CS61B, Spring 2003 **Discussion #12** **Amir Kamil**
UC Berkeley **4/7/03**

Topics: Balanced Search Trees, Graphs

1 Balanced Search Trees

In the worst case, binary search trees can run as slowly as in linear time due to imbalance introduced by insertions and removals. We can guarantee logarithmic time algorithms by forcing our search trees to be balanced.

There are three main approaches to balanced trees. The first is to use *rotations* to restore balance to a tree after it becomes unbalanced. The second is to grow a tree upwards rather than downwards. The last is to use probablistic algorithms to ensure balance.

1.1 AVL Trees

An example of a structure that uses rotations to retain balance is an *AVL tree*. An AVL tree is a binary search tree that remains balanced as long as no insertion or removal is done. When one of those operations is executed, the tree is checked for balanced and restructured if necessary. This ensures that when an insertion or removal occurs, at most one node will be unbalanced.

It actually is possible to define AVL restructuring without explicitly using rotations, and the resulting algorithm can be applied to any of the cases of imbalance. This is the algorithm we will discuss.

1.1.1 Search

Searching for an element in an AVL tree is exactly the same as in a binary search tree. Since an AVL tree is guaranteed to be balanced, searching is always in $O(\lg n)$.

1.1.2 Insertion

Insertion into an AVL tree proceeds initially as in a binary search tree. The position at which the element is to be inserted is calculated as in a BST, using a search operation. The element is inserted at that position, as in a BST. Now the tree may be unbalanced, so the AVL tree must check for imbalance and perform a restructuring if necessary. In order to quickly determine whether or not the tree is unbalanced, each node must keep track of its height. Then imbalance can be detected in $O(\lg n)$ time by following parent pointers up from the newly inserted node, and comparing the heights of each node's children at each point.

When the first unbalanced node is detected, it must be restructured, and in fact restructuring it will restore balance to the tree. In order to keep track of the relevant nodes, let's name them. Let z be the unbalanced node, y be its child with greater height, and x be y's child with greater height. Thus z is the grandparent of x. In addition, z and y each have another child, and x has two children. Let these children be t_1, t_2, t_3, and t_4, in increasing order of their keys. Also, give x, y, and z additional labels a, b, and c in order of their keys. Figure 1 shows an example of this labeling.

Now move a such that it is the left child of b, c such that it is the right child of b. Assign t_1 to be the left child of a, t_2 to be its right child, t_3 to be the left child of c, and t_4 to be the right child of c. Finally, move b to be in the position that z used to occupy. The tree is now balanced, and still obeys the BST property. Of course, the height field of each node must also be readjusted.

Insertion proceeds initially as in a BST, then must detect imbalance, then perform a restructuring if necessary. The first two take $O(\lg n)$ time, while a restructuring can be done in constant (albeit a large constant) time. So insertion takes $O(\lg n)$ time total.

1.1.3 Removal

Like insertion, removal initially proceeds as in a BST. The node whose key is to be removed is located using a search, and the key is replaced with its successor or predecessor. Now the AVL tree may be unbalanced. So

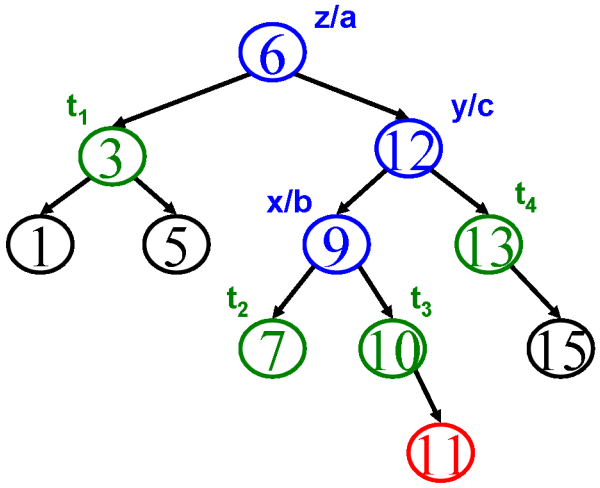


Figure 1: Insertion of element *11* into an AVL tree, causing imbalance. The nodes are labeled as required for restructuring.

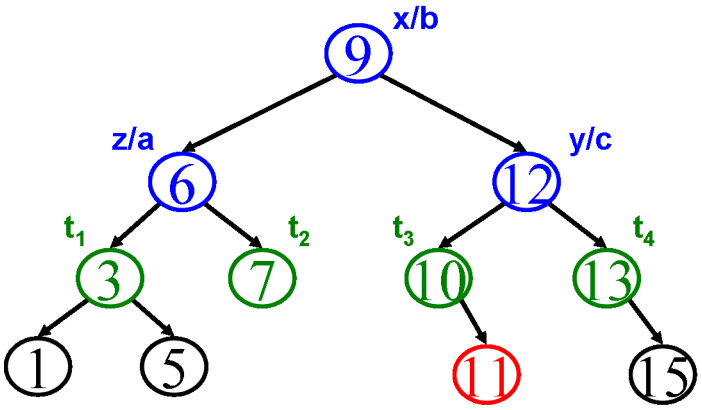


Figure 2: The AVL tree after restructuring.

as in insertion, the first unbalanced ancestor of the removed node (the leaf node that contained the successor or predecessor) is located and restructured. Unlike in insertion, however, a single restructuring may not restore balance to the tree. So after the restructuring operation, it is necessary to keep going upward in the tree up to the root, restructuring as necessary.

Removal proceeds initially as in a BST, then must detect imbalance, then perform multiple restructurings if necessary. The first two take $O(\lg n)$ time, while a restructuring takes constant time. A maximum of one restructuring per level is required, so at most $O(\lg n)$ restructuring operations need to be done. So removal takes $O(\lg n)$ time total.

1.1.4 Disadvantages

If you look in the Java API, you'll find that there is no implementation of AVL trees in the library. Why is this, if an AVL tree guarantees logarithmic operations? This is because AVL trees are difficult to implement, and have high constant factors for some operations. Restructuring is an expensive operation, and an AVL tree may have to perform $\lg n$ such operations in a removal. Java does have a balanced tree that uses rotations, the `TreeMap` class, which uses *red-black* trees. Such trees require only one restructuring for a removal or an insertion.

1.2 B-Trees

Another approach to balanced trees is to grow a tree *upward*. In order for this to make any sense, we must modify our tree structure. We examine *B-trees* as an example.

The major modification B-trees make is that each node may contain multiple elements. B-trees have an *order*, which is the maximum number of children a node can have. All internal nodes except the root node in an order n tree must have at least $\lceil \frac{n}{2} \rceil$ children. A node that has k children must contain $k-1$ elements. Leaf nodes must be all on the last level and are always empty. For simplicity, we will not draw them in, but they are required to maintain the above properties.

B-trees also have an ordering property similar to that of BSTs. Within a single level, all elements must be less than the element to its right. The children and elements in a node are arranged in an alternating pattern, so between any two children is an element. The subtree to the left of a key may only contain elements less than it, and the subtree to the right only elements greater than it.

Searching a B-tree is almost exactly like searching a BST. Insertion also starts out the same way, with a position in the last internal level located for the new element, and the element placed there. However, this may cause *overflow*, in which the node the element was inserted in may now contain more than n elements. In such a case, the node is split, with each side getting half the elements, but one element promoted to the next level. So the parent node gets another child due to the split, but it also gets another element, so the property that there is one more child than elements is preserved. This may cause the parent node to overflow, but that can be dealt with in the same manner.

Removal is also done as in a BST. The element to be removed is swapped to the last internal level as in a BST and then deleted. This deletion may cause an *underflow*, in which the affected node now contains fewer than $\lceil \frac{n}{2} \rceil$ elements. In such a case, the node is merged with one if its siblings, and the element between them in the parent node is demoted to the newly combined node. Thus the parent node loses a child but also loses an element, so the child to element relationship is preserved. This may cause either overflow of the combined node or underflow of the parent node, but either case can be dealt with as before. (*NOTE: The removal algorithm in the text and lecture notes is different than this one, and is somewhat more complicated. You should know that algorithm. I have not reproduced it here since it won't improve one's understanding of a B-tree.*)

1.3 Skiplists (optional)

One structure that is not much of a tree at all is a *skiplist*. It is actually a two-dimensional linked list, but with elements duplicated in the vertical dimension. This duplication is probabilistic and results in logarithmic operations for the structure.

The bottom level of a skiplist always contains all the elements in the skiplist, as well as two more items representing $-\infty$ and $+\infty$. The elements in a particular level are ordered, so $-\infty$ is always all the way

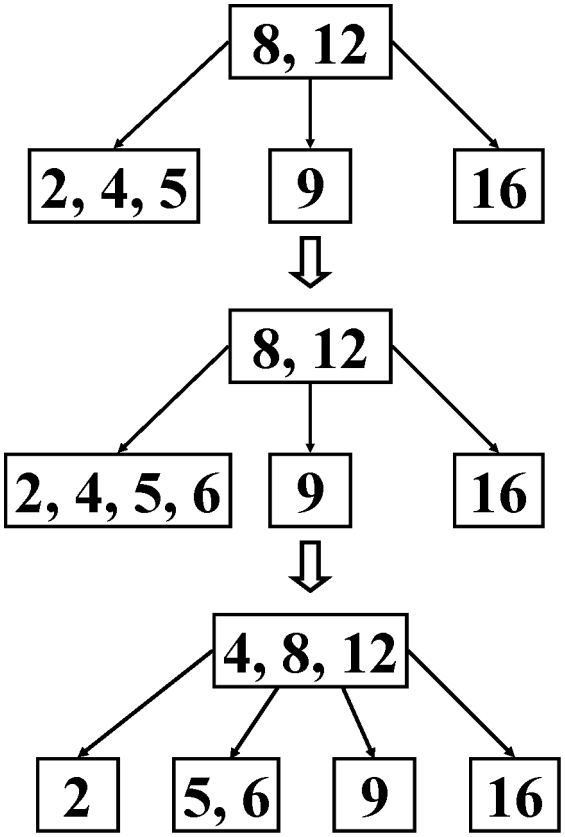


Figure 3: Insertion into an order 4 B-tree. When an overflow occurs, the overflowed node is split, and one key is promoted.

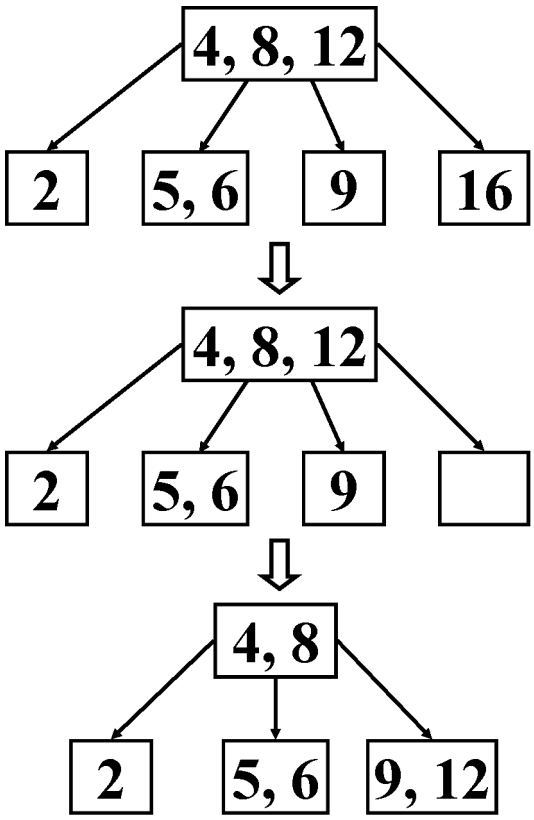


Figure 4: Removal from an order 4 B-tree. When an underflow occurs, the underflowed node is merged with a sibling, and the parent key between them is demoted. A resulting overflow is dealt with as in insertion.

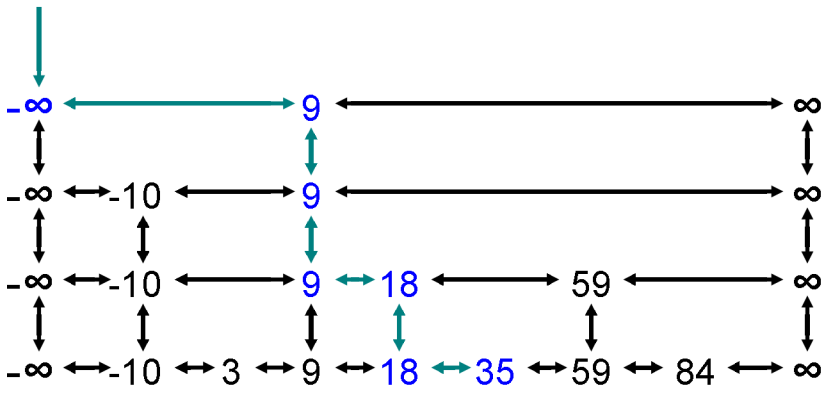


Figure 5: Searching for an element in a skiplist.

to the left and $+\infty$ all the way to the right. Now each subsequent level contains probabilistically half the elements of the previous level. This is accomplished by flipping a coin when an item is inserted to see if it should be added to the next level, and repeating until the coin orders us to stop. Each level must contain $-\infty$ and $+\infty$. The start of the skiplist is the top left corner.

To search for an element in a skiplist, we start at the top left. If the element to our right is larger than the element we are looking for, we go down a level. If it is smaller, we go right. If it is the element, well then we're done. We repeat this procedure from the next position, until we've located the element. Insertion consists of searching for the proper location to insert the new element, placing it there, and repeatedly flipping coins and adding the element to subsequent levels until the coin tells us to stop. Removal is just a search for the element in question, and removing it from each level in which it is.

It is not a trivial task to prove that the above operations take logarithmic time in a skiplist. We will not attempt such a proof here, but just state it as a fact.

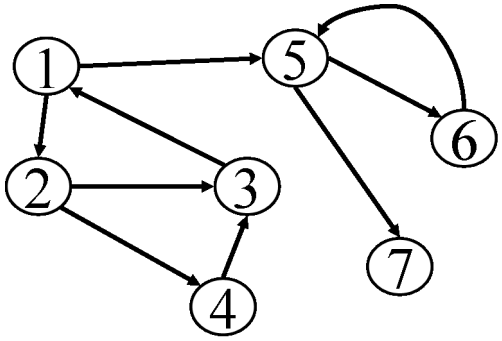
2 Graphs

Recall that trees are extensions of linked lists such that each node may have multiple children. However, trees are still prohibited from having cycles or nodes with multiple parents. If we remove these restrictions, the result is a *graph*.

A graph is defined as a set of *vertices* (nodes) and *edges* that connect vertices. Edges may be *directed* or *undirected*, and may have *weights*. A graph with only undirected edges is an undirected graph, and a graph with directed edges is a directed graph. An undirected graph is *connected* if there exists a path from each node to every other node. A directed graph is *strongly connected* if it satisfies this condition. Directed graphs also have *strongly connected components*, subsets of the graph that are strongly connected. A graph is *cyclic* if a cycle exists in the graph or *acyclic* if none do. Vertices have *in-degree*, the number of edges coming into a vertex, and *out-degree*, the number of edges originating from a vertex. Thus a tree is just a directed acyclic graph (DAG) in which each node has in-degree of 1, except the root which has in-degree of 0.

There are two major representations for graphs, *adjacency matrices* and *adjacency lists*. An adjacency matrix has a column for each vertex, and a row for each. Where there is an edge between vertices, the adjacency matrix has a 1 in the corresponding position, or the edge weight if the graph is weighted. The row corresponds to the start of the edge and the column the end. An undirected graph is actually equivalent to a directed graph in which each edge has a corresponding reverse edge, so its adjacency matrix represention is as such. When a graph is *sparse*, or only has few edges, the adjacency matrix representation wastes a lot of space since it requires quadratic space. A *dense* graph has nearly n^2 edges, so an adjacency matrix is efficient in that case.

The adjacency list representation requires actual objects corresponding to each edge. Then each vertex stores both incoming to and outgoing edges from itself. Such a representation requires space linear in the



$$\begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Node	In	Out
1	31	12, 15

Node	In	Out
2	12	23, 24

Node	In	Out
3	23, 43	31

Node	In	Out
4	24	43

Node	In	Out
5	15, 65	56, 57

Node	In	Out
6	56	65

Node	In	Out
7	57	

Figure 6: A directed graph and its adjacency matrix and adjacency list representations.

number of vertices and edges.