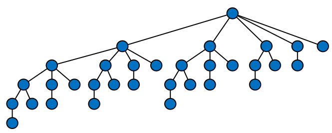
Lecture 8. Binary Search Trees

CpSc 8400: Algorithms and Data Structures

Brian C. Dean



School of Computing
Clemson University
Spring, 2016

Dictionary / Set Data Structures

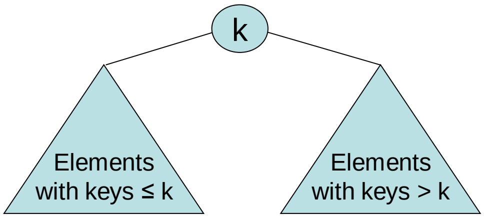
- A dictionary maintains a set of elements with associated keys, supporting the operations:
  - *find*(k) : return a pointer to the element having key k, or indicates that such an element does not exist.
  - *insert*(e, k) : insert a new element e with key k.
  - *delete(e)* : delete element e, given a pointer to e.

    To delete an element given its key: *delete*(*find*(k)).
- Common dictionary implementations:
  - Arrays and linked lists: not very efficient (at least one of the above operations takes Θ(n) worst-case time).
  - Balanced BSTs: all ops take O(log n) time.
  - Universal Hash tables: O(1) (expected) time per op.

2

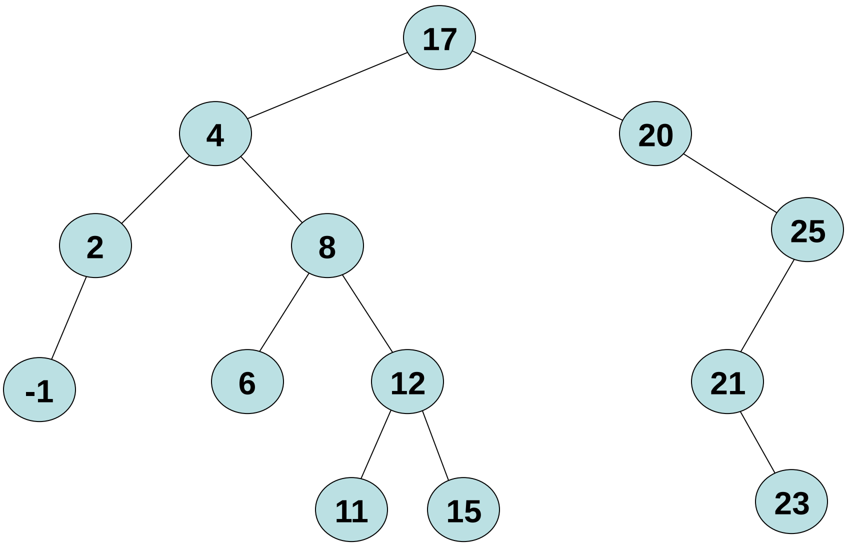
Binary Search Trees

- A BST is a binary tree satisfying the **binary search tree property**:



- Each node typically maintains a pointer to its parent, left child, and right child.
- Tree is **balanced** if height = O(log n).

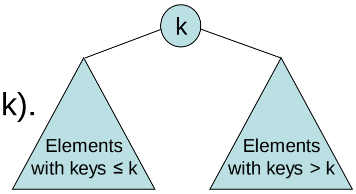
Example



Fundamental Operations

Most BST operations can be implemented in a simple recursive fashion in O(height) time (O(log n) if balanced). For example:

- **Find(T, k):**
  if T == NULL, then return NULL.
  if k == T.key, then return T.
  if k < T.key, then return Find(T.left, k).
  else return Find(T.right, k).



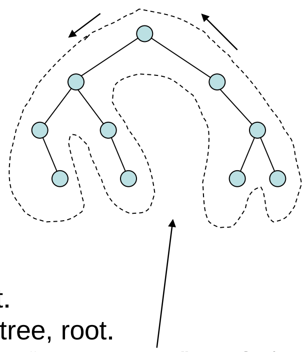
- **Insert(T, e, k):**
  if T == NULL, then return NewNode(e, k).
  if k < T.key, then T.left = Insert(T.left, e, k).
  else T.right = Insert(T.right, e, k).
  return T.

5

Tree Traversals

- We often enumerate the contents of an n-element BST in O(n) time using an **inorder** tree traversal:

  Inorder(T):
  if T == NULL, then return.
  Inorder(T.left).
  print T.key.
  Inorder(T.right).



- Other types of common traversals:
  - Preorder: root, left subtree, right subtree.
  - Postorder: left subtree, right subtree, root.
  - Eulerian: root, left subtree, root, right subtree, root.
    (sequence of nodes encountered when we “walk around” a BST)

6

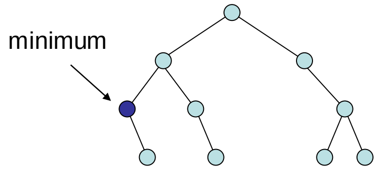
Sorting with a BST

- An inorder traversal prints the contents of an n-element BST **in sorted order** in O(n) time.
- Therefore, we can use BSTs to sort: *insert* n elements, then do an inorder traversal.
- Since the BST is a comparison-based data structure, this means *insert* must run in Ω(log n) time in the worst case; otherwise we could sort faster than O(n log n) in the comparison model.

7

Minimum and Maximum

- It's easy to locate the minimum and maximum elements in a BST in O(h) time.
- For the minimum, start at the root and walk left as long as possible:

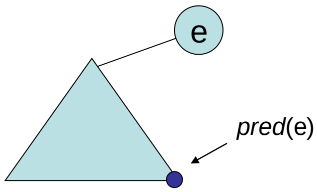


- Vice-versa for the maximum…

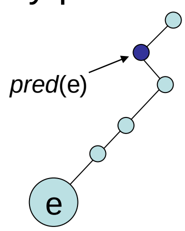
8

Predecessor and Successor

- The *pred*(e) operation takes a pointer to e and locates the element immediately preceding *e* in an inorder traversal.



If e has a left child, then *pred*(e) is the maximum element in *e's* left subtree.



If e has no left child, then *pred*(e) is the first "left parent" we encounter when walking from e up to the root.

- The successor operation, succ(e), is completely symmetric to this.

9

Deletion

- It's easy to *delete* an element with zero or one children.
- To *delete* an element e with two children, first swap e with either *pred*(e) or *succ*(e), then proceed to delete e.
  - Note that if e has two children, then *pred*(e) and *succ*(e) can both have at most one child.
  - Also note that replacing e with *pred*(e) or *succ*(e) is o.k. according to the BST property.
- Can also replace e with the *join* of e's two subtrees.

10

Using pred and succ for Inexact Searches

- In addition to *pred*(e) and *succ*(e) that take pointers to elements, we could implement:
  - *pred*(k) : find the element with largest key ≤ k.
  - *succ*(k) : find the element with smallest key ≥ k.
- This allows us to perform inexact searches: if an element with key k isn't present, we can at least find the closest nearby elements in the BST.
- This is one reason a BST is so powerful. Other dictionary data structures, notably hash tables, don't support *pred / succ* and cannot perform inexact searches.

Paging Through Nearby Elements

- Starting from element e, we can use *pred / succ* to find elements near e very quickly.
- Prototypical application: library catalog.
  - After searching for an author's name, you are presented with a alphabetized list in which you can scroll through nearby names.
- Can also answer **range queries**: output all elements whose keys are in the range [a, b].
  - Start at the element e = *succ*(a). Then repeatedly call *succ*(e) until you reach b.
  - Running time O(k + log n), where k = # of elements written as output.

Rank-Based Access

- A BST augmented with subtree sizes can support two additional useful operations:
  - *select*(k) : return a pointer to the kth largest element in the tree (k = 1 is the min, k = n is the max, and k = n/2 is the median).
  - *rank*(e) : given a pointer to e, returns the number of elements with keys ≤ e's key.

    (that is, e's index within an inorder traversal).
- The element of rank k in a set is also called the **kth order statistic** of the set. Accordingly, the CLRS textbook calls a BST augmented this way an **order statistic tree.**

13

Select and Rank

Select(T, k):

r = size(T.left) + 1. (*r = rank of root*)

if k = r, then return T.

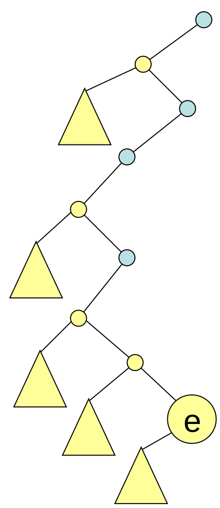
if k < r, then return Select(T.left, k).

else, return Select(T.right, k – r).

Rank(e):

Add up the total # of yellow elements as we walk from e up to the root.

(these are all the elements less than e)



14

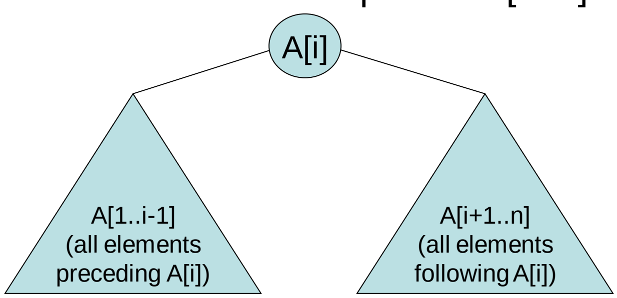
Dynamic Sequences

- Suppose we use an array or linked list to encode a sequence of n elements.
  - We can insert/delete at the endpoints of the sequence in O(1) time,
  - But insertions/deletions in the middle of the sequence take O(n) worst-case time.
- Using a balanced BST augmented with subtree sizes, we can perform all the following operations in O(log n) time:
  - Insert / delete anywhere in the sequence.
  - Access or modify any element by its index in the sequence.

15

Dynamic Sequences

- The "BST property" is slightly different when we use a BST to encode a sequence A[1..n]:

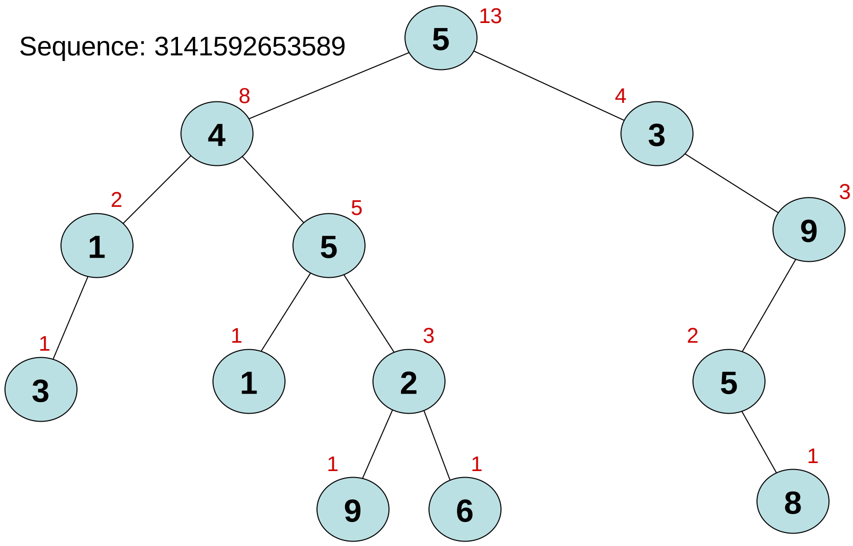


- Elements no longer have keys, and we no longer use *find*. Rather, we rely on the *select* to access elements based their index within the sequence.

16

Dynamic Sequences : Example

Sequence: 3141592653589

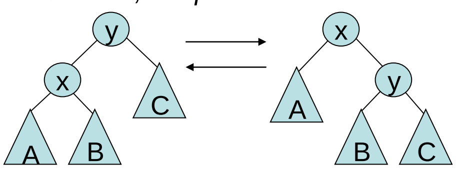


Dynamic Sequences

- Many operations have a nice meaning:
  - *select(i)* accesses the ith element
  - An inorder traversal prints the sequence in order
  - *succ* and *pred* allow us to move between successive elements (as in a linked list)
  - *min* and *max* jump to the beginning and end
- You might want to think of a BST as a structure that fundamentally encodes an arbitrary sequence from left to right. In the case of a dictionary, this sequence happens to be the sorted ordering of the elements we are storing.

Balancing a BST

- There are many ways to modify the insert and delete operations of a BST to keep it balanced:
  - **Worst-case** mechanisms: h = O(log n) always.
    AVL trees, BB[α] trees, red-black trees
  - **Amortized** mechanisms: starting with an empty tree, any sequence of m operations takes O(m log n) time.
    splay trees, "batch" rebalancing methods
  - **Randomized** mechanisms: always balanced "with high probability". *randomly-balanced BSTs, treaps*
- Most of these are based on **rotations:**



Example of Balancing with Rotations: AVL Trees

- AVL: <u>A</u>del'son-<u>V</u>el'skiĭ, and <u>L</u>andis '62.
- A binary tree is **height balanced** if:
  - Its left and right subtrees differ in height by at most 1, and
  - Its left and right subtrees are themselves height balanced.
- It's easy to show that height-balanced trees are balanced (proof on next slide).
- An AVL tree is a height-balanced tree where every node is augmented with the height of its subtree.
  - Note: easy to keep this info up to date after a rotation.
- After each insertion or deletion, we restore the height balance property by performing O(log n) carefully chosen rotations.

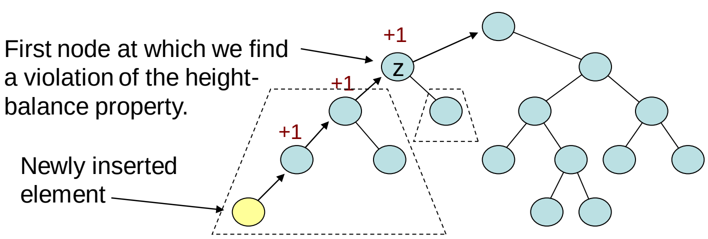
Height Balanced → Balanced

- **Claim:** A height-balanced tree of height h contains ≥ F_h nodes, where F_h is the h^{th} Fibonacci number.
- Recall that $F_h = \Theta(\Phi^h)$, where $\Phi = (1+\sqrt{5}) / 2$. Therefore, an n-element height-balanced tree can have height at most O(log n).
- Easy proof by induction:
  - Consider any arbitrary height-balanced tree of height h.
  - Without loss of generality, suppose its left subtree is tallest.
  - By induction, left subtree contains ≥ F_{h-1} elements.
  - By induction, right subree contains ≥ F_{h-2} elements.
  - So total # of elements is ≥ $1 + F_{h-1} + F_{h-2} = 1 + F_h$.
  - Don't forget the base cases:
    - •Tree of height 0 contains ≥ 1 element ($F_0 = 0$).
    - •Tree of height 1 contains ≥ 2 element ($F_1 = 1$).

21

Restoring Balance After an Insertion

- After an *insert*, walk back up the insertion path and increment augmented subtree heights.
- Stop at the first node (z in the figure below) at which we encounter a height-balancing violation (a height imbalance of exactly 2).

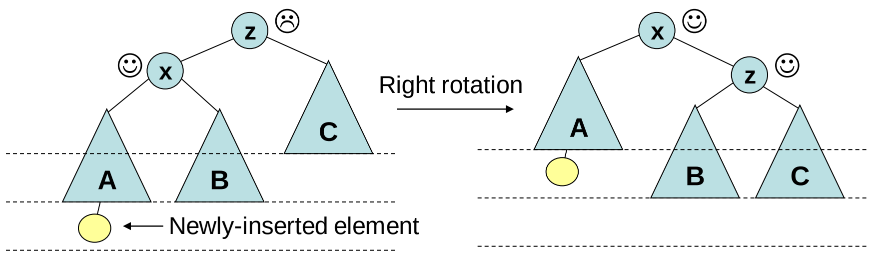


- Using 1 or 2 rotations, we will be able to rebalance our tree at z.
- The height of z's subtree will decrease by 1 to its original value, so we don't need to visit the remaining nodes on the insertion path above z.

22

Restoring Balance After an Insertion

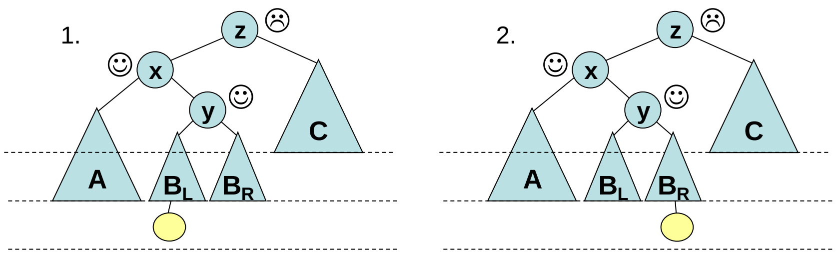
- Let's suppose we have an imbalance at node z due to an insertion that makes z's left subtree too tall (the same argument will apply to the right subtree as well…)
- **Easy case:** Insertion into subtree A makes it too tall:



- **Harder case:** Insertion into subtree B makes it too tall…

Restoring Balance After an Insertion

- **Harder case:** Subtree B too tall. There are now 2 possible subcases:



- In all both cases, a left rotation about the edge (x, y) brings us back to the previous case (with A being too tall).
- So 2 total rotations needed for these cases.