

MA/CSSE 473

Day 29

Hashing review

B-tree overview

Dynamic
Programming



MA/CSSE 473 Day 29

- HW 12 due Thursday
- Exam 2 is Friday, Oct 31
 - Covers through Chapter 7 and HW12.
- **Student Questions**
- Hashing: Review of algorithms and analysis
- B-Trees – a quick look
- Dynamic Programming



Hashing Review

- What problem do we try to solve by hashing?
- What is the brute force approach?
- What alternatives have we seen?
- What is the general idea of how hashing works?
- Why does it fit into Chapter 7?
- What are the main issues to be addressed when discussing hashing implementation?



Terminology and analysis

- collision
- load factor (λ)
- separate chaining
- open addressing
- linear probing
- cluster
- quadratic probing
- rehashing
- expected lookup time (as a function of λ)
 - For successful search
 - For unsuccessful search



Some Hashing Details

- The next slides are from CSSE 230.
- They are here in case you didn't "get it" the first time.
- We will not go over all of them in detail in class.
- If you don't understand the effect of clustering, you might find the animation that is linked from the slides especially helpful.



Collision Resolution: Linear Probing

- When an item hashes to a table location occupied by a non-equal item, simply use the next available space.
- Try $H+1$, $H+2$, $H+3$, ...
 - With wraparound at the end of the array
- Problem: Clustering (picture on next slide)
- http://www.cs.auckland.ac.nz/software/AlgAnim/hash_tables.html
- We'll let it keep running while we look at analysis.



```

hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9

```

After insert 89 *After insert 18* *After insert 49* *After insert 58* *After insert 9*

0			49	49	49
1				58	58
2					9
3					
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89

Figure 20.4
 Linear probing hash
 table after each
 insertion

Analysis of linear probing

- Dependent on the **load factor**, λ , which is the ratio of the number of items in the table to the size of the table. Thus $0 \leq \lambda \leq 1$.
- For a given λ , what is the expected number of probes before an empty location is found?
- For simplicity, assume that all locations are equally likely to be occupied, and equally likely to be the next one we look at. Then the probability that a given cell is empty is $1 - \lambda$, and thus the expected number of probes before finding an empty cell is (write it as a summation).

```
> simplify(sum(i*(1-lambda)*lambda^(i-1), i=1..infinity));
```

$$-\frac{1}{\lambda - 1}$$



Analysis of linear probing (continued)

- The "equally likely" probability is not realistic, because of **clustering**
- Large blocks of consecutive occupied cells are formed. Any attempt to place a new item in any of those cells results in extending the cluster by at least one item
- Thus items collide not only because of identical hash values, but also because of hash values that happen to put them into the cluster
- Average number of probes when λ is large:
 - $0.5 [1 + 1/(1 - \lambda)^2]$.
 - For a proof, see Knuth, *The Art of Computer Programming*, Vol 3: Searching Sorting, 2nd ed, Addison-Wesley, Reading, MA, 1998.
 - What are the values for $\lambda = 0, 0.5, 0.75, 0.9$?
 - **When λ approaches 1, this gets bad!**
 - But if λ is close to zero, then the average is near 1.0



So why consider linear probing?

- Easy to implement
- Simple code has fast run time per probe
- Works well when load factor is low
 - It could be more efficient just to rehash using a bigger table once it starts to fill.
 - What is often done in practice: rehash to an array that is double in size once the load factor reaches 0.5
- What about other fast, easy-to-implement strategies?



Quadratic probing

- With linear probing, if there is a collision at H , we try $H, H+1, H+2, H+3, \dots$ until we find an empty spot.
 - Causes (primary) clustering
- With quadratic probing, we try $H, H+1^2, H+2^2, H+3^2, \dots$
 - Eliminates primary clustering, but can cause secondary clustering.



Hints for quadratic probing

- **Choose a prime number for the array size**

- If the array used for the table is not more than half full, finding a place to do the insertion is guaranteed, and no cell is probed twice
- Suppose the array size is P , a prime number greater than 3
- Show by contradiction that if i and j are $\leq \lfloor P/2 \rfloor$, and $i \neq j$, then $H + i^2 \pmod{P} \neq H + j^2 \pmod{P}$.

- **Use an algebraic trick to calculate next index**

- Replaces mod and general multiplication with subtraction and a bit shift
- Difference between successive probes:
 - $H + (i+1)^2 = H + i^2 + (2i+1)$ [can use bit-shift for the multiplication].
 - `nextProbe = nextProbe + (2i+1);`
if (`nextProbe >= P`) `nextProbe -= P;`



Quadratic probing analysis

- No one has been able to analyze it
- Experimental data shows that it works well
 - Provided that the array size is prime, and is the table is less than half full

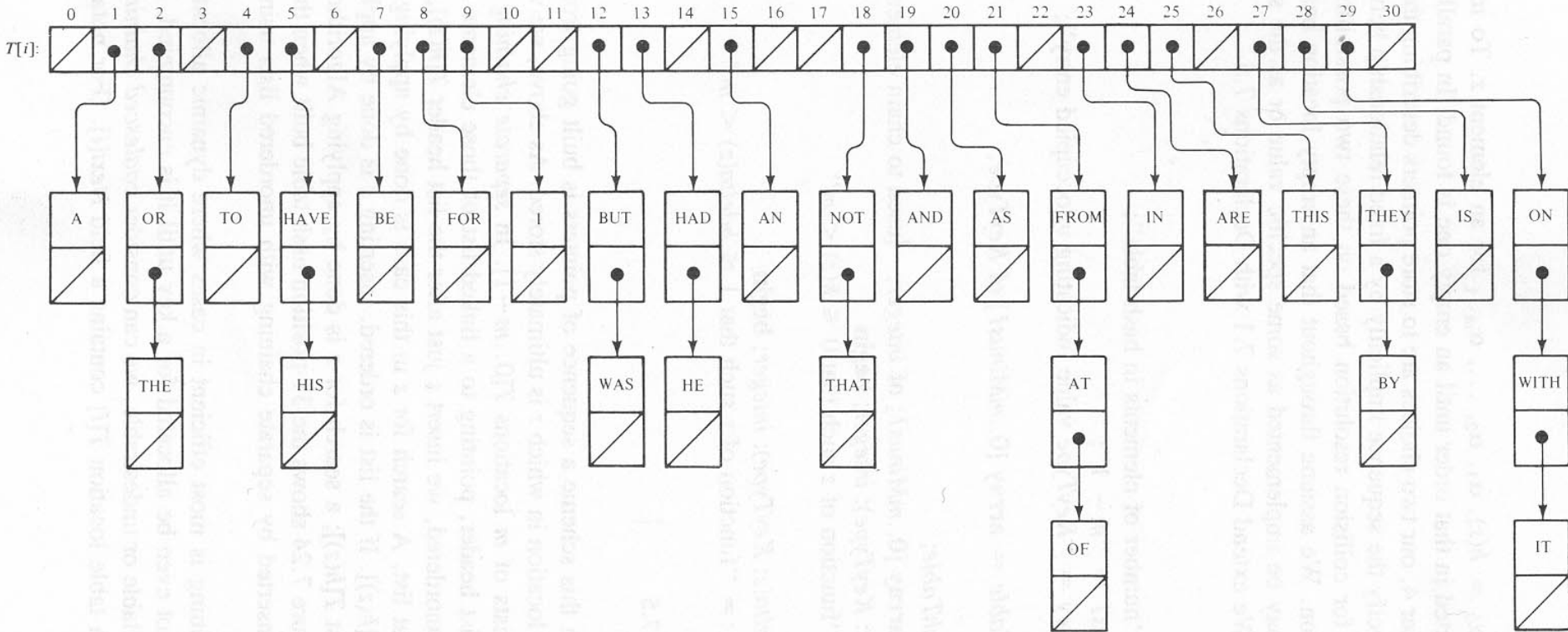


Other approaches to collision resolution

- Double hashing
 - A second hash function is used to calculate an offset d to use in probing. Try locations $h+d$, $h+2d$, $h+3d$, etc
- Separate chaining
 - Rather than an array of items, we use an array of linked lists. When multiple items hash to the same location, we add them to the list for that location
 - Picture on next slide
 - No clustering effect
 - But we use space (that could have been used to make the array larger) for the links.
 - If many items have the same hash code, the chains can become long.



Hashing with Chaining



Analysis: Hashing with Chaining

- With chaining, the load factor may be > 1 .
- Assume a hash function that distributes keys evenly in the table. If there are n keys in the table (backed by an array of size m), the average chain should be λ elements long
- So it takes constant time to compute the hash function plus $\lambda / 2$ to search within the chain.
- If $\lambda \approx 1$, this is VERY fast
- But there is the extra space for the pointers, which could have been used to make the table larger if open addressing was used



B-trees

- We will do a quick overview here.
- For the whole scoop on B-trees (Actually B+ trees), take CSSE 433, Advanced Databases.
- Nodes can contain multiple keys and pointers to other to subtrees



B-tree nodes

- Each node can represent a block of disk storage; pointers are disk addresses
- This way, when we look up a node (requiring a disk access, we can get a lot more information than if we used a binary tree
- In an n -node of a B-tree, there are $n-1$ pointers to subtrees, and thus $n-1$ keys

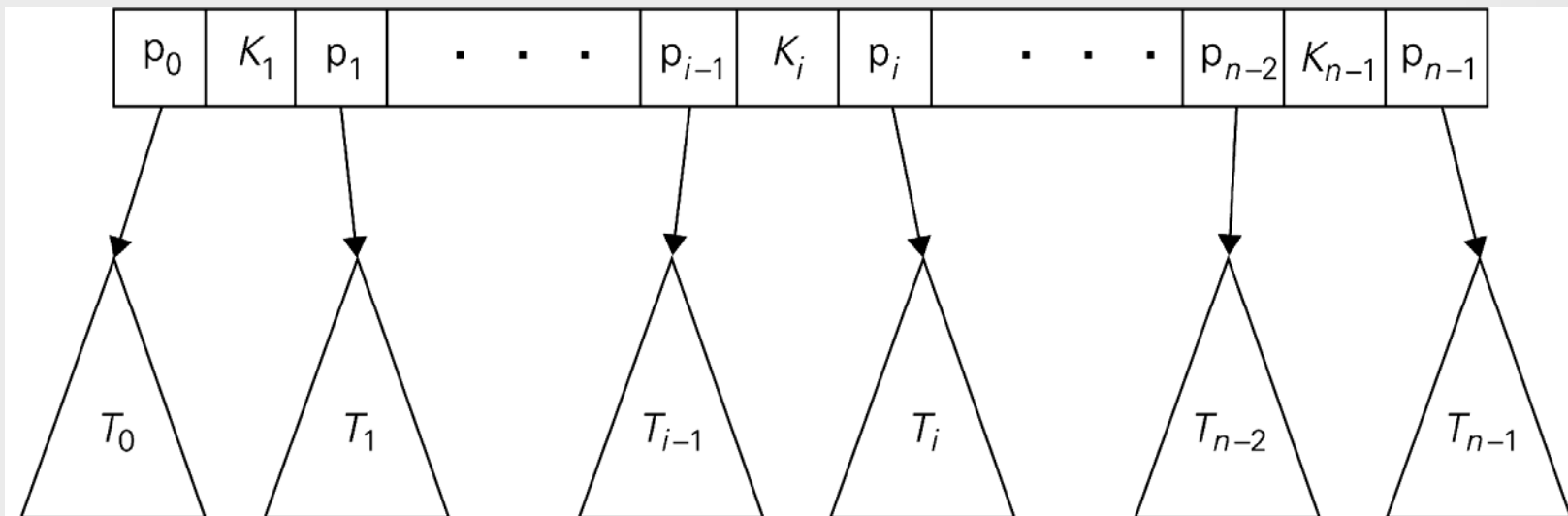


FIGURE 7.7 Parental node of a B-tree

Order of keys and subtrees

- All keys in T_i are $\geq K_i$ and $< K_{i+1}$
 K_i is the smallest key that appears in T_i
- This way, when we look up a node (requiring a disk access, we can get a lot more information than if we used a binary tree
- In an n -node of a B-tree, there are n pointers to subtrees, and thus $n-1$ keys

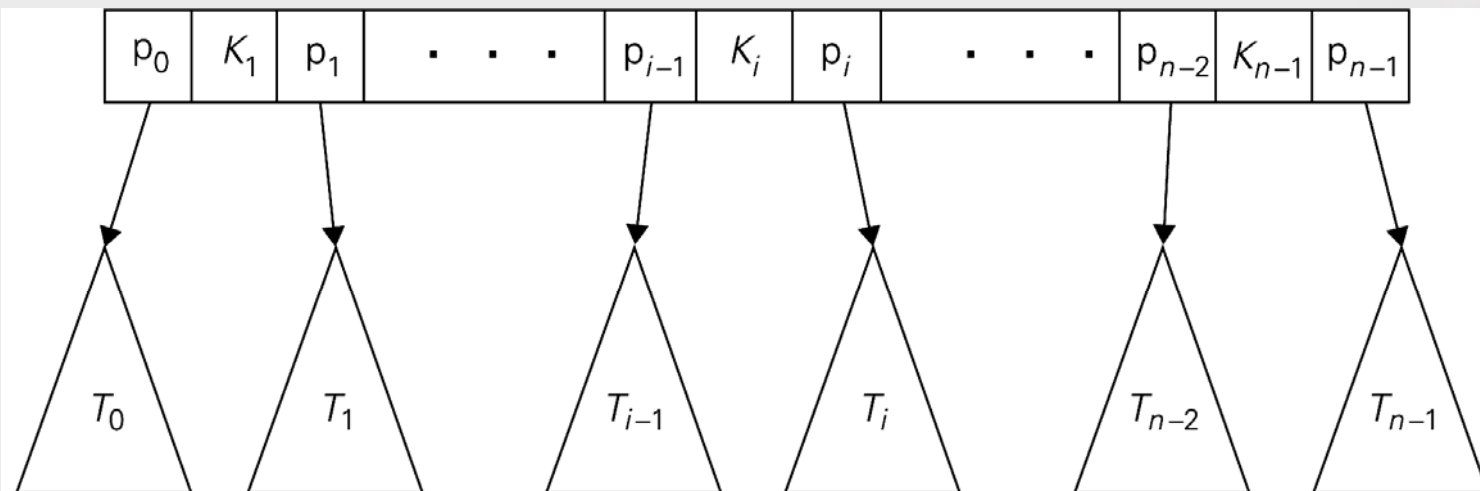


FIGURE 7.7 Parental node of a B-tree

B-tree nodes (tree of order m)

- All nodes have at most $m-1$ keys
- All keys and associated data are stored in special leaf nodes (that thus need no child pointers)
- The other (parent) nodes are index nodes
- All index nodes except the root have between $\lceil m/2 \rceil$ and m children
- root has between 2 and n children
- All leaves are at the same level
- The space-time tradeoff is because of duplicating some keys at multiple levels of the tree.
- Example on next slide



Example B-tree(order 4)

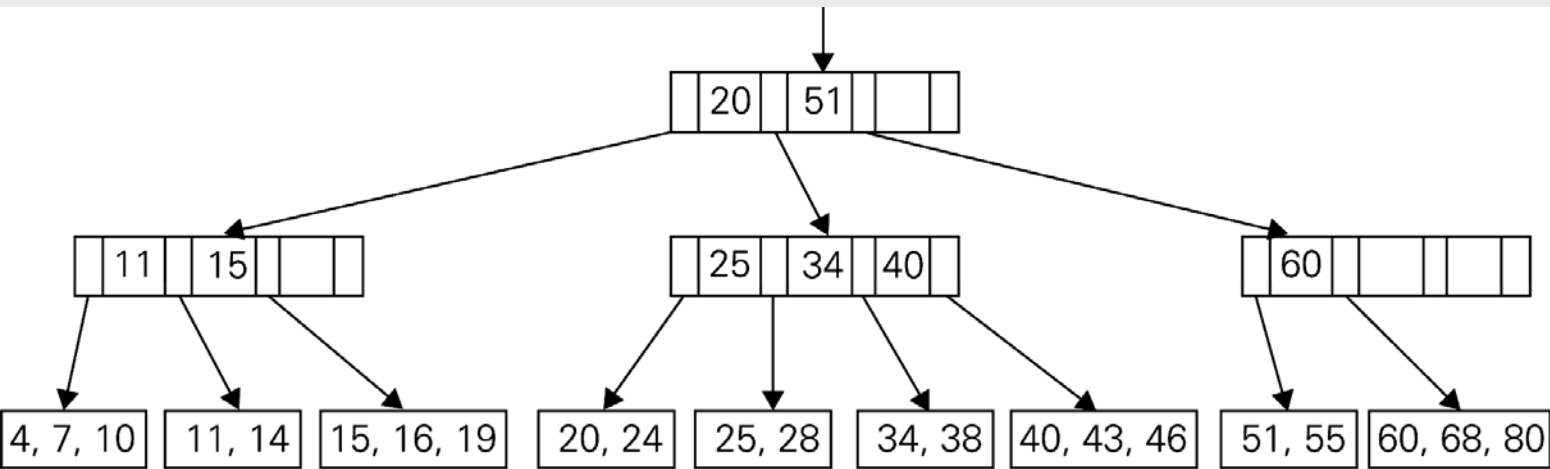


FIGURE 7.8 Example of a B-tree of order 4

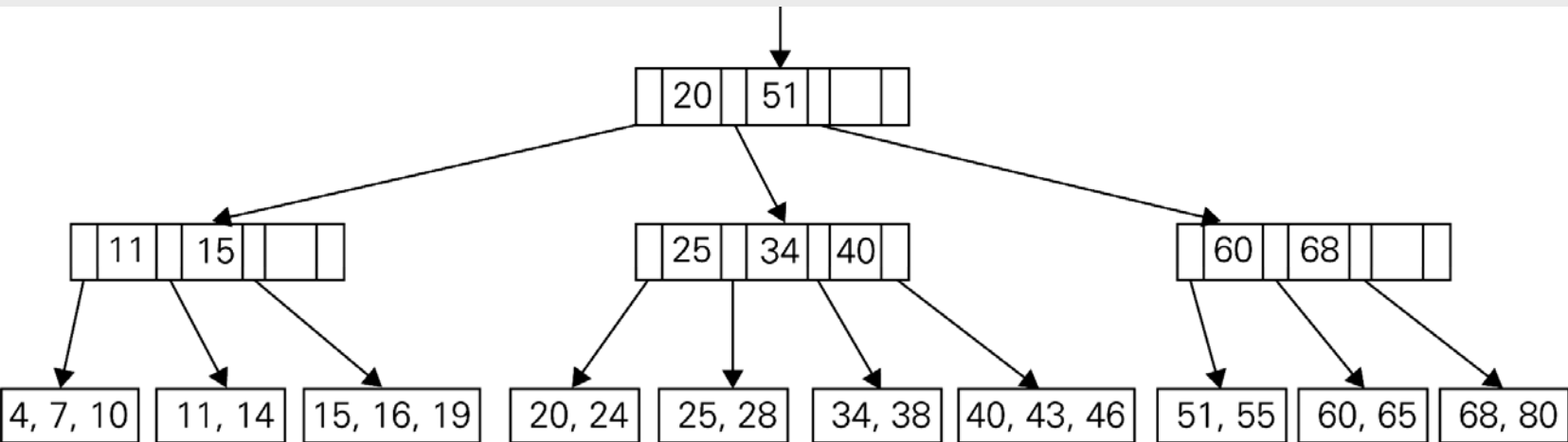


FIGURE 7.9 B-tree obtained after inserting 65 into the B-tree in Figure 7.8



B-tree Animation

- <http://slady.net/java/bt/view.php?w=800&h=600>



Search for an item

- Within each parent or leaf node, the items are sorted, so we can use binary search ($\log m$), which is a constant with respect to n , the number of items in the table
- Thus the search time is proportional to the height of the tree
- Max height is approximately $\log_{\lceil m/2 \rceil} n$
- **Exercise for you:** Read and understand the straightforward analysis on pages 273-274
- Insert and delete are also proportional to height of the tree



Dynamic programming

- Used for problems with overlapping subproblems
- Typically, we save (memoize) solutions to the subproblems, to avoid recomputing them.

