CS146-Lab7 Name: Michael Huang Due date: April 6th, 2016

Ex-1: Implement the BinaryHeap as shown in page 228.

Code:

|  |
| --- |
| public class BinaryHeap<AnyType extends Comparable<? super AnyType>> {  private static final int DEFAULT\_CAPACITY = 10;  private int currentSize;  private AnyType[] array;  public BinaryHeap() {  this(DEFAULT\_CAPACITY);  }  @SuppressWarnings("unchecked")  public BinaryHeap(int capacity) {  currentSize = 0;  array = (AnyType[]) new Comparable[capacity + 1];  }  @SuppressWarnings("unchecked")  public BinaryHeap(AnyType[] items) {  currentSize = items.length;  array = (AnyType[]) new Comparable[(currentSize + 2) \* 11 / 10];  int i = 1;  for (AnyType item : items) {  array[i++] = item;  }  buildHeap();  }  public void insert(AnyType object) {  if (currentSize == array.length - 1) {  enlargeArray(array.length \* 2 + 1);  }  // Percolate up.  int hole = ++currentSize;  for (array[0] = object; object.compareTo(array[hole / 2]) < 0; hole /= 2) {  array[hole] = array[hole / 2];  }  array[hole] = object;  }  public AnyType findMin() throws Exception {  if (isEmpty()) {  throw new Exception();  }  return array[1];  }  public AnyType deleteMin() throws Exception {  if (isEmpty()) {  throw new Exception();  }  AnyType min = findMin();  array[1] = array[currentSize];  percolateDown(1);  array[currentSize] = null;  currentSize--;  return min;  }  public boolean isEmpty() {  if (currentSize == 0) {  return true;  } else {  return false;  }  }  public void makeEmpty() {  currentSize = 0;  }  private void percolateDown(int hole) {  int child;  AnyType temp = array[hole];  for (; hole \* 2 <= currentSize; hole = child) {  child = hole \* 2;  if (child != currentSize && array[child + 1].compareTo(array[child]) < 0) {  child++;  }  if (array[child].compareTo(temp) < 0) {  array[hole] = array[child];  } else {  break;  }  }  array[hole] = temp;  }  private void buildHeap() {  for (int i = currentSize / 2; i > 0; i--) {  percolateDown(i);  }  }  @SuppressWarnings("unchecked")  private void enlargeArray(int newSize) {  AnyType[] oldArray = array;  array = (AnyType[]) new Comparable[newSize];  for (int i = 0; i < oldArray.length; i++) {  array[i] = oldArray[i];  }  }  public void print() {  int allowed = 1;  int row = 0;  for (int i = 1; i < array.length; i++) {  if (array[i] == null) {  break;  }  row++;  if (row == allowed) {  System.out.println(array[i]);  allowed \*= 2;  row = 0;  } else {  System.out.print(array[i] + " ");  }  }  if (row != 0) {  System.out.println("\n");  } else {  System.out.println();  }  }  public static void main(String[] args) throws Exception {  BinaryHeap h = new BinaryHeap(50);  Comparable[] a = new Comparable[] { 10, 12, 1, 14, 6, 5, 8, 15, 3, 9, 7, 4, 11, 13, 2 };  for (Comparable i : a) {  h.insert(i);  }  h.print();  /\*  h.deleteMin();  h.print();  h.deleteMin();  h.print();  h.deleteMin();  h.print();  \*/  }  } |

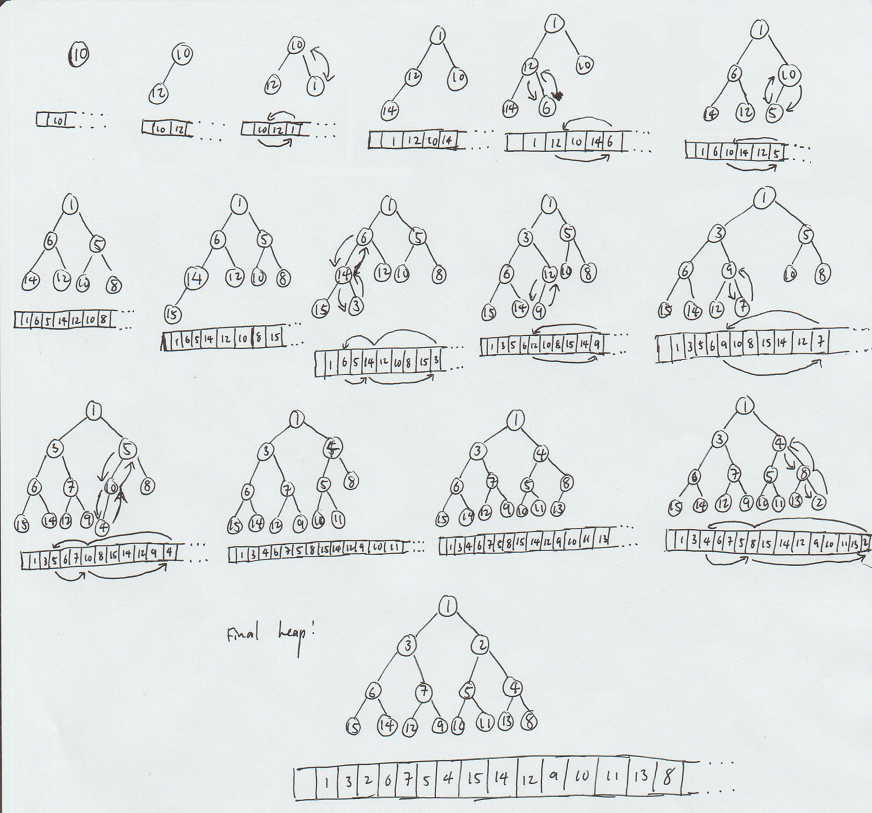
Ex-2: Use the above implementation to do exercise 6-2 and 6-3 of page 263. Include the code to display the tree. Make sure to have the screen shot of the result.

The MinHeap:

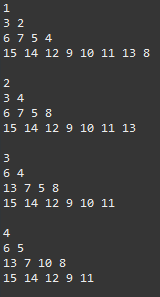


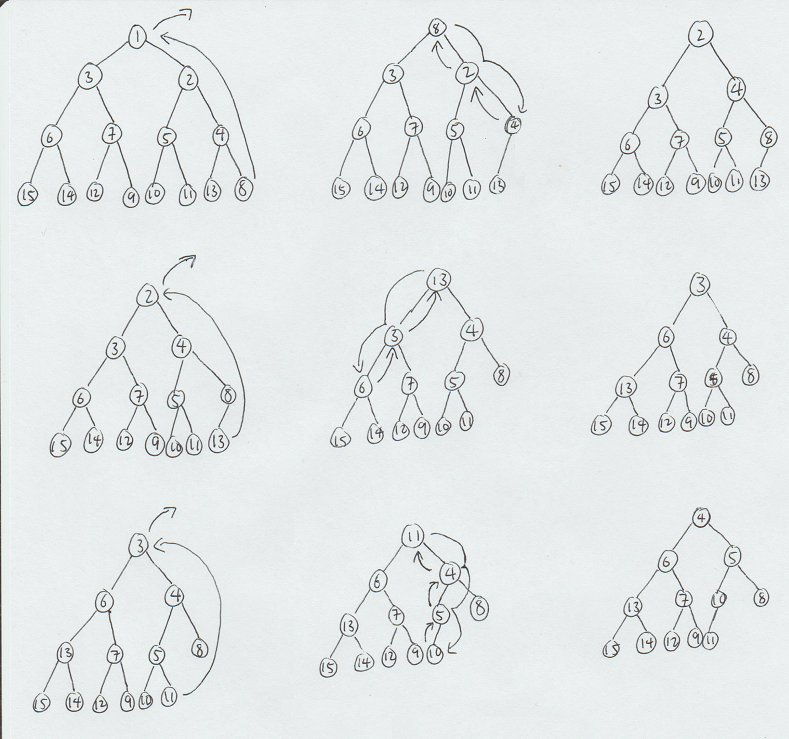
6.2a: Show the result of inserting 10, 12, 1, 14, 6, 5, 8, 15, 3, 9, 7, 4, 11, 13, and 2, one at a time, into an initially empty binary heap.

Process step by step:



6.3: Show the result of performing three deleteMin operations in the heap of the previous exercise.





Ex-3: Rewrite the BinaryHeap insert method by placing a reference to the inserted item in position 0.

Code:

|  |
| --- |
| public void insert(AnyType object) {  if (currentSize == array.length - 1) {  enlargeArray(array.length \* 2 + 1);  }  array[0] = object;  // Percolate up.  int hole = ++currentSize;  for (; object.compareTo(array[hole / 2]) < 0; hole /= 2) {  array[hole] = array[hole / 2];  }  array[hole] = array[0];  } |

Ex-4: Implement heapsort to perform descending sort.

Code:

|  |
| --- |
| public class HeapSort {  public static void main(String[] args) {  Comparable[] b = new Comparable[] {142, 543, 123, 65, 453, 879, 572, 434, 111, 242, 811, 102};  System.out.print("Not sorted: ");  for (Comparable f : b) {  System.out.print(f + " ");  }  System.out.print("\nSorted: ");  heapSort(b);  for (Comparable f : b) {  System.out.print(f + " ");  }  }  private static int leftChild(int i) {  return 2 \* i + 1;  }  private static <AnyType extends Comparable<? super AnyType>> void percDown(AnyType[] a, int i, int n) {  int child;  AnyType tmp;  for (tmp = a[i]; leftChild(i) < n; i = child) {  child = leftChild(i);  if (child != n - 1 && a[child].compareTo(a[child + 1]) > 0) {  child++;  }  if (tmp.compareTo(a[child]) > 0) {  a[i] = a[child];  } else {  break;  }  }  a[i] = tmp;  // print(a);  }  public static <AnyType extends Comparable<? super AnyType>> void heapSort(AnyType[] a) {  for (int i = a.length / 2 - 1; i >= 0; i--) {  percDown(a, i, a.length);  }  // System.out.println("MinHeap built.");  for (int i = a.length - 1; i > 0; i--) {  swapReferences(a, 0, i);  percDown(a, 0, i);  }  }  public static <AnyType> void swapReferences(AnyType[] a, int index1, int index2) {  AnyType tmp = a[index1];  a[index1] = a[index2];  a[index2] = tmp;  // print(a);  }  public static <AnyType> void print(AnyType[] a) {  for (AnyType i: a) {  System.out.print(i + " ");  }  System.out.println();  }  } |

Ex-5: Show how heapsort processes the input 142, 543, 123, 65, 453, 879, 572, 434, 111, 242, 811, 102 (show each step). Also use the code in Ex-4 to show how it sorts these values. Include the screen shot of the result.

Console output:

