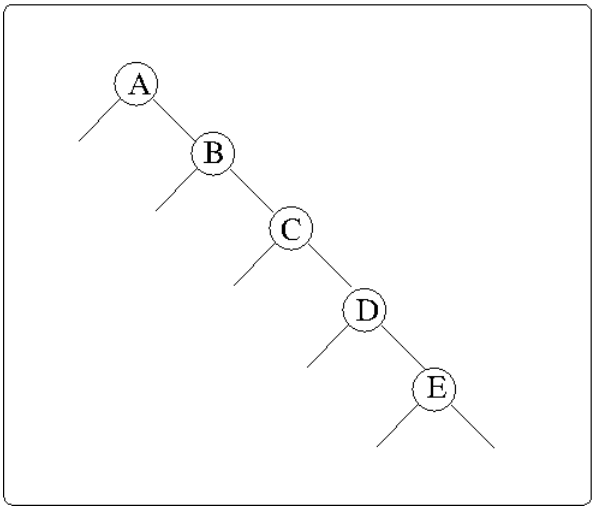
1. Drzewa 2-3-4

W przypadku drzewa typu Binary Search Tree dołączanie nowych węzłów o kluczach tworzących ciąg uporządkowany prowadzi do otrzymania drzewa niezrównoważonego. Struktura takiego drzewa w najgorszym przypadku przypomina listę jednostronnie wiązaną. Przykład drzewa niezrównoważonego:

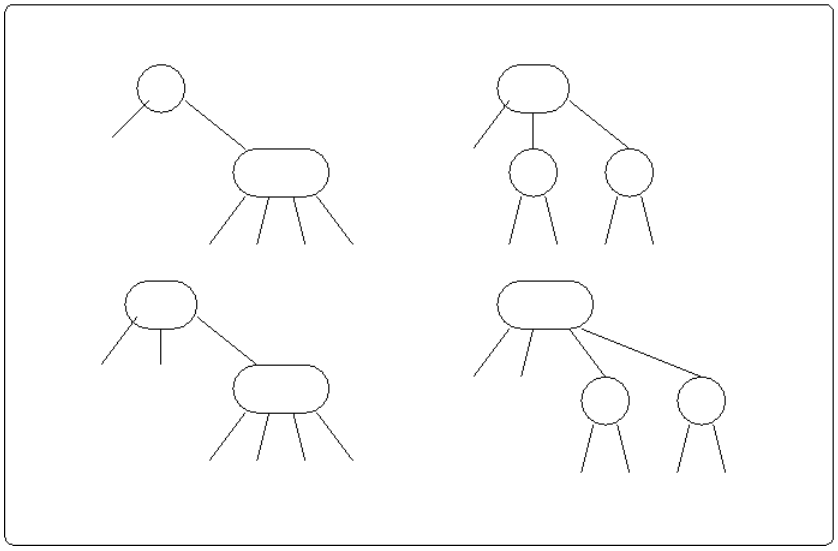


Rys. 1. Przykład drzewa niezrównoważonego.

W takich przypadkach wyszukiwanie węzła o zadanym kluczu zwykle będzie rzędu O(N) zamiast $O(\log_2 N)$. Drzewa 2-3-4 oraz RBT zapewniają zrównoważenie drzewa bliskie optymalnemu.

Drzewo 2-3-4 to takie, które nie zawiera węzłów albo węzły typu:
2-węzeł: 1 klucz (A) oraz 2 wskaźniki (na potomka o kluczu mniejszym od A oraz na potomka o kluczu większym od A),
3-węzeł: 2 klucze (A, B) oraz 3 wskaźniki (na potomka: z kluczem mniejszym niż A, z kluczem między A i B, z kluczem większym od B),
4-węzeł: 3 klucze (A, B, C) oraz 4 wskaźniki (na potomków z kluczami zdefiniowanymi przez zakresy kluczy A, B i C).

Tworzenie drzewa 2-3-4:
W przypadku dołączania nowego węzła do 2-węzła należy go zamienić na 3-węzeł.
Podobnie gdy dołączamy do 3-węzła zamieniamy go na 4-węzeł.
W przypadku gdy napotkamy w drodze na dół drzewa 4-węzeł, rozdzielamy go na dwa 2-węzły z kluczem środkowym przekazanym dla ojca 4-węzła.
Ilekroć korzeń drzewa staje się 4-węzłem zamieniamy go na trójkąt złożony z trzech 2-węzłów.



Rys. 2. Zamiana 4-węzła na dwa 2-węzły z przekazaniem klucza do ojca węzła który zamieniamy na 3-węzeł (górna część rysunku), zamiana 4-węzła na dwa 2-węzły z przekazaniem klucza do ojca węzła który zamieniamy na 4-węzeł (dolna część rysunku).

Przykład 1 (C++). Realizacja drzewa 2-3-4.

```
//tree234.cpp
//demonstrates 234 tree
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class DataItem
   {
   public:
      double dData;                 //one piece of data
//--------------------------------------------------------------
   DataItem() : dData(0.0)          //default constructor
      {  }
//--------------------------------------------------------------
   DataItem(double dd) : dData(dd)  //1-arg constructor
      {  }
//--------------------------------------------------------------
   void displayItem()               //format "/27"
      { cout << "/" << dData; }
   };  //end class DataItem
////////////////////////////////////////////////////////////////
class Node
   {
   private:
      enum {ORDER=4};
      int numItems;
      Node* pParent;
      Node* childArray[ORDER];      //array of ptrs to nodes
      DataItem* itemArray[ORDER-1]; //array of ptrs to data
   public:
//--------------------------------------------------------------
   Node() : numItems(0)             //constructor
      {
      for(int j=0; j<ORDER; j++)    //initialize arrays
         childArray[j] = NULL;
      for(int k=0; k<ORDER-1; k++)
```

```
         itemArray[k] = NULL;
      }
//--------------------------------------------------------------
   //connect child to this node
   void connectChild(int childNum, Node* pChild)
      {
      childArray[childNum] = pChild;
      if(pChild != NULL)
         pChild->pParent = this;
      }
//--------------------------------------------------------------
   //disconnect child from this node, return it
   Node* disconnectChild(int childNum)
      {
      Node* pTempNode = childArray[childNum];
      childArray[childNum] = NULL;
      return pTempNode;
      }
//--------------------------------------------------------------
   Node* getChild(int childNum)
      { return childArray[childNum]; }
//--------------------------------------------------------------
   Node* getParent()
      { return pParent; }
//--------------------------------------------------------------
   bool isLeaf()
      { return (childArray[0]==NULL) ? true : false; }
//--------------------------------------------------------------
   int getNumItems()
     { return numItems; }
//--------------------------------------------------------------
   DataItem getItem(int index)       //get DataItem at index
      { return *( itemArray[index] ); }
//--------------------------------------------------------------
   bool isFull()
      { return (numItems==ORDER-1) ? true : false; }
//--------------------------------------------------------------
   int findItem(double key)          //return index of
      {                                      //item (within node)
      for(int j=0; j<ORDER-1; j++)           //if found,
         {                                   //otherwise,
         if(itemArray[j] == NULL)            //return -1
            break;
         else if(itemArray[j]->dData == key)
            return j;
         }
      return -1;
      }  //end findItem
//--------------------------------------------------------------
   int insertItem(DataItem* pNewItem)
      {
      //assumes node is not full
      numItems++;                            //will add new item
      double newKey = pNewItem->dData;       //key of new item

      for(int j=ORDER-2; j>=0; j--)          //start on right,
         {                                   //   examine items
         if(itemArray[j] == NULL)            //if item null,
            continue;                        //go left one cell
         else                                //not null,
            {                                //get its key
```

```
            double itsKey = itemArray[j]->dData;
            if(newKey < itsKey)             //if it's bigger
               itemArray[j+1] = itemArray[j]; //shift it right
            else
               {
               itemArray[j+1] = pNewItem;  //insert new item
               return j+1;                 //return index to
               }                           //   new item
            }  //end else (not null)
         }  //end for                      //shifted all items,
      itemArray[0] = pNewItem;             //insert new item
      return 0;
      }  //end insertItem()
//--------------------------------------------------------------
   DataItem* removeItem()           //remove largest item
      {
      //assumes node not empty
      DataItem* pTemp = itemArray[numItems-1]; //save item
      itemArray[numItems-1] = NULL;            //disconnect it
      numItems--;                              //one less item
      return pTemp;                            //return item
      }
//--------------------------------------------------------------
   void displayNode()               //format "/24/56/74/"
      {
      for(int j=0; j<numItems; j++)
         itemArray[j]->displayItem();          //format "/56"
      cout << "/";                             //final "/"
      }
//--------------------------------------------------------------
   };  //end class Node
////////////////////////////////////////////////////////////////
class Tree234
   {
   private:
      Node* pRoot;                  //root node
   public:
//--------------------------------------------------------------
   Tree234()
      { pRoot = new Node; }
//--------------------------------------------------------------
   int find(double key)
      {
      Node* pCurNode = pRoot;              //start at root
      int childNumber;
      while(true)
         {
         if(( childNumber=pCurNode->findItem(key) ) != -1)
            return childNumber;            //found it
         else if( pCurNode->isLeaf() )
            return -1;                     //can't find it
         else                              //search deeper
            pCurNode = getNextChild(pCurNode, key);
         }  //end while
      }
//--------------------------------------------------------------
   void insert(double dValue)       //insert a DataItem
      {
      Node* pCurNode = pRoot;
      DataItem* pTempItem = new DataItem(dValue);
```

```
      while(true)
         {
         if( pCurNode->isFull() )          //if node full,
            {
            split(pCurNode);               //split it
            pCurNode = pCurNode->getParent(); //back up
                                           //search once
            pCurNode = getNextChild(pCurNode, dValue);
            }  //end if(node is full)

         else if( pCurNode->isLeaf() )     //if node is leaf,
            break;                         //go insert
         //node is not full, not a leaf; so go to lower level
         else
            pCurNode = getNextChild(pCurNode, dValue);
         }  //end while

      pCurNode->insertItem(pTempItem);     //insert new item
      }  //end insert()
//-------------------------------------------------------------
   void split(Node* pThisNode)       //split the node
      {
      //assumes node is full
      DataItem *pItemB, *pItemC;
      Node *pParent, *pChild2, *pChild3;
      int itemIndex;

      pItemC = pThisNode->removeItem();    //remove items from
      pItemB = pThisNode->removeItem();    //this node
      pChild2 = pThisNode->disconnectChild(2); //remove children
      pChild3 = pThisNode->disconnectChild(3); //from this node

      Node* pNewRight = new Node;          //make new node

      if(pThisNode==pRoot)                 //if this is the root,
         {
         pRoot = new Node();               //make new root
         pParent = pRoot;                  //root is our parent
         pRoot->connectChild(0, pThisNode); //connect to parent
         }
      else                                 //this node not the root
         pParent = pThisNode->getParent(); //get parent

      //deal with parent
      itemIndex = pParent->insertItem(pItemB); //item B to parent
      int n = pParent->getNumItems();      //total items?

      for(int j=n-1; j>itemIndex; j--)     //move parent's
         {                                 //connections
         Node* pTemp = pParent->disconnectChild(j); //one child
         pParent->connectChild(j+1, pTemp); //to the right
         }
                                    //connect newRight to parent
      pParent->connectChild(itemIndex+1, pNewRight);

      //deal with newRight
      pNewRight->insertItem(pItemC);       //item C to newRight
      pNewRight->connectChild(0, pChild2); //connect to 0 and 1
      pNewRight->connectChild(1, pChild3); //on newRight
      }  //end split()
//-------------------------------------------------------------
```

```
   //gets appropriate child of node during search for value
   Node* getNextChild(Node* pNode, double theValue)
      {
      int j;
      //assumes node is not empty, not full, not a leaf
      int numItems = pNode->getNumItems();
      for(j=0; j<numItems; j++)          //for each item in node
         {                               //are we less?
         if( theValue < pNode->getItem(j).dData )
            return pNode->getChild(j);   //return left child
         }  //end for                    //we're greater, so
      return pNode->getChild(j);         //return right child
      }
//--------------------------------------------------------------
   void displayTree()
      {
      recDisplayTree(pRoot, 0, 0);
      }
//--------------------------------------------------------------
   void recDisplayTree(Node* pThisNode, int level,
                                             int childNumber)
      {
      cout << "level=" << level
           << " child=" << childNumber << " ";
      pThisNode->displayNode();            //display this node
      cout << endl;

      //call ourselves for each child of this node
      int numItems = pThisNode->getNumItems();
      for(int j=0; j<numItems+1; j++)
         {
         Node* pNextNode = pThisNode->getChild(j);
         if(pNextNode != NULL)
            recDisplayTree(pNextNode, level+1, j);
         else
            return;
         }
      }  //end recDisplayTree()
//--------------------------------------------------------------
   };  //end class Tree234
////////////////////////////////////////////////////////////////
int main()
   {
   double value;
   Tree234* pTree = new Tree234;
   pTree->insert(50);
   pTree->insert(40);
   pTree->insert(60);
   pTree->insert(30);
   pTree->insert(70);

   while(true)
      {
      int found;

      cout << "Enter first letter of show, insert, or find: ";
      char choice;
      cin >> choice;
      switch(choice)
         {
         case 's':
```

```
            pTree->displayTree();
            break;
         case 'i':
            cout << "Enter value to insert: ";
            cin >> value;
            pTree->insert(value);
            break;
         case 'f':
            cout << "Enter value to find: ";
            cin >> value;
            found = pTree->find(value);
            if(found != -1)
               cout << "Found " << value << endl;
            else
               cout << "Could not find " << value << endl;
            break;
         default:
            cout << "Invalid entry\n";
         }  //end switch
      }  //end while
   delete pTree;
   return 0;
   }  //end main()
```

Zadanie 1. Wstaw do drzewa 2-3-4 300000 losowo wygenerowanych kluczy. Powtórz tworzenie drzewa 2-3-4 i wstawianie kluczy 100 razy (100 rund). Dla każdej rundy zmierz czas potrzebny na wstawienie 300000 kluczy. Oblicz wartość średnią, medianę oraz odchylenie standardowe z uzyskanych pomiarów.

Zadanie 2. Wykonaj analogiczny pomiar jak w zadaniu 1 lecz z użyciem drzewa Red-Black tree.

Opracowano na podstawie:
Robert Lafore – "Teach Yourself Data Structures And Algorithms In 24 Hours"