Balanced Binary trees

Based on the PC-notes by A. Rauzy

M. Vazirgiannis
mvazirg@lix.polytechnique.fr

February 20 2013
Introduction - context

• **Priority Queues**
• A *priority queue* is a data structure that defines (at least) three operations on ordered elements: *add*, *peek*, and *remove*.

```java
public interface PriorityQueue<E extends Comparable<E>> {
    void add(E obj);
    E peek(); // returns the most important value in the queue
    E remove(); // removes and returns the most important value
}
```
Introduction - context

- **Binary Heaps**
  - One way to implement a priority queue is by using a *binary heap*: binary heap is a binary tree with two additional properties:
    - The tree is *complete*, i.e. every level of the tree is full, except perhaps the bottom level, and it is filled *left-to-right*.
    - For *every node* in the tree, the *value of the node's children* are *larger than or equal* to the value of the node.
    - Thus, the *smallest value* in the heap is always the *root* of the tree.

Also, because the tree is complete, we don't actually have to use a tree data structure to represent the tree; we can use an ArrayList and just number the nodes from left-to-right, top-to-bottom, starting at the root.
Introduction - context

- **Braun Trees**
- A *Braun tree* is a binary tree which is as balanced as it can possibly be. Whereas a binary heap requires a *complete* binary tree, in a Braun tree, *every* node satisfies the following conditions:

  - The left subtree has either the same number of nodes as the right subtree or one more.

```
1 node: X
2 nodes: X X
3 nodes: X X X
4 nodes: X X X X
5 nodes: X X X X X
6 nodes: X X X X X X
7 nodes: X X X X X X X
```
Braun Trees

We consider that the tree consists of nodes containing each one one element and the leaves have no content.

The Braun tree fulfills the following properties:
1. The root node element is the smallest of all the elements in the right and left subtrees.
2. Assume the root of the tree and then if $s$ is the size of the left subtree and $t$ the size of the right one then: $t <= s <= t+1$ (equilibrium).
3. If the root is a binary node then the left and the right subtrees are also Braun trees.

Two important observations:
- **Braun trees are a container objects.** A general principle of good programming of containers is not to give direct access to structures for storing objects. Always a class mediates between these structures and the user. For example, if you want to implement a list, create a List class that encapsulates the structures that will actually list cell the elements of the list.
- **The leaves of the tree does not bear elements.** They actually serve to represent the empty tree. It is therefore unnecessary to duplicate. A single leave will suffice.
Braun trees

Definition

a binary tree is a Braun tree if
- it is empty or
- if its left and right subtrees are Braun trees
- the size of the left one minus the size of the right one is either 0 or 1.
Question 1

• Propose a data structure to implement the Braun trees
• Assume that the elements stored in the tree
• are objects of type *KeyedObject* having the method *getKey* that returns an integer used for the objects comparison.
Question 1

```java
public class BraunTree {
    private BraunNode rootNode;
    private BraunNode emptyTree;
    public BraunTree() {
        this.emptyTree = new BraunNode();
        this.rootNode = emptyTree;
    }
}

class BraunNode {
    public final KeyedObject item;
    public final BraunNode leftTree;
    public final BraunNode rightTree;
    public BraunNode(KeyedObject item, BraunNode leftTree, BraunNode rightTree) {
        this.item = item;
        this.leftTree = leftTree;
        this.rightTree = rightTree;
    }
    ...
    ...
    ...
}
```
Question 2

• **Write a method to check if a braun tree is balanced.**

• **Hint:** Define a method `getBalancedSize` for the nodes that returns the size of the tree for a balanced one and -1 otherwise.

• This method would be called from the `isBalanced` method of the BraunTree.

```java
// in class BraunTree
public bool isBalanced()
{
    return(this.rootNode.getBalancedSize()) ;
}

// in class BraunNode
public int getBalancedSize()
{
    if (this==BraunTree.this.emptyTree)
        return(0) ;
    int s1, s2 ;
    s1 = this.leftTree.getBalancedSize() ;
    if (s1==-1) return(-1) ;
    s2 = this.rightTree.getBalancedSize() ;
    if (s2==-1) return(-1) ;
    return( s1>2 && s1<=s2+1 ? s1+s2+1 : -1) ;
}
```
Node insertion

- To insert an object $a$ on a Braun tree we do it in the following manner:
  - If the object to be inserted is greater than the object in the root: $< T1; b; T2 >$ we insert it in right subtree$(T2)$. It returns a node with value $b$, with the $left$ subtree the one just obtained and the subtree $T1$ as the $right$ subtree.
  - else, we do the same operation with inverse roles for $a$ and $b$
Question 3

- Write a method for inserting a node in the tree. What is the complexity of this method?

```java
// in class BraunTree
public void insertItem(KeyedObject item) {
    this.rootNode =
    this.rootNode.insertItem(item);
}
```

```java
// in class BraunNode
public BraunNode insertItem(KeyedObject s) {
    if (this==BraunTree.this.emptyTree)
        return(new
        BraunNode(s,BraunTree.this.emptyTree, 
        BraunTree.this.emptyTree)); 
    KeyedObject u, v ;
    BraunNode lt, rt ;
    if (s.getKey()<this.item.getKey()) {
        u = s ;
        v = this.item ;
    }
    else {
        u = this.item ;
        v = s ;
    }
    lt = this.rightTree.insertItem(v) ;
    rt = this.leftTree ;
    return( new BraunNode(u, lt, rt) ) ;
}
```
Complexity

• The above method is correct because it preserves the three properties defining trees Braun (see over).
• Braun trees are balanced, their maximum depth is $O(\log_2 n)$, where $n$ the number of elements in the tree.
• The method \textit{insertItem} descends one level in the tree for each each recursive call
• the number of operations per recursive call is bounded by a constant
• Therefore \textit{insertItem complexity} is $O(\log n)$. 
Question 4

- Describe the tree with the insertions of 3; 4; 1; 2; 5; 6.
Question 5

• Write an effective method to *retrieve an object from a tree* (any).
  The idea is:
  - To extract the object in the root if the tree consists of a single node.
  - otherwise to extract an object from the *left subtree* and switch *subtrees*.

• Find the complexity of this method.
Question 5

The implementation of this idea poses a technical problem:

• it must return an object

• modify the tree and then return new root.

• To return two objects, we will introduce a new class to return pairs (node, object)

```java
public class BraunPair {
    public final BraunNode node;
    public final KeyedObject item;

    public BraunPair(BraunNode node, KeyedObject item) {
        this.node = node;
        this.item = item;
    }
}
```
We verify that the method preserves the three properties defining Braun trees.
For the same reasons as above, this method is $O(\log_2 n)$.
Question 6

Write an effective method to replace the smallest object in a tree by an object $s$. Find the complexity of this method.
// in class BraunTree
public void replaceLeastObject(KeyedObject object)
{
    if (this==BraunTree.this.emptyTree) return; // error
    this.rootNode = this.rootNode.replaceLeastObject(object);
}

// in class BraunNode
public BraunNode replaceLeastObject(KeyedObject object)
{
    BraunNode node;
    if (object.getKey()<=this.item.getKey())
        || this.leftTree==BraunTree.this.emptyTree)
        node = new BraunNode(object, this.leftTree, this.rightTree);
    else if (this.rightTree==BraunTree.this.emptyTree
        || this.leftTree.item.getKey()<=this.rightTree.item.getKey()) {
            BraunNode lt = this.leftTree.replaceLeastItem(object);
            node = new BraunNode(this.leftTree.item, lt, this.rightTree);
    } else {
            BraunNode rt = this.rightTree.replaceLeastItem(object);
            node = new BraunNode(this.rightTree.item, this.leftTree, rt);
    }
    return (node);
Question 7

• Write an effective method to merge two trees A and B such that the size A is equal to the size of B, or to the size of B plus 1.
• Assume that these two trees are sub-trees of the same tree and can therefore be modified.
• We can use methods defined above.
• Give the complexity of this method.
Question 7

```java
// in class BraunNode
public BraunNode merge(BraunNode other)
{
    if (this==BraunTree.this.emptyTree) return (BraunTree.this.emptyTree);
    if (other==BraunTree.this.emptyTree) return (this);
    if (this.getKey()<=other.getKey()) {
        BraunNode rt = this.leftBranch.merge(this.rightBranch);
        return ( new BraunNode(this.item, other, rt) );
    }

    BraunPair pair;
    BraunNode lt;
    KeyedObject leastObject;
    pair = this.removeInternalItem();
    leastObject = other.item;
    lt = other.replaceLeastItem(pair.item);
    return ( new BraunNode(leastObject, lt, pair.node) );
}
```
Question 8

• Write an effective method to extract the smallest element of a tree.
• Give the complexity of this method.

```java
// in class BraunTree
public KeyedObject extractLeastItem()
{
    if (this.rootNode==BraunTree.this.emptyTree) return (0);
    KeyedObject leastItem = this.rootNode.item;
    this.rootNode = this.rootNode.leftTree.merge(this.rootNode.rightTree);
    return (leastItem);
}
```
Skew heaps

- substantially the same data structure of binary trees as above.
- a tree is either a leaf <> or a node <T1, a, T2>, a integer, T1 and T2 are the left and right subtrees
- the heap notion (be partially ordered) remains the same.

Algorithms for *inserting an element* and *extraction of the smallest* element of the heap are described by the following recursive expressions:

```plaintext
extract(< T1; a; T2 >) = (a; merge(T1; T2))

insert(a; T) = merge(T; <<>; a;<>>) // <<>; a;<>>: single node

merge(T;<> ) = T
merge(<>; T) = T
merge(T;U) = join(T;U) // if the root T < root U
merge(T;U) = join(U; T) // if the root U < root T

join(< T1; a; T2 >;U) = < T2; a; merge(T1;U) >
```
Question 10

• Insert successively into the empty tree the integers 1, 2, 3, 4. What is the resulting tree? Is it balanced? What happens when it is inserted successively 4, 3, 2, 1?

Intuitive visualization of Skew Heaps at:
http://www.cse.yorku.ca/~aaw/Pourhashemi/AAWLeftistHeap/AAWLeftistHeap.htm