B-trees:

They’re not just binary anymore!
The balance problem

• Binary search trees provide efficient search mechanism only if they’re balanced
• Balance depends on the order in which nodes are added to a tree

This tree is balanced because data arrived in this order: 4, 2, 6, 1, 5, 3, 7

If data arrive in this order: 7, 6, 5, 4, 3, 2, 1

…result is this
One possible solution: B-trees

- B-tree nodes hold data
- B-tree nodes have (many) more than 2 children
- Each node contains more than one data entry
- Set of rules governs behavior of B-trees
B-tree rules

• Two constants need to be defined to determine the number of entries stored in a node:
  – MINIMUM: every node (other than root) has at least MINIMUM entries
  – MAXIMUM: twice the value of minimum
B-tree rules

• Rule 1: Root may have as few as one entry; every other node has at least MINIMUM entries

• Rule 2: Maximum number of entries in a node is MAXIMUM (2 * MINIMUM)

• Rule 3: Each node of a B-tree contains a partially-filled arrays of entries, sorted from smallest to largest
B-tree rules

• Rule 4: The number of subtrees below a non-leaf node is one more than the number of entries in the node
  – example: if a node has 10 entries, it has 11 children
  – entries in subtrees are organized according to rule #5
B-tree rules

• Rule 5: For any non-leaf node:
  – an entry at index $n$ is greater than all entries in subtree $n$ of the node;
  – an entry at index $n$ is less than all entries in subtree $n+1$ of the node

• Rule 6: Every leaf has the same depth
Example B-tree

MINIMUM=2
MAXIMUM=4

For this node, count=2, children=3
Using a B-tree to implement the Set ADT

• Characteristics of a set:
  – similar to bag (container holding a collection of items)
  – each item is unique; bag can contain more than one instance of a particular value, but a set only contains one
Set operations

• construct / clone
• add: add one item (if value not already in set)
• remove: searches for target value; if found, deletes value and returns true – otherwise returns false
• contains: returns true if set contains target value
Set implemented as B-tree

• We will use the Set ADT to illustrate the use of a B-tree
• The class we’re defining (IntBalancedSet) describes a single object, the root node of a B-tree
• Keep in mind that, as with most of the trees we have studied, the concept of a B-tree is inherently recursive; every node can be considered the root node of a subtree
Set class definition

public class IntBalancedSet implements Cloneable {

    private final int MINIMUM = 1;
    private final int MAXIMUM = 2*MINIMUM;
    int dataCount;
    int[ ] data = new int[MAXIMUM + 1];
        // # of items stored at this node
    int childCount;
        // # of children of this node
    IntBalancedSet[ ] subset = new IntBalancedSet[MAXIMUM + 2];
        // each element of subset is a reference to a set – represented
        // here as a partially filled array of sets

Set class definition - constructor

public IntBalancedSet() {
    dataCount = 0;
    childCount = 0;
}
Invariant for IntBalancedSet class

• Items in the set are stored in a B-tree; each child node is the root of a smaller B-tree
• A tally of the number of items in the root node is kept in member variable count
• The items in the root node are stored in the data array in data[0] … data[count-1]
• If the root has subtrees, they are stored in sets pointed to by pointers in the subset array in subset[0] … subset[children-1]
Searching for item in a B-tree

- Check for target in root; if found there, return true
- If target isn’t found in root, and root has no children, return false
- If root has children but doesn’t contain target, make recursive call to search the subtree that could contain the target
Implementation of Set member method contains()

```java
public boolean contains(int target) {
    int i;
    for (i=0; i<data.length && data[i] < target; i++);
    if (i < data.length && data[i] == target) //we found it
        return true;
    if (childCount == 0) // root has no subsets
        return false;
    return subset[i].contains(target);
}
```
Inserting an item into a B-tree

• Easiest method: relax the rules!
• Perform “loose” insertion: allow the root node to end up with one entry too many
• After loose insertion, can split root node if necessary, creating new root node and increasing height of the tree
Insertion example

MINIMUM = 1
MAXIMUM = 2

Data entered in this order: 0,1,2,3,4,5,6,7,8

Regardless of data entry order, tree will remain balanced
Methods needed for insertion

• Public add method:
  – performs “loose” insertion;
  – if loose insertion results in excess entries in a child node, grows the tree upward

• Private methods looseAdd and fixExcess are called by the public method
Loose insertion

• Loose insertion does most of the work of inserting a value:
  – finds slot where value should go, saving index; if correct slot not found in root, index set to root’s count value
  – if index is within root’s data array, and root has no children, shift entries to the right and add new entry, incrementing count
  – if root has children make recursive call on subset at index
Implementation of looseAdd

private void looseAdd(int entry) {
    int i;
    for (i = 0; i < dataCount && data[i] < entry; i++);
    if (i < data.length && data[i] == entry)
        return;
    if (childCount == 0) { // add entry at this node
        for (int x = data.length-1; x > i; x--)
            data[x] = data[x-1]; // shift elements to make room
        data[i] = entry;
        dataCount++;
    }
    else { // add entry to a subset, housekeep
        subset[i].looseAdd(entry);
        if (subset[i].dataCount > MAXIMUM)
            fixExcess(i);
    }
}
Fixing nodes with excess entries

• Loose insertion can result in a node containing one too many entries

• A node with an excess will always have an odd number of entries – to fix:
  – middle entry is pushed up to the parent node
  – remaining entries, along with any subsets, are split between the existing child and a new child
fixExcess method

• Called by looseAdd when a child node is involved
• Called by add when action of looseAdd causes there to be an excess entry in the root node (of the entire tree)
Implementation of fixExcess

private void fixExcess(int i)
{
    int ct;
    // copy middle entry of subset to root:
    for(ct = dataCount; ct > i; ct--)
        data[ct] = data[ct-1];
    data[i] = subset[i].data[MINIMUM];
    dataCount++;

    // continued on next slide …
Implementation of fixExcess

// split child into 2 subsets:
IntBalancedSet leftChild = new IntBalancedSet(),
    rightChild = new IntBalancedSet();
leftChild.dataCount = MINIMUM;
rightChild.dataCount = MINIMUM;
// copy data from original subset into 2 splits:
for (ct = 0; ct < MINIMUM; ct++) {
    leftChild.data[ct] = subset[i].data[ct];
    rightChild.data[ct] = subset[i].data[ct+MINIMUM+1];
}
// continued on next slide ...
Implementation of fixExcess

// copy subsets of child if they exist:

int subChCt = (subset[i].childCount)/2;
for (ct = 0; ct < subChCt; ct++) {
    leftChild.subset[ct] = subset[i].subset[ct];
    rightChild.subset[ct] = subset[i].subset[ct+subChCt];
}
if(subChCt > 0) {
    leftChild.childCount = MINIMUM + 1;
    rightChild.childCount = MINIMUM + 1;
}
// continued next slide
Implementation of fixExcess

// make room in root's subset array for new children:
subset[childCount] = new IntBalancedSet();
for (ct = childCount; ct > i; ct--)
    subset[ct] = subset[ct-1];
childCount++;

// add new subsets to root's subset array:
subset[i] = leftChild;
subset[i+1] = rightChild;
} // end of method
public void add(int element) {
    looseAdd(element);
    // add data, then check to see if node still OK; if not:
    if (dataCount > MAXIMUM) {
        // get ready to split root node
        IntBalancedSet child = new IntBalancedSet();
        // transfer data to new child:
        for (int x=0; x<dataCount; x++)
            child.data[x] = data[x];
        for (int y=0; y<childCount; y++)
            child.subset[y] = subset[y];
        // continued on next slide
Public add method

// finish setting up child set:
    child.childCount = childCount;
    child.dataCount = dataCount;
// reset current node as empty, with 1 child
    dataCount = 0;
    childCount = 1;
// make new child subset of current node
    subset[0] = child;
// fix problem of empty root node
    fixExcess(0);
}
Removing an item from a B-tree

• Again, simplest method involves relaxing the rules

• Perform “loose” erase -- may end up with an invalid B-tree:
  – might leave root of entire tree with 0 entries
  – might leave root of subtree with less than MINIMUM entries

• After loose erase, restore B-tree
Removing a B-tree entry

- Four methods involved; three are analogous to insertion methods:
  - `remove`: public method -- performs “loose” `remove`, then calls other methods as necessary to restore B-tree
  - `looseRemove`: performs actual removal of data entry; may leave B-tree invalid, with root node having 0 or subtree root having `MINIMUM-1` entries
Removing a B-tree entry

• Additional removal methods:
  – `fixShortage`: deals with the problem of a subtree’s root having MINIMUM-1 entries
  – `removeLargest`: helper method called by `looseRemove` to ensure that root node contains children-1 data entries; works by copying largest data value from a subtree into root
Pseudocode for public remove method

public boolean remove(int target) {
    if (!(looseRemove(target)))
        return false;  // target not found
    if (dataCount == 0 && childCount == 1)
        // root was emptied by looseRemove: shrink the
        // tree by :
        // - setting temporary reference to subset
        // - copying all member variables from
        //   temp to root
        // - deleting original child node
Pseudocode for looseRemove

public boolean looseRemove(int target) {
    
    find first index such that data[index] >= target;
    if no such index found, index = count
    if (target not found and isLeaf())
        return false;
    if (target found and isLeaf())
        remove target from data array;
        shift contents to the left and decrement count
    return true;
}
Pseudocode for looseRemove

if (target not found and root has children)
{
    subset[index].loose_remove(target);
    if(subset[index].dataCount < MINIMUM)
        fixShortage(index);
    return true;
}

Pseudocode for looseRemove

if (target found and root has children)
{
    subset[index].removeLargest(data[index]);
    if(subset[index].dataCount < MINIMUM)
        fixShortage(index);
    return true;
}

Action of fixShortage method

• In order to remedy a shortage of entries in subset[n], do one of the following:
  – borrow an entry from the node’s left neighbor (subset[n-1]) or right neighbor (subset[n+1]) if either of these two has more than MINIMUM entries
  – combine subset[n] with either of its neighbors if they don’t have excess entries to give
Pseudocode for fixShortage

public void fixShortage(int x)
{
    if (subset[x-1].dataCount > MINIMUM)
        • shift existing entries in subset[x] over one,
          copy data[x-1] to subset[x].data[0]
          and increment subset[x].dataCount
        • data[x-1] = last item in subset[x-1].data
          and decrement subset[x-1].dataCount
        • if(!(subset[x-1].isLeaf()))
          transfer last child of subset[x-1] to front of
          subset[x], incrementing subset[x].childCount
          and decrementing subset[x-1].childCount
Example 1 for fixShortage

MINIMUM = 2
x = 1
Example 1 for fixShortage

MINIMUM = 2
x = 1
Example 1 for fixShortage

MINIMUM = 2
x = 1
Example 1 for fixShortage

MINIMUM = 2
x = 1
Example 1 for fixShortage

MINIMUM = 2
x = 1
Pseudocode for fixShortage

else if (subset[x+1].dataCount > MINIMUM)
  • increment subset[x].dataCount and copy data[x] to subset[x].data[subset[x].dataCount-1]
  • data[x] = subset[x+1].data[0] and shift entries in subset[x+1].data to the left and decrement subset[x+1].dataCount
  • if (!(subset[x+1].isLeaf()))
     transfer first child of subset[x+1] to subset[x], incrementing subset[x].childCount and decrementing subset[x+1].childCount
Example 2 for fixShortage

MINIMUM = 2

x = 1
Example 2 for fixShortage

MINIMUM = 2
x = 1
Example 2 for fixShortage

MINIMUM = 2
x = 1
Example 2 for fixShortage

MINIMUM = 2
x = 1
Example 2 for fixShortage

MINIMUM = 2
x = 1
Pseudocode for fixShortage

else if (subset[x-1].dataCount == MINIMUM)
  • add data[x-1] to the end of subset[x-1].data
    shift data array leftward, decrementing dataCount and incrementing subset[x-1].dataCount
  • transfer all data items and children from subset[x] to end of subset[x-1]; update values of subset[x-1].dataCount and subset[x-1].childCount, and set subset[x].dataCount and subset[x].childCount to 0
  • delete subset[x] and shift subset array to the left and decrement children
Example 3 for fixShortage

MINIMUM = 2
x = 1
Example 3 for fix_shortage

MINIMUM = 2

x = 1

Diagram showing the allocation of elements.
Example 3 for fix_shortage

MINIMUM = 2
x = 1
Example 3 for fix_shortage

\[ \text{MINIMUM} = 2 \]
\[ x = 1 \]
Example 3 for fix_shortage

MINIMUM = 2
x = 1
else

combine subset[x] with subset[x+1] --
work is similar to previous combination operation:
• borrow an entry from root and add to subset[x]
• transfer all private members from subset[x+1]
to subset[x], and zero out subset[x+1]’s childCount
and dataCount variables
• delete subset[x-1] and update root’s subset information
Example 4 for fixShortage

MINIMUM = 2
x = 0
Example 4 for fixShortage

MINIMUM = 2
x = 0
Trees, Logs and Time Analysis

That’s logs as in logarithms, not Lincoln Logs!
Worst-case times for tree operations

• For a tree of depth $d$, all of the following are $O(d)$ applications in the worst case:
  – adding an entry to a binary search tree, heap or B-tree
  – deleting an entry from a binary search tree, heap or B-tree
  – search for an entry in a binary search tree or B-tree
Depth is not the whole story for binary search trees

• Time analysis on the basis of depth is not always the most useful measure -- these two binary search trees have the same depth:
Analysis based on number of entries in a BST

• The maximum depth of a binary search tree is n-1 (because there must be at least one node at each level)

• So the worst-case time for a binary search tree (O(d)) converts to O(n-1), or just O(n)
Heap analysis

• By definition, a heap is a complete binary tree

• Maximum nodes at each level:
  – root node (level 0): 1 \( (2^0) \) nodes
  – root’s children (level 1): 2 \( (2^1) \) nodes
  – root’s grandchildren (level 2): 4 \( (2^2) \) nodes
  – At level \( d \), there are \( 2^d \) nodes
Heap analysis

• So, for a heap to reach depth d, it must have 
  \[(1 + 2 + 4 + \ldots + 2^{(d-1)}) + 1\] nodes

• Simplifying the formula:
  
  \[
  1 + 1 + 2 + 4 + \ldots + 2^{(d-1)}
  \]
  
  \[
  2 + 2 + 4 + \ldots + 2^{(d-1)}
  \]
  
  \[
  4 + 4 + \ldots + 2^{(d-1)}
  \]
  
  \[2^{(d-1)} + 2^{(d-1)} = 2^d\]
Worst-case times for heap operations

• Since
  
  \[ d \text{ (depth)} = \log_2 2^d \quad \text{and} \]
  
  \[ n \text{ (number of nodes)} \geq 2^d \]
  
  \[ \log_2 n \geq \log_2 2^d \quad \text{so} \quad \log_2 n \geq d \]

• Adding or deleting an entry is \( O(d) \); since \( d \leq \log_2 n \), worst-case scenario for heap operations is \( O(\log_2 n) \) or just \( O(\log n) \)
B-tree analysis

• For all three functions, the number of total steps is a constant (MAXIMUM in the worst case) times the height of the B-tree
• Height is no more than $\log_{M} n$ (where $M$ is MINIMUM and $n$ is the number of entries in the tree)
• Thus, all three functions require no more than $O(\log n)$ operations
Significance of logarithms

• Logarithmic algorithms are those (such as heap and B-tree operations) with worst-case time of $O(\log n)$

• For a logarithmic algorithm, doubling the input size ($N$) will make the time required increase by a (small) fixed number of operations
Significance of logarithms

• Example: adding a new entry to a heap with n entries
  – In worst case, the algorithm may examine as many as $\log_2 n$ nodes
  – Doubling the number of nodes to $2n$ would require the algorithm to examine as many as $\log_2 2n$ nodes -- but that is just 1 more than $\log_2 n$ (example: $\log_2 1024 = 10$, $\log_2 2048 = 11$)