CSE 332
Review Slides

Tyler Robison

Summer 2010

Terminology

- Abstract Data Type (ADT)
  - Mathematical description of a “thing” with set of operations on that “thing”; doesn’t specify the details of how it’s done
    - Ex, Stack: You push stuff and you pop stuff
      - Could use an array, could use a linked list
- Algorithm
  - A high level, language-independent description of a step-by-step process
    - Ex: Binary search
- Data structure
  - A specific family of algorithms & data for implementing an ADT
    - Ex: Linked list stack
- Implementation of a data structure
  - A specific implementation in a specific language

Big Oh's Family

- Big Oh: Upper bound: O(f(n)) is the set of all functions asymptotically less than or equal to f(n)
  - g(n) is in O(f(n)) if there exist constants c and n_0 such that

$$g(n) \le c\, f(n) \text{ for all } n \ge n_0$$

- Big Omega: Lower bound: Ω(f(n)) is the set of all functions asymptotically greater than or equal to f(n)
  - g(n) is in Ω(f(n)) if there exist constants c and n_0 such that

$$g(n) \ge c\, f(n) \text{ for all } n \ge n_0$$

- Big Theta: Tight bound: θ(f(n)) is the set of all functions asymptotically equal to f(n)
  - Intersection of O(f(n)) and Ω(f(n)) (use *different c* values)

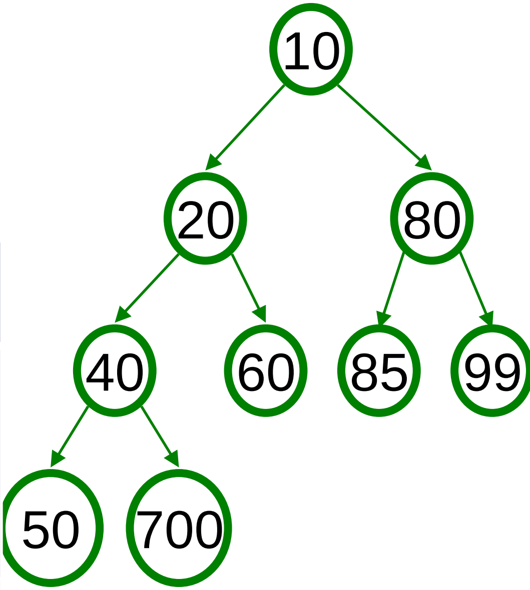
Common recurrence relations

$T(n) = O(1) + T(n-1)$	linear
$T(n) = O(1) + 2T(n/2)$	linear
$T(n) = O(1) + T(n/2)$	logarithmic
$T(n) = O(1) + 2T(n-1)$	exponential
$T(n) = O(n) + T(n-1)$	quadratic
$T(n) = O(n) + T(n/2)$	linear
$T(n) = O(n) + 2T(n/2)$	$O(n \log n)$

- Solving to a closed form (summary):
  - Ex: $T(n)=2+T(n-1)$, $T(1)=5$
  - Expand: $T(n)=2+T(n-1)=2+2+T(n-2)=\ldots=2+2+2+\ldots+2+5$
  - Determine # of times recurrence was applied to get to the base case; call it k
  - $T(n)=2(k-1)+5=2k+3$
  - Determine k in terms of n; here $k=n$; plug into equation
  - $T(n)=2n+3$

Binary Heap: Priority Queue DS

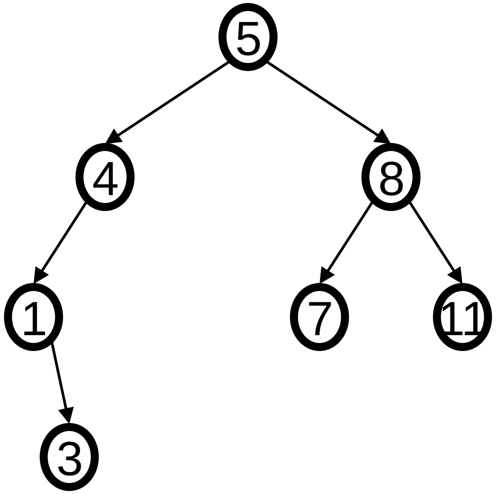
- Structure property : A complete binary tree
- Heap ordering property: For every (non-root) node the parent node’s value is less than the node’s value
- Array representation; index starting at 1
- Poor performance for general ‘find’



Operation	Description	Run-time
Insert	Place in next available spot; percUp	O(logn) worst; O(1) expected
DeleteMin	Remember root value; place last node in root; percDown	O(logn)
BuildHeap	Treat array as heap; percDown elements index <= size/2	O(n)

Binary Search Tree: Dictionary ADT

- Structure property : Binary tree; values in left subtree < this node’s value; values in right subtree > this node’s value
- Height O(logn) if balanced; O(n) if not
- No guarantees on balance

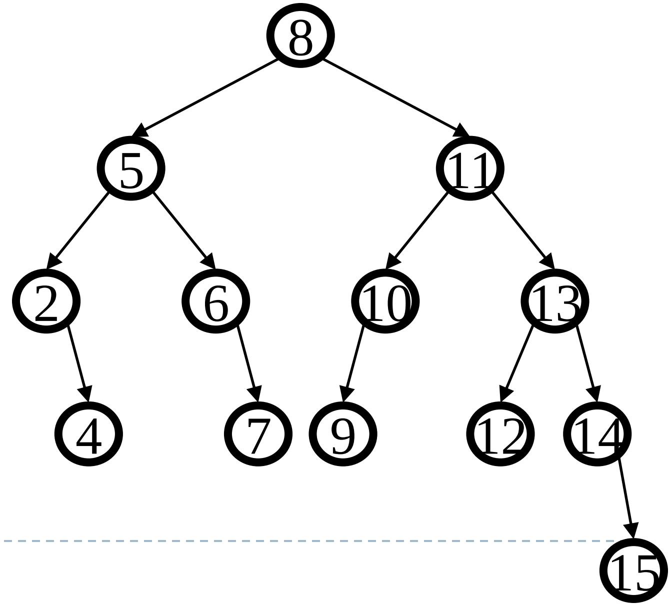


Operation	Description	Run-time
Find	Check my value against node’s: go left or right	O(n) worst
Insert	Traverse like in find; create new node there	O(n) worst
Delete	Traverse like in find; 3 cases: has no children, 1 child or 2 children	O(n) worst

AVL Tree: Dictionary ADT

- Structure property : BST
- Balance property: |left.height-right.height|<=1
- Balance guaranteed; O(logn) height
- Perform O(1) rotations to fix balance; at most one required per insert
- 4 rotation cases; depend on
  - At what node the imbalance is detected
  - At which of 4 subtrees the insertion was performed, relative to the detecting node

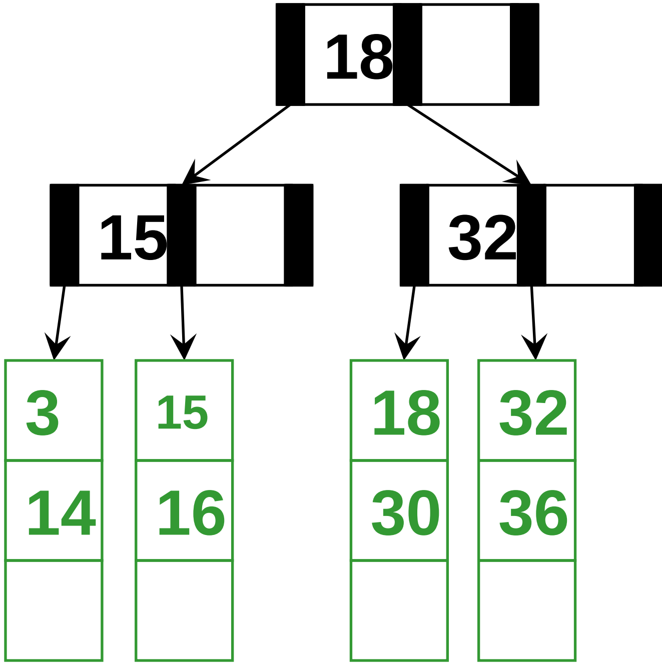
Operation	Description	Run-time
Find	BST find	O(logn)
Insert	BST insert, then recurse back up, check for imbalance & perform necessary rotations	O(logn)



B-Tree: Dictionary ADT

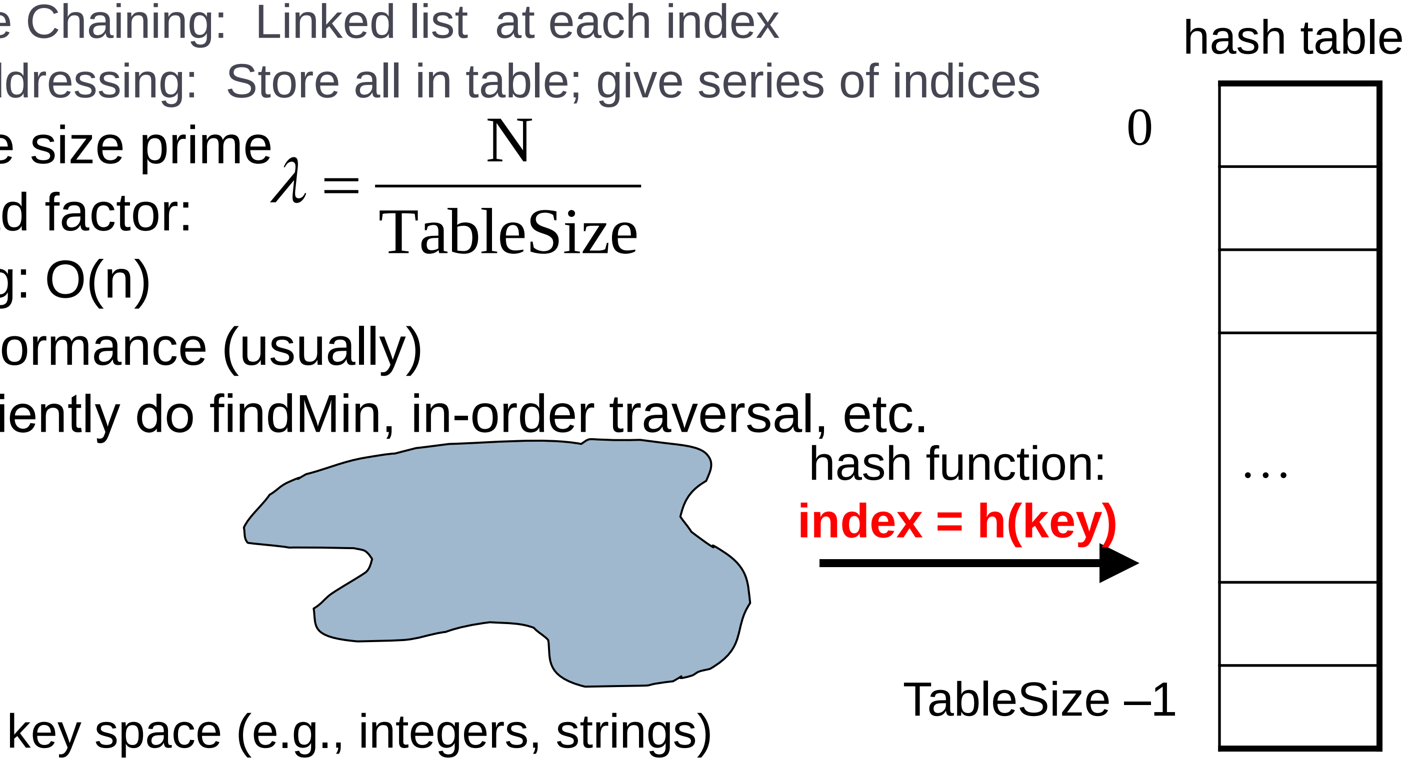
- 2 constants: M & L
  - Internal nodes (except root) have between $\lceil M/2 \rceil$ and M children (inclusive)
  - Leaf nodes have between $\lceil L/2 \rceil$ and L data items (inclusive)
  - Root has between 2 & M children (inclusive); or between 0 & L data items if a leaf
  - Base M & L on disk block size
- All leaves on same level; all data at leaves
- If in child branch, value is >= prev key in parent, < next key
- Goal: Shallow tree to reduce disk accesses
- Height: $O(\log_M n)$

Operation	Description	Run-time
Find	Binary Search to find which child to take on each node; Binary Search in leaf to find data item	$O(\log_2 M \log_M n)$
Insert	Find leaf; insert in sorted order; if overflow, split leaf; if parent overflows, split parent; may need to recursively split all the way to root	$O(L + M \log_M n)$ worst (split root) $O(L + \log_2 M \log_M n)$ expected
Delete	Find leaf; remove value, shift others as appropriate; if underflow, adopt and/or merge; may need to merge all the way to the root	$O(L + M \log_M n)$ worst (replace root) $O(L + \log_2 M \log_M n)$ expected



Hash tables (in general): Dictionary ADT (pretty much)

- Store everything in an array
- To do this, provide a mapping from key to index
  - Ex: “Jean Valjean” → 24601
- Keyspace >> tablesize; need to deal with ‘collisions’; we consider 2 varieties:
  - Separate Chaining: Linked list at each index
  - Open Addressing: Store all in table; give series of indices
- Keep table size prime
- Define load factor: $\lambda = \dfrac{N}{\text{TableSize}}$
- Rehashing: O(n)
- Great performance (usually)
- Can’t efficiently do findMin, in-order traversal, etc.



hash table

0

…

TableSize –1

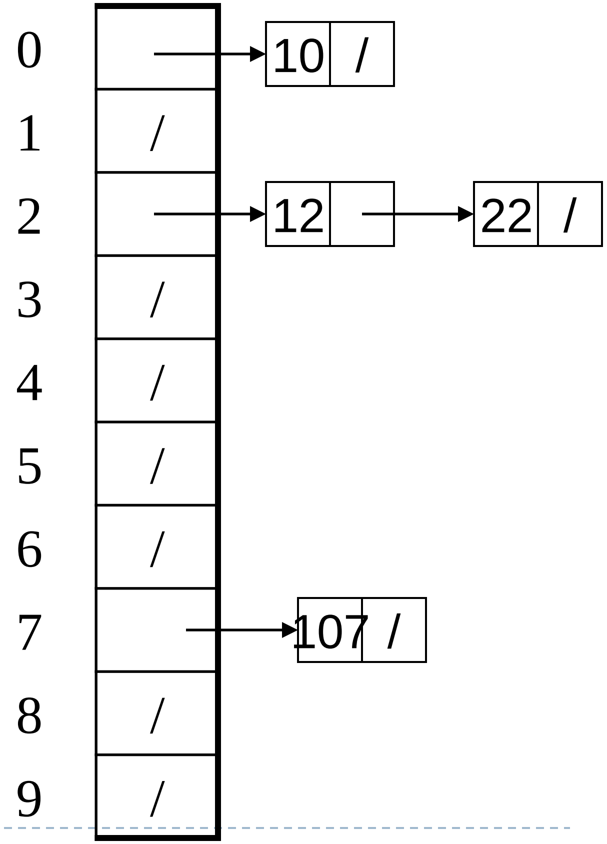
hash function:

index = h(key)

key space (e.g., integers, strings)

Hash tables: Separate Chaining

- Each array cell is a ‘bucket’ that can store several items (say, using a linked sort); each (conceptually) boundless
- λ: average # items per bucket
- λ can be greater than 1



Oper-ation	Description	Run-time
Find	Go to list at index, step through until we find correct item or reach the end	O(λ) expected O(n) worst
Insert	Go to list at index, insert at start	O(1)
Delete	Go to list at index, find and delete	O(λ) expected O(n) worst

Hash tables: Open Addressing

- Keep all items directly in table
- ‘Probe’ indices according to
  `(h(key) + f(i)) % TableSize`
  where i is the # of the attempt (starting at i=0)
- Linear probing: f(i)=i
  - Will always find a spot if one is available
  - Problem of primary clustering
- Quadratic probing: $f(i)=i^2$
  - Will find space if $\lambda<1/2$ & TableSize is prime
  - Problem of secondary clustering
- Double Hashing: f(i)=i*g(key)
  - Avoids clustering problems (if g is well chosen)
  - g(key) must never evaluate to 0
- $\lambda=1$ means table is full; no inserts possible

Operation	Description	Run-time
Find	Probe until found (success) or empty space hit (fail)	O(1), O(n); specific estimates in slides
Insert	Probe until found – replace value, or until empty - place at that index	O(1), O(n); specific estimates in slides
Delete	Use lazy deletion	Same as find

0	8
1	109
2	10
3	/
4	/
5	/
6	/
7	/
8	38
9	19

Insert: 38, 19, 8, 109, 10
Using Linear Probing