BALANCED SEARCH TREES

David Kauchak
CS 62 – Fall 2026

1

Admin

Checkpoint 2

Hex-A-Pawn
- Game trees

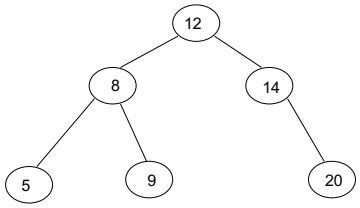
Schedule next week:
- No class Tuesday
- No office hours Mon/Wed
- Asynchronous lab Wednesday (optionally with partner) to be completed by end of day Thursday

BST lab next week
- Can work with a partner
- Needs to be completed by Thursday (11/12)
- I'd recommend doing it **before** the assignment

2

Binary Search Trees

BST – A binary tree where each each node has a key, and every node's key is:
- Larger than all keys in its left subtree. (everything left is smaller)
- Smaller than all keys in its right subtree. (everything right is larger)



3

Operations

Search – Does the key exist in the tree

Insert – Insert the key into tree

Delete – Delete the key from the tree

4

Operations

Search – Does the key exist in the tree
- Start at root and work down the tree based on comparison with current node

Insert – Insert the key into tree
- Start at root and work down the tree based on comparison with current node
- When the search encounters a null child, add node there

Delete – Delete the key from the tree
- No children: delete node
- 1 child: splice out node
- 2 children: replace with successor

5

Previous Worksheet #3

insert(5)
insert(7)
insert(9)
insert(6)
insert(4)
insert(5)
delete(7)
insert(8)
delete(9)

6

Height of the tree

Most of the operations take time O(height)

We said trees built from random data have height O(log n), which is asymptotically tight

Two problems:
- We can't always insure random data
- What happens when we delete nodes and insert others after building a tree?

Worst case height for binary search trees is O(n) ☹

7

Height of tree

What are some types of data where inserting would result in worst case?

8

Height of tree

What are some types of data where inserting would result in worst case?

- sorted
- reverse sorted
- 100, 1, 99, 2, 98, 3, …

9

Balanced trees

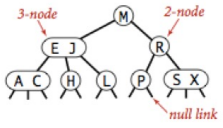
Make sure that the trees remain balanced!

- Red-black trees
- AVL trees
- 2-3 trees
- 2-3-4 trees
- B-trees
- …

Height is guaranteed to be O(log n)

10

2-3 trees



Anatomy of a 2-3 search tree

2-node: one key and two children (left and right)

- everything left is smaller than key
- everything right is greater than (or equal to) key

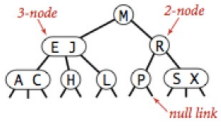
3-node: two keys (k_1, k_2) and three children, left, middle and right

- $k_1 < k_2$
- left values $< k_1$
- $k_1 \leq$ middle values $< k_2$ (greater than or equal to k_1 and less than k_2)
- $k_2 \leq$ right values

11

Search

How do we search for a key?

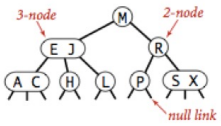


12

Search

Almost identical to BST search

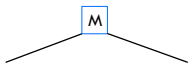
Only difference: sometimes we have two keys



13

Search

Search(H)

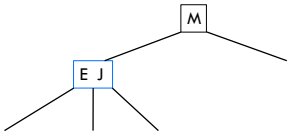


Which child?

14

Search

Search(H)

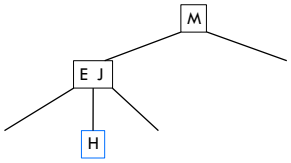


Which child?

15

Search

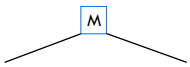
Search(H)



16

Search

Search(B)

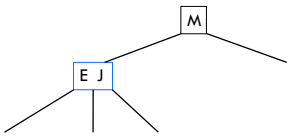


Which child?

17

Search

Search(B)

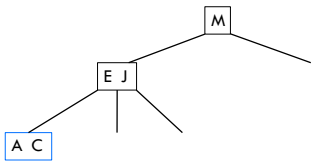


Which child?

18

Search

Search(B)

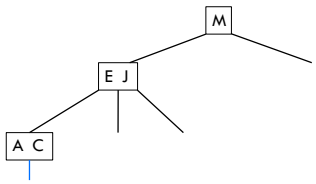


Which child?

19

Search

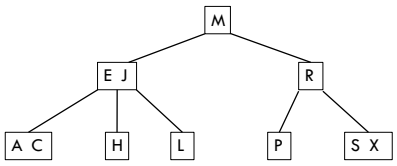
Search(B)



Not found!

20

Search



21

Insertion

Like BST, insert always happens at a leaf

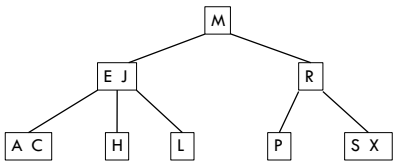
If the leaf is a 2-node, just insert it directly

22

Insertion

If the leaf is a 2-node, just insert it directly

Insert(F)



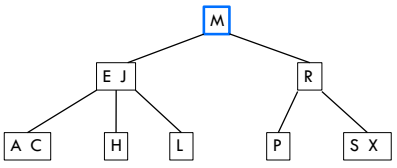
Where should it go?

23

Insertion

If the leaf is a 2-node, just insert it directly

Insert(F)

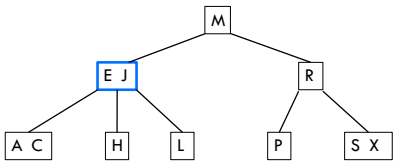


24

Insertion

If the leaf is a 2-node, just insert it directly

Insert(F)

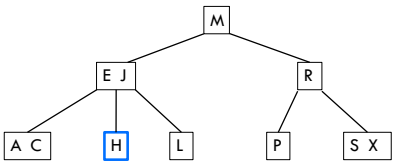


25

Insertion

If the leaf is a 2-node, just insert it directly

Insert(F)

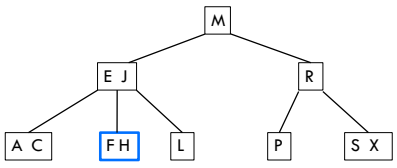


26

Insertion

If the leaf is a 2-node, just insert it directly

Insert(F)



27

Insertion

Like BST, insert always happens at a leaf

If the leaf is a 2-node, just insert it directly

If the leaf is a 3-node:

- We now have three values at this leaf
- Send the middle value up a node
- Make new 2-nodes out of the smallest and largest

28

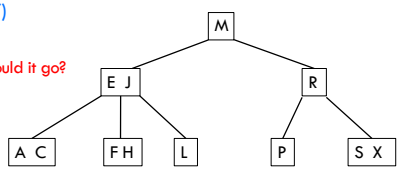
Insertion

If the leaf is a 3-node:

- We now have three values at this leaf
- Send the middle value up a node
- Make new 2-nodes out of the smallest and largest

Insert(T)

Where should it go?



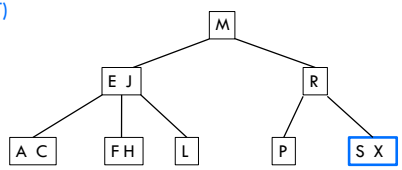
29

Insertion

If the leaf is a 3-node:

- We now have three values at this leaf
- Send the middle value up a node
- Make new 2-nodes out of the smallest and largest

Insert(T)



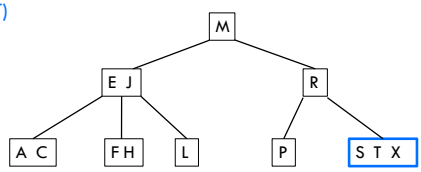
30

Insertion

If the leaf is a 3-node:

- We now have three values at this leaf
- Send the middle value up a node
- Make new 2-nodes out of the smallest and largest

Insert(T)



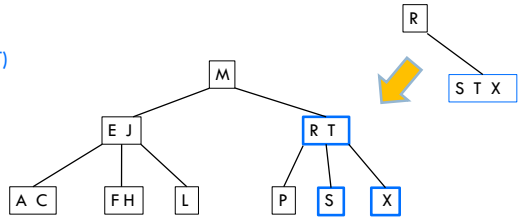
31

Insertion

If the leaf is a 3-node:

- We now have three values at this leaf
- Send the middle value up a node
- Make new 2-nodes out of the smallest and largest

Insert(T)



32

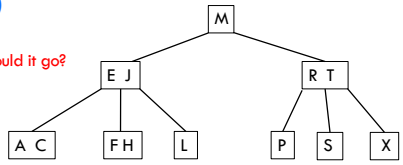
Insertion

If the leaf is a 3-node:

- We now have three values at this leaf
- Send the middle value up a node
- Make new 2-nodes out of the smallest and largest

Insert(I)

Where should it go?



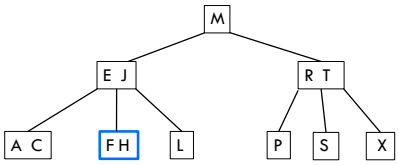
33

Insertion

If the leaf is a 3-node:

- We now have three values at this leaf
- Send the middle value up a node
- Make new 2-nodes out of the smallest and largest

Insert(I)



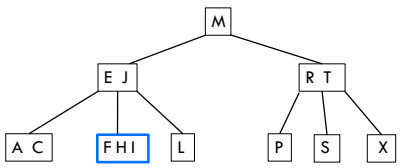
34

Insertion

If the leaf is a 3-node:

- We now have three values at this leaf
- Send the middle value up a node
- Make new 2-nodes out of the smallest and largest

Insert(I)



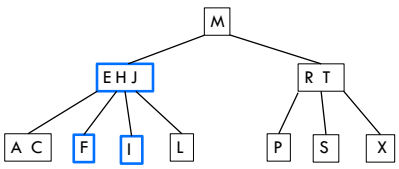
35

Insertion

If the leaf is a 3-node:

- We now have three values at this leaf
- Send the middle value up a node
- Make new 2-nodes out of the smallest and largest

Insert(I)



36

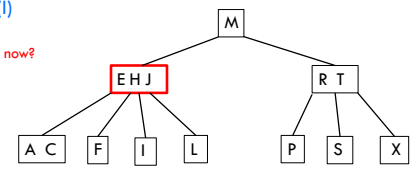
Insertion

If the leaf is a 3-node:

- We now have three values at this leaf
- Send the middle value up a node
- Make new 2-nodes out of the smallest and largest

Insert(I)

What now?



37

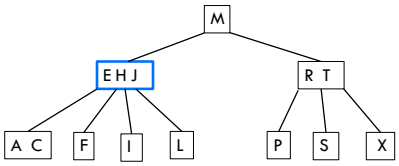
Insertion

If the leaf is a 3-node:

- We now have three values at this leaf
- Send the middle value up a node
- Make new 2-nodes out of the smallest and largest

Insert(I)

Repeat!



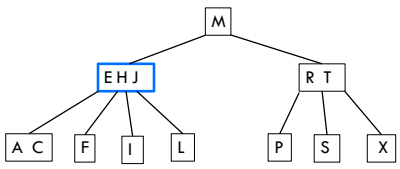
38

Insertion

If the leaf is a 3-node:

- We now have three values at this leaf
- Send the middle value up a node
- Make new 2-nodes out of the smallest and largest

Insert(I)



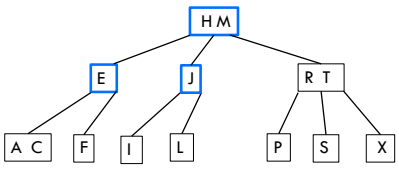
39

Insertion

If the leaf is a 3-node:

- We now have three values at this leaf
- Send the middle value up a node
- Make new 2-nodes out of the smallest and largest

Insert(I)



40

Insertion

If the leaf is a 2-node, just insert it directly

If the leaf is a 3-node:

- We now have three values at this leaf
- Send the middle value up a node
- Make new 2-nodes out of the smallest and largest

When will the height of the tree change?

41

Insertion

If the leaf is a 2-node, just insert it directly

If the leaf is a 3-node:

- We now have three values at this leaf
- Send the middle value up a node
- Make new 2-nodes out of the smallest and largest

Only when the root is a 3-node and we insert into a path that is all 3-nodes!

Effect: The tree can hold quite a few values before having to increase the height

42

Practice

Draw the 2-3 tree that results when you insert the keys:

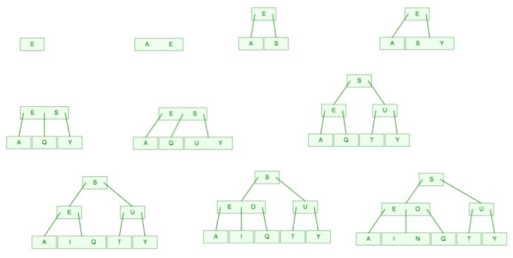
E A S Y Q U T I O N in that order in an initially empty tree.

43

Practice

Draw the 2-3 tree that results when you insert the keys:

E A S Y Q U T I O N in that order in an initially empty tree.



44

Deletion

Similar idea to insert, but in reverse:

- Deletion always happens at a leaf
- If the value to be deleted isn't in a leaf, swap it with its successor
- If it's a 3-node, just delete the value
- If it's a 2-node, need to delete the child from the parent
  - This may cause a recursive deletion

45

Running time

Worst case height: O(log n)

What does that mean?

46

Running time

Worst case height: O(log n)

Insert, search and delete are all O(log n)

47

2-3 search trees in practice

A pain to implement

Overhead can often make slower than standard BST

Other balanced trees exist that provide the same worst case guarantee, but are faster (e.g, red-black trees)

48

B trees

Defined by a parameter, *m*, the order:

Every node has at most at most *m*-1 keys
- keys are stored in sorted order
- the keys delimit the values of the children

A node with c children will have exactly c-1 keys

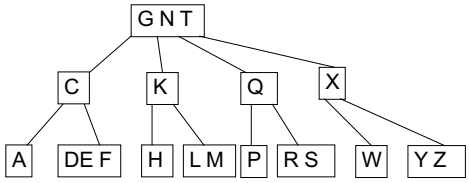
All internal nodes except the root have at least $m/2$ children

All children are at the same depth

Root has at least 2 children

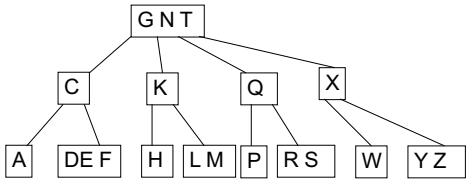
49

Example B-tree: m = 4



50

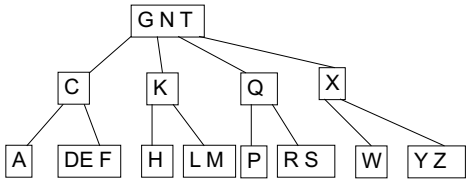
Example B-tree: m = 4



Balanced: all leaves have the same depth

51

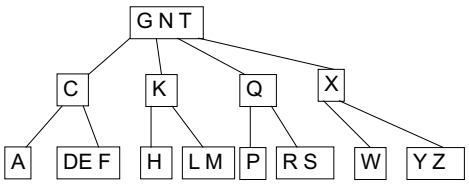
Example B-tree: m = 4



Every node has at most at most *m*-1 keys stored in sorted order

52

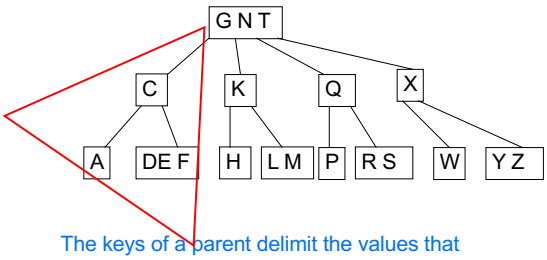
Example B-tree: m = 4



Nodes have at most m children

53

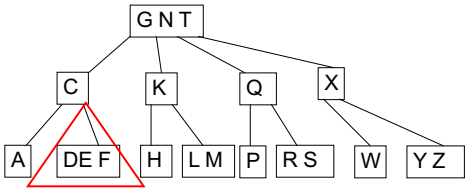
Example B-tree: m = 4



The keys of a parent delimit the values that a child's keys can take

54

Example B-tree: m = 4



The keys of a parent delimit the values that a child's keys can take

55

B-trees vs. 2-3 trees?

56

B-trees vs. 2-3 trees?

2-3 trees are B-trees with m = 3!

57

B-tree usage

B-trees are generally an **on-disk** data structure

Memory is limited or there is a large amount of data to be stored

In the extreme, only one node is kept in memory and the rest on disk

Size of the nodes is often determined by a page size on disk.

Databases frequently use B-trees

58

Red-black tree high-level

https://www.cs.usfca.edu/~galles/visualization/RedBlack.html

59

Balanced trees in Java

https://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html

- Compare with HashSet

https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html

- Compare with HashMap

60