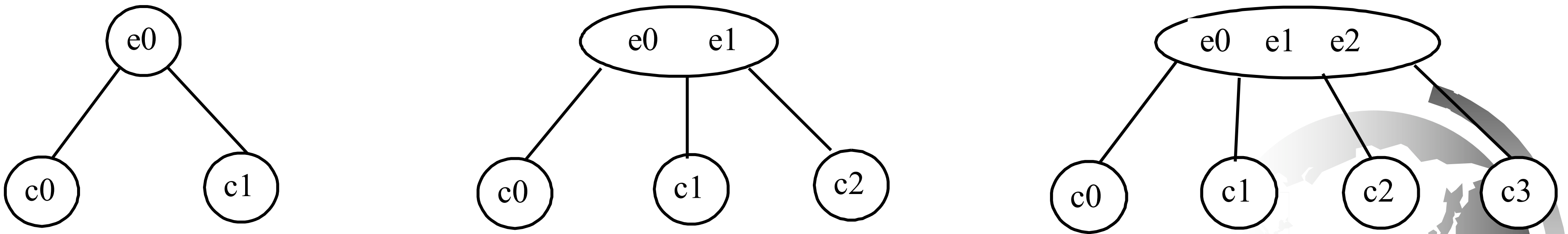
Chapter 42 2-4 Trees and B-Trees

Objectives

- To know what a 2-4 tree is (§42.1).
- To design the Tree24 class that implements the Tree interface (§42.2).
- To search an element in a 2-4 tree (§42.3).
- To insert an element in a 2-4 tree and know how to split a node (§42.4).
- To delete an element from a 2-4 tree and know how to perform transfer and fusion operations (§42.5).
- To traverse elements in a 2-4 tree (§42.6).
- To know how to implement the Tree24 class (§42.7).
- To use B-trees for indexing large amount of data (§42.10).

What is 2-4 Tree?

A *2-4 tree*, also known as a *2-3-4 tree*, is a *complete balanced* search tree with all leaf nodes appearing on the same level. In a 2-4 tree, a node may have one, two, or three elements. An interior *2-node* contains one element and two children. An interior *3-node* contains two elements and three children. An interior *4-node* contains three elements and four children.

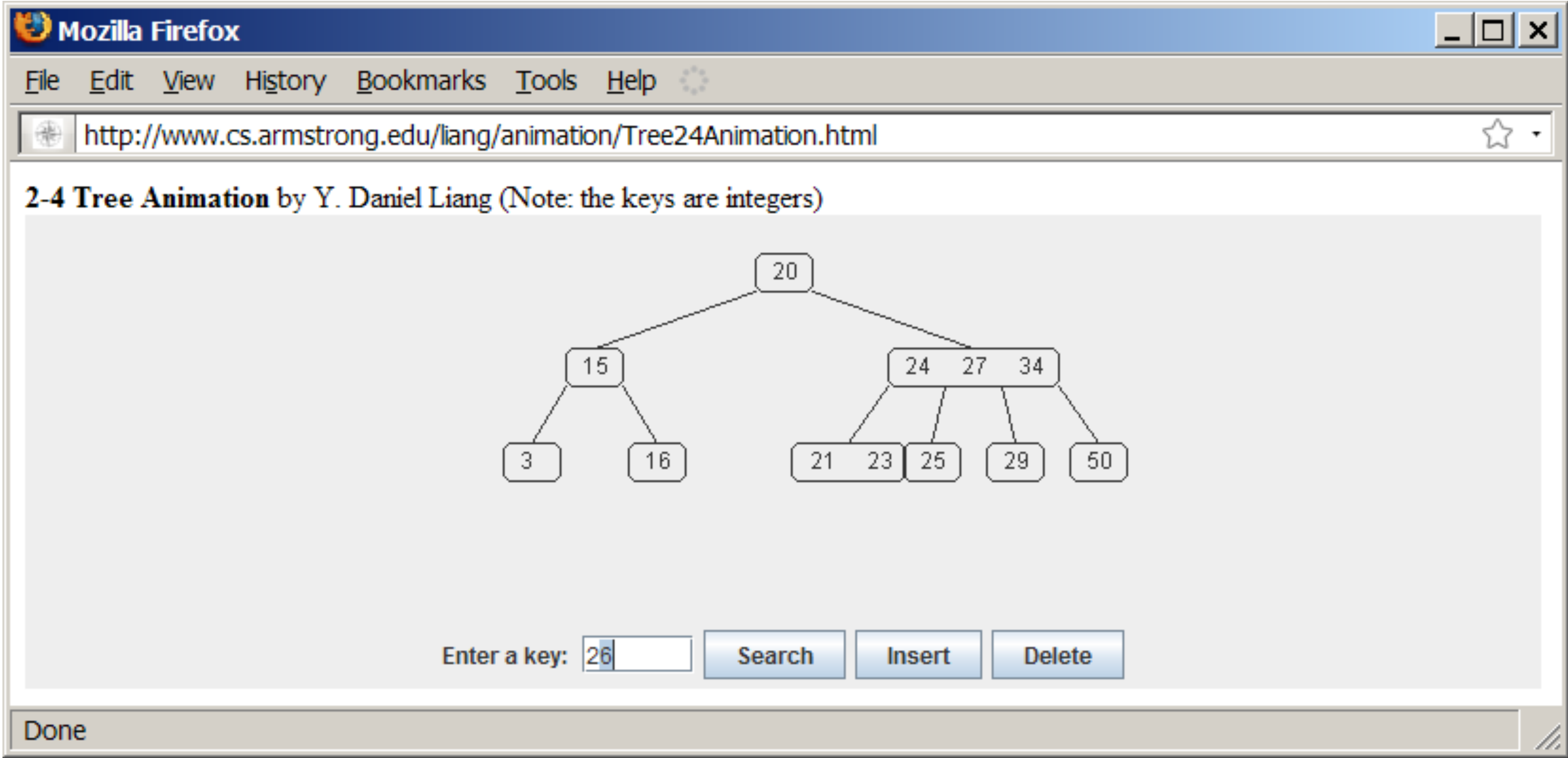


Why 2-4 Tree?

A 2-4 tree tends to be shorter than a corresponding binary search tree, since a 2-4 tree node may contain two or three elements.

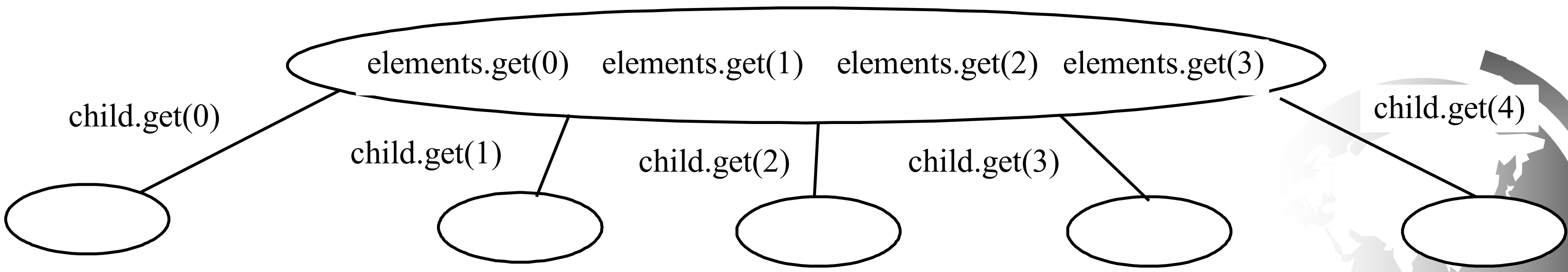
2-4 Tree Animation

www.cs.armstrong.edu/liang/animation/Tree24Animation.html



Searching an Element

Searching an element in a 2-4 tree is similar to searching an element in a binary tree. The difference is that you have to also search an element within a node in addition to searching elements along the path. To search an element in a 2-4 tree, you start from the root and scan down. If an element is not in the node, move to an appropriate subtree. Repeat the process until a match is found or you arrive at an empty subtree.



Inserting an Element

To insert an element to a 2-4 tree, locate a leaf node in which the element will be inserted. If the leaf node is a 2-node or 3-node, simply insert the element into the node. If the node is a 4-node, inserting a new element would cause an *overflow*. To resolve overflow, perform a *split* operation.

Deleting an Element

To delete an element from a 2-4 tree, first search the element in the tree to locate the node that contains the element. If the element is not in the tree, the method returns false. Let be the node that contains the element and be the parent of . Consider three cases:

Case 1: is a leaf 3-node or 4-node. Delete from .

Case 2: is a leaf 2-node. Delete from . Now is empty. This situation is known as *underflow*. Perform appropriate operations to remedy an underflow.

Case 3: is a non-leaf node.

Designing Classes for 2-4 Trees

Tree<E>

Tree24<E>	
-root: Tree24Node<E>;	The root of the tree.
+size: int	The size of the tree.
+Tree24()	Creates a default 2-4 tree.
+Tree24(objects: E[])	Creates a 2-4 tree from an array of objects.
+search(e: E): boolean	Returns true if the element is in the tree.
+insert(e: E): boolean	Returns true if the element is added successfully.
+delete(e: E): boolean	Returns true if the element is removed from the tree successfully.
-matched(e: E, node: TreeNode<E>): boolean	Returns true if element e is in the specified node.
-getChildNode(e: E, node: TreeNode<E>): Tree24Node<E>	Returns the next child node to search for e.
-insert23(e: E, rightChildOfe: Tree24Node<E>, node: Tree24Node<E>): void	Inserts element and along with the reference to its right child to a 2- or 3- node.
-split(e: E, rightChildOfe: Tree24Node<E>, u: Tree24Node<E>, v: Tree24Node<E>): E	Splits a 4-node u into u and v, inserts e to u or v, and returns the median element.
-locate(e: E, node: Tree24Node<E>): int	Locates the insertion point of the element in the node.
-delete(e: E, node: Tree24Node<E>): void	Deletes the specified element from the node.
-validate(e: E, u: Tree24Node<E>, path: ArrayList<Tree24Node<E>>): void	Performs a transfer and confusion operation if node u is empty.
-path(e: E): ArrayList<E>	Return a search path that leads to element e
-leftSiblingTransfer(e: E): ArrayList<E>	

m 0

Tree24Node<E>	
elements: ArrayList<E>	An array list for storing the elements.
child: ArrayList<Tree24Node<E>>	An array list for storing the links to the child nodes.
+Tree24()	Creates an empty tree node.
+Tree24(o: E)	Creates a tree node with an initial element.

1

Link

Tree24

TestTree24

Why B-Tree?

So far we assume that the entire data set is stored in main memory. What if the data set is too large and cannot fit in the main memory, as in the case with most databases where data is stored on disks. Suppose you use an AVL tree to organize a million records in a database table. To find a record, the average number of nodes traversed is . This is fine if all nodes are stored in main memory. However, for nodes stored on a disk, this means 20 disk reads. Disk I/O is expensive and it is thousands of times slower than memory access. To improve performance, we need to reduce the number of disk I/Os. An efficient data structure for performing search, insertion, and deletion for data stored on secondary storage such as hard disks is the B-tree, which is a generalization of the 2-4 tree.

What is a B-Tree?

A B-tree of order is defined as follows:

- Each node except the root contains between and number of keys.
- The root may contain up to number of keys.
- A non-leaf node with number of keys has number of children.
- All leaf nodes have the same depth.