Lecture 6: Balanced Search Trees

Michael Dinitz

September 16, 2021
601.433/633 Introduction to Algorithms
Introduction

Today, and next few weeks: data structures.

- Since “Data Structures” a prereq, focus on advanced structures and on interesting analysis
Introduction

Today, and next few weeks: data structures.

- Since “Data Structures” a prereq, focus on advanced structures and on interesting analysis

Today and later: data structures for *dictionaries*
Introduction

Today, and next few weeks: data structures.
- Since “Data Structures” a prereq, focus on advanced structures and on interesting analysis

Today and later: data structures for dictionaries

Definition

A dictionary data structure is a data structure supporting the following operations:
- \texttt{insert(key,object)}: insert the (key, object) pair.
- \texttt{lookup(key)}: return the associated object
- \texttt{delete(key)}: remove the key and its object from the data structure. We may or may not care about this operation.
Obvious Approaches

Reminder: all running times for worst case
Obvious Approaches

Reminder: all running times for *worst case*

Approach 1: Sorted array
Obvious Approaches

Reminder: all running times for worst case

Approach 1: Sorted array
  ▸ Lookup:
Obvious Approaches

Reminder: all running times for worst case

Approach 1: Sorted array
  - Lookup: $O(\log n)$
Obvious Approaches

Reminder: all running times for worst case

Approach 1: Sorted array
  - Lookup: $O(\log n)$
  - Insert:

Approach 2: Unsorted (linked) list
  - Insert: $O(1)$
  - Lookup: $\Omega(n)$
Obvious Approaches

Reminder: all running times for worst case

Approach 1: Sorted array
  ▶ Lookup: $O(\log n)$
  ▶ Insert: $\Omega(n)$
Obvious Approaches

Reminder: all running times for worst case

Approach 1: Sorted array
  - Lookup: $O(\log n)$
  - Insert: $\Omega(n)$

Approach 2: Unsorted (linked) list

Goal: $O(\log n)$ for both.
Obvious Approaches

Reminder: all running times for worst case

Approach 1: Sorted array
  - Lookup: $\mathcal{O}(\log n)$
  - Insert: $\Omega(n)$

Approach 2: Unsorted (linked) list
  - Insert:
Obvious Approaches

Reminder: all running times for *worst case*

Approach 1: Sorted array
  - Lookup: $O(\log n)$
  - Insert: $\Omega(n)$

Approach 2: Unsorted (linked) list
  - Insert: $O(1)$
Obvious Approaches

Reminder: all running times for *worst case*

Approach 1: Sorted array
- Lookup: $O(\log n)$
- Insert: $\Omega(n)$

Approach 2: Unsorted (linked) list
- Insert: $O(1)$
- Lookup:

Goal: $O(\log n)$ for both.

Approach today: search trees
Obvious Approaches

Reminder: all running times for \textit{worst case}

Approach 1: Sorted array
  - Lookup: $O(\log n)$
  - Insert: $\Omega(n)$

Approach 2: Unsorted (linked) list
  - Insert: $O(1)$
  - Lookup: $\Omega(n)$
Obvious Approaches

Reminder: all running times for *worst case*

Approach 1: Sorted array
- Lookup: $O(\log n)$
- Insert: $\Omega(n)$

Approach 2: Unsorted (linked) list
- Insert: $O(1)$
- Lookup: $\Omega(n)$

Goal: $O(\log n)$ for both.
Obvious Approaches

Reminder: all running times for worst case

Approach 1: Sorted array
  - Lookup: $O(\log n)$
  - Insert: $\Omega(n)$

Approach 2: Unsorted (linked) list
  - Insert: $O(1)$
  - Lookup: $\Omega(n)$

Goal: $O(\log n)$ for both.
Approach today: search trees
Binary Search Tree Review

Binary search tree:
- All nodes have at most 2 children
- Each node stores (key, object) pair
- All descendants to left have smaller keys
- All descendants to the right have larger keys
Binary Search Tree Review

Binary search tree:

- All nodes have at most 2 children
- Each node stores (key, object) pair
- All descendants to left have smaller keys
- All descendants to the right have larger keys

Lookup: follow path from root!
Dictionary Operations in Simple Binary Search Tree

`insert(x)`:

- If tree empty, put `x` at root
- Else if `x < root.key` recursively insert into left child
- Else (if `x > root.key`) recursively insert into right child
Dictionary Operations in Simple Binary Search Tree

insert(x):
- If tree empty, put x at root
- Else if x < root.key recursively insert into left child
- Else (if x > root.key) recursively insert into right child

Example: H O P K I N S
Simply Binary Search Tree: Analysis

Pluses: easy to implement
Simply Binary Search Tree: Analysis

Pluses: easy to implement

(Worst-case) Running time:
Simply Binary Search Tree: Analysis

Pluses: easy to implement

(Worst-case) Running time: if depth $d$, then $\Theta(d)$
Simply Binary Search Tree: Analysis

Pluses: easy to implement

(Worst-case) Running time: if depth \(d\), then \(\Theta(d)\)
  - If very unbalanced \(d\) could be \(\Omega(n)\)!
Simply Binary Search Tree: Analysis

Pluses: easy to implement

(Worst-case) Running time: if depth $d$, then $\Theta(d)$

- If very unbalanced $d$ could be $\Omega(n)$!

Want to make tree \textit{balanced}.
Simply Binary Search Tree: Analysis

Pluses: easy to implement

(Worst-case) Running time: if depth $d$, then $\Theta(d)$
- If very unbalanced $d$ could be $\Omega(n)$!

Want to make tree balanced.

Rest of today:
- B-trees: perfect balance, not binary
- Red-black trees: approximate balance, binary
- Turn out to be related!
B-Trees
B-tree Definition

Parameter $t \geq 2$. 
B-tree Definition

Parameter $t \geq 2$.

**Definition (B-tree with parameter $t$)**

1. Each node has between $t-1$ and $2t-1$ keys in it (except the root has between 1 and $2t-1$ keys). Keys in a node are stored in a sorted array.

2. Each non-leaf has degree (number of children) equal to the number of keys in it plus 1. If $v$ is a node with keys $[a_1, a_2, \ldots, a_k]$ and the children are $[v_1, v_2, \ldots, v_{k+1}]$, then the tree rooted at $v_i$ contains only keys that are at least $a_{i-1}$ and at most $a_i$ (except the the edge cases: the tree rooted at $v_1$ has keys less than $a_1$, and the tree rooted at $v_{k+1}$ has keys at least $a_k$).

3. All leaves are at the same depth.

When $t=2$ known as a 2-3-4 tree, since # children either 2, 3, or 4
B-tree Definition

Parameter $t \geq 2$.

**Definition (B-tree with parameter $t$)**

1. Each node has between $t - 1$ and $2t - 1$ keys in it (except the root has between 1 and $2t - 1$ keys). Keys in a node are stored in a sorted array.

2. Each non-leaf has degree (number of children) equal to the number of keys in it plus 1. If $v$ is a node with keys $[a_1, a_2, \ldots, a_k]$ and the children are $[v_1, v_2, \ldots, v_{k+1}]$, then the tree rooted at $v_i$ contains only keys that are at least $a_{i-1}$ and at most $a_i$ (except the edge cases: the tree rooted at $v_1$ has keys less than $a_1$, and the tree rooted at $v_{k+1}$ has keys at least $a_k$).

3. All leaves are at the same depth.

When $t = 2$ known as a 2-3-4 tree, since # children either 2, 3, or 4
An important idea: the problem with the basic binary search tree was that we were not maintaining balance. On the other hand, if we try to maintain a perfectly balanced tree, we will spend too much time rearranging things. So, we want to be balanced but also give ourselves some slack. It’s a bit like how in the median-finding algorithm, we gave ourselves slack by allowing the pivot to be “near” the middle. For B-trees, we will make the tree perfectly balanced, but give ourselves slack by allowing some nodes to have more children than others.

9.4 B-trees and 2-3-4 trees

A B-tree is a search tree where for some pre-specified $t \geq 2$ (think of $t = 2$ or $t = 3$):

- Each node has between $t-1$ and $2t-1$ keys in it (except the root has between 1 and $2t-1$ keys). Keys in a node are stored in a sorted array.
- Each non-leaf has degree (number of children) equal to the number of keys in it plus 1. So, node degrees are in the range $[t, 2t]$ except the root has degree in the range $[2, 2t]$. The semantics are that the $i$th child has items between the $(i-1)$st and $i$th keys. E.g., if the keys are $[a_1, a_2, \ldots, a_{10}]$ then there is one child for keys less than $a_1$, one child for keys between $a_1$ and $a_2$, and so on, until the rightmost child has keys greater than $a_{10}$.
- All leaves are at the same depth.

The idea is that by using flexibility in the sizes and degrees of nodes, we will be able to keep trees perfectly balanced (in the sense of all leaves being at the same level) while still being able to do inserts cheaply. Note that the case of $t = 2$ is called a 2-3-4 tree since degrees are 2, 3, or 4.

Example: here is a tree for $t = 3$ (so, non-leaves have between 3 and 6 children—though the root can have fewer—and the maximum size of any node is 5).

```
H M R

A B C D  K L  N O  T Y Z
```

Now, the rules for lookup and insert turn out to be pretty easy:

**Lookup:**
Just do binary search in the array at the root. This will either return the item you are looking for (in which case you are done) or a pointer to the appropriate child, in which case you recurse on that child.

**Insert:**
To insert, walk down the tree as if you are doing a lookup, but if you ever encounter a full node (a node with the maximum $2t-1$ keys in it), perform a split operation on it (described below) before continuing. Finally, insert the new key into the leaf reached.

**Split:**
To split a node, pull the median of its keys up to its parent and then split the remaining $2t-2$ keys into two nodes of $t-1$ keys each (one with the elements less than the median and
Lookups

Binary search in array at root. Finished if find item, else get pointer to appropriate child, recurse.

```
A B C D
H M R
K L
N O
T Y Z
```

An important idea: the problem with the basic binary search tree was that we were not maintaining balance. On the other hand, if we try to maintain a perfectly balanced tree, we will spend too much time rearranging things. So, we want to be balanced but also give ourselves some slack. It’s a bit like how in the median-finding algorithm, we gave ourselves slack by allowing the pivot to be “near” the middle. For B-trees, we will make the tree perfectly balanced, but give ourselves slack by allowing some nodes to have more children than others.

9.4 B-trees and 2-3-4 trees

A B-tree is a search tree where for some pre-specified $t \geq 2$ (think of $t = 2$ or $t = 3$):

- Each node has between $t - 1$ and $2t - 1$ keys in it (except the root has between 1 and $2t - 1$ keys). Keys in a node are stored in a sorted array.
- Each non-leaf has degree (number of children) equal to the number of keys in it plus 1. So, node degrees are in the range $[t, 2t]$ except the root has degree in the range $[2, 2t]$. The semantics are that the $i$th child has items between the $(i - 1)$st and $i$th keys. E.g., if the keys are $[a_1, a_2, \ldots, a_{10}]$ then there is one child for keys less than $a_1$, one child for keys between $a_1$ and $a_2$, and so on, until the rightmost child has keys greater than $a_{10}$.
- All leaves are at the same depth.

The idea is that by using flexibility in the sizes and degrees of nodes, we will be able to keep trees perfectly balanced (in the sense of all leaves being at the same level) while still being able to do inserts cheaply. Note that the case of $t = 2$ is called a 2-3-4 tree since degrees are 2, 3, or 4.

Example: here is a tree for $t = 3$ (so, non-leaves have between 3 and 6 children—though the root can have fewer—and the maximum size of any node is 5).

```
H M R
A B C D
K L
N O
T Y Z
```

Now, the rules for lookup and insert turn out to be pretty easy:

**Lookup:** Just do binary search in the array at the root. This will either return the item you are looking for (in which case you are done) or a pointer to the appropriate child, in which case you recurse.

**Insert:** To insert, walk down the tree as if you are doing a lookup, but if you ever encounter a full node (a node with the maximum $2t - 1$ keys in it), perform a split operation on it (described below) before continuing. Finally, insert the new key into the leaf reached.

**Split:** To split a node, pull the median of its keys up to its parent and then split the remaining $2t - 2$ keys into two nodes of $t - 1$ keys each (one with the elements less than the median and
An important idea: the problem with the basic binary search tree was that we were not maintaining balance. On the other hand, if we try to maintain a perfectly balanced tree, we will spend too much time rearranging things. So, we want to be balanced but also give ourselves some slack. It's a bit like how in the median-finding algorithm, we gave ourselves slack by allowing the pivot to be "near" the middle. For B-trees, we will make the tree perfectly balanced, but give ourselves slack by allowing some nodes to have more children than others.

9.4 B-trees and 2-3-4 trees

A **B-tree** is a search tree where for some pre-specified $t \geq 2$ (think of $t = 2$ or $t = 3$):

- Each node has between $t - 1$ and $2t - 1$ keys in it (except the root has between $1$ and $2t - 1$ keys). Keys in a node are stored in a sorted array.
- Each non-leaf has degree (number of children) equal to the number of keys in it plus 1. So, node degrees are in the range $[t, 2t]$ except the root has degree in the range $[2, 2t]$. The semantics are that the $i$th child has items between the $(i-1)$st and $i$th keys. E.g., if the keys are $[a_1, a_2, \ldots, a_{10}]$ then there is one child for keys less than $a_1$, one child for keys between $a_1$ and $a_2$, and so on, until the rightmost child has keys greater than $a_{10}$.
- All leaves are at the same depth.

The idea is that by using flexibility in the sizes and degrees of nodes, we will be able to keep trees perfectly balanced (in the sense of all leaves being at the same level) while still being able to do inserts cheaply. Note that the case of $t = 2$ is called a **2-3-4 tree** since degrees are 2, 3, or 4.

**Example:** here is a tree for $t = 3$ (so, non-leaves have between 3 and 6 children—though the root can have fewer—and the maximum size of any node is 5).

```
  K  L  N  O
 /   /   /   /
H   M   R   T Y Z
A   B   C   D
```

Now, the rules for lookup and insert turn out to be pretty easy:

**Lookup:**
Just do binary search in the array at the root. This will either return the item you are looking for (in which case you are done) or a pointer to the appropriate child, in which case you recurse on that child.

**Insert:**
To insert, walk down the tree as if you are doing a lookup, but if you ever encounter a full node (a node with the maximum $2t - 1$ keys in it), perform a **split** operation on it (described below) before continuing. Finally, insert the new key into the leaf reached.

**Split:**
To split a node, pull the median of its keys up to its parent and then split the remaining $2t - 2$ keys into two nodes of $t - 1$ keys each (one with the elements less than the median and the other with the elements greater than the median). Reconnect appropriately.

Obvious approach: do a lookup, put $x$ in leaf where it should be.

- Example: insert $E$
An important idea: the problem with the basic binary search tree was that we were not maintaining balance. On the other hand, if we try to maintain a perfectly balanced tree, we will spend too much time rearranging things. So, we want to be balanced but also give ourselves some slack. It's a bit like how in the median-finding algorithm, we gave ourselves slack by allowing the pivot to be "near" the middle. For B-trees, we will make the tree perfectly balanced, but give ourselves slack by allowing some nodes to have more children than others.

**9.4 B-trees and 2-3-4 trees**

A B-tree is a search tree where for some pre-specified \( t \geq 2 \) (think of \( t = 2 \) or \( t = 3 \)):

- Each node has between \( t - 1 \) and \( 2t - 1 \) keys in it (except the root has between 1 and \( 2t - 1 \) keys). Keys in a node are stored in a sorted array.
- Each non-leaf has degree (number of children) equal to the number of keys in it plus 1. So, node degrees are in the range \([t, 2t]\) except the root has degree in the range \([2, 2t]\). The semantics are that the \( i \)th child has items between the \((i-1)\)st and \( i \)th keys. E.g., if the keys are \([a_1, a_2, \ldots, a_{10}]\) then there is one child for keys less than \( a_1 \), one child for keys between \( a_1 \) and \( a_2 \), and so on, until the rightmost child has keys greater than \( a_{10} \).
- All leaves are at the same depth.

The idea is that by using flexibility in the sizes and degrees of nodes, we will be able to keep trees perfectly balanced (in the sense of all leaves being at the same level) while still being able to do inserts cheaply. Note that the case of \( t = 2 \) is called a 2-3-4 tree since degrees are 2, 3, or 4.

**Example:** here is a tree for \( t = 3 \) (so, non-leaves have between 3 and 6 children—though the root can have fewer—and the maximum size of any node is 5).

```
H    M    R
/ \   / \   / \  
A  B  C  D  K  L  N  O  T  Y  Z
```

Now, the rules for lookup and insert turn out to be pretty easy:

**Lookup:**
- Just do binary search in the array at the root. This will either return the item you are looking for (in which case you are done) or a pointer to the appropriate child, in which case you recurse on that child.

**Insert:**
- To insert, walk down the tree as if you are doing a lookup, but if you ever encounter a full node (a node with the maximum \( 2t - 1 \) keys in it), perform a split operation on it (described below) before continuing.
- Finally, insert the new key into the leaf reached.

**Split:**
- To split a node, pull the median of its keys up to its parent and then split the remaining \( 2t - 2 \) keys into two nodes of \( t - 1 \) keys each (one with the elements less than the median and the other with the elements greater than the median).

Obvious approach: do a lookup, put \( x \) in leaf where it should be.

- Example: insert \( E \)
- Problem: What if leaf is full (already has \( 2t - 1 \) keys)?
An important idea: the problem with the basic binary search tree was that we were not maintaining balance. On the other hand, if we try to maintain a perfectly balanced tree, we will spend too much time rearranging things. So, we want to be balanced but also give ourselves some slack. It's a bit like how in the median-finding algorithm, we gave ourselves slack by allowing the pivot to be "near" the middle. For B-trees, we will make the tree perfectly balanced, but give ourselves slack by allowing some nodes to have more children than others.

### B-trees and 2-3-4 trees

A **B-tree** is a search tree where for some pre-specified $t \geq 2$ (think of $t = 2$ or $t = 3$):

- Each node has between $t - 1$ and $2t - 1$ keys in it (except the root has between 1 and $2t - 1$ keys). Keys in a node are stored in a sorted array.
- Each non-leaf has degree (number of children) equal to the number of keys in it plus 1. So, node degrees are in the range $[t, 2t]$ except the root has degree in the range $[2, 2t]$. The semantics are that the $i$th child has items between the $(i-1)$st and $i$th keys. E.g., if the keys are $[a_1, a_2, \ldots, a_{10}]$ then there is one child for keys less than $a_1$, one child for keys between $a_1$ and $a_2$, and so on, until the rightmost child has keys greater than $a_{10}$.
- All leaves are at the same depth.

The idea is that by using flexibility in the sizes and degrees of nodes, we will be able to keep trees perfectly balanced (in the sense of all leaves being at the same level) while still being able to do inserts cheaply. Note that the case of $t = 2$ is called a **2-3-4 tree** since degrees are 2, 3, or 4.

**Example:** here is a tree for $t = 3$ (so, non-leaves have between 3 and 6 children—though the root can have fewer—and the maximum size of any node is 5).

```
H   M   R
A  B  C  D K  L N  O T  Y  Z
```

Now, the rules for lookup and insert turn out to be pretty easy:

- **Lookup:** Just do binary search in the array at the root. This will either return the item you are looking for (in which case you are done) or a pointer to the appropriate child, in which case you recurse on that child.
- **Insert:** To insert, walk down the tree as if you are doing a lookup, but if you ever encounter a full node (a node with the maximum $2t - 1$ keys in it), perform a **split** operation on it (described below) before continuing. Finally, insert the new key into the leaf reached.
- **Split:** To split a node, pull the median of its keys up to its parent and then split the remaining $2t - 2$ keys into two nodes of $t - 1$ keys each (one with the elements less than the median and one with the elements greater than the median). Reconnect appropriately.

Obvious approach: do a lookup, put $x$ in leaf where it should be.

- Example: insert $E$

Problem: What if leaf is full (already has $2t - 1$ keys)?

**Split:**
- Only used on full nodes (nodes with $2t - 1$ keys) whose parents are not full.
- Pull median of its keys up to its parent
- Split remaining $2t - 2$ keys into two nodes of $t - 1$ keys each. Reconnect appropriately.
Insert (continued)

Insert: do a lookup and insert at leaf, but when we encounter a full node on way down, split it.
Insert (continued)

Insert: do a lookup and insert at leaf, but when we encounter a full node on way down, split it.

![Diagram](image)

Insert E, F into example.
Insert (continued)

Insert: do a lookup and insert at leaf, but when we encounter a full node on way down, split it.

```
  H M R
 /   \   
A B C D  K L  N O  T Y Z
```

Insert \textbf{E, F} into example.

```
  C H M R
 /   \   
A B  DEF  K L  N O  T Y Z
```
Insert (continued)

Insert: do a lookup and insert at leaf, but when we encounter a full node on way down, split it.

```
            H M R
           /   \
          /     \
         A B C D   K L   N O   T Y Z
```

Insert E, F into example.

```
          C H M R
         /     \
        /       \
       A B     DEF   K L   N O   T Y Z
```

Note: since split on the way down, when a node is split, its parent is not full!
Example continued

\[
\text{CHMR}
\]

\[
\text{ABDEFKLNOYZ}
\]
Example continued

Insert S, U, V:

![Tree diagram](image-url)
Example continued

Insert \( S, U, V \):
Example continued

Insert $S, U, V$:

Insert $P$:
Example continued

Insert \(S, U, V\):

\[
\begin{array}{c}
\text{C H M R U} \\
A B \quad \text{DEF} \quad K L \quad NO \quad \text{STVYZ}
\end{array}
\]

Insert \(P\):

\[
\begin{array}{c}
\text{M} \\
\text{C H R U} \\
A B \quad \text{DEF} \quad K L \quad NOP \quad \text{STVYZ}
\end{array}
\]
Insert: Correctness sketch

Induction. Start with a valid B-tree, insert $x$. 
Insert: Correctness sketch

Induction. Start with a valid B-tree, insert $x$.

Third property (all leaves at same depth):
Insert: Correctness sketch

Induction. Start with a valid B-tree, insert $x$.

Third property (all leaves at same depth): Tree grows up. ✓
Insert: Correctness sketch

Induction. Start with a valid B-tree, insert $x$.

Third property (all leaves at same depth): Tree grows up. ✓

First property (all non-leaves other than root have between $t - 1$ and $2t - 1$ keys):
Insert: Correctness sketch

Induction. Start with a valid B-tree, insert $x$.

Third property (all leaves at same depth): Tree grows up. ✓

First property (all non-leaves other than root have between $t - 1$ and $2t - 1$ keys):
  - No split:
Insert: Correctness sketch

Induction. Start with a valid B-tree, insert $x$.

Third property (all leaves at same depth): Tree grows up. ✓

First property (all non-leaves other than root have between $t - 1$ and $2t - 1$ keys):
  - No split: only leaf changes, was not full (or would have split)
Insert: Correctness sketch

Induction. Start with a valid B-tree, insert $x$.

Third property (all leaves at same depth): Tree grows up. ✓

First property (all non-leaves other than root have between $t-1$ and $2t-1$ keys):
  - No split: only leaf changes, was not full (or would have split)
  - Split:

/uni25B8
Insert: Correctness sketch

Induction. Start with a valid B-tree, insert $x$.

Third property (all leaves at same depth): Tree grows up. ✓

First property (all non-leaves other than root have between $t - 1$ and $2t - 1$ keys):
- No split: only leaf changes, was not full (or would have split)
- Split: Parent was not full. New nodes have exactly $t - 1$ keys.
Insert: Correctness sketch

Induction. Start with a valid B-tree, insert $x$.

Third property (all leaves at same depth): Tree grows up. ✓

First property (all non-leaves other than root have between $t - 1$ and $2t - 1$ keys):
  - No split: only leaf changes, was not full (or would have split)
  - Split: Parent was not full. New nodes have exactly $t - 1$ keys.

Second property (correct degrees, subtrees have keys in correct ranges):

Hooked nodes up correctly after split.
Insert: Correctness sketch

Induction. Start with a valid B-tree, insert $x$.

Third property (all leaves at same depth): Tree grows up. ✓

First property (all non-leaves other than root have between $t-1$ and $2t-1$ keys):
  ▶ No split: only leaf changes, was not full (or would have split)
  ▶ Split: Parent was not full. New nodes have exactly $t-1$ keys.

Second property (correct degrees, subtrees have keys in correct ranges): Hooked nodes up correctly after split. ✓
B-tree running time

Suppose $n$ keys, depth $d$
B-tree running time

Suppose \( n \) keys, depth \( d \leq O(\log_t n) \)
B-tree running time

Suppose $n$ keys, depth $d \leq O(\log_t n)$

Lookup:
  - Binary search on array in each node we pass through
B-tree running time

Suppose \( n \) keys, depth \( d \leq O(\log_t n) \)

Lookup:
- Binary search on array in each node we pass through \( \implies O(\log t) \) time per node.

Lookup time:
- Total time \( O(d \times \log t) \)
  - \( = O(\log t \times \log t \times \log t) \)
  - \( = O(\log n \log \log t) \) total

Insert:
- Same as insert, but need to split on the way down & insert into leaf

Insert into leaf:
- \( O(t) \)

Splitting time:
- \( O(t) \) per split \( \implies O(td) = O(t \log t \log n) \) total

Insert:
- \( O(t \log t \log n) \) total
B-tree running time

Suppose \( n \) keys, depth \( d \leq O(\log_t n) \)

Lookup:
- Binary search on array in each node we pass through \( \implies O(\log t) \) time per node.
- Total time \( O(d \times \log t) = O(\log_t n \times \log t) = O\left(\frac{\log n}{\log t} \times \log t\right) = O(\log n) \)
B-tree running time

Suppose $n$ keys, depth $d \leq O(\log_t n)$

Lookup:
- Binary search on array in each node we pass through $\implies O(\log t)$ time per node.
- Total time $O(d \times \log t) = O(\log_t n \times \log t) = O(\frac{\log n}{\log t} \times \log t) = O(\log n)$

Insert:
- Same as insert, but need to split on the way down & insert into leaf
- Total: lookup time + splitting time + time to insert into leaf
- Insert into leaf: $O(t)$
- Splitting time: $O(t)$ per split $\implies O(td) = O(t \log_t n) = O(t \log t \log n)$ total
B-tree running time

Suppose $n$ keys, depth $d \leq O(\log_t n)$

Lookup:
- Binary search on array in each node we pass through $\implies O(\log t)$ time per node.
- Total time $O(d \times \log t) = O(\log_t n \times \log t) = O(\frac{\log n}{\log t} \times \log t) = O(\log n)$

Insert:
- Same as insert, but need to split on the way down & insert into leaf

$O(\log n)$
B-tree running time

Suppose \( n \) keys, depth \( d \leq O(\log t n) \)

Lookup:
- Binary search on array in each node we pass through \( \Rightarrow O(\log t) \) time per node.
- Total time \( O(d \times \log t) = O(\log_t n \times \log t) = O\left(\frac{\log n}{\log t} \times \log t\right) = O(\log n) \)

Insert:
- Same as insert, but need to split on the way down & insert into leaf
- Total: lookup time + splitting time + time to insert into leaf

\( O\left(\frac{\log n}{\log t} \times \log t\right) = O(\log n) \)
B-tree running time

Suppose $n$ keys, depth $d \leq O(\log_t n)$

Lookup:
- Binary search on array in each node we pass through $\implies O(\log t)$ time per node.
- Total time $O(d \times \log t) = O(\log_t n \times \log t) = O(\frac{\log n}{\log t} \times \log t) = O(\log n)$

Insert:
- Same as insert, but need to split on the way down & insert into leaf
- Total: lookup time + splitting time + time to insert into leaf
  - Insert into leaf:
B-tree running time

Suppose \( n \) keys, depth \( d \leq O(\log_t n) \)

Lookup:
- Binary search on array in each node we pass through \( \implies O(\log t) \) time per node.
- Total time \( O(d \times \log t) = O(\log_t n \times \log t) = O(\frac{\log n}{\log t} \times \log t) = O(\log n) \)

Insert:
- Same as insert, but need to split on the way down & insert into leaf
- Total: lookup time + splitting time + time to insert into leaf
  - Insert into leaf: \( O(t) \)
B-tree running time

Suppose $n$ keys, depth $d \leq O(\log_t n)$

Lookup:
- Binary search on array in each node we pass through $\implies O(\log t)$ time per node.
- Total time $O(d \times \log t) = O(\log_t n \times \log t) = O(\frac{\log n}{\log t} \times \log t) = O(\log n)$

Insert:
- Same as insert, but need to split on the way down & insert into leaf
- Total: lookup time + splitting time + time to insert into leaf
  - Insert into leaf: $O(t)$
  - Splitting time:
B-tree running time

Suppose $n$ keys, depth $d \leq O(\log_t n)$

Lookup:
- Binary search on array in each node we pass through $\implies O(\log t)$ time per node.
- Total time $O(d \times \log t) = O(\log_t n \times \log t) = O(\frac{\log n}{\log t} \times \log t) = O(\log n)$

Insert:
- Same as insert, but need to split on the way down & insert into leaf
- Total: lookup time + splitting time + time to insert into leaf
  - Insert into leaf: $O(t)$
  - Splitting time: $O(t)$ per split
B-tree running time

Suppose \( n \) keys, depth \( d \leq O(\log_t n) \)

Lookup:
- Binary search on array in each node we pass through \( \implies O(\log t) \) time per node.
- Total time \( O(d \times \log t) = O(\log_t n \times \log t) = O\left(\frac{\log n}{\log t} \times \log t\right) = O(\log n) \)

Insert:
- Same as insert, but need to split on the way down & insert into leaf
- Total: lookup time + splitting time + time to insert into leaf
  - Insert into leaf: \( O(t) \)
  - Splitting time: \( O(t) \) per split \( \implies O(td) = O(t \log_t n) \) total
B-tree running time

Suppose \( n \) keys, depth \( d \leq O(\log_t n) \)

Lookup:
- Binary search on array in each node we pass through \( \implies O(\log t) \) time per node.
- Total time \( O(d \times \log t) = O(\log_t n \times \log t) = O(\frac{\log n}{\log t} \times \log t) = O(\log n) \)

Insert:
- Same as insert, but need to split on the way down & insert into leaf
- Total: lookup time + splitting time + time to insert into leaf
  - Insert into leaf: \( O(t) \)
  - Splitting time: \( O(t) \) per split \( \implies O(td) = O(t \log_t n) \) total
- \( O(t \log_t n) = O(\frac{t}{\log_t \log n}) \) total
B-tree notes

Used a lot in databases

- Large $t$: shallow trees. Fits well with memory hierarchy
B-tree notes

Used a lot in databases
  - Large $t$: shallow trees. Fits well with memory hierarchy

$t = 2$:
  - 2-3-4 tree
  - Can be implemented as *binary* tree using *red-black trees*
Red-Black Trees
Red-Black Trees: Intro

B-Trees great, but binary is nice: lookups very simple!
Want *binary* balanced tree.
Red-Black Trees: Intro

B-Trees great, but binary is nice: lookups very simple!
Want *binary* balanced tree.
- Classical and super important data structure question
- Many solutions!
Red-Black Trees: Intro

B-Trees great, but binary is nice: lookups very simple!
Want *binary* balanced tree.
- Classical and super important data structure question
- Many solutions!

Most famous: *red-black trees*
- Default in Linux kernel, used to optimize Java HashMap, . . .
- Today: Quick overview, connection to 2-3-4 trees.
- *Not* traditional or practical point of view on red-black trees. See book!
2-3-4 trees to binary

Can we turn a 2-3-4 tree into a binary tree with all the same properties?
2-3-4 trees to binary

Can we turn a 2-3-4 tree into a binary tree with all the same properties?

- No: can’t have perfect balance!
2-3-4 trees to binary

Can we turn a 2-3-4 tree into a binary tree with all the same properties?

- No: can’t have perfect balance!
- Just need depth $O(\log n)$
2-3-4 trees to binary
Can we turn a 2-3-4 tree into a binary tree with all the same properties?
  - No: can’t have perfect balance!
  - Just need depth $O(\log n)$

Nodes in 2-3-4 tree have degree 2, 3, or 4
Two easy cases.

2-3-4 trees to binary

Can we turn a 2-3-4 tree into a binary tree with all the same properties?

- No: can’t have perfect balance!
- Just need depth $O(\log n)$

Nodes in 2-3-4 tree have degree 2, 3, or 4

- Degree 2: good!
2-3-4 trees to binary

Can we turn a 2-3-4 tree into a binary tree with all the same properties?

- No: can’t have perfect balance!
- Just need depth $O(\log n)$

Nodes in 2-3-4 tree have degree 2, 3, or 4

- Degree 2: good!
- Degree 4:
2-3-4 trees to binary

Can we turn a 2-3-4 tree into a binary tree with all the same properties?

- **No**: can’t have perfect balance!
- Just need depth $O(\log n)$

Nodes in 2-3-4 tree have degree 2, 3, or 4

- Degree 2: good!
- Degree 4:
2-3-4 trees to binary

Can we turn a 2-3-4 tree into a binary tree with all the same properties?

- No: can’t have perfect balance!
- Just need depth $O(\log n)$

Nodes in 2-3-4 tree have degree 2, 3, or 4

- Degree 2: good!
- Degree 4:

- Degree 3:
2-3-4 trees to binary

Can we turn a 2-3-4 tree into a binary tree with all the same properties?

- No: can’t have perfect balance!
- Just need depth $O(\log n)$

Nodes in 2-3-4 tree have degree 2, 3, or 4

- Degree 2: good!
- Degree 4:

![Diagram of degree 4 nodes]

- Degree 3:

![Diagram of degree 3 nodes]
Important Properties

Two easy cases. Switch colors.

Two hard cases. Use rotations.
  - Do single rotation
  - Double rotation
Important Properties

1. Never have two red edges in a row.
   - Red edge is “internal”, never have more than one “internal” edge in a row.
Important Properties

1. Never have two red edges in a row.
   - Red edge is “internal”, never have more than one “internal” edge in a row.

2. Every leaf has same number of black edges on path to root (black-depth)
   - Each black edge is a 2-3-4 tree edge
   - All leaves in 2-3-4 tree at same distance from root
Important Properties

1. Never have two red edges in a row.
   - Red edge is “internal”, never have more than one “internal” edge in a row.

2. Every leaf has same number of black edges on path to root (black-depth)
   - Each black edge is a 2-3-4 tree edge
   - All leaves in 2-3-4 tree at same distance from root

\[ \text{path from root to deepest leaf} \leq 2 \times \text{path to shallowest leaf} \]
Important Properties

1. Never have two red edges in a row.
   - Red edge is “internal”, never have more than one “internal” edge in a row.

2. Every leaf has same number of black edges on path to root (black-depth)
   - Each black edge is a 2-3-4 tree edge
   - All leaves in 2-3-4 tree at same distance from root

\[ \text{path from root to deepest leaf } \leq 2 \times \text{path to shallowest leaf} \]
\[ \text{depth } \leq O(\log n) \]
Insert

Want to insert while preserving two properties.
Insert

Want to insert while preserving two properties.
2-3-4 trees: split full nodes on way down.
Insert

Want to insert while preserving two properties.

2-3-4 trees: split full nodes on way down.

Easy cases:
Insert

Want to insert while preserving two properties.
2-3-4 trees: split full nodes on way down.

Easy cases:
Insert

Want to insert while preserving two properties.
2-3-4 trees: split full nodes on way down.

Easy cases:
Insert

Want to insert while preserving two properties.
2-3-4 trees: split full nodes on way down.

Easy cases:
**Insert**

Want to insert while preserving two properties.
2-3-4 trees: split full nodes on way down.

Easy cases:
Insert

Want to insert while preserving two properties.

2-3-4 trees: split full nodes on way down.

Easy cases:

Harder cases:
Tree Rotations

Used in many different tree constructions.
Tree Rotations

Used in many different tree constructions.
Using Rotations

Can use rotations to “fix” hard cases. Example:

- Change colors inserting $G$
- Right rotate $R$ →
- Left rotate $E$ →
A few more complications to deal with – see lecture notes, textbook.
A few more complications to deal with – see lecture notes, textbook.

Main points:

- Red-Black trees can be thought of as a binary implementation of 2-3-4 trees
- Approximately balanced, so $O(\log n)$ lookup time
- Insert time (basically) same as 2-3-4 tree, so also $O(\log n)$.
- See book for direct approach (not through 2-3-4 trees).