DM507

Project part III

Esben H. Christensen 300790, M1

Delivered 25/05/2012

Indhold

1 Task 1

I will argue that FingerInsert overholds inorder after insertion of a new node. If we enter the if statement ($x_j \geq T.\max.key$) we insert the new node z as the right child of $T.max$. Since $z.key = x_j \geq T.\max.key$ we still have inorder because all nodes beside z are inorder by the original tree and $z.key \geq T.\max.key \geq$ all other nodes.

If we enter the else statement ($x_j < T.\max.key$) then we search from $v = T.\max$ to it's parent with a while loop and if $x_j < v.p.key$ we go up to it's parent and we do this until $x_j \geq v.key$ or $v = T.root$. If we hit the root of the tree then it is just a normal search and after insertion inorder is upheld. If we end the while loop with $x_j \geq v.key$ then $x_j \geq v.p.key$ because nodes we go through is in decreasing order when going from right children to parents by the the figure in the task description. This means that $x_j \geq$ all nodes besides $v's$ subtree. This means that a normal binary search from the root would go through v and it will be the same as when searching from v. This means that searching from v and inserting as a result of this will be an inorder tree since a normal search from the root will result in an inorder tree.

2 Task 2

In this task I will argue the runtime of FingerTreeSort by analysing each part of the sorting process i.e.:

- Insertion = Search for insertion point + replace leaf with node.
- Rebalancing of the tree after each insertion.
- Retrieve the sortet input.

The search part of insertion for all nodes can be shown to be in $O\left(n + n\log\left(\frac{INV}{n} + 1\right)\right)$ and relpacing a leaf by a node takes constant time. The total work of rebalancing the tree after each insertion has been shown to take $O(n)$ in part II of the project because the tree is a red-black tree. Retrieving the sortet input takes $O(n)$ beacuse we only touch each node once.

This means that the total work will be:

$$\begin{aligned} O\left(n + n\log\left(\frac{INV}{n} + 1\right) + n + n\right) &= O\left(3n + n\log\left(\frac{INV}{n} + 1\right)\right) \\ &= O\left(n + n\log\left(\frac{INV}{n} + 1\right)\right) \end{aligned}$$

3 Task 3

I will start by looking at the asymptotic runtime for InsertionSort for different number of inversions since we know that InsertionSort runs in $\Theta(n + INV)$:

InsertionSort	
INV	$\Theta(n + INV)$
0	$\Theta(n + 0) = \Theta(n)$
n	$\Theta(n + n) = \Theta(n)$
$n^{3/2}$	$\Theta(n + n^{3/2}) = \Theta(n^{3/2})$
$\frac{n^2}{4}$	$\Theta(n + \frac{n^2}{4}) = \Theta(n^2)$

The asymptotic runtime for FingerTreeSort for different number of inversions has been shown to be $O(n+n\log(\frac{INV}{n}+1))$ and it can be shown that it is also $\Omega(n+n\log(\frac{INV}{n}+1))$ $\Rightarrow \Theta\left(n + n\log(\frac{INV}{n} + 1)\right)$:

FingerTreeSort	
INV	$\Theta(n + n\log\left(\frac{INV}{n} + 1\right))$
0	$\Theta(n + n\log\left(\frac{0}{n} + 1\right)) = \Theta(n + n \cdot 0) = \Theta\left(n\right)$
n	$\Theta(n + n\log\left(\frac{n}{n} + 1\right)) = \Theta(n + n \cdot \log\left(2 + 1\right)) = \Theta\left(n\right)$
$n^{3/2}$	$\Theta\left(n + n\log\left(\frac{n^{3/2}}{n} + 1\right)\right) = \Theta(n + n\log\left(n^{1/2} + 1\right)) = \Theta\left(n + n\frac{1}{2}\log(n)\right) = \Theta(n\log(n))$
$\frac{n^2}{4}$	$\Theta\left(n + \log\left(\frac{n^2}{4n} + 1\right)\right) = \Theta\left(n + n\log\left(\frac{1}{4}n + 1\right)\right) = \Theta\left(n + n\log\left(n\right)\right) = \Theta\left(n\log\left(n\right)\right)$

The asymptotic runtime for MergeSort does not depend on the number of inversions and the runtime for MergeSort has been shown in part I of the project to be $\Theta(n\log\left(n\right))$.

4 Task 4

In the implementation of FingerTreeSort I will reuse the implementation of RedBlack trees from the second part of the project, where the insert method will use FingerInsert as described in the project description. The inorderTraversal method has also been modified so it will return an integer arraylist so when measuring the runtime this is not included because the integers are sorted in the integer arraylist. The tree will also have an extra attribute *max* which will indicate the largest key in the tree which is needed in FingerInsert but the rest is exactly as in the second part of the project. I have implemented FingerInsert with a method insert which will create a new node with the given key and call the method FingerInsert with the tree and this node:

```
public void insert(int k) {
 RBnode z = new RBnode(k,this.nil);
 FingerInsert(z);
}%
%EndExpansion
\ The implicit pseudo code of FingerInsert where we use this $max$ attribute
of the tree and insert the new node as $max$ if it is larger than the $max$
value and going upwards in the tree until it is larger than the next parent\
is implemented as follows:
%TCIMACRO{%
%\TeXButton{FingerInsert Pseudo}{\begin{verbatim}
%public void FingerInsert(RBnode z) {
% RBnode y = this.nil;
% if (z.key >= this.max.key) {
%   y = this.max;
%   this.max = z;
% } else {
%   RBnode v = this.max;
%   while (z.key < v.key && v != this.root) {
%     v = v.p;
%   }
%   ...
% }
%}
%

public void FingerInsert(RBnode z) {
 RBnode y = this.nil;
```

```
 if (z.key >= this.max.key) {
   y = this.max;
   this.max = z;
 } else {
   RBnode v = this.max;
   while (z.key < v.key && v != this.root) {
     v = v.p;
   }
   ...
 }
}
```

The search for insertionpoint in the else statement after going upwards in the tree is executed by a while loop which implements a simple binary search and we will remember the last node before we hit the *nil* node:

```
public void FingerInsert(RBnode z) {
 ...
 else {
   while (v != this.nil) {
     y = v;
     if (z.key < v.key) {
       v = v.left;
     } else {
       v = v.right;
     }
   }
 }
 z.p = y;
 ...
}
```

With the search we have found the parent of the new node so we can now replace the leaf with the node depending only on the relationship between the keys of the parent and the node:

```
public void FingerInsert(RBnode z) {
 ...
 if (y == this.nil) {
   this.root = z;
   this.max = z;
```

```
 } else if (z.key < y.key) {
   y.left = z;
 } else {
   y.right = z;
 }

 z.left = this.nil;
 z.right = this.nil;

 z.color = false;  // RED
 RBInsertFixup(z);
}
```

Notice that if the parent of the node is the *nil* node then we originally have an empty tree and the new node should be both the *root* and the *max* node. And finally we use the method RBInsertFixup from the second part of the project to rebalance the tree. MergeSort and InsertionSort has been implemented as separate methods in my program which will measure runtime and the implementation can be found in the appendix of Time.java. MergeSort has been implemented as in the first part of the project along with FastInv which will count the number of inversions, and InsertionSort was implemented in an exercise session.

5 Task 5

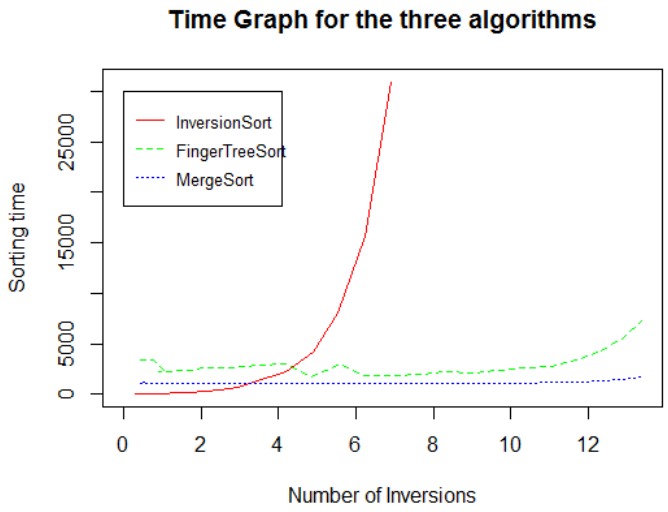
In this task I will compare runtimes of the three search algorithms of Task 4. I have written a program with a main method where the user will be prompted to choose which algorithm it should use and will at the end output a file with the given runtimes measured and inversions counted for a given k as described in the task desription. My computer used with MergeSort about one second to sort the array with input size 8 million. I have then calculated largest i so $k = 2^i$ is not larger than 8 million which is calculated by:

$$\begin{aligned} i.\max &= \log_2\left(8 \cdot 10^6\right) = 22.9316 < 23 \\ \Rightarrow i.\max &= 22 \end{aligned}$$

I will use a *switch* statement for each algorithm so I only run one of the algorithms at each run. Since InsertionSort will take a very long time to sort an array with many inversions the max value of i will in this case be 12 as it will take about half of a minute with InsertionSort. With a for-loop going from 0 to $i.max$ the program will generate input with the given method generateInput, count the number of inversions with FastInv on a copy of the generated input and sort the input with the given algorithm where we measure

the time by the difference of the current time found with System.currentTimemillis(). The program will print both the inversions and time used in the command window but will also add these to a file which can then be used to make a graph of the runtimes.

When running the program for the three different algorithms I am able to get the following graph with time on the second axis and $\log(\frac{INV}{n})$ on the first axis which will give a graph where the three algorithms all depend on the number of inversions relative to the input size and in logarithm:



[1]

We see that InsertionSort is fastest when sorting an array with few inversions relative to the input size but when the number of inversions increase it begins to take longer and longer to sort the input. We also notice that MergeSort is as suspected constant and does not depend on the number of inversions. FingerTreeSort is steady with a growing number of inversions but when we get close to the limit of inversions (n^2) it starts to increase but not as much as InsertionSort. I did not however get the result that FingerTreeSort is better than MergeSort with any number of inversions which should have been the case according to Task 3, but this might be because the tree from the implementation takes up a lot of memory along with the fact that the original array still is in the memory, but I am not sure of this.

[1]The graph has been made using the program R and the script used has also been added to the appendix.

6 Appendix

6.1 Time.java

```
import java.io.*;
import java.util.*;
import java.lang.*;

public class Time {

  public static void main(String[] args) throws IOException {

    // size of input
    int n = 8000000;

    Scanner sc = new Scanner(System.in);
    System.out.println("Which algorithm shoud be used? \n 1: MergeSort \n 2: FingerTr
    // 1: MergeSort \n 2: FingerTreeSort \n 3: InsertionSort

    int Alg = sc.nextInt();

    int max = 22; // = floor(lg2(8000000))

    if (Alg == 3) {
      max = 12;
    }
    File f; // file which will hold inversions and runtime
    switch (Alg) {
      case 1: {
        f = new File("MergeSortTime.txt");
        if(!f.exists()){
          f.createNewFile();
        }
        break;}
      case 2: {
        f = new File("FingerTreeSortTime.txt");
        if(!f.exists()){
          f.createNewFile();
        }
```

```
    break;}
  case 3: {
    f = new File("InsertionSortTime.txt");
    if(!f.exists()){
      f.createNewFile();
    }
    break;}
  default: {
    f = new File("Invalid Algorithm.txt");
    System.out.println("Invalid Algorithm!!");
    break;}
}

Writer out = new FileWriter(f, /*append = */true);
out.write("Inversions Time\n");
out.close();

for (int i = 0; i <= max; i++) {
  int k = (int) Math.pow(2,i);

  int[] A = generateInput(n,k);
  int[] Acopy = new int[A.length];
  // Create a copy of the input
  for (int j = 0; j < A.length; j++) {
    Acopy[j] = A[j];
  }
  /* Count inversions which will be stored in a long
  because int cannot hold the larger number of inversions*/
  long INV = FastInv(Acopy);

  long time = 0;

  int[] Asorted;

  switch (Alg) {
    case 1: {
      long t1 = System.currentTimeMillis();
      Asorted = MergeSort(A,0,A.length-1);
      long t2 = System.currentTimeMillis();
```

```
        time = t2-t1;
        break;}
      case 2: {
        long t1 = System.currentTimeMillis();
        ArrayList<Integer> ASorted = FingerTreeSort(A);
        long t2 = System.currentTimeMillis();

        time = t2-t1;
        break;}
      case 3: {
        long t1 = System.currentTimeMillis();
        Asorted = InsertionSort(A);
        long t2 = System.currentTimeMillis();

        time = t2 - t1;
        break;}
    }
    System.out.println(" Inversions: " + INV + "\n Time:       " + time);
    out = new FileWriter(f, /*append = */true);
    out.write(INV + " " + time + "\n");

    out.close();
  }
}

public static long FastInv(int[] A) {
  return MergeSortINV(A,0,A.length-1);

}

  public static long MergeSortINV(int[] A, int p, int r) {
  if (p < r) {
    int q = (p+r)/2;
    long z = MergeSortINV(A,p,q) + MergeSortINV(A,q+1,r);
    z =+ MergeINV(A,p,q,r);
    return z;
  } else {
    return 0;
```

```
  }

}

public static long MergeINV(int[] A, int p, int q, int r) {
  int n1 = q - p + 1;
  int n2 = r - q;
  int[] L = new int[n1+1];
  int[] R = new int[n2+1];
  for (int i = 0; i < n1; i++) {
    L[i] = A[p+i];
  }
  for (int j = 0; j < n2; j++) {
    R[j] = A[q+j+1];
  }
  L[n1] = Integer.MAX_VALUE;
  R[n2] = Integer.MAX_VALUE;

  int i = 0;
  int j = 0;

  long zCounter = 0;

  for (int k = p; k <= r; k++) {
    if (L[i] <= R[j]) {
      A[k] = L[i];
      i++;
    } else {
      A[k] = R[j];
      j++;
      /* When we enter the else then we know that the j'th position
      in R is smaller than the rest of L so the count will increase
      with the number of slots in L that has not been checked */
      zCounter = zCounter + L.length-1-i;
    }
  }

  return zCounter;
```

```
}

public static int[] MergeSort(int[] A, int p, int r) {
  if (p < r) {
    int q = (p+r)/2;
    A = MergeSort(A,p,q);
    A = MergeSort(A,q+1,r);
    A = Merge(A,p,q,r);
  }
  return A;
}

public static int[] Merge(int[] A, int p, int q, int r) {
  int n1 = q - p + 1;
  int n2 = r - q;
  int[] L = new int[n1+1];
  int[] R = new int[n2+1];
  for (int i = 0; i < n1; i++) {
    L[i] = A[p+i];
  }
  for (int j = 0; j < n2; j++) {
    R[j] = A[q+j+1];
  }
  L[n1] = Integer.MAX_VALUE;
  R[n2] = Integer.MAX_VALUE;

  int i = 0;
  int j = 0;

  for (int k = p; k <= r; k++) {
    if (L[i] <= R[j]) {
      A[k] = L[i];
      i++;
    } else {
      A[k] = R[j];
      j++;
    }
  }
```

```
    return A;

  }

  public static ArrayList<Integer> FingerTreeSort(int[] A) {
    RBTree T = new RBT();
    for (int j = 0; j < A.length; j++) {
      T.insert(A[j]);
    }
    return T.inOrderTraversal();
  }

  public static int[] InsertionSort(int[] A) {
    for (int j = 1; j < A.length; j++) {
      int key = A[j];
      int i = j-1;
      while (i >= 0 && A[i] > key) {
        A[i+1] = A[i];
        i = i-1;
      }
      A[i+1] = key;
    }
    return A;
  }

  private static int[] generateInput(int n, int k){

// Create array of length n
  int[] array = new int[n];

// Fill array with sorted numbers
  for (int i = 0; i<n; i++){
    array[i] = i;
  }

// Now k times swap a random pair
    Random randomGenerator = new Random();
  int k1, k2;
  int temp;
```

```
    for (int j = 0; j<k; j++){
    k1 = randomGenerator.nextInt(n);
    k2 = randomGenerator.nextInt(n);
    temp = array[k1];
    array[k1] = array[k2];
      array[k2] = temp;
  }

// Return resulting array
return array;

  }
}
```

6.2 RBT.java

```
import java.util.*;

public class RBT implements RBTree {

  // the three attributes of RBT objects
  private RBnode nil;
  private RBnode root;
  private RBnode max;

  // the constructor of RB-trees which takes no arguments
  public RBT() {
    this.nil = new RBnode(0,nil);
    this.root = this.nil;
    this.max = this.nil;
  }

  private class RBnode { // class for creating the nodes

    public int key;
    public RBnode p;
    public RBnode left;
    public RBnode right;
    public boolean color;
```

```
  public RBnode(int key, RBnode nil) {
    this.key = key;
    this.p = nil;
    this.left = nil;
    this.right = nil;
    this.color = true; // color is a boolean where true
                       // means black and false means red
  }
}

/* The search method as specified in the interface which
will call my own method TreeSearch which is recursive
and it will start at the root */
public boolean search(int k) {
  RBnode x = this.root;
  return TreeSearch(x,k);
}

// TreeSearch as in the book where everything is trivial
private boolean TreeSearch(RBnode x, int k) {
  if (x == this.nil) {
    return false;
  } else if (k == x.key) {
    return true;
  } else if (k < x.key) {
    return TreeSearch(x.left, k);
  } else {
    return TreeSearch(x.right,k);
  }
}

/* the insert method as specified in the interface which
will call the method FingerInsert with a node which has
been created with the specified key and the nil object of
the tree as parent and both children */
public void insert(int k) {
  RBnode z = new RBnode(k,this.nil);
  FingerInsert(z);
```

```
}

public void FingerInsert(RBnode z) {
  RBnode y = this.nil;
  if (z.key >= this.max.key) {
  // new node is larger than all other nodes
    y = this.max;
    this.max = z;
  } else {
    RBnode v = this.max;
    while (z.key < v.key && v != this.root) {
      // search upwards
      v = v.p;
    }
    while (v != this.nil) {
      // search downwards
      y = v;
      if (z.key < v.key) {
        v = v.left;
      } else {
        v = v.right;
      }
    }
  }
  // Insert new node at correct position
  z.p = y;
  if (y == this.nil) {
    this.root = z;
    this.max = z;
  } else if (z.key < y.key) {
    y.left = z;
  } else {
    y.right = z;
  }

  // set appropriate attributes according to RB-tree
  z.left = this.nil;
  z.right = this.nil;
  z.color = false;  // RED
```

```
  RBInsertFixup(z);
}



/* This is the method which will make sure that the tree after
insertion will be a RB-tree again and is the same as the pseudo
code in the book */
private void RBInsertFixup(RBnode z) {
  while (z.p.color == false) {
    if (z.p == z.p.p.left) {  // if z's parent is a left child
      RBnode y = z.p.p.right;
      if (y.color == false) {   // Case 1
        z.p.color = true;       // Case 1
        y.color = true;         // Case 1
        z.p.p.color = false;    // Case 1
        z = z.p.p;              // Case 1
      } else {
        if (z == z.p.right) {   // Case 2
          z = z.p;              // Case 2
          LeftRotate(z);        // Case 2
        }
        z.p.color = true;       // Case 3
        z.p.p.color = false;    // Case 3
        RightRotate(z.p.p);     // Case 3
      }
    } else {                  // if z's parent is a right child
      RBnode y = z.p.p.left;
      if (y.color == false) {   // Case 1
        z.p.color = true;       // Case 1
        y.color = true;         // Case 1
        z.p.p.color = false;    // Case 1
        z = z.p.p;              // Case 1
      } else {
        if (z == z.p.left) {    // Case 2
          z = z.p;              // Case 2
          RightRotate(z);       // Case 2
        }
```

```
        z.p.color = true;       // Case 3
        z.p.p.color = false;    // Case 3
        LeftRotate(z.p.p);      // Case 3
      }
    }
  }
  this.root.color = true;
}

/* Both LeftRotate is as in the book and RightRotate is symmetric
to RightRotate and it is created with figure 13.2 as background */
private void LeftRotate(RBnode x) {
  RBnode y = x.right;
  x.right = y.left;
  if (y.left != this.nil) {
    y.left.p = x;
  }
  y.p = x.p;
  if (x.p == this.nil) {
    this.root = y;
  } else if (x == x.p.left) {
    x.p.left = y;
  } else {
    x.p.right = y;
  }
  y.left = x;
  x.p = y;
}

private void RightRotate(RBnode y) {
  RBnode x = y.left;
  y.left = x.right;
  if (x.right != this.nil) {
    x.right.p = y;
  }
  x.p = y.p;
  if (y.p == this.nil) {
    this.root = x;
  } else if (y == y.p.right) {
```

```
    y.p.right = x;
  } else {
    y.p.left = x;
  }
  x.right = y;
  y.p = x;
}

/* The method inorderTraversal as specified in the interface which
will call the method inorderTreeWalk where I use an ArrayList where
the length isn't fixed, so we do not need to know the size before hand.
*/
public ArrayList<Integer> inOrderTraversal() {
  ArrayList<Integer> A = new ArrayList<Integer>();
  // We start at the root
  return inorderTreeWalk(this.root, A);
}

/* My own method inorderTreeWalk as in the book except instead of
printing the key it will save the key in the ArrayList with the add
method of ArrayList */
private ArrayList<Integer> inorderTreeWalk(RBnode x, ArrayList<Integer> A) {
  if (x != this.nil) {
    A = inorderTreeWalk(x.left,A);
    A.add(x.key);
    A = inorderTreeWalk(x.right, A);
  }
  return A;
}

/* The method isRedBlack as specified in the interface. This method
along with BlackHeight and TwoRedsInRow have been implemented using
the pseudo code from the appendix of the problem description */

public boolean isRedBlack(){
  /* Because this.root.color is a boolean where black means true
  I do not need to write this.root.color == true */
  return (this.root.color && BlackHeight(this.root) >= 0 &&
          (! TwoRedsInRow(this.root)));
```

```
  }

  private int BlackHeight(RBnode v) {
    if (v == this.nil) {
      return 0;
    } else {
      int h1 = BlackHeight(v.left);
      int h2 = BlackHeight(v.right);
      if (h1 != h2 || h1 == -1) {
        return -1;
      } else if (v.color == true) {
        return h1 + 1;
      } else {
        return h1;
      }
    }
  }

  private boolean TwoRedsInRow(RBnode v) {
    if (v == this.nil) {
      return false;
    } else if (v.color == false &&
               (v.left.color == false || v.right.color == false) ) {
      return true;
    } else {
      return (TwoRedsInRow(v.left) || TwoRedsInRow(v.right) );
    }
  }
}
```

6.3 RBTree.java

```
import java.util.*;

public interface RBTree {
  public boolean search(int k);
  public void insert(int k);
  public ArrayList<Integer> inOrderTraversal();
  public boolean isRedBlack();
```

```
}
```

6.4 InversionTimeGraph.R

```
# DM507 Project del III
n = 8000000

Ins.df <- read.table("InsertionSortTime.txt", header = TRUE)
INVI <- Ins.df$Inversions
TimeI <- Ins.df$Time

Fin.df <- read.table("FingerTreeSortTime.txt", header = TRUE)
INVF <- Fin.df$Inversions
TimeF <- Fin.df$Time

Mer.df <- read.table("MergeSortTime.txt", header = TRUE)
INVM <- Mer.df$Inversions
TimeM <- Mer.df$Time

plot(TimeI~log(INVI/n+1), type = "l",
     xlim = c(0,max(log(INVF/n+1))), main = "Time Graph for the three algorithms",
     xlab = "Number of Inversions", ylab = "Sorting time")
lines(TimeI~log(INVI/n+1), col = "red", lty = 1)
lines(TimeF~log(INVF/n+1), col = "green", lty = 2)
lines(TimeM~log(INVM/n+1), col = "blue", lty = 3)
legend(0, 30000, c("InversionSort","FingerTreeSort","MergeSort"), cex=0.8,
       col=c("red","green","blue"), lty=1:3)
```