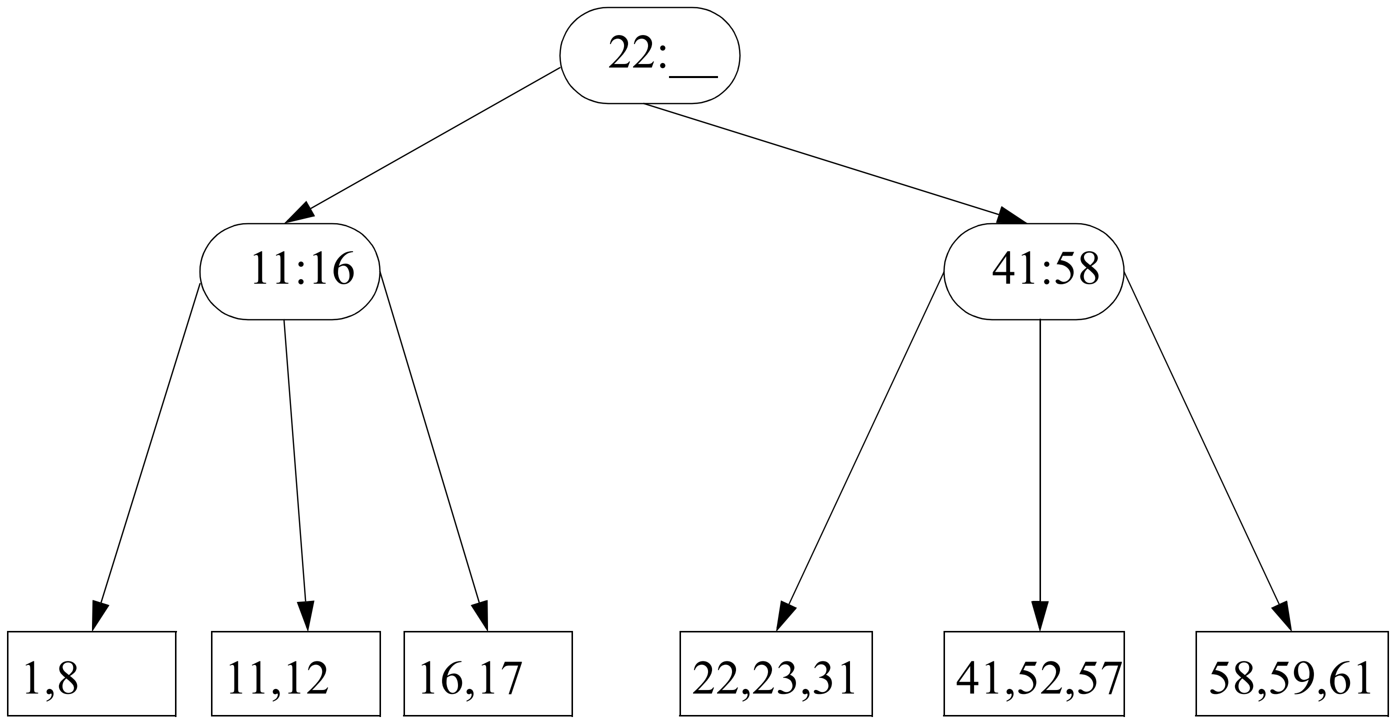
Insert in a 2-3 tree, still cont’d

- Suppose insertions have continued until this tree is produced:



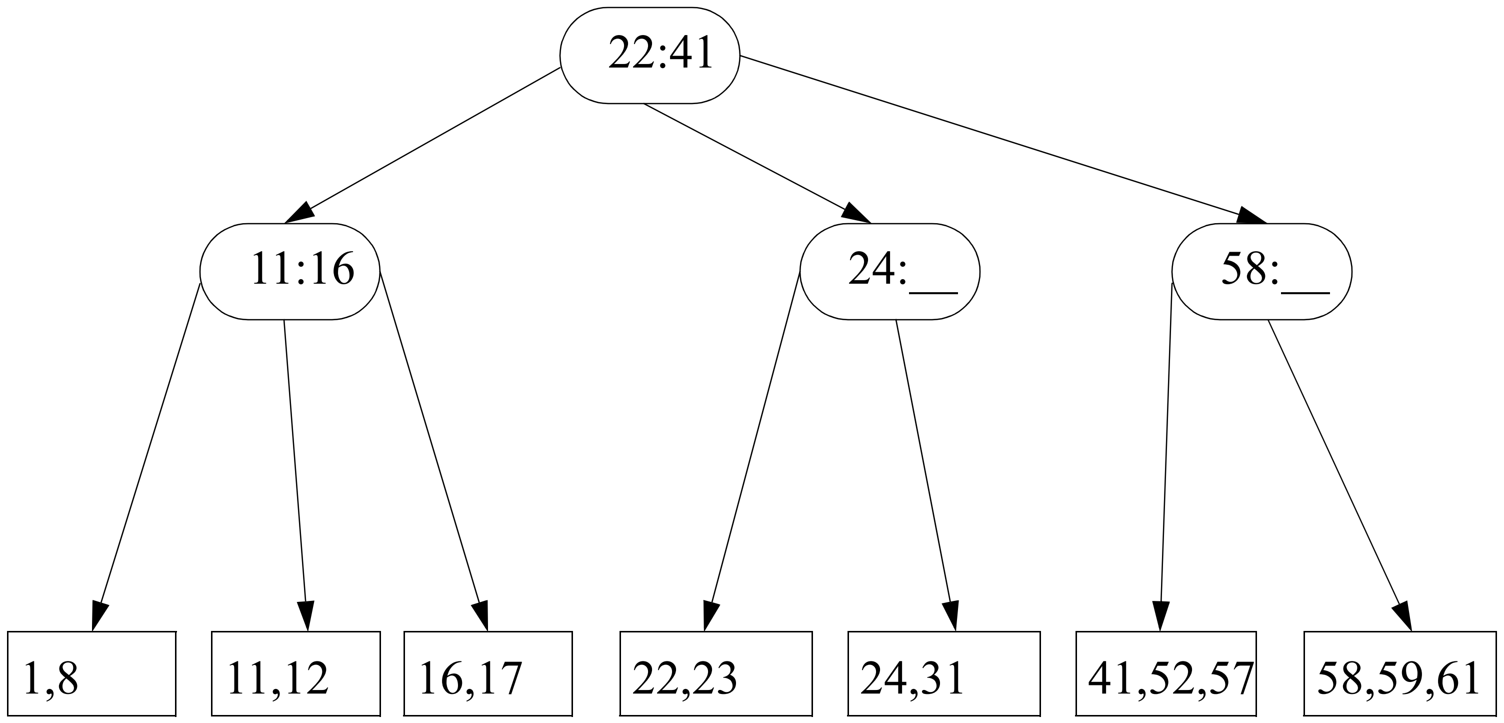
- What happens if 57 is inserted?
- What happens if 1 is inserted?

After inserting 57 and 1



- Now what happens if 24 is inserted?

After also inserting 24



- Now what happens if 55 is inserted? The root will be split, and the tree will grow in height by 1

Next time...

- Self-organizing data structures
- Self-organizing lists
- Splay trees
- Spatial data structures
- K-D trees
- Final Review