Data Structures and Algorithms

Conf. dr. ing. Guillaume Ducoffe

guillaume.ducoffe@fmi.unibuc.ro

Searching an element

One of the most basic problems in Computer Science.

- In an unordered structure: $\mathcal{O}(n)$ time.
- In an ordered vector: $\mathcal{O}(\log n)$ time.

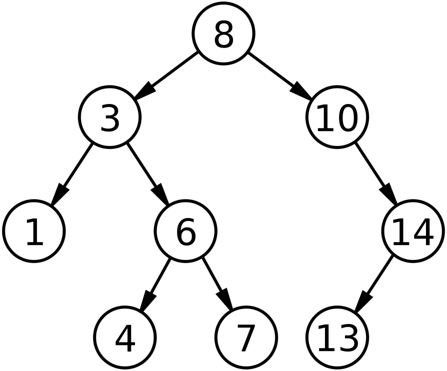
  (binary search)
- In an ordered skip list: expected $\mathcal{O}(\log n)$ time.
- **In a hash table**: expected $\mathcal{O}(1)$ time.

Goal: comparable performances with a dynamic + deterministic data structure.

Binary Research Tree

Each node stores a value i.

- Nodes in the left subtree store values $< i$.
- Nodes in the right subtree store values $> i$.



Remark: we may assume that all values stored are pairwise different (just add a counter for the number of occurrences of each element)

Implementation

Just a binary tree...

```
struct node {
   int value;
   node *father, *left, *right; //sometimes father is omitted
};
typedef node *BinaryResearchTree;
```

- Four standard operations:
  - Emptiness test
  - Search for an element
  - Insertion of an element
  - Deletion of an element

Operations (1/3)

Naive Implementation

```
//Complexity: O(1)
bool empty(const BinaryResearchTree& T) { return (T==nullptr); }

//Can be modified to output any desired information: node pointer, height, etc.
//Complexity: O(height)
bool search(const BinaryResearchTree& T, int e) {
    if(empty(T))
        return 0;
    else if(T->value == e)
        return 1;
    else if(T->value < e)
        return search(T->right,e);
    else
        return search(T->left,e);
}
```

Operations (2/3)

Naive Implementation

```
//Complexity: O(height)
void add(BinaryResearchTree& T, int e) {
    if(empty(T)) {
        T = new node;
        T->father = T->left = T->right = nullptr;
        T->value = e;
    }else if(e < T->value) {
        add(T->left,e); if(!empty(T->left)) T->left->father = T;
    }else if(e > T->value) {
        add(T->right,e); if(!empty(T->right)) T->right->father = T;
    }//nothing to be done if e == value
}
```

Remark: if multiple occurrences allowed, put all equal elements to left/right.

Operations (3/3)

Naive Implementation

```
//Complexity: O(height)
void remove(BinaryResearchTree& T, int e) {
    if(!empty(T)) {
        if(e < T->value) remove(T->left,e);
        else if(e > T->value) remove(T->right,e);
        else { //e == value
            if(empty(T->left) && empty(T->right)) { //emptied tree
                BinaryResearchTree tmp(T);
                T = nullptr; delete tmp;
            } else if(!empty(T->left)) {
                T->value = maximum(T->left);
                remove(T->left, T->value);
            } else {
                T->value = minimum(T->right);
                remove(T->right, T->value);
    }}}
}
```

Remark: min/max is called at most once. Removing the min/max of a subtree can be done in $\mathcal{O}(1)$ because it is a leaf node.

Min/Max element

```
//Complexity: O(height)
int minimum(const BinaryResearchTree& T) {
   if(empty(T->left)) return T->value;
   else return minimum(T->left);
}
```

```
//Complexity: O(height)
int maximum(const BinaryResearchTree& T) {
   if(empty(T->right)) return T->value;
   else return maximum(T->right);
}
```

Generalization: Order statistics

Definition (k^{th} order statistic)

The k^{th} smallest element.

Special cases:

- $k = 1$ (Minimum) and $k = n$ (Maximum)
- $k = n/2$ (if n even): Median

If n odd then we have Lower/Upper Median, and the Median equals their average

- $k = n/4, n/2, 3n/4$ (Quartiles), etc.

Computation

Every node stores in some integer variable the size of its left subtree.

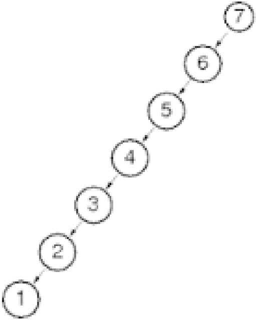
→ To be updated after each insertion/deletion.

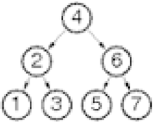
```
//Complexity: O(height)
int stat(const BinaryResearchTree& T, int k){
   if(T->leftSize==k-1) return T->value;
   else if(T->leftSize >= k) return stat(T->left,k);
   else return stat(T->right,k-1-T->leftSize);
}
```

Remark: slight changes needed if multiple occurrences allowed...

The height of a Binary Research Tree

- In the worst-case: $\mathcal{O}(n)$.
- In the best-case: $\mathcal{O}(\log n)$.





Objective: Keep the height to $\mathcal{O}(\log n)$ (**Balanced Tree**).

Balanced Tree: Offline Construction

Key observation: A Binary Tree is balanced if for every node with p descendants:

- $\leq c \cdot p$ nodes are in its left subtree;
- $\leq c \cdot p$ nodes are in its right subtree;

for some constant $c \in [1/2; 1)$.

- Natural choice: $c = 1/2 \Longrightarrow$ Each subtree has for root its median value!

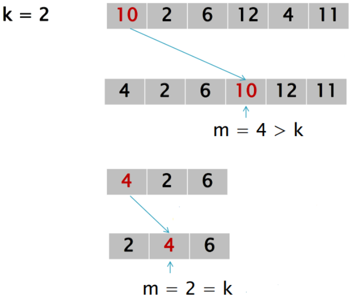
Algorithmic problem: fast computation of the median (or of the cn^{th}-order statistic, for some $c \in [1/2; 1)$)?

Statistic search: Divide & Conquer strategy

Quickselect

1) Select an arbitrary element p (sometimes called **pivot**) and partition in two sub-vectors: one for smaller elements, and one for larger elements.

2) If element p is not the k^{th} order statistic, then recurse on one of the two sub-vectors.



Implementation

```
//Returns the final position of the pivot element
int partition(vector<int>& v, int lb, int ub) {
    int e = v[lb]; //to be discussed...
    int p = ub; //position of the pivot element
    for(int i = ub; i > lb; i--)
        if(v[i] >= v[lb]) {
            swap(v,i,p--); //new larger element found
        }
    swap(v,lb,p);
    return p;
}


int quickselect(vector<int>& v, int k, int lb, int ub) {
    int p = partition(v,lb,ub);
    if(p==k-1) return v[p];
    else if(p >= k) return quickselect(v,k,lb,p-1);
    else return quickselect(v,k,p+1,ub);
}
```

Complexity

- Fix the n elements of the vector and consider a random permutation.

$\rightarrow$ Every element is put in position 0 with same probability $\frac{1}{n}$. In particular, the left sub-vector has length k with probability $\frac{1}{n}$.

$\rightarrow$ The Average complexity satisfies the following inequation:

$$T(n) \leq n - 1 + \sum_{k=1}^{n-1} \frac{1}{n} \times \max\{T(k), T(n-1-k)\}$$

By induction: $T(n) \leq 4 \cdot n$.

Alternative strategy: choose a **random pivot**.
Expected complexity in $\mathcal{O}(n)$.

Median of medians

Key observation: To ensure Linear-time **Worst-case** complexity, it would suffice to choose as pivot a cn^{th} order statistic, for some $0 < c < 1$.

$$T(n) \leq n + T(cn) \leq n + cn + T(c^2n) \leq \ldots \leq n + cn + c^2n + \ldots + c^in + \ldots$$
$$= n \times \sum_i c^i = \mathcal{O}(n)$$

- We subdivide the vector in sub-vectors of size 5 and we keep the median of each of them.

`[1,4,`**`5`**`,22,9,13,`**`67`**`,91,0,15,33,`**`50`**`,16,12,87,19,`**`14`**`]` $\Longrightarrow$ `[5,67,50,14]`

The new vector has size $\leq n/5$ (*Sufficiently small for recursive calls*). Its median is the cn^{th} order statistic of the original vector, for some $c \in [3/10; 7/10]$.

Sketch Implementation

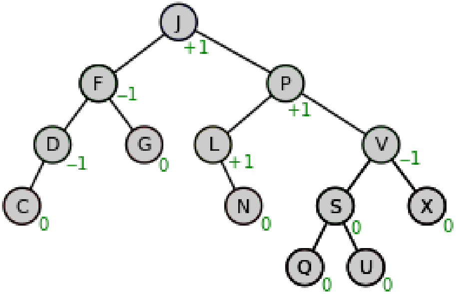
```
int quickselect(vector<int>& v, int k, int lb, int ub) {
   vector<int> med;
   for(int i = lb; i <= ub; i+=5) {
      med.push_back(quickselect(v,i+2,i,i+4));
      //border effects for last subvector...
   }
   int p = quickselect(med,med.size()/2,0,med.size()-1);
   //Find p in the vector, then use it as pivot
...
}
```

Complexity: $T(n) \leq n - 1 + T(n/5) + T(7n/10) = \mathcal{O}(n)$.

Online construction: AVL trees

Adelson-Velsky and Landis (1962).

- Every node keeps track of the order of its height
- After each operation, we modify these subtrees so that $|left.height - right.height| \leq 1$.



Every AVL of height h contains $\geq F(h)$ nodes (by induction). $\Longrightarrow$ Balanced

AVL trees
Implementation

```
struct node {
   int value;
   node *father, *left, *right;
   int height;
};
typedef node *AVL;
```

In practice. we may only store `left.height - right.height` (only 2 bits needed). However, various complications would arise after each insertion/deletion.

Actualizing the height value

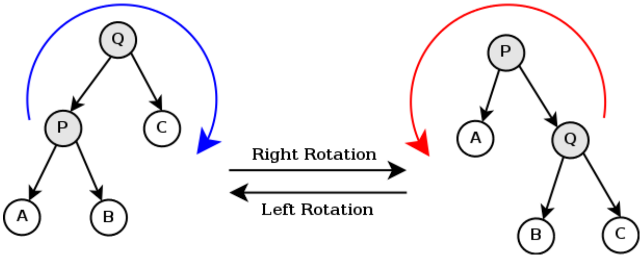
Just to have it somewhere for future reference ...

```
void update(AVL& T) {
   if(!empty(T)) {
      if(empty(T->left) && empty(T->right))
         T->height = 0;
      else if(empty(T->left))
         T->height = 1 + T->right->height;
      else if(empty(T->right))
         T->height = 1 + T->left->height;
      else if(T->left->height >= T->right->height)
         T->height = 1 + T->left->height;
      else T->height = 1 + T->right->height;
   }
}
```

Tree rotation

The left/right child becomes the new root.

Left/right grandchildren need to be redistributed. $\Longrightarrow$ We need to use the pointer to parent



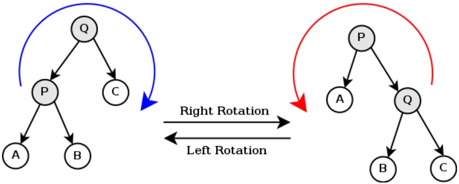
Tree rotation

Motivation

Consider the following scenario:

- subtree(P) and subtree(C) are balanced
- P.height = C.height + 2 (subtree(Q) is not balanced)
- A.height = P.height -1 (highest subtree) = C.height +1

After a right rotation, the tree becomes balanced!



Tree rotation

Implementation

```
void rotateRight(AVL& T) { //Assumption: left is nonempty
    AVL newRoot = T->left; newRoot->father = T->father;
    if(!empty(T->father)){
        if(T->father->left == T) T->father->left = newRoot;
        else T->father->right = newRoot; }
    T->left = T->left->right; if(!empty(T->left)) T->left->father = T;
    newRoot->right = T; T->father = newRoot;
    update(T); update(newRoot);
    T = newRoot; }

void rotateLeft(AVL& T) { //Assumption: right is nonempty
    AVL newRoot = T->right; newRoot->father = T->father;
    if(!empty(T->father)){
        if(T->father->left == T) T->father->left = newRoot;
        else T->father->right = newRoot;}
    T->right = T->right->left; if(!empty(T->right)) T->right->father = T;
    newRoot->left = T; T->father = newRoot;
    update(T); update(newRoot);
    T = newRoot; }
```

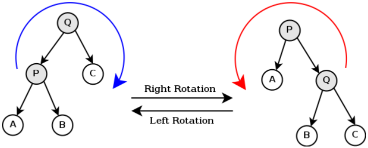
Adding a value

Suppose we add a new element **to the left**

$\Longrightarrow$ We may have left.height = right.height + 2 (Not balanced)

- Case 1: the left-left subtree is higher than the left-right subtree

  $\longrightarrow$we only need one rotation



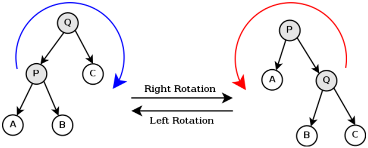
Adding a value

Suppose we add a new element **to the left**

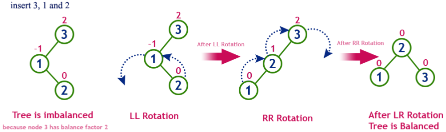
$\Longrightarrow$ We may have left.height = right.height + 2 (Not balanced)

- Case 1: the left-left subtree is higher than the left-right subtree

$\longrightarrow$we only need one rotation



- Case 2: Left rotation (make left-left subtree heavier + AVL) then Right rotation (as for Case 1)



Adding a value

Implementation

```
void add(AVL& T, int e) {
    if(empty(T)) { /*already discussed*/ }
    else if(e == T->value) return;
    else if(e < T->value) {
        add(T->left,e);
        if(!empty(T->left)) {
            T->left->father = T;
            int r = (empty(T->right)) ?  0 :  T->right->height;
            if(T->left->height == r+2) {
                if(!empty(T->left->left) && T->left->left->height == r+1) //Case 1
                    rotateRight(T);
                else { //Case 2
                    rotateLeft(T->left); rotateRight(T);
        }}}
    } else { /*add to the right*/ }
    update(T);
}
```

Removing a value

- Case we remove an element **to the right**
  $\Longrightarrow$ We may have left.height = right.height + 2 (Not balanced)

Proceed as before...

- Case we remove an element **to the left**: Same as above...

Removing a value

- Case we remove an element **to the right**
  $\Longrightarrow$ We may have left.height = right.height + 2 (Not balanced)

Proceed as before...

- Case we remove an element **to the left**: Same as above...

- Case we remove the root. The **new root** can be:
  - either the maximum to the left
  - or the minimum to the right

$\Longrightarrow$ We choose the element in the highest subtree.

Removing a value

```
void remove(AVL& T, int e) {
    if(!empty(T)) {
        if(e < T->value) {
            remove(T->left,e);
            //Do as for insertion
        } else if(e > T->value) {
            remove(T->right,e);
            //Do as for insertion
        } else { //e == value
            if(T->height == 0) { /* emptied tree: proceed as before*/ }
            else if(!empty(T->left) && T->left->height = T->height - 1) {
                T->value = maximum(T->left); remove(T->left,T->value);
            } else {
                T->value = minimum(T->right); remove(T->right,T->value);
            }
        }
        update(T);
    }
}
```

Beyond Binary Research Trees

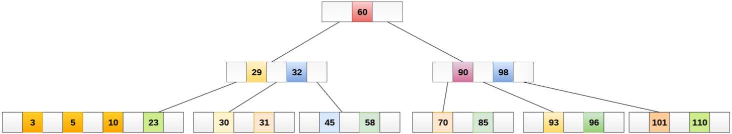
- Research Trees of larger Arity
  - B-trees
  - $2-3-4$ trees

- Research Trees for multi-dimensional points
  - Interval trees
  - k-range trees

Storing more values per node: B-trees

Standard Data Structure for indexing in Data Bases and File Systems.

- Two parameters L, U (in general, $U = 2L$):
  - Each internal node contains $\geq L-1$ values;
  - Each node contains $\leq U-1$ values.
- If the values stored in an internal node are $a_1, \ldots, a_k,\ L-1 \leq k \leq U-1$ then there are exactly $k+1$ subtrees:
  - Nodes with values $\leq a_1$
  - Nodes with values $> a_{i-1}$ and $\leq a_i,\ \forall 2 \leq i \leq k$
  - Nodes with values $> a_k$



Implementation

```
struct Bnode {
   vector<int> values;
   vector<Bnode*> children;
   Bnode *father;
};
typedef Bnode *Btree;
```

Remark 1: If `values` is nonempty, then `children.size() == values.size() +1`

Remark 2: `values.size()` lies between $L - 1$ and $U - 1$ (varying node sizes)

Remark 3: `values` is sorted so as to find efficiently (using binary search) the child node where to continue the search.

Path-balanced property

A B-tree must preserve the following invariant:

Definition

All leaves must stay at the same level (=distance to the root).

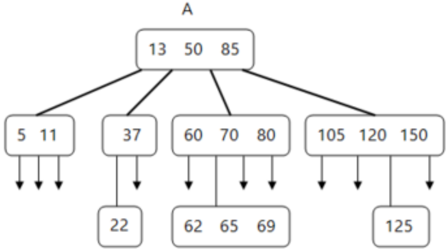
$\Longrightarrow$ The number of nodes grows by a factor $\geq L - 1$ at each step, and therefore there are at most $\mathcal{O}(\log n / \log L)$ levels.

Remark: path-balanced cannot be checked locally. We need operations that always preserve this property (*i.e.*, we cannot correct the tree if it becomes unbalanced after each operation).

$2-3-4$ trees

- Special case of B-trees for $L = 2, U = 4$.
- Can be implemented with Binary Research Trees!

Figure 1: Multiway Search Trees



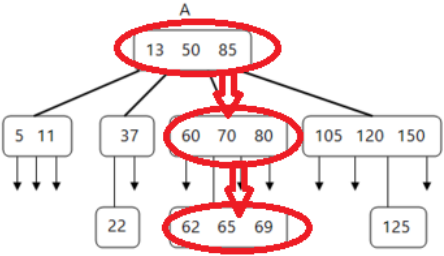
Insertion of a value (1/3)

- Search for the element in the current tree

  (we may assume that we did not find it. . .)

<u>Example</u> for $e = 67$

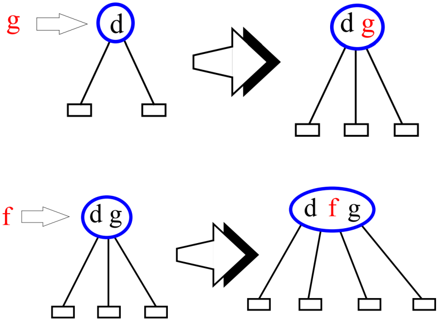
Figure 1: Multiway Search Trees



- Consider the last node on the search path (it is a leaf).

Insertion of a value (2/3)

- **Try to** insert the new element in the leaf node

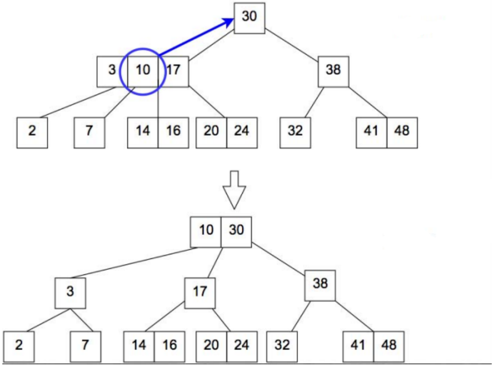


- Only possible if the node is not already full (4-node)

Insertion of a value (3/3)

Solution: Split all the 4-nodes on the search path!

The middle element is now stored in the parent node



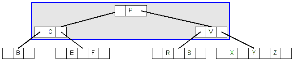
Remark: preserves the path-balanced property...

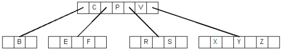
Deletion of a value

1) Locate the value to be removed in some node `N`.

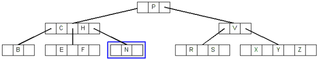
2) Ensure **recursively** that `N` contains at least two values.

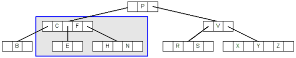
- If the root contains one value, merge it with its two children.



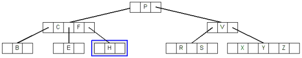


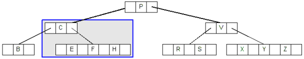
- If the next node has one value *but* a closest sibling with > 1 values then make a transfer.





- Else, merge the node with a sibling and transfer one value from the parent.



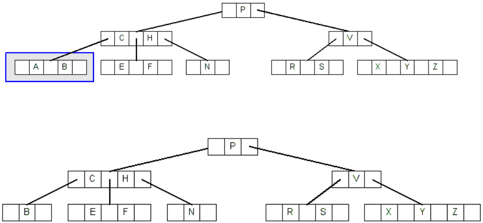


Deletion of a value

Case of a Leaf Node

Our pre-processing ensures that `N` contains at least one more element than the one to be removed.

$\Longrightarrow$ Just remove the element



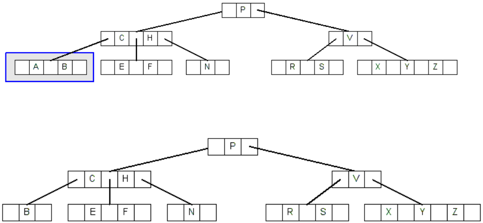
- What was the problem if there were exactly one value?

Deletion of a value

Case of a Leaf Node

Our pre-processing ensures that `N` contains at least one more element than the one to be removed.

$\Longrightarrow$ Just remove the element



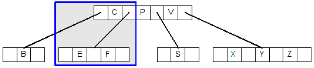
- What was the problem if there were exactly one value?

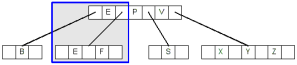
$\longrightarrow$ path-balanced property

Deletion of a value

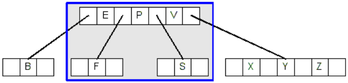
Case of an Internal Node

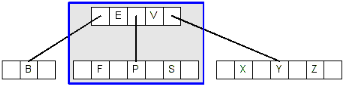
- Subcase # 1: One child has $>$ 1 values
  $\Longrightarrow$ Replace the deleted value with its predecessor/successor.





- Subcase # 2: Both children have 1 value
  $\Longrightarrow$ Merge the two children with the value to be deleted.





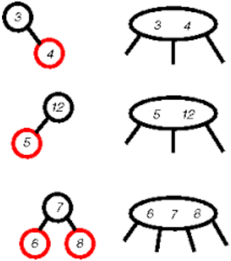
Proceed **recursively**!

Encoding: Red-Black Trees

A $2-3-4$-tree can be encoded as a **binary research tree**, with one colour (red/black) being assigned to each node.

Encoding of a Bnode:

- The middle value of a node is coloured black
- The (at most) two other values are coloured red.

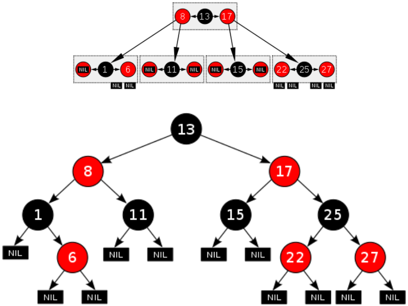


Application: simulate all operations on a 2-3-4-tree by operations on Binary Research Trees (mostly, left/right rotations).

Properties of Red-Black Trees

- Every node is either red or black
- The root is black
- Any child of a red node is black.
- There is the same number of black nodes on any root-leaf path

"*all leaves of a* $2-3-4$ *tree are at the same level*"



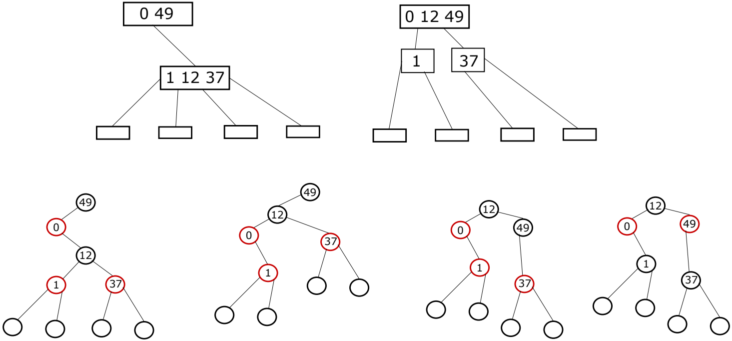
Implementation

```
struct RedBlackNode {
   int value;
   RedBlackNode *father, *left, *right;
   bool color; //false for black
};

typedef RedBlackNode *RedBlackTree;
```

Splitting a 4-node

- Recolour black the two children nodes
- $+$ At most two rotations
- $+$ At most $\mathcal{O}(1)$ recolouring.



Splitting a 4-node

```
void checkAndSplit(RedBlackTree& T) {

    if( (!empty(T->left)&& T->left->color)
     && (!empty(T->right)&& T->right->color) ) {
        //full node

        T->left->color = T->right->color = 0; //black

        if(!empty(T->father)) { //not the root

            if(!T->father->color) { //black node
                T->color = 1;

            } else {
                if(T->father->left == T) {

                    rotateRight(T->father);
                    T->right->color = 1;

                    //new rotation
                    if(T->father->left == T) {
                        rotateRight(T->father); T->right->color=1;
                    } else {
                        rotateLeft(T->father); T->left->color=1;
                    }
                } else { /*right child*/ }
}}}
```

More dimensions: **Interval tree**

<u>Def.</u>: an interval = an ordered pair of two values.

An interval tree maintains a dynamic collection of intervals, while supporting the following operations:

- Insertion/Deletion/Membership queries.

  in $\mathcal{O}(\log n)$ time.

- Output an interval that contains a given point x.

  in $\mathcal{O}(\log n)$ time.

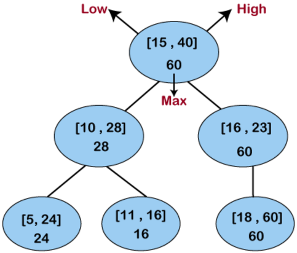
- Output an interval that intersects a given interval $[a; b]$

  in $\mathcal{O}(\log n)$ time.

Implementation: Augmented Tree

- We put all intervals in some **balanced** binary reseach tree (*e.g.*, AVL)

$\rightarrow$ Use lexicographic ordering



- Insertion/Deletion/Membership queries automatically in $\mathcal{O}(\log n)$ time.
- Additional storage of the largest upper values amongst all intervals in the subtree of a node.

Remark: other implementations exist with comparable performances.

Intersection with a point

Input: a point x

Output: any interval containing x.

- If `x >= root->low` and `x <= root->high` then output `root`.
- Else if `x < root->low`, then we recurse on `root->left`.
- Else, `x > root->high`.
  - If `!empty(root->left)` and `x <= root->left->max`, then there is an interval on the left containing x. We recurse on `root->left`.
  - Else, we recurse on `root->right`.

Complexity: $\mathcal{O}(\log n)$.

Intersection with an interval

Input: an interval $[x; y]$

Output: any interval intersecting $[x; y]$.

- If `x <= root->high` and `root->low <= y` then output `root`.
- Else if `root->low > y`, then we recurse on `root->left`.
- Else, `x > root->high`.
  - If `!empty(root->left)` and `x <= root->left->max`, then there is an interval on the left containing x. We recurse on `root->left`.
  - Else, we recurse on `root->right`.

Complexity: $\mathcal{O}(\log n)$

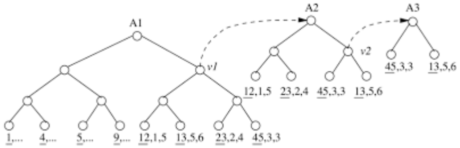
Range Trees

Input: a static collection of k-dimensional points.

Recursive construction:

- An 1-range tree is a balanced binary research tree.
- A k-range tree, $k > 1$ is a balanced binary research tree on the first coordinate of each point.

  Each node stores a $(k-1)$-dimensional range tree, for all remaining coordinates of all points in its rooted subtree.



Construction

- Find a point whose first coordinate is a median. – $\mathcal{O}(n)$.
- Construct k-range trees for left/right. – $2 \times C(n/2, k)$.
- Construct a $(k-1)$-range tree over the remaining coordinates and store it at the root. – $C(n, k-1)$.

Complexity:

$$C(n,k) = C(n,k-1) + 2 \times C(n/2,k) = \mathcal{O}(C(n,k-1)\log n) = \mathcal{O}(n \log^k n)$$

Queries

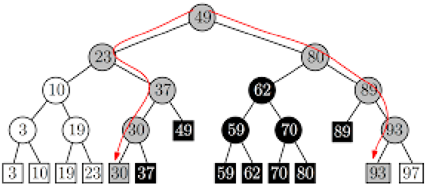
"Multi-dimensional range queries"

Input: a box = lower/upper bounds for all k coordinates

Output: enumerate all points in the box.

Variant: assign values to each point and output the max/min point, the sum of all values, etc.

A 1-dimensional range query with [25, 90]



Answer to a query (Sketch)

1) Consider the upper/lower bounds $[a_1, b_1]$ for the first coordinate.

2) In the binary research tree for the first coordinate, find the smallest/largest values x, y in the interval. Let $z = lca(x, y)$.

Remark: all searched points must be in the subtree rooted at z (otherwise, x or y could not be the smallest or largest value in $[a_1, b_1]$).

3) Consider the xz-path. For each edge uv on this path, if v is the left child of u, then all the right subtree of u contains points whose first coordinate lies between a_1 and b_1.

4) Answer to queries on the remaining coordinates for the $(k-1)$-range trees on the paths between x, z and y, z.

Questions