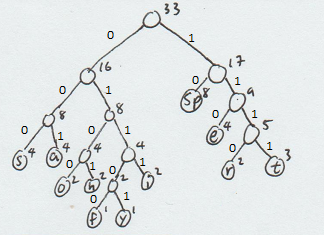
CS146-Lab6 Name: Michael Huang Due Date: March 21th, 2016

Ex-1: Show the Huffman tree and the resulting codes for each character:

it is safe to stay here as a hero

Final tree:



Code with traversal:

it is safe to stay here as a hero

0111 (i) 1111 (t) 10 (sp) 0111 (i) 000 (s) 10 (sp) 000 (s) 001 (a) 01100 (f) 110 (e) 10 (sp)

1111 (t) 0100 (o) 10 (sp) 000 (s) 1111 (t) 001 (a) 01101 (y) 10 (sp) 0101 (h) 110 (e)

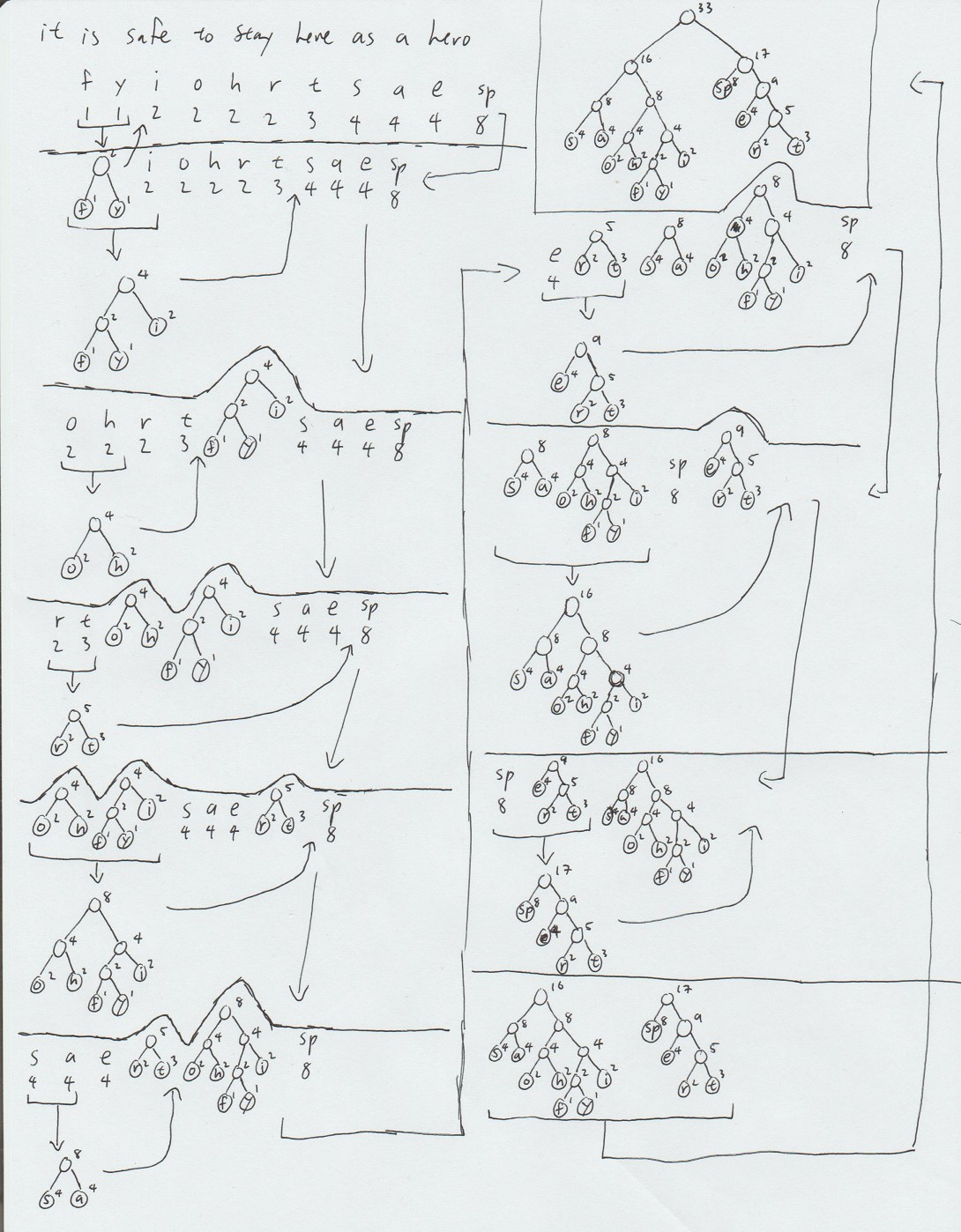
1110 (r) 110 (e) 10 (sp) 001 (a) 000 (s) 10 (sp) 001 (a) 10 (sp) 0101 (h) 110 (e) 1110 (r)

0100 (o)

Code:

0111111110011100010000001011001101011110100100001111001011011001011101110110100010001000110010111011100100

This is the entire process step by step, characters with the same priority is sorted by their order of appearance in the original string. Zoom in if characters are too small.



Ex-2: Write an implementation of the TreeSet class, with associated iterators using a binary search tree. Add to each node a link to the parent node.

Also, your remove method missed the case if the node is a leaf.

Code:

|  |
| --- |
| // Received help from computer science tutors.  import java.util.\*;  class UnderflowException extends Exception {};  public class MyTreeSet<AnyType extends Comparable<? super AnyType>> {  private static class BinaryNode<AnyType> {  BinaryNode(AnyType theElement) {  this(theElement, null, null, null);  }  BinaryNode(AnyType theElement, BinaryNode<AnyType> lt, BinaryNode<AnyType> rt, BinaryNode<AnyType> pt) {  element = theElement;  left = lt;  right = rt;  parent = pt;  }  AnyType element;  BinaryNode<AnyType> left;  BinaryNode<AnyType> right;  BinaryNode<AnyType> parent;  }  public java.util.Iterator<AnyType> iterator() {  return new MyTreeSetIterator();  }  private class MyTreeSetIterator implements java.util.Iterator<AnyType> {  // private BinaryNode<AnyType> current = findMin(root);  private BinaryNode<AnyType> previous;  private int expectedModCount = modCount;  private boolean okToRemove = false;  private boolean atEnd = false;  private Stack<BinaryNode<AnyType>> nodes;  public MyTreeSetIterator() {  nodes = new Stack<BinaryNode<AnyType>>();  BinaryNode<AnyType> start = root;  while (start != null) {  nodes.push(start);  start = start.left;  }  }  public boolean hasNext() {  return !atEnd && !nodes.isEmpty();  }  public AnyType next() {  if (modCount != expectedModCount)  throw new java.util.ConcurrentModificationException();  if (!hasNext())  throw new java.util.NoSuchElementException();  // \*\*\*\*\*\*\*\*\*\*\*\*\*\*NEEDS TO BE COMPLETED remember it is sorted order  BinaryNode<AnyType> current = nodes.pop();  AnyType data = current.element;  if (current.right == null) {  return data;  } else {  current = current.right;  while (current != null) {  nodes.push(current);  current = current.left;  }  }  return data;  }  public void remove() {  if (modCount != expectedModCount)  throw new java.util.ConcurrentModificationException();  if (!okToRemove)  throw new IllegalStateException();  MyTreeSet.this.remove(previous.element);  okToRemove = false;  }  }  public MyTreeSet() {  root = null;  }  public void makeEmpty() {  modCount++;  root = null;  }  public boolean isEmpty() {  return root == null;  }  public boolean contains(AnyType x) {  return contains(x, root);  }  public AnyType findMin() throws UnderflowException {  if (isEmpty())  throw new UnderflowException();  else  return findMin(root).element;  }  public AnyType findMax() throws UnderflowException {  if (isEmpty())  throw new UnderflowException();  else  return findMax(root).element;  }  public void insert(AnyType x) {  root = insert(x, root, null);  }  public void remove(AnyType x) {  root = remove(x, root);  }  public void printTree() {  if (isEmpty())  System.out.println("Empty tree");  else  printTree(root);  }  private void printTree(BinaryNode<AnyType> t) {  // MUST COMPLETE THIS PART  if (t != null) {  printTree(t.left);  System.out.println(t.element);  printTree(t.right);  }  }  private boolean contains(AnyType x, BinaryNode<AnyType> t) {  // MUST COMPLETE  if (t == null) {  return false;  } else {  int compare = x.compareTo(t.element);  if (compare < 0) {  contains(x, t.left);  } else if (compare > 0) {  contains(x, t.right);  } else {  return true;  }  }  return false;  }  private BinaryNode<AnyType> findMin(BinaryNode<AnyType> t) {  // MUST COMPLETE  while (t.left != null) {  t = t.left;  }  return t;  }  private BinaryNode<AnyType> findMax(BinaryNode<AnyType> t) {  // MUST COMPLETE  while (t.right != null) {  t = t.right;  }  return t;  }  private BinaryNode<AnyType> insert(AnyType x, BinaryNode<AnyType> t, BinaryNode<AnyType> pt) {  // Must complete but remember to have the implement the parent node as well  if (t == null) {  return new BinaryNode<AnyType>(x, null, null, pt);  }  int compare = x.compareTo(t.element);  if (compare < 0) {  t.left = insert(x, t.left, t);  } else if (compare > 0) {  t.right = insert(x, t.right, t);  }  return t;  }  private BinaryNode<AnyType> remove(AnyType x, BinaryNode<AnyType> t) {  if (t == null)  return t; // not found  int compareResult = x.compareTo(t.element);  if (compareResult < 0)  t.left = remove(x, t.left);  else if (compareResult > 0)  t.right = remove(x, t.right);  else if (t.left != null && t.right != null) { // two children  t.element = findMin(t.right).element;  t.right = remove(t.element, t.right);  } else if (t.left == null && t.right == null) { // leaf  t = null;  } else { // single child  modCount++;  BinaryNode<AnyType> oneChild;  oneChild = (t.left != null) ? t.left : t.right;  oneChild.parent = t.parent; // update parent link  t = oneChild;  }  return t;  }  private BinaryNode<AnyType> root;  int modCount = 0;  public static void main(String[] args) throws UnderflowException {  MyTreeSet<Integer> a = new MyTreeSet<Integer>();  a.insert(4);  a.insert(3);  a.insert(9);  a.insert(7);  a.printTree();  System.out.println("Removing 7:");  a.remove(7);  a.printTree();  System.out.println("Iterating through TreeSet:");  Iterator<Integer> iter = a.iterator();  while(iter.hasNext()) {  System.out.println(iter.next());  }  System.out.println("Contains 4: " + a.contains(4));  System.out.println("Find the min: " + a.findMin());  System.out.println("Find the max: " + a.findMax());  }  } |

Console output:

