Joanna Klukowska
joannakl@cs.nyu.edu

Lecture 9: Balanced Binary Search Trees, Priority Queues, Heaps, Binary Trees for Compression, General Trees

Reading materials

Dale, Joyce, Weems: 9.1, 9.2, 8.8

Liang: 26 (comprehensive edition)

OpenDSA: 2.11 (Heap), 2.12 (Huffman Coding)

Topics Covered

Joanna Klukowska
joannakl@cs.nyu.edu

1 Balanced Binary Search Trees

We often discussed the fact that the performance of insertion and removal operations in a binary search tree can become very poor (comparable to linked list) if the binary search tree is not nice and bushy. We will look into ways of avoiding such degenerative situation.

For the discussion of balanced trees we will use the following two definitions:

Full binary tree A binary tree in which all of the leaves are on the same level and every non-leaf node has two children. If we count levels starting at zero at the root, then the full binary tree has exactly 2^{L} nodes at level L.

Complete binary tree A binary tree that is either full or full through the next-to-last level, with the leaves on the last level as far to the left as possible.

1.1 Adding a `balance()` Method

The textbook describes an approach of balancing a tree. The burden and responsibility is put on the user of the data structure to periodically call a balance() method that takes the current tree and turns it into a nice and bushy structure with the height of approximately $\log_2 \mathrm{N}$ where N is the number of nodes in the tree.

There are good and bad approaches to implementing such a method. See the section 8.8 in the textbook for details of several different approaches. The best one uses our favorite tool of recursion and inserts existing nodes into a new tree in an order that guarantees a nice balanced tree. **The computational complexity of this balancing of the tree is O(**$\mathrm{N} \log_2 \mathrm{N}$**).**

The following pseudocode describes the steps needed to implement a balance method for the tree.

```
balance()

    perform inorder traversal of the current tree and save the nodes/elements
      nodeList = currentTree.inorderTraversal()
    create a new empty binary search tree called newBST
      newBST = new BST();
    call method insertNodes on the newBST as follows:
      newBST.insertNodes( nodeList, 0, nodeList.size()-1 )



isertNodes( nodeList, first, last )

    if (first == last)
        this.insert( nodeList[first] )
    else if (first+1 == last)
        this.insert( nodeList[first] )
        this.insert( nodeList[last] )
```

Joanna Klukowska
joannakl@cs.nyu.edu

```
else
    mid = (first + last) / 2
    this.insert ( nodeList[mid] )
    this.insertNodes ( first, mid-1 )
    this.insertNodes ( mid+1, last )
```

The `balance()` method performs an inorder traversal saving all the nodes in a list, creates a new empty tree and then adds all the nodes into the tree one by one using the recursive method `insertNodes`. The recursive `insertNodes` method continues by finding the best candidate for the root of the tree (that is the middle element) and then finds the best candidates for roots of left and right subtrees of the root (the middle of the first half and the middle of the second half), etc.

The big **advantage** of this approach is that the user of our binary tree can decide when it is worth to spend the extra computation time to balance the tree. Unfortunately, this is also the **drawback** - the user has to anticipate when it makes sense to balance the tree. If balancing is done too frequently, then that process adds a lot of computation to the tree processing. If it is not done frequently enough, then the tree may become a linked list.

Additionally, the cost of running the balance() method is exactly the same for a tree that is almost balanced and for a tree that looks like a linked list.

1.2 Self-Balancing Binary Search Trees

Another solution (not discussed in the book) is implementation of the **self-balancing binary search trees**. The add and remove operations of the tree are "willing to tolerate" some level of imbalance (i.e. the tree does not have to be complete), but as soon as this is violated, these methods readjust the tree to keep it balanced. Such readjustments do not require the same type of procedures as re-balancing the entire tree. They can be done **locally**, because the imbalance was caused by most recent addition and removal.

Keeping the binary search tree balanced becomes up to the tree itself and not its user.

There are several different approaches to implementing self-balancing binary search trees.

1.3 AVL Tree

The AVL tree (named after its inventors Adelson-Velskii and Landis) is the original self-balancing binary search tree. The computational time of insertions and removals (even with re-balancing) remains $O(\log_2 N)$. **The balancing is done based on height: the heights of the two subtrees of any given node cannot differ by more than 1**. This does not mean that the tree is complete, but it is close enough to complete that the performance of insertions and removals stays even in the worst case at $O(\log_2 N)$, not $O(N)$ as was the case with the ordinary BST. This is achievable by applying one or two (relatively) simple rotations at the nodes that become unbalanced.

Wikipedia has an interesting article about AVL trees with good explanation of rotations and some pseudocode: http://en.wikipedia.org/wiki/AVL_tree

And you can animate creation of the tree and re-balancing at http://www.cs.usfca.edu/~galles/visualization/AVLtree.html.

Joanna Klukowska
joannakl@cs.nyu.edu

Rminder: The **height** of any node is calculated from the bottom of the tree up. The leaves are all at height 0 (or 1 according to some definitions). The height of any internal node is calculated recursively as the maximum of the heights of its children + 1.

After an insert or remove operations we need to check all the nodes in the tree from the one added/removed up to the root to determine if any imbalance was created. Because the rebalancing is performed on each operation, the changes are always small and they are only needed along the path from the root to the newly created or removed node. In fact, there are only 4 different ways in which the node can be out of balance (according to the rule that requires that the heights of its two subtrees differ by no more than 1).

1.3.1 When balancing is necessary

In order to keep the tree balanced, we need to make adjustments when they become necessary. This is decided based on the height difference of the subtrees of a given node. The balance factor is the difference between the hights of the left and right subtrees.

```
int balanceFactor ( Node n )
    if ( n.right == null )
        return -node.height
    if ( n.left == null )
        return node.height
    return node.right.height - node.left.height;
```

In order to compute the balance factor and determine which rotation (if any) is needed at that node, we need to know the heights associated with every node. When we update the height of any given node, we make an assumption (easily satisfiable) that the heights stored in the children of the node are correct.

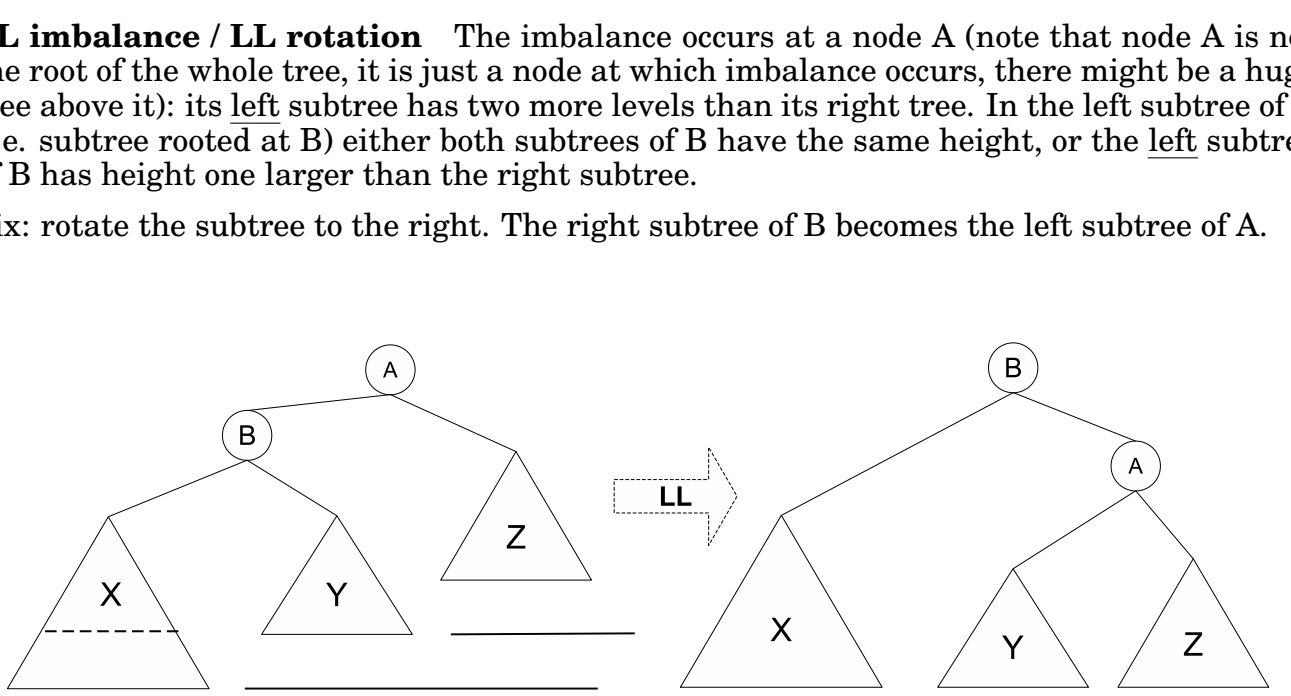
```
void updateHeight ( Node n )
    if node is a leaf
        node.height = 0 //this is sometime set to 1
    else if node.left == null
        node.height = node.right.height + 1
    else if node.right == null
        node.height = node.left.height + 1
    else

        node.height = max( node.right.height, node.left.height) + 1
```

When we add or remove a node in an AVL tree, we need to go back up to the root (along the path from the modified node to the root) and check one by one if any of the nodes requires rebalancing. If the call to `balanceFactor` returns -2 or 2, then we need to perform one of the four possible rotations described below. Notice that the call to balanceFactor should never return a value smaller than -2 or larger than 2 if the tree is maintained properly.

It is important to note that in all of the rotations below we change the structure of the tree (relink the nodes) and the data is never copied from one node to another node.

Joanna Klukowska
joannakl@cs.nyu.edu



LL imbalance / LL rotation The imbalance occurs at a node A (note that node A is not the root of the whole tree, it is just a node at which imbalance occurs, there might be a huge tree above it): its left subtree has two more levels than its right tree. In the left subtree of A (i.e. subtree rooted at B) either both subtrees of B have the same height, or the left subtree of B has height one larger than the right subtree.

Fix: rotate the subtree to the right. The right subtree of B becomes the left subtree of A.

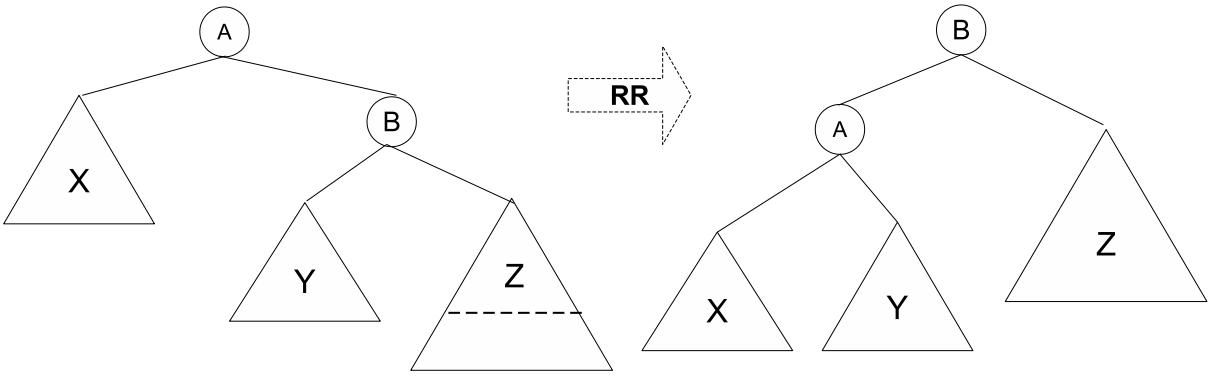
The following pseudocode describes steps required to perform an LL rotation

```
1 //returns a reference to the new root of the subtree after
2 //the LL rotation has been performed
3
4 Node  balanceLL ( Node A )
5
6     Node B = A.left
7
8     A.left = B.right
9     B.right = A
10
11    updateHeight ( A )
12    updateHeight ( B )
13
14    return B
```

RR imbalance / RR rotation The imbalace occurs at a node A: its right subtree has two more levels than its right subtree. In the right subtree of A (i.e. subtree rooted at B) either both subtrees of B have the same height, or the right subtree of B has height one larger than the left subtree.

Fix: rotate the subtree to the left. The left subtree of B becomes the right subtree of A.

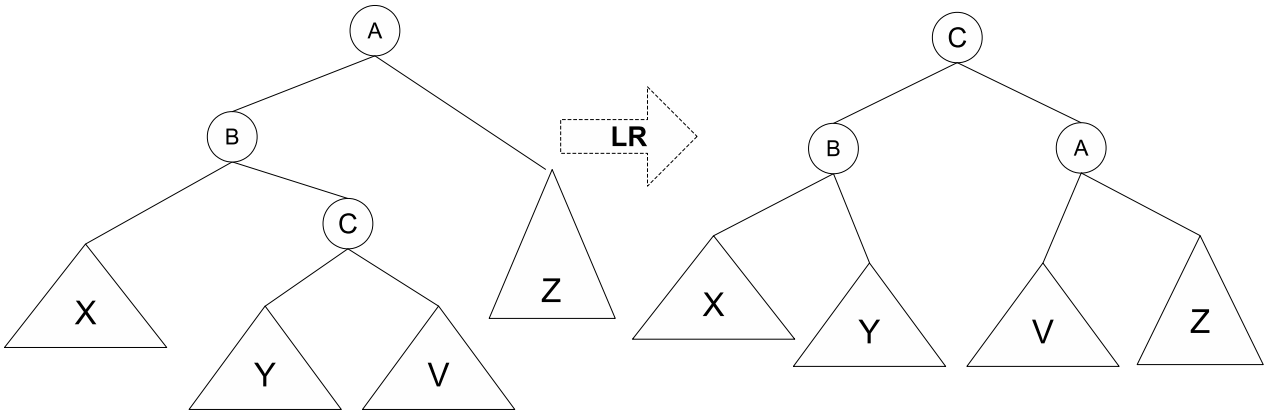
Joanna Klukowska
joannakl@cs.nyu.edu



The code that performs this rotation is very similar to the one for the LL rotation.

LR imbalance / LR rotation The imbalance occurs at a node A: its left subtree has two more levels than its right subtree. In the left subtree of A (i.e. subtree rooted at B) the right subtrees of B (whose root is C) has height one larger than the left subtree of B.

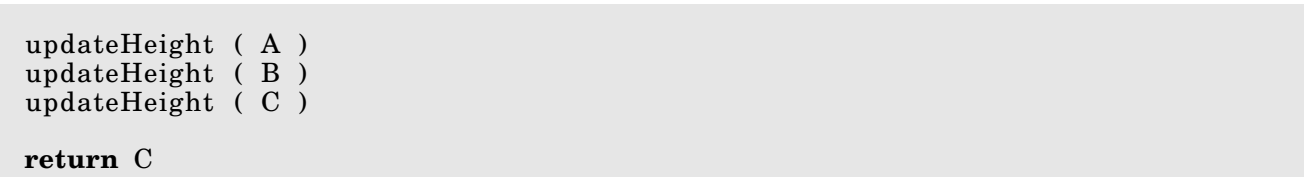
Fix: rotate to the left at node B and then rotate to the right at node A.



The following pseudocode describes steps required to perform an LR rotation

```
1 //returns a reference to the new root of the subtree after
2 //the LR rotation has been performed
3
4 Node  balanceLR ( Node A )
5
6     Node B = A.left
7     Node C = B.right;
8
9     A.left = C.right
10    B.right = C.left
11    C.left = B
12    C.right = A
```

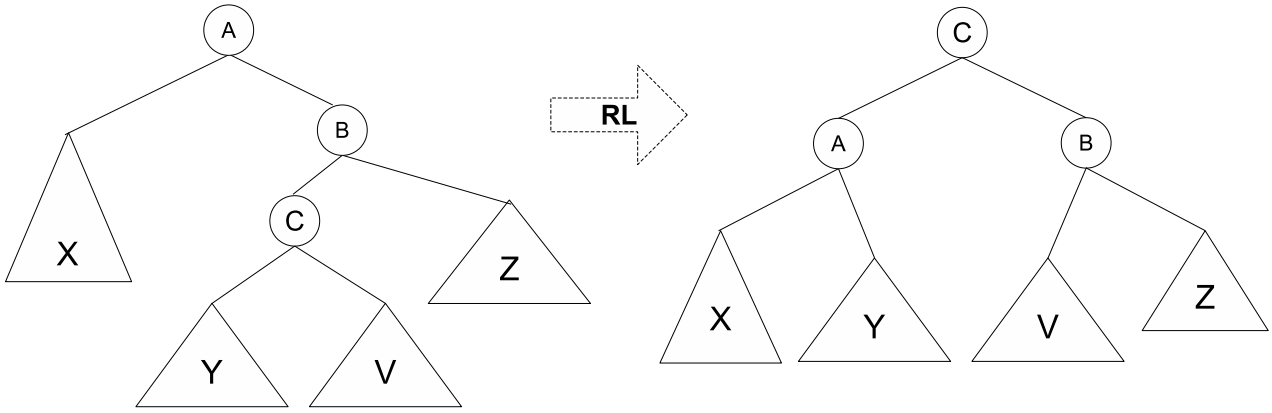
Joanna Klukowska
joannakl@cs.nyu.edu



```
13
14      updateHeight ( A )
15      updateHeight ( B )
16      updateHeight ( C )
17
18      return C
```

RL imbalance / RL rotation The imbalance occurs at a node A: its right subtree has two more levels than its left subtree. In the right subtree of A (i.e. subtree rooted at B) the left subtrees of B (whose root is C) has height one larger than the left subtree of B.

Fix: rotate to the right at node B and then rotate to the left at node A.



The code that performs this rotation is very similar to the one for the LR rotation.

1.4 Red-Black Trees

The red-black tree (invented by R Bayer) is another self-balancing search tree. The computation time of insertions and removals is as well O($\log_2$ N), but the balancing is done differently than with the AVL trees. All nodes are assigned one of the two colors (red or black, hence the name) and the coloring has to follow certain properties. If these properties are violated the insert or remove method has to fix the tree.

For the list of all the properties and their consequences see the Wikipedia article at http://en.wikipedia.org/wiki/Red-black_tree.

And the animation of creation and modification of a tree is at http://www.cs.usfca.edu/~galles/visualization/RedBlack.html.

Java's library `TreeSet<E>` and `TreeMap<K,V>` classes are implemented using the Red-Black trees:

- TreeSet (http://docs.oracle.com/javase/7/docs/api/java/util/TreeSet.html),
- TreeMap (http://docs.oracle.com/javase/7/docs/api/java/util/TreeMap.html).

Joanna Klukowska
joannakl@cs.nyu.edu

2 Priority Queues and Heaps

At the beginning of the semester we looked at queue data structure. In a typical queue, the items that arrive at the queue first are removed from the queue first (first in, first out, or FIFO, structure). Our typical queue is a special case of a priority queue in which the priority is decided based on time of arrival. In more general priority queue, the priority of an element can be decided based on anything. The next element to be removed (or processed, or served, etc) is the element whose priority is highest.

Examples:

- Printer's print queue - usually FIFO, but could be reconfigured to print the shortest (fewest pages) job first. The fewer pages the print job has, the higher priority is has. (What is the potential problem with this arrangement?)
- Boarding the plane is done according to a (partial) priority queue - the passengers who paid the most for their tickets board first (they have higher priority).
- Job scheduling on a processor - all computer processes have some priority assigned to them and the ones with highest priority get the time on a processor first. The assignment of priorities for computer processes is a complicated and challenging problem - you'll see more about it in your operating systems course.
- Registering for classes on Albert (partial priority queue) - the seniors get to register sooner than the other students.

2.1 Priority Queue Interface

A priority queue needs to provide the following operations (they are practically the same as the operations provided by the regular queue):

`boolean isFull( )`	returns true if the priority queue is full (this may never happen), false otherwise
`boolean isEmpty( )`	returns true if the priority queue is empty, false otherwise
`void enqueue( T element)`	adds another element to the queue
`T dequeue( )`	removes and returns the element with highest priority from the queue (ties are resolved arbitrarily and are implementation dependent)

2.2 Implementation

Throughout the semester, we discussed several different structures that can be used for implementation of a priority queue. Below we review different options.

Joanna Klukowska
joannakl@cs.nyu.edu

An Array-based Sorted List

`enqueue()` operation can be implemented using binary search to find the correct locations - this can be done in O(log_2N) time, but then other elements in the array need to be shifted to accomodate the new element - this makes it O(N)

`dequeue()` operation is simple as we need to remove the item with highest priority which is at the front of the list which requires only constant time O(1) (well, assuming that we do not insist on keeping the front of the queue at the index 0 of the array - the front of the queue needs to be able to move through the array, otherwise the dequeue operation becomes O(N))

In addition we need to introduce finite size or resize the array when it becomes full.

A Reference-based Sorted List

`enqueue()` operation has to traverse through the list in order to find the correct place to insert the element, that is O(N) time

`dequeue()` operation requires removing the head of the list which is constant time operation, O(1)

The queue can grow in size indefinitely (well, within our memory limitations).

A Binary Search Tree

`enqueue()` operation is always proportional to the height of the tree

`dequeue()` operation is always proportional to the height of the tree

This is good and bad. If the tree is nice and bushy, that this means that we can perform both enqueue and dequeue in O(log_2N) time. But the binary search trees can become unbalance and the true worst case is O(N) for both operations. We could use a self-balancing binary search tree, but there is something better.

The queue can grow in size indefinitely (well, within our memory limitations).

2.3 Heaps as Implementation of Priority Queues

Our goal is to find an implementation that guarantees better than O(N) performance on both `enqueue` and `dequeue` operations, even in the worst case. This is where the **heap** data structure comes in.

2.3.1 Max-Heap

Max-heap is a complete binary tree with a property that each node contains a value (priority in our case) that is greater than or equal to the value of its children. When we talk about heaps, we need to make sure that a heap maintains two properties:

shape property the heap has to be a complete binary tree (a *complete binary* tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible),

Joanna Klukowska
joannakl@cs.nyu.edu

order property each node has a value greater than or equal to the values of both of its children.

`enqueue()` operation is always proportional to the height of the tree, but because of the shape property, the height is always guaranteed to be O(log_2N)

`dequeue()` operation can be done fast by accessing the node at the root of the tree, O(1), but since we will need to replace that node with something else, it also turns out to be O(log_2N)

Both of these running times are guaranteed even in the worst case.

Enqueing an element During enqueue operation, we add the new node as the last element in the bottom row of the tree. This maintains the shape property (the tree is still complete), but it may violate the order property. We need to *heapify* the tree, by moving the new node up (swapping with the parent node) until its parent is greater than, or equal to it.

Dequeueing an element During the dequeue operation, we remove and return the root of the tree. But that leaves a hole in the tree. To fill it, we take the rightmost node from the bottom level of the tree and place it at the root. This preserves the shape property, but we need to do some work to restore the order property. As with the enqueue, we heapify the tree by moving the new root down (swapping with the larger of its children) until we find the position at which both of its children are smaller.

As you remember, in a binary tree (complete or not) the parents have references to the children, but not the other way around. This means that the heapifying the tree might be a problem. Well, the heaps are not implemented as trees, they are implemented as arrays. It is actually much more efficient to implement them as arrays because the heaps are complete trees and the relationship between index of a parent and its children is easy to establish. On the other hand, it is much easier to think about the operations performed on the heaps when we think about them as trees. There is a one-to-one correspondence between an array and a tree representation of any heap.

2.3.2 Min-Heaps

It is intuitive to think of largest elements at the front of a priority queue because these are the elements with the highest priority. But often it is the smallest elements that have higher priority. We can implement a min-heap in exactly same way as the max-heap. The only difference is that the order property changes to ”each node has a value smaller than or equal to the values of both of its children”.

2.3.3 Using Arrays to Implement Heaps

We can populate an array with values stored in nodes of a complete binary tree by reading them off from the root one level at a time (this is breadth first search traversal of a tree). This gives us the following properties:

- the root of a tree is at index `0`

Joanna Klukowska
joannakl@cs.nyu.edu

- if an element is not a root, its parent is at position `(index-1)/2`
- if an element has a left child, the child is at position `index*2+1`
- if an element has a right child, the child is at position `index*2+2`

This allows us to move from parent to child and child to parent very quickly (no need to worry about storing extra references) and, because of the heap property, we do not need to traverse the entire array to find the right place for a node.

See simulations on the OpenDSA website: http://algoviz.org/OpenDSA/Books/OpenDSA/html/Heaps.html.

2.4 Java Libraries Implementation of a Priority Queue

Java implements a heap based priority queue in the class PriorityQueue<E>, see http://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html for details. You should also take a look at the actual implementation at http://cs.nyu.edu/~joannakl/cs102_f14/assignments/PriorityQueue.java (this can be downloaded with the source code for JDK).

The method names are slightly different than what we have been using. The enqueue operation is provided by `add(E element)` method and the dequeue operation is provided by `poll()` method.

3 File Compression

When a text file is stored on a computer, each letter is represented by a binary code corresponding to that letter. Depending on the file encoding (ASCII or UNICODE), we might be using anywhere from 1 to 4 bytes per character. The War and Peace text file that we used for project 3, has 3226614 characters, that is over 3MB (and that file uses only 1 byte to represent each character).

Do we really need to use all that space to save this file?

We compress files all the time. How can this be done and what does it have to do with data structures?

In 1952 David Huffman showed that we do not need all the bits to represent the text (well, I am not sure if he was thinking about bits, but his idea of minimal prefix encoding allows us to do this). Here is the basic idea.

Use different length of bit sequences to represent different letters. If some letters occur much more frequently than others, we want to use a very short bit sequence to represent them. If some letters occur very rarely, then we can afford to use a long sequence of bits to represent them. The necessary condition for this to work is the existence of a prefix-code: coding of letters such that no valid code is ever a prefix of another valid code, otherwise we could not recognize boundaries between letters.

Knowing statistical properties of letters used to produce English text, we can figure out best bit sequences for each letter. But we can do even better if the sequences are based on specific text that we want to make ”smaller” (i.e., compress).

The method to come up with such encoding (they are not unique) is as follows:

Joanna Klukowska
joannakl@cs.nyu.edu

1. compute frequencies of every single letter occurring in a given text
2. create individual nodes that contain a letter and its frequency (this is a forest of single nodes of binary trees)
3. create a priority queue of the trees (the order is based on the frequencies: the lower frequency, the higher the priority of the tree)
4. as long as the priority queue has more than one tree in it
   (a) pick two trees with the smallest frequencies and merge them by creating a new node and settings its children to the two trees, assign the sum of the frequencies of the two trees to the new tree
   (b) put the new tree back on the priority queue

Once the Huffman tree is complete the actual letters (and their counts) are located at the leaves. To determine the binary sequences that should be assigned to them, label all the left children references (edges from parent to its left child) with 0 and all the right children references (edges from parent to its right child) with 1. For each leaf construct the sequence by following the edges from the root to the leaf and reading the labels on the edges.

Visualization of building a Huffman tree and establishing the binary sequences can be found on OpenDSA website: `http://algoviz.org/OpenDSA/Books/OpenDSA/html/Huffman.html`.

You can find complete Java code for a program that compresses and decompresses files using Huffman coding at:

http://nayuki.eigenstate.org/page/huffman-coding-java

4 General Trees

A **general tree** is useful for representing hierarchies in which the number of children varies.

- In a file system, a node represents each file, and if the file is a directory, then it is an internal node whose children are the files contained in the directory. Some file systems do not restrict the number of files per folder, implying that the number of children per node is varying and unbounded.
- In computational linguistics, as sentences are parsed, the parser creates a representation of the sentence as a tree whose nodes represent grammatical elements such as predicates, subjects, prepositional phrases, and so on. Some elements such as subject elements are always internal nodes because they are made up of simpler elements such as nouns and articles. Others are always leaf nodes, such as nouns. The number of children of the internal nodes is unbounded and varying.
- In genealogical software, the tree of descendants of a given person is a general tree because the number of children of a given person is not fixed.

Joanna Klukowska
joannakl@cs.nyu.edu

Tree implementations Because one does not know the maximum degree that a tree may have, and because it is inefficient to create a structure with a very large fixed number of child entries, the best implementation of a general tree uses a linked list of siblings and a single child, as follows.

```
class TreeNode <E>{

    E data
    TreeNode firstChild
    TreeNode nextSibling

}
```