CS 561, Lecture 5

Jared Saia
University of New Mexico

Binary Search Trees

- Overview
- Red Black Trees
- AVL, B-Trees, Splay Trees

Binary Search Trees

- Binary Search Trees are another data structure for implementing the dictionary ADT

Red-Black Trees

Red-Black trees (a kind of binary tree) also implement the Dictionary ADT, namely:

- Insert(x) - $O(\log n)$ time
- Lookup(x) - $O(\log n)$ time
- Delete(x) - $O(\log n)$ time

Why BST?

- Q: When would you use a Search Tree?
- A1: When need a hard guarantee on the worst case run times (e.g. “mission critical” code)
- A2: When want something more dynamic than a hash table (e.g. don’t want to have to enlarge a hash table when the load factor gets too large)
- A3: Search trees can implement some other important operations...

Search Tree Operations

- Insert
- Lookup
- Delete
- *Minimum/Maximum*
- *Predecessor/Successor*

What is a BST?

- It's a binary tree
- Each node holds a key and record field, and a pointer to left and right children
- *Binary Search Tree Property* is maintained

Binary Search Tree Property

- Let x be a node in a binary search tree. If y is a node in the left subtree of x, then key(y)$\leq$key(x). If y is a node in the right subtree of x then key(x)$\leq$key(y)

Example BST

Inorder Walk

- BSTs are arranged in such a way that we can print out the elements in sorted order in $\Theta(n)$ time
- Inorder Tree-Walk does this

Inorder Tree-Walk

```
Inorder-TW(x){
  if (x is not nil){
    Inorder-TW(left(x));
    print key(x);
    Inorder-TW(right(x));
}
```

Example Tree-Walk

Analysis

- Correctness?
- Run time?

Search in BT

```
Tree-Search(x,k){
  if (x=nil) or (k = key(x)){
    return x;
  }
  if (k<key(x)){
    return Tree-Search(left(x),k);
  }else{
    return Tree-Search(right(x),k);
  }
}
```

Analysis

- Let h be the height of the tree
- The run time is $O(h)$
- Correctness???

In-Class Exercise

- Q1: What is the loop invariant for Tree-Search?
- Q2: What is Initialization?
- Q3: Maintenance?
- Q4: Termination?

Binary Search Tree Property

- Let x be a node in a binary search tree. If y is a node in the left subtree of x, then key(y)$\leq$key(x). If y is a node in the right subtree of x then key(x)$\leq$key(y)

Search in BT

```
Tree-Search(x,k){
  if (x=nil) or (k = key(x)){
    return x;
  }
  if (k<key(x)){
    return Tree-Search(left(x),k);
  }else{
    return Tree-Search(right(x),k);
  }
}
```

Analysis

- Let h be the height of the tree
- The run time is $O(h)$
- Correctness???

Previous In-Class Exercise

- Q1: What is the loop invariant for Tree-Search?
- Q2: What is Initialization?
- Q3: Maintenance?
- Q4: Termination?

Loop Invariant Review

A useful tool for proving correctness is loop invariants. Three things must be shown about a loop invariant

- **Initialization:** Invariant is true before first iteration of loop
- **Maintenance:** If invariant is true before iteration i, it is also true before iteration $i+1$
- **Termination:** When the loop terminates, the invariant gives a property which can be used to show the algorithm is correct

Loop Invariant Review

- When **Initialization** and **Maintenance** hold, the loop invariant is true prior to every iteration of the loop
- Similar to mathematical induction: must show both base case and inductive step
- Showing the invariant holds before the first iteration is like the base case. Showing the invariant holds from iteration to iteration is like the inductive step

Loop Invariant Review

- **Termination** shows that if the loop invariant is true after the last iteration of the loop, then the algorithm is correct
- The termination condition is different than induction

Choosing Loop Invariants

- Q: How do we choose the right loop invariant for an algorithm?
- A1: There is no standard recipe for doing this. It's like choosing the right guess for the solution to a recurrence relation.
- A2: Following is one possible recipe:
  1. Study the algorithm and list what important invariants seem true during iterations of the loop - it may help to simulate the algorithm on small inputs to get this list of invariants
  2. From the list of invariants, select one which seems strong enough to prove the correctness of the algorithm
  3. Try to show Initialization, Maintenance and Termination for this invariant. If you're unable to show all three properties, go back to the step 1.

Answers

- To show: If key k exists in the tree, Tree-Search returns the elem with key k, otherwise Tree-Search returns nil.
- *Loop Invariant: If key k exists in the tree, then it exists in the subtree rooted at node x*

Answers

- Initialization: Before the first iteration, x is the root of the entire tree, therefor if key k exists in the tree, then it exists in the subtree rooted at node x

Maintenance

- Maintenance: Assume at the beginning of the procedure, it's true that if key k exists in the tree that it is in the subtree rooted at node x. There are three cases that can occur during the procedure:
  - Case 1: key(x) is k. In this case, the procedure terminates and returns x, so the invariant continues to hold
  - Case 2: k<key(x). In this case, by the *BST Property*, all keys in the subtree rooted on the right child of x are greater than k (since key(x)>k). Thus, if k exists in the subtree rooted at x, it must exist in the subtree rooted at left(x).
  - Case 3:k>key(x). In this case, by the *BST Property*, All keys in the subtree rooted on the right child of x are less than k (since key(x)<k). Thus, if k exists in the subtree rooted at x, it must exist in the subtree rooted at right(x).

Termination

- By the loop invariant, we know that when the procedure terminates, if k is in the tree, then it is in the subtree rooted at x. If k is in fact in the tree, then x will never be nil, and so the procedure will only terminate by returning a node with key k. If k is not in the tree, then the only way the procedure will terminate is when x is nil. Thus, in this case also, the procedure will return the correct answer.

Tree Min/Max

- Tree Minimum(x): Return the leftmost child in the tree rooted at x
- Tree Maximum(x): Return the rightmost child in the tree rooted at x

Successor

- The successor of a node x is the node that comes after x in the sorted order determined by an in-order tree walk.
- If all keys are distinct, the successor of a node x is the node with the smallest key greater than x

Tree-Successor

```
Tree-Successor(x){
  if (right(x) != null){
    return Tree-Minimum(right(x));
  }
  y = parent(x);
  while (y!=null and x=right(y)){
    x = y;
    y = parent(y);
  }
  return y;
}
```

Successor Intuition

- Case 1: If right subtree of x is non-empty, successor(x) is just the leftmost node in the right subtree
- Case 2: If the right subtree of x is empty and x has a successor,x then successor(x) is the lowest ancestor of x whose left child is also an ancestor of x. (i.e. the lowest ancestor of x whose key is $\geq$ key(x))

Insertion

Insert(T,x)

1. Let r be the root of T.
2. Do Tree-Search(r,key(x)) and let p be the last node processed in that search
3. If p is nil (there is no tree), make x the root of a new tree
4. Else if key(x) $\leq$ p, make x the left child of p, else make x the right child of p

Deletion

- Code is in book, basically there are three cases, two are easy and one is tricky
- Case 1: The node to delete has no children. Then we just delete the node
- Case 2: The node to delete has one child. Then we delete the node and "splice" together the two resulting trees

Case 3

Case 3: The node, x to be deleted has two children

1. Swap x with Successor(x) (Successor(x) has no more than 1 child (why?))
2. Remove x, using the procedure for case 1 or case 2.

Analysis

- All of these operations take $O(h)$ time where h is the height of the tree
- If n is the number of nodes in the tree, in the worst case, h is $O(n)$
- However, if we can keep the tree *balanced*, we can ensure that $h = O(\log n)$
- Red-Black trees can maintain a balanced BST

Randomly Built BST

- What if we build a binary search tree by inserting a bunch of elements at random?
- Q: What will be the average depth of a node in such a randomly built tree? We'll show that it's $O(\log n)$
- For a tree T and node x, let $d(x, T)$ be the depth of node x in T
- Define the total path length, $P(T)$, to be the sum over all nodes x in T of $d(x, T)$

Analysis

"Shut up brain or I'll poke you with a Q-Tip" - Homer Simpson

- Note that the average depth of a node in T is

$$\frac{1}{n}\sum_{x\in T} d(x,T) = \frac{1}{n}P(T)$$

- Thus we want to show that $P(T) = O(n \log n)$

Analysis

- Let T_l, T_r be the left and right subtrees of T respectively. Let n be the number of nodes in T
- Then $P(T) = P(T_l) + P(T_r) + n - 1$. Why?

Analysis

- Let $P(n)$ be the expected total depth of all nodes in a randomly built binary tree with n nodes
- Note that for all i, $0 \leq i \leq n-1$, the probability that T_l has i nodes and T_r has $n-i-1$ nodes is $1/n$.
- Thus $P(n) = \frac{1}{n}\sum_{i=0}^{n-1}(P(i) + P(n-i-1) + n - 1)$

Analysis

$$
\begin{aligned}
P(n) &= \frac{1}{n}\sum_{i=0}^{n-1}(P(i)+P(n-i-1)+n-1) && (1)\\
&= \frac{1}{n}(\sum_{i=0}^{n-1}(P(i)+P(n-i-1))+\frac{1}{n}(\sum_{i=0}^{n-1}n-1)) && (2)\\
&= \frac{1}{n}(\sum_{i=0}^{n-1}(P(i)+P(n-i-1))+\Theta(n) && (3)\\
&= \frac{2}{n}(\sum_{k=1}^{n-1}P(k))+\Theta(n) && (4)\\
& && (5)
\end{aligned}
$$

Analysis

- We have $P(n) = \frac{2}{n}(\sum_{k=1}^{n-1} P(k)) + \Theta(n)$
- This is the same recurrence for randomized Quicksort
- In your hw (problem 7-2), you show that the solution to this recurrence is $P(n) = O(n \log n)$

Take Away

- $P(n)$ is the expected total depth of all nodes in a randomly built binary tree with n nodes.
- We've shown that $P(n) = O(n \log n)$
- There are n nodes total
- Thus the expected average depth of a node is $O(\log n)$

Take Away

- The expected average depth of a node in a randomly built binary tree is $O(\log n)$
- This implies that operations like search, insert, delete take expected time $O(\log n)$ for a randomly built binary tree

Warning!

- In many cases, data is not inserted randomly into a binary search tree
- I.e. many binary search trees are not “randomly built”
- For example, data might be inserted into the binary search tree in almost sorted order
- Then the BST would not be randomly built, and so the expected average depth of the nodes would not be $O(\log n)$

What to do?

- A Red-Black tree implements the dictionary operations in such a way that the height of the tree is always $O(\log n)$, where n is the number of nodes
- This will guarantee that no matter how the tree is built that all operations will always take $O(\log n)$ time
- Next time we'll see how to create Red-Black Trees

Outline

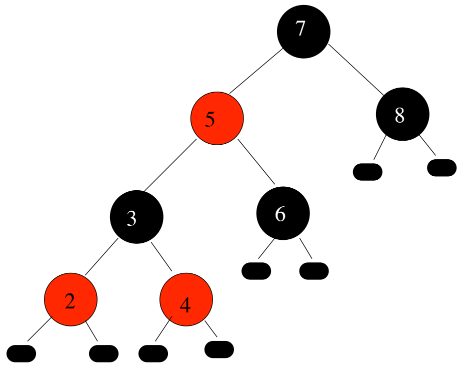
- Red Black Trees (Chapter 13)

Red-Black Properties

A BST is a red-black tree if it satisfies the RB-Properties

1. Every node is either red or black
2. The root is black
3. Every leaf (NIL) is black
4. If a node is red, then both its children are black
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes

Example RB-Tree



Black Height

- *Black-height* of a node x, bh(x) is the number of black nodes on any path from, but not including x down to a leaf node.
- Note that the black-height of a node is well-defined since all paths have the same number of black nodes
- The black-height of an RB-Tree is just the black-height of the root

Key Lemma

- *Lemma: A RB-Tree with n internal nodes has height at most* $2\log(n+1)$
- Proof Sketch:
  1. The subtree rooted at the node x contains at least $2^{bh(x)}-1$ internal nodes
  2. For the root r, $bh(r) \geq h/2$, thus $n \geq 2^{h/2}-1$. Taking logs of both sides, we get that $h \leq 2\log(n+1)$

Proof

1) The subtree rooted at the node x contains at least $2^{bh(x)} - 1$ internal nodes. Show by induction on the height of x.

- BC: If the height of x is 0, then x is a leaf, and subtree rooted at x does indeed contain $2^0 - 1 = 0$ internal nodes
- IH: For all nodes y of height less than x, the subtree rooted at y contains at least $2^{bh(y)} - 1$ internal nodes.
- IS: Consider a node x which is an internal node with two children(all internal nodes have two children). Each child has black-height of either $bh(x)$ or $bh(x) - 1$ (the former if it is red, the latter if it is black). Since the height of these children is less than x, we can apply the inductive hypothesis to conclude that each child has at least $2^{bh(x)-1} - 1$ internal nodes. This implies that the subtree rooted at x has at least $(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$ internal nodes. This proves the claim.

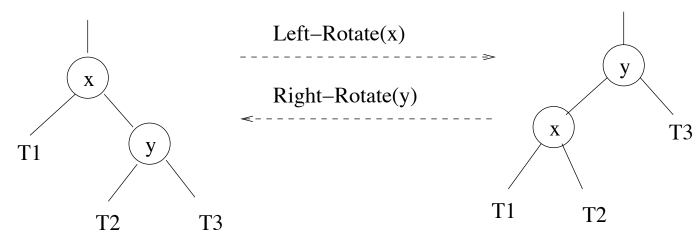
Maintenance?

- How do we ensure that the Red-Black Properties are maintained?
- I.e. when we insert a new node, what do we color it? How do we re-arrange the new tree so that the Red-Black Property holds?
- How about for deletions?

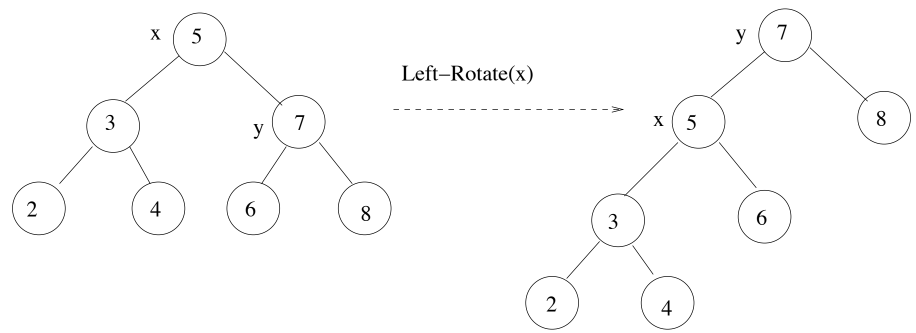
Left-Rotate

- Left-Rotate(x) takes a node x and "rotates" x with its right child
- Right-Rotate is the symmetric operation
- *Both Left-Rotate and Right-Rotate preserve the BST Property*
- We'll use Left-Rotate and Right-Rotate in the RB-Insert procedure

Picture



Example



Binary Search Tree Property

- Let x be a node in a binary search tree. If y is a node in the left subtree of x, then key(y)$\leq$key(x). If y is a node in the right subtree of x then key(y)$\geq$key(x)

In-Class Exercise

Show that Left-Rotate(x) maintains the BST Property. In other words, show that if the BST Property was true for the tree before the Left-Rotate(x) operation, then it's true for the tree after the operation.

- Show that after rotation, the BST property holds for the entire subtree rooted at x
- Show that after rotation, the BST property holds for the subtree rooted at y
- Now argue that after rotation, the BST property holds for the entire tree

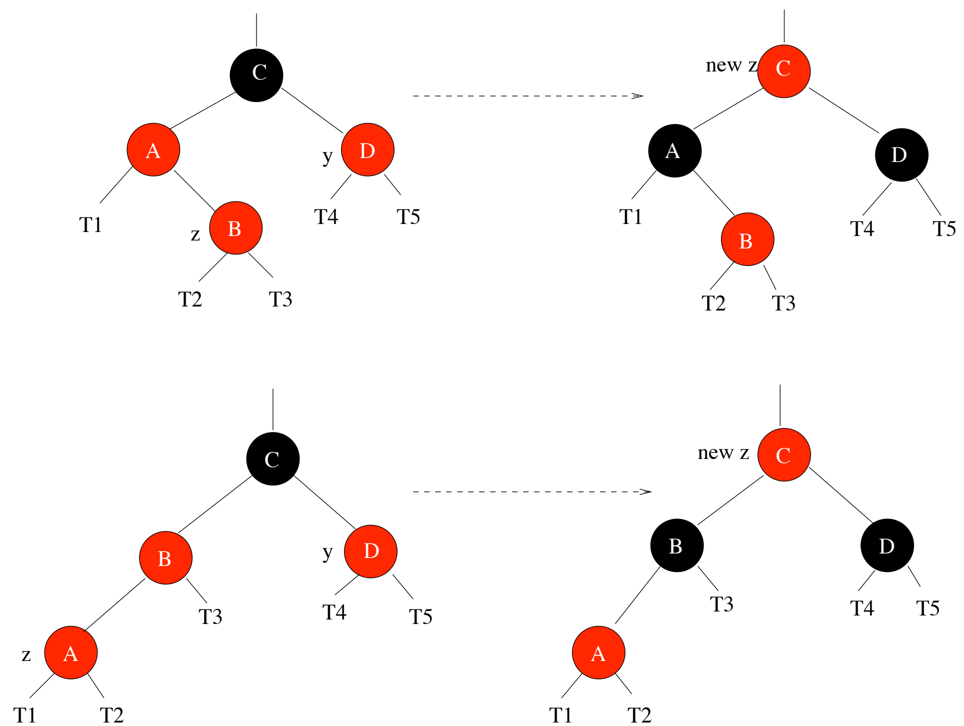
RB-Insert(T,z)

1. Set left(z) and right(z) to be NIL
2. Let y be the last node processed during a search for z in T
3. Insert z as the appropriate child of y (left child if key(z)$\leq$ y, right child otherwise)
4. Color z red
5. Call the procedure RB-Insert-Fixup

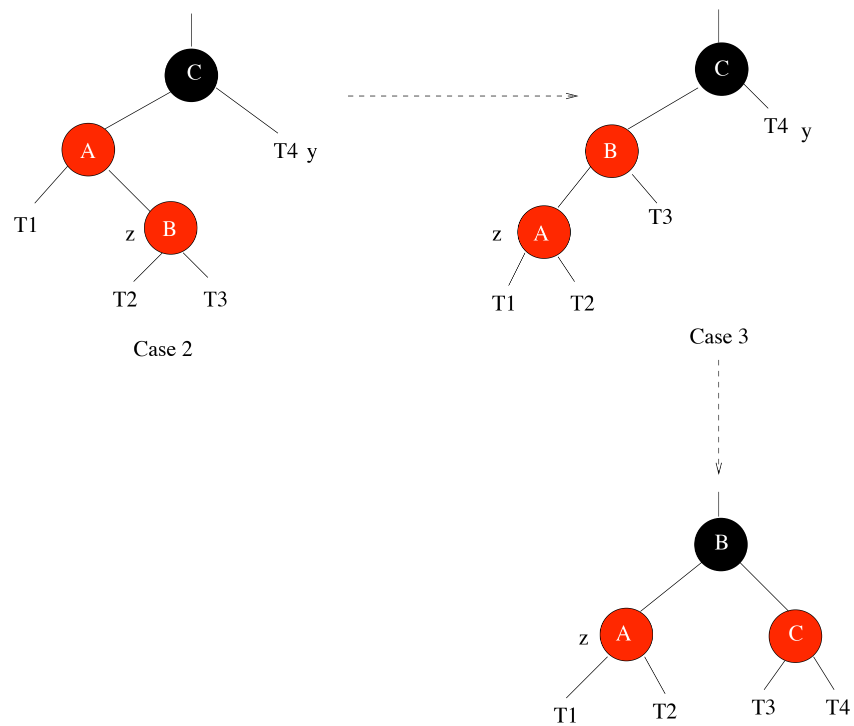
RB-Insert-Fixup(T,z)

```
RB-Insert-Fixup(T,z){
  while (color(p(z)) is red){
    case 1: z's uncle, y, is red{
      do case 1
    }
    case 2: z's uncle, y, is black and z is a right child{
      do case 2
    }
    case 3: z's uncle, y, is black and z is a left child{
      do case 3
    }
  }
  color(root(T)) = black;
}
```

Case 1


C
A
y
D
T1
z
B
T4
T5
T2
T3
new z
C
A
D
T1
B
T4
T5
T2
T3
C
B
y
D
T3
T4
T5
z
A
T1
T2
new z
C
B
D
T3
T4
T5
A
T1
T2

Case 2 and 3



Loop Invariant

At the start of each iteration of the loop:

- Node z is red
- If parent(z) is the root, then parent(z) is black
- If there is a violation of the red-black properties, there is at most one violation, and it is either property 2 or 4. If there is a violation of property 2, it occurs because z is the root and is red. If there is a violation of property 4, it occurs because both z and parent(z) are red.

Pseudocode

- Detailed Pseudocode for RB-Insert and RB-Insert-Fixup is in the book, Chapter 13.3
- A detailed proof of correctness for RB-Insert-Fixup in the the same Chapter
- Code for *RB-Deletion* is also in Chapter 13

Other Balanced BSTs

- We'll now *briefly* discuss some other balanced BSTs
- They all implement Insert, Delete, Lookup, Successor, Predecessor, Maximum and Minimum efficiently

AVL Trees

- An AVL tree is height-balanced: For each node x, the heights of the left and right subtrees of x differ by at most 1
- Each node has an additional height field $h(x)$
- Claim: An AVL tree with n nodes has height $O(\log n)$

AVL Trees

- Claim: An AVL tree with n nodes has height $O(\log n)$
- Q: For an AVL tree of height h, how many nodes must it have in it?
- A: We can write a recurrence relation. Let $T(h)$ be the minimum number of nodes in a tree of height h
- Then $T(h) = T(h-1) + T(h-2) + 1$, $T(2) = T(1) \geq 1$
- This is similar to the recurrence relation for Fibonnaci numbers! Solution:

$$T(h) = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^h - 2$$

- So we have the equation $n > T(h)$. Let $\phi = \frac{1+\sqrt{5}}{2}$. Then:

$$\begin{aligned}
n &\geq \frac{1}{\sqrt{5}}(\phi^h) - 2 &(6)\\
\log n &\geq \log(\frac{1}{\sqrt{5}}) + h\log\phi - 1 &(7)\\
\log n - \log(\frac{1}{\sqrt{5}}) + 1 &\geq h\log\phi &(8)\\
C * \log n &\geq h &(9)
\end{aligned}$$

- Where the final inequality holds for appropriate constant C, and for n large enough. The final inequality implies that $h = O(\log n)$

AVL Tree Insertion

- After insert into an AVL tree, the tree may no longer be height-balanced
- Need to "fix-up" the subtrees so that they become height-balanced again
- Can do this using rotations (similar to case for RB-Trees)
- Similar story for deletions

B-Trees

- B-Trees are balanced search trees designed to work well on disks
- B-Trees are *not* binary trees: each node can have many children
- Each node of a B-Tree contains *several* keys, not just one
- When doing searches, we decide which child link to follow by finding the correct interval of our search key in the key set of the current node.

Disk Accesses

- Consider any search tree
- The number of disk accesses per search will dominate the run time
- Unless the entire tree is in memory, there will usually be a disk access every time an arbitrary node is examined
- The number of disk accesses for most operations on a B-tree is proportional to the height of the B-tree
- I.e. The info on each node of a B-tree can be stored in main memory

B-Tree Properties

The following is true for every node x

- x stores keys, $key_1(x), \ldots key_l(x)$ in sorted order (nondecreasing)
- x contains pointers, $c_1(x), \ldots, c_{l+1}(x)$ to its children
- Let k_i be any key stored in the subtree rooted at the i-th child of x, then $k_1 \leq key_1(x) \leq k_2 \leq key_2(x) \cdots \leq key_l(x) \leq k_{l+1}$

B-Tree Properties

- All leaves have the same depth
- Lower and upper bounds on the number of keys a node can contain. Given as a function of a fixed integer t
  - Every node other than the root must have $\geq (t-1)$ keys, and t children. If the tree is non-empty, the root must have at least one key (and 2 children)
  - Every node can contain at most $2t-1$ keys, so any internal node can have at most $2t$ children

Note

- The above properties imply that the height of a B-tree is no more than $\log_t \frac{n+1}{2}$, for $t \geq 2$, where n is the number of keys.
- If we make t, larger, we can save a larger (constant) fraction over RB-trees in the number of nodes examined
- A (2-3-4)-tree is just a B-tree with $t = 2$

In-Class Exercise

We will now show that for any B-Tree with height h and n keys, $h \leq \log_t \frac{n+1}{2}$, where $t \geq 2$.

Consider a B-Tree of height $h > 1$

- Q1: What is the minimum number of nodes at depth 1, 2, and 3
- Q2: What is the minimum number of nodes at depth i?
- Q3: Now give a lowerbound for the total number of *keys* (e.g. $n \geq$???)
- Q4: Show how to solve for h in this inequality to get an upperbound on h

Splay Trees

- A Splay Tree is a kind of BST where the standard operations run in $O(\log n)$ *amortized* time
- This means that over l operations (e.g. Insert, Lookup, Delete, etc), where l is sufficiently large, the total cost is $O(l * \log n)$
- In other words, the average cost per operation is $O(\log n)$
- However a single operation could still take $O(n)$ time
- In practice, they are very fast

Skip Lists

- Technically, not a BST, but they implement all of the same operations
- Very elegant randomized data structure, simple to code but analysis is subtle
- They guarantee that, with high probability, all the major operations take $O(\log n)$ time
- We'll discuss them more next class

High Level Analysis

Comparison of various BSTs

- RB-Trees: $+$ guarantee $O(\log n)$ time for each operation, easy to augment, $-$ high constants
- AVL-Trees: $+$ guarantee $O(\log n)$ time for each operation, $-$ high constants
- B-Trees: $+$ works well for trees that won't fit in memory, $-$ inserts and deletes are more complicated
- Splay Tress: $+$ small constants, $-$ amortized guarantees only
- Skip Lists: $+$ easy to implement, $-$ runtime guarantees are probabilistic only

Which Data Structure to use?

- Splay trees work very well in practice, the "hidden constants" are small
- Unfortunately, they can not guarantee that *every* operation takes $O(\log n)$
- When this guarantee is required, B-Trees are best when the entire tree will not be stored in memory
- If the entire tree will be stored in memory, RB-Trees, AVL-Trees, and Skip Lists are good