

Learning Outcomes

COMP202 Complexity of Algorithms

Binary Search Trees and Other Search Trees

[See relevant sections in chapters 2 and 3 in
Goodrich and Tamassia.]

At the conclusion of this set of lecture notes, you should:

1. Recall the "binary search" method on sorted arrays.
2. Comprehend how binary search trees are used to mimic the binary search method for arrays.
3. Understand the concept of AVL trees and the mechanism used to maintain binary search trees that satisfy the "height balance" property.
4. Understand the methods for using (2,4)-trees to store and maintain sorted data.

Data Structures

- ▶ Algorithmic computations require data (information) upon which to operate.

The speed of algorithmic computations is (partially) dependent on efficient use of this data.

Data structures are specific methods for storing and accessing information.

We study different kinds of data structures as one kind may be more appropriate than another depending upon the *type* of data stored, and *how* we need to use it for a particular algorithm.

Data Structures (cont.)

Many modern high-level programming languages, such as C++, Java, and Perl, have implemented various data structures in some format.

For specialized data structures that aren't implemented in these programming languages, many can typically be constructed using the more general features of the programming language (e.g. pointers in C++, references in Perl, etc.).

Data Structures: Arrays

Arrays are available in some manner in every high-level programming language. You have encountered this data structure in some programming course(s) before.

- ▶ Abstractly, an *array* is simply a collection of items, each of which has its own unique "index" used to access this data. So if A is an array having n elements, we (typically) access its members as $A[0], A[1], \dots, A[n - 1]$.

Data Structures: Arrays (cont.)

One possible difficulty using arrays in real-life programs is that we must (usually) specify the *size* of the array *as it is created*. If we later need to store more items than the array can currently hold, then we (usually) have to create a new array and copy the old array elements into the newly created array.

So we have quick access to individual array members via indexing, but we need to know how many elements we want to store *beforehand*.

Another possible disadvantage is that it is difficult to insert new items into the "middle" of an array. To do so, we have to copy (shift) items to make space for the new item in the desired position.

So it can be difficult (i.e. *time-consuming*) to maintain data that is *sorted* by using an array.

Data Structures: Linked Lists

You have previously seen linked lists in COMP 108.

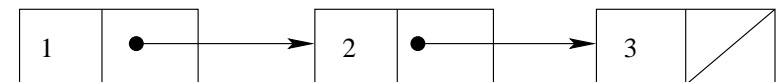
In contrast to arrays, data can be easily inserted *anywhere* in a linked list by inserting a new node into the list and reassigning pointers.

A list abstract data type (ADT) supports: *referring*, *update* (both *insert* and *delete*) as well as *searching* methods.

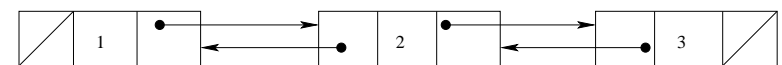
We can implement the list ADT as either a *singly*-, or *doubly*-linked list.

Linked List

- ▶ A *node* in a *singly*-linked list stores a *next* link pointing to next element in list (**null** if element is last element).



- ▶ A *node* in a *doubly*-linked list stores two links: a *next* link, pointing to the next element in list, and a *prev* link, pointing to the previous element in the list.



From now on, we will concentrate on doubly-linked lists.

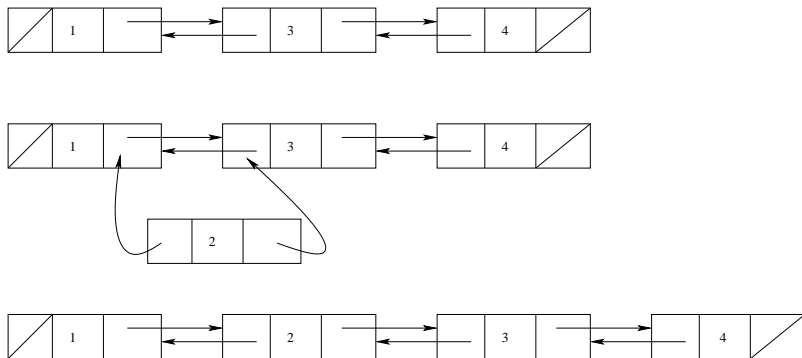
List ADT: Update methods

A list ADT supports the following *update* methods:

- ▶ *replaceElement(p,e)*: p - position, e - element.
- ▶ *swapElements(p,q)*: p, q - positions.
- ▶ *insertFirst(e)*: e - element.
- ▶ *insertLast(e)*: e - element.
- ▶ *insertBefore(p,e)*: p - position, e - element.
- ▶ *insertAfter(p,e)*: p - position, e - element.
- ▶ *remove(p)*: p - position.

List update: Element insertion

Example: *insertAfter(1,2)*



List update: Element insertion

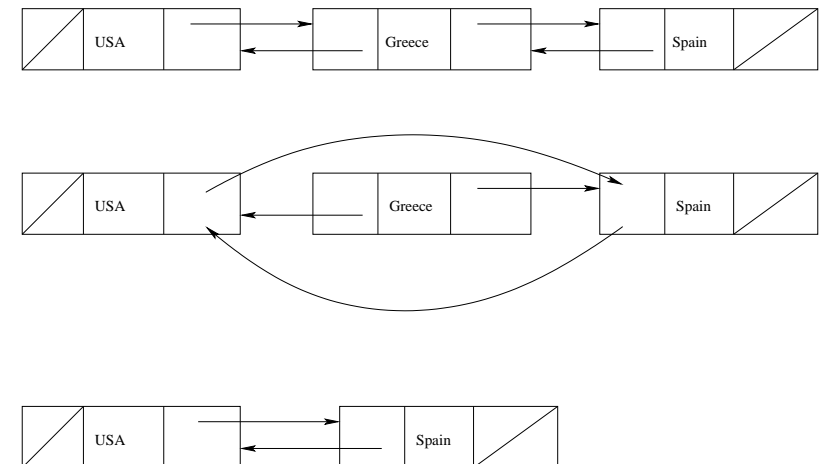
Pseudo-code for *insertAfter(p,e)* :

INSERTAFTER(p, e)

- 1 ▷ Create a new node v
- 2 $v.element \leftarrow e$
- 3 ▷ Link v to its predecessor
- 4 $v.prev \leftarrow p$
- 5 ▷ Link v to its successor
- 6 $v.next \leftarrow p.next$
- 7 ▷ Link p 's old successor to v
- 8 $(p.next).prev \leftarrow v$
- 9 ▷ Link p to its new successor v
- 10 $p.next \leftarrow v$
- 11 **return** v

List update: Element removal

Example: *remove(2)*



List update: Complexity

What is the cost of the *insertion* and *removal* methods?

- ▶ If the address of element p is known, then the cost is $O(1)$.
- ▶ If only the address of the *head* of the list is known, then the cost of an update is $O(p)$ (we need to traverse the list from positions $0, \dots, p$ to first find p).

Ordered Data

We often wish to store data that is ordered in some fashion (typically in numeric order, or alphabetical order, but there may be other ways of ordering it).

Arrays and lists are obvious structures that may be used to store this type of data.

Based on our previous discussions, arrays aren't generally efficient to maintain data where we must add/delete items *and* maintain a sorted order. Linked lists may provide a better method in that case, but it is generally harder to *search for* an item in a list.

Lists: Implementation in Java

Java implements a List class (and related classes that vary in functionality) so you need not create your own.

Method names could differ from the ones given here for our List ADT, and users should check carefully how Java defines its supporting methods to avoid unexpected behavior when using Java List classes.

Ordered Dictionary

A *dictionary* is a *set of elements* and *keys*, supported by dictionary operations:

- ▶ *findElement(k)*: position k
- ▶ *insertElement(k,e)*: position k , element e
- ▶ *removeElement(k)*: position k

An *ordered* dictionary maintains an order relation for its elements, where the items are ordered by their keys.

There are times when the keys and elements are the same, i.e. $k = e$ for each pair.

Sorted Tables

If a dictionary D , is *ordered*, we can store its items in a vector or array, S , by *non-decreasing* order of keys. (This generally assumes that we're *not going to add more items* into the dictionary.)

Storing the dictionary in an array in this fashion allows faster searching than if S is stored using a linked list.

An ordered array implementation is typically referred to as a *lookup table*.

Binary Search - Algorithm

Here's a recursive search algorithm.

`BINARYSEARCH($S, k, low, high$)`

- ▷ Input is an ordered array of elements.
- ▷ Output: Element with key k if it exists, otherwise an error.

```
1 if  $low > high$ 
2   then return NO_SUCH_KEY
3 else
4    $mid \leftarrow \lfloor (low + high) / 2 \rfloor$ 
5   if  $k = key(mid)$ 
6     then return  $elem(mid)$ 
7   elseif  $k < key(mid)$ 
8     then return BINARYSEARCH( $S, k, low, mid - 1$ )
9   else return BINARYSEARCH( $S, k, mid + 1, high$ )
```

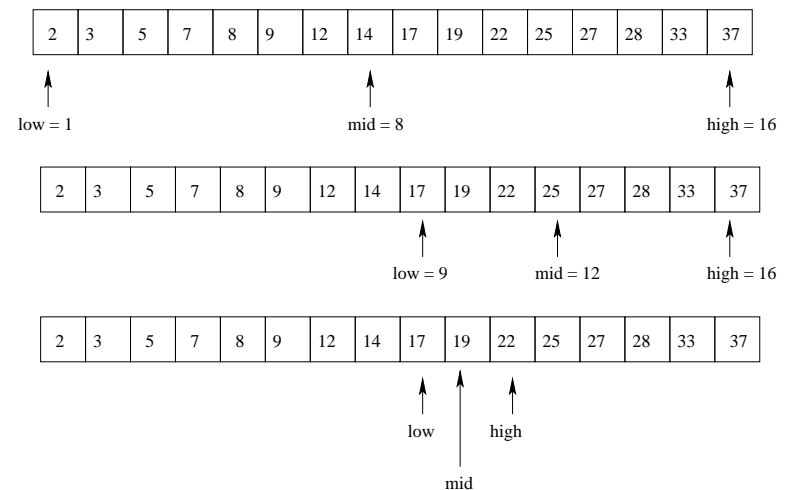
Binary Search

Assumptions:

- ▶ S is an array-based representation of our data having n elements.
- ▶ Accessing an element of S by its index takes $O(1)$ time.
- ▶ The item at index i has a key no smaller than keys of the items of ranks $1, \dots, i - 1$ and no larger than keys of the items of ranks $i + 1, i + 2, \dots, n$.
- ▶ $key(i)$ denotes the key at index i .
- ▶ $elem(i)$ denotes the element at index i .
- ▶ Keys are unique.

Binary Search - Example

`BINARYSEARCH($S, 19, 1, size(S)$)`



Complexity of Binary Search

Let the function $T(n)$ denote the running time of the binary search method.

We can characterize the running time of the recursive binary search algorithm as follows

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ T(n/2) + b & \text{otherwise} \end{cases}$$

where b is a constant that denotes the time for updating *low* and *high* and other overhead.

As you did in COMP108, it can be shown that binary search runs in time $O(\log n)$ on a list with n keys.

Can we find something else?

We see some of the contrast between linked lists and arrays.

List are somewhat costly (in terms of time) to find an item, but it is easy (i.e. quick) to add an item once we know where to insert it.

Arrays are efficient when we search for items, but it is expensive to update the list and maintain items in a sorted order.

Can we find a data structure that might let us mimic the efficiency of binary search on arrays ($O(\log n)$), *and* lets us insert/delete items more efficiently than arrays or lists (possibly as big as $O(n)$) to maintain an ordered collection?

Complexity of Binary Search (cont.)

Comparison of linked list vs. a lookup table (sorted array).

Method	Linked list	Lookup table
findElement	$O(n)$	$O(\log n)$
insertItem (having located its position)	$O(1)$	$O(n)$
removeElement	$O(n)$	$O(n)$
closestKeyBefore	$O(n)$	$O(\log n)$

Another option

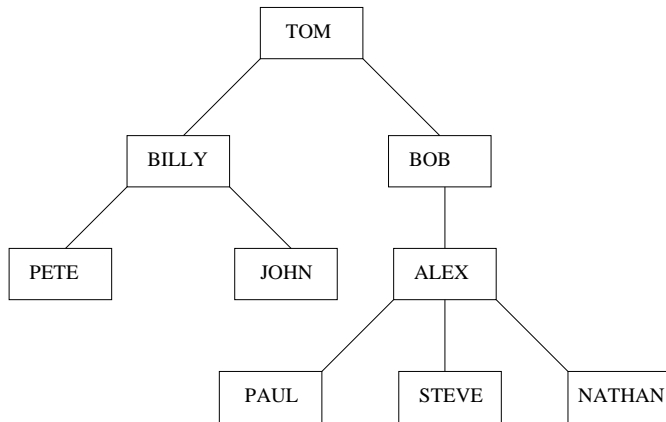
One data structure that seems to have this possibility is a *binary search tree*.

We pause first to review *rooted trees* before we get to binary search trees.

You have previously encountered binary search trees in COMP 108.

Data Structures: Rooted Trees

- ▶ A *rooted tree*, T , is a set of nodes which store elements in a parent-child relationship.
- ▶ T has a special node, r , called the *root* of T .
- ▶ Each node of T (excluding the root node r) has a *parent* node.



Rooted trees: terminology

- ▶ If node u is the *parent* of node v , then v is a *child* of u .
- ▶ Two nodes that are *children* of the same *parent* are called *siblings*.
- ▶ A node is a *leaf* (external) if it has no *children* and *internal* otherwise.
- ▶ A tree is *ordered* if there is a *linear* ordering defined for the *children* of each internal node (i.e. an internal node has a distinguished first child, second child, etc).

Binary Trees

- ▶ A *binary tree* is a rooted ordered tree in which every node has *at most two children*.
- ▶ A binary tree is *proper* if each internal node has *exactly two children*.
- ▶ Each *child* in a binary tree is labeled as either a *left child* or a *right child*.

Depth of a node in a tree

- ▶ The *depth* of a node, v , is number of ancestors of v , excluding v itself. This is easily computed by a recursive function.

DEPTH(T, v)

```
1 if  $T.isRoot(v)$   
2   then return 0  
3   else return 1 + DEPTH( $T, T.parent(v)$ )
```

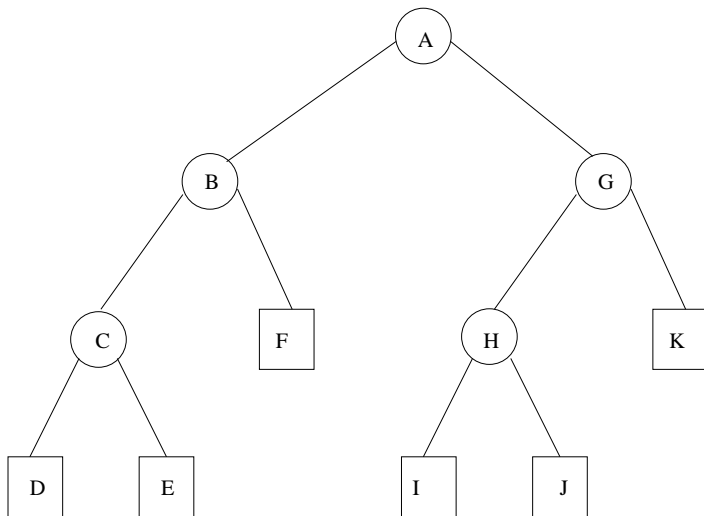
The height of a tree

- ▶ The *height* of a tree is equal to the maximum depth of an external node in it. The following pseudo-code computes the height of the subtree rooted at v .

HEIGHT(T, v)

```
1 if ISEXTERNAL( $v$ )
2   then return 0
3   else
4      $h = 0$ 
5     for each  $w \in T.CHILDREN(v)$ 
6       do
7          $h = \text{MAX}(h, \text{HEIGHT}(T, w))$ 
8     return  $1 + h$ 
```

Postorder traversal of trees



- ▶ A post-order traversal of this tree (printing out the vertex label after recursively visiting the left and right children) would produce: D,E,C,F,B,I,J,H,K,G,A.

Traversal of trees

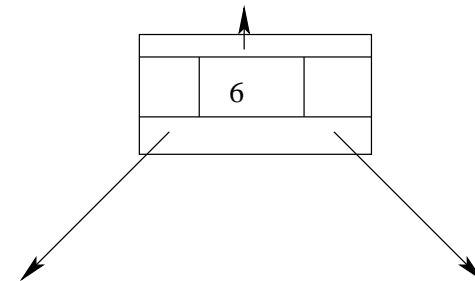
As a brief reminder, there are essentially three ways that trees are generally explored/traversed. These methods are the

1. Pre-order traversal
2. Post-order traversal
3. In-order traversal

I will not give formal pseudo-code for these methods, and refer you to the COMP108 notes for a more full description of these methods.

Data structures for trees

- ▶ Linked structure: each node v of T is represented by an *object* with *references* to the *element* stored at v and positions of its parents and children.



Data structures for rooted t -ary trees

For rooted trees where each node has at most t children, and is of bounded depth, you can store the tree in an array A . This is most useful when you're working with (almost) complete t -ary trees.

Consider, for example, a *binary* tree. The root is stored in $A[0]$.

The (possibly) two children of the root are stored in $A[1]$ and $A[2]$. The two children of $A[1]$ are stored in $A[3]$ and $A[4]$, and the two children of $A[2]$ are in $A[5]$ and $A[6]$, and so forth.

Data structures for rooted t -ary trees (cont.)

In general, the two children of node $A[i]$ are in $A[2 * i + 1]$ and $A[2 * i + 2]$.

The parent of node $A[i]$ (except the root) is the node in position $A[\lfloor (i - 1) / 2 \rfloor]$.

This can be generalized to the case when each node has at most t children, and we will see this implementation for rooted trees later when we discuss with heaps.

Binary Search Tree

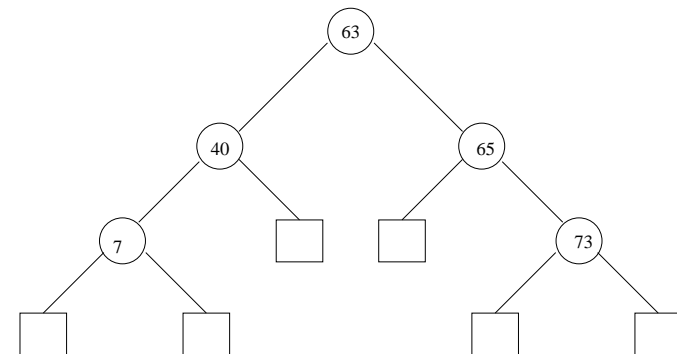
A Binary Search Tree (*BST*) applies the motivation of binary search to a *tree-based* data structure.

In a *BST* each internal node stores an element, e (or, more generally, a key k which defines the ordering, and some element e).

A *BST* has the property that, given a node v storing the element e , all elements in the *left subtree* at v are *less than or equal to* e , and those in the *right subtree* at v are *greater than or equal to* e .

Binary Search Tree (BST)

An *inorder* traversal of a *BST* visits the nodes in non-decreasing order.



Searching in a BST

Here's a recursive searching method:

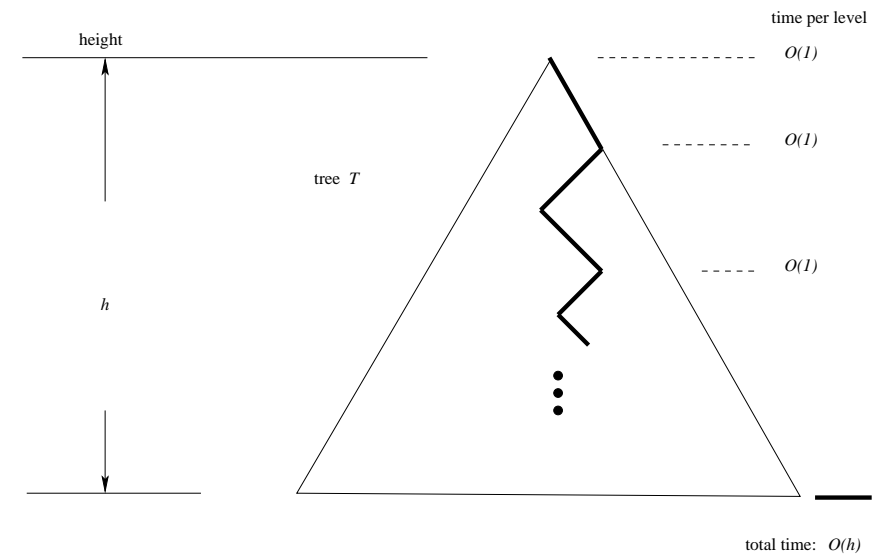
TREESearch(k, v)

- ▷ Input: A key k and a node v of a binary search tree.
- ▷ Output: A node w in the subtree $T(v)$, either w is an internal node with key k or w is an external node where the key k would belong if it existed.

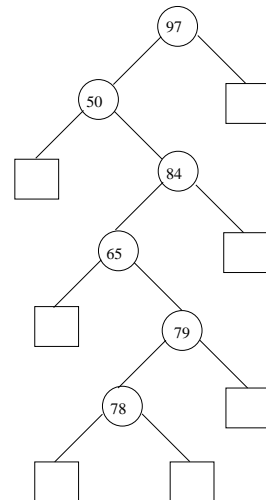
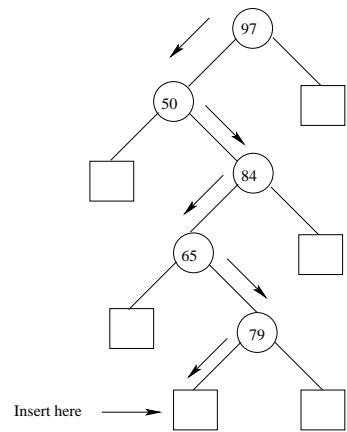
```

1  if ISEXTERNAL( $v$ )
2    then return  $v$ 
3  elseif  $k < key(v)$ 
4    then return TREESearch( $k, T.LEFTCHILD(v)$ )
5  else return TREESearch( $k, T.RIGHTCHILD(v)$ )
    
```

Complexity of searching in a BST

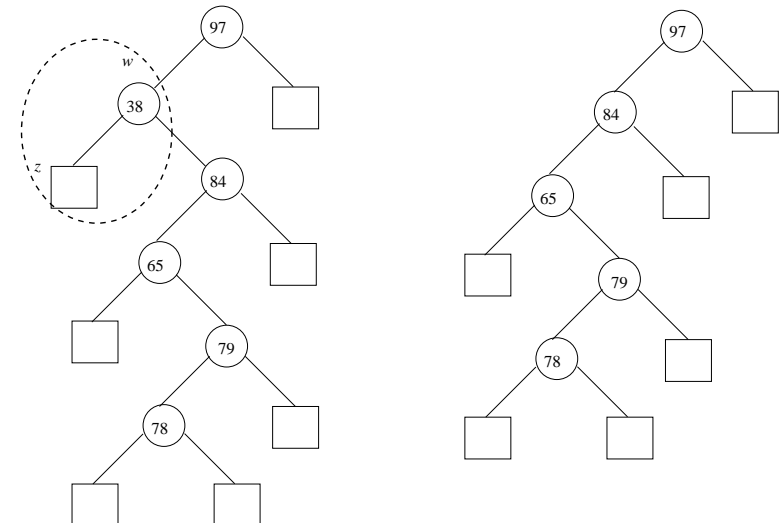


Insertion in a BST



Insertion of 78 is performed in time $O(h)$.

Deletion in a BST



Removal of a node (38) with one *external* (non-empty) child can be performed in time $O(h)$.

Inefficiency of general BSTs

- ▶ All operations in a BST are performed in $O(h)$, where h is the height of the tree.

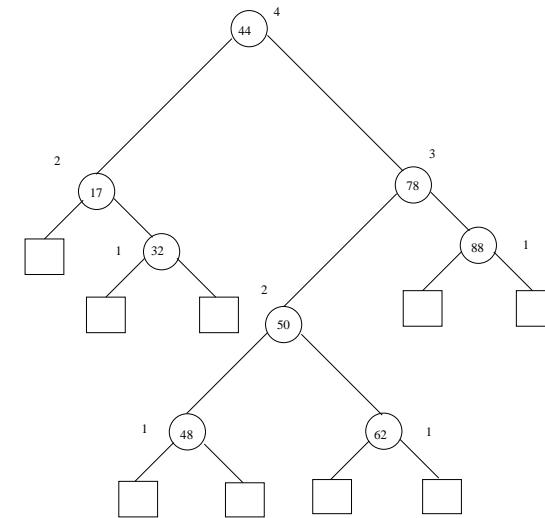
Unfortunately, h may be as large as n , e.g. for a *degenerate tree*.

- ▶ The main advantage of binary searching (i.e. the $O(\log n)$ search time) may be lost if the BST is not *balanced*. In the worst case, the search time may be $O(n)$.

- ▶ Can we do better?

AVL trees

- ▶ *Height-Balance Property*: for every *internal node*, v , of T , the heights of the children of v can differ by at most 1.



AVL trees (cont.)

An *AVL tree* is a tree that has the Height-Balance Property.

- ▶ **Theorem**: The height of an *AVL tree*, T , storing n items is $O(\log n)$.

Consequence 1: A search in an AVL tree can be performed in time $O(\log n)$.

Consequence 2: Insertions and removals in AVL need more careful implementations (using *rotations* to maintain the height-balance property).

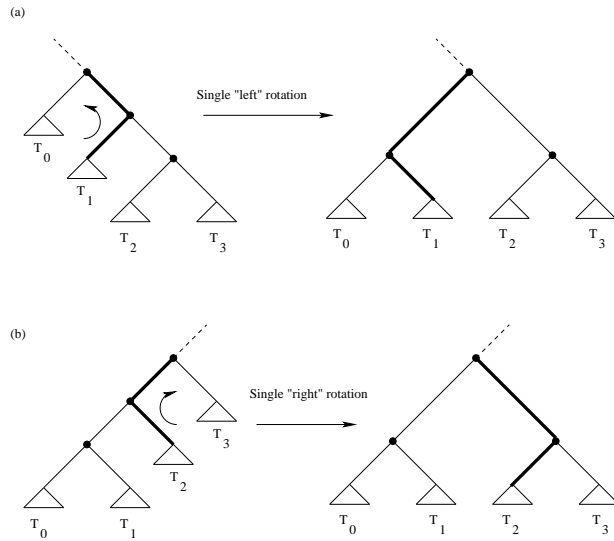
Insertion in AVL trees

An *insertion* in an *AVL tree* begins as an insertion in a *general BST*, i.e., attaching a new *external node* (leaf) to the tree.

- ▶ This action may result in a tree that *violates* the *height-balance property* because the heights of some nodes increase by 1.

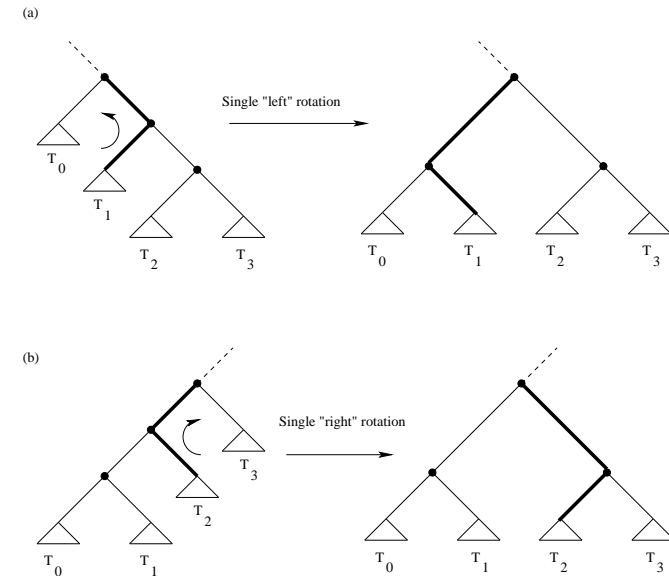
A *bottom-up* mechanism (based on *rotations*) is applied to *rebalance* the unbalanced subtrees.

AVL: Rebalancing after insertion

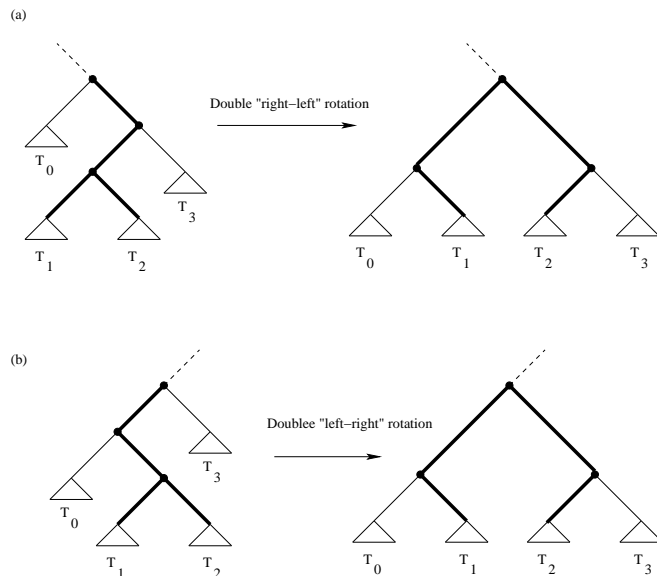


Inserting the number 54 in the previous example will require a rebalancing action (a "right" rotation).

Single rotations (zig) in AVL trees



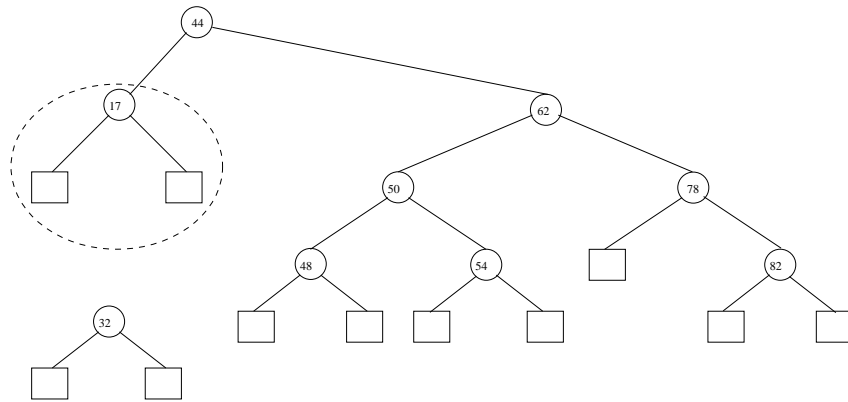
Double rotations (zig-zag) in AVL trees



Deletion in AVL trees

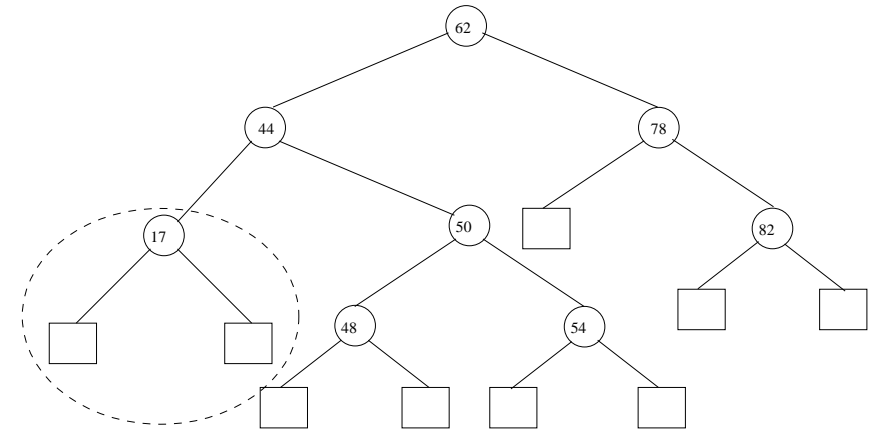
- ▶ A deletion in an *AVL* tree begins as a removal in a general *BST*.
- ▶ Action may *violate* the *height-balance property*.
- ▶ *Bottom-up* mechanism (based on *rotations*) is applied to rebalance the tree.

Rebalancing after a deletion in AVL tree (Stage 1)



Remove 32 from AVL tree

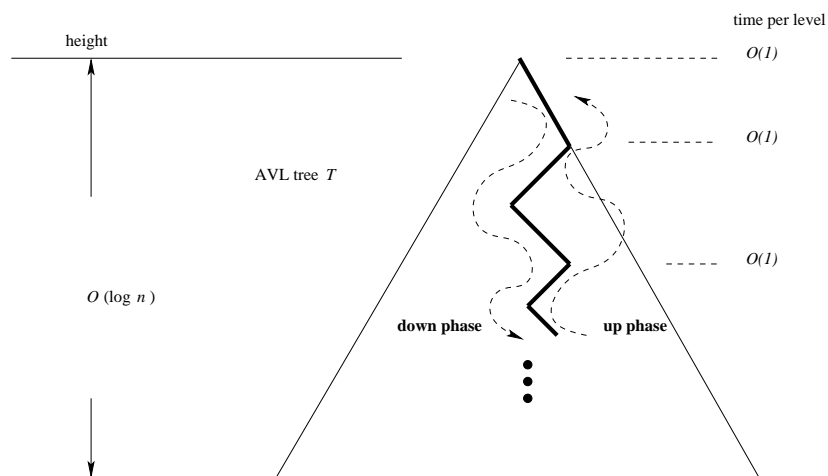
Rebalancing after a deletion in AVL tree (Stage 2)



Rebalancing the tree with a left rotation

AVL performance

- ▶ All operations (*search*, *insertion*, and *removal*) on an AVL tree with n elements can be performed in $O(\log n)$ time.



AVL trees (a demonstration)

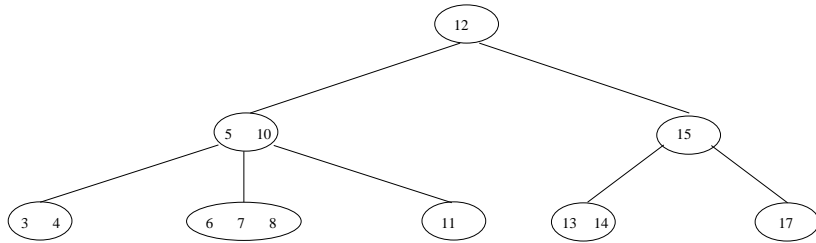
Visit the website

<http://webdiis.unizar.es/asignaturas/EDA/AVLTree/avltree.html>
for a Java applet in which you can view insertion and deletion operations on AVL trees.

You can control the animation speed, and insert and delete items (either random elements or ones that you specify).

(2, 4) trees

Every node in a (2, 4) tree has at *least* 2 and at *most* 4 children.
Each internal node v in a (2, 4) tree contains, 1, 2 or 3 keys defining the range of keys stored in its subtrees.



(2, 4) trees (cont.)

- ▶ All *external* nodes (leaves) in a (2, 4) tree have the same depth.
- ▶ **Theorem:** The height of a (2, 4) tree storing n items is $\Theta(\log n)$.

(2, 4) trees - Search

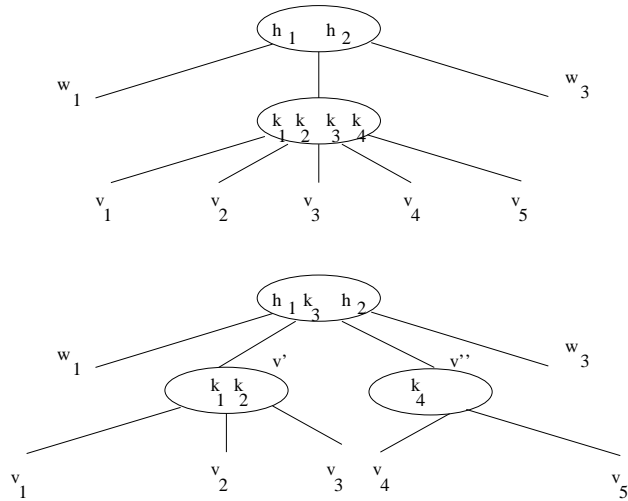
- ▶ Search for a key k in a (2, 4) tree T is done via tracing the *path* in T starting at the root in a *top-down* manner.
- ▶ *Visiting* a node v we compare k with keys (k_i) stored at v :
 - ▶ If $k = k_i$ the search is completed.
 - ▶ if $k_i \leq k < k_{i+1}$, the $i + 1^{\text{th}}$ subtree of v is searched *recursively*.

(2, 4) tree - Insertion

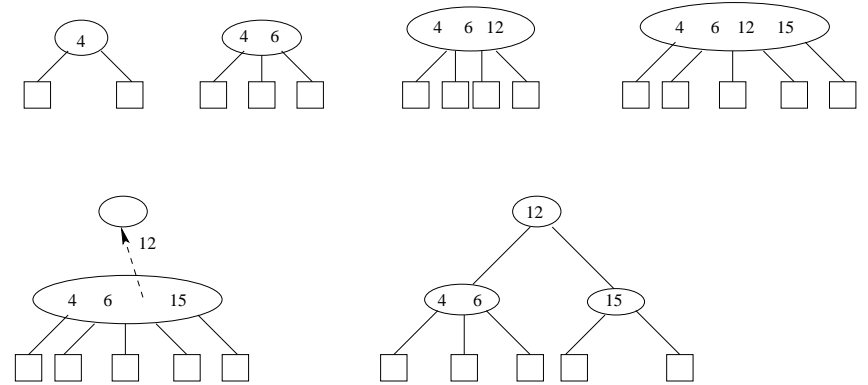
- ▶ An insertion of k into a (2, 4) tree T begins with a *search* for an *internal* node on the *lowest* level that could accommodate k without violating the *range*.
- ▶ This action *may overflow* the *node-size* of a node v acquiring the new key k .
- ▶ *Bottom-up* mechanism (based on *split-operation*) is applied to fix *overflows*.

(2, 4) trees - Split operation

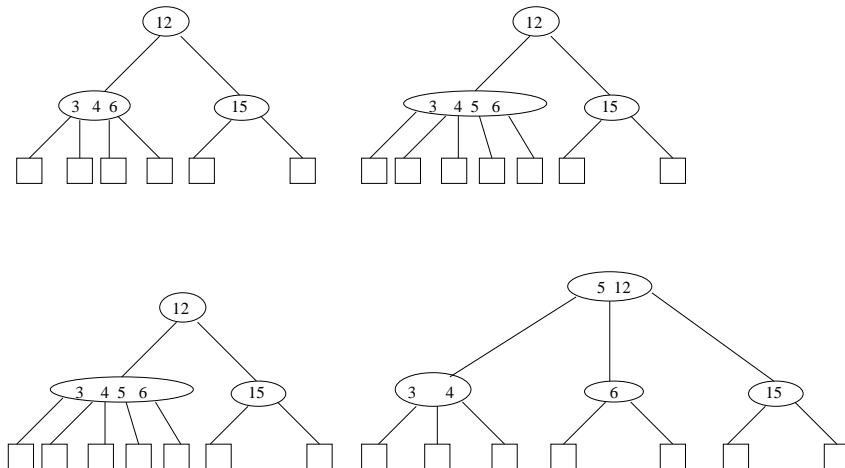
Here's an example of a split operation.



Sequence of insertions (Pt. 1)



Sequence of insertions (Pt. 2)



(2, 4) trees - Deletion

- ▶ Deletion of key k from a (2, 4) tree T begins with a search for node v possessing key $k_i = k$.
- ▶ Key k_i is replaced by largest key in the i^{th} consecutive subtree of v .
- ▶ *Bottom-up* mechanism (based on *transfer and fusion* operation) fixes any *underflows*.

(2, 4) tree performance

- ▶ The *height* of a (2, 4) tree storing n elements in $O(\log n)$.
- ▶ *Split, transfer and fusion* operations take $O(1)$ time.
- ▶ *Search, insertion and removal* of an elements in a tree visits $O(\log n)$ nodes.