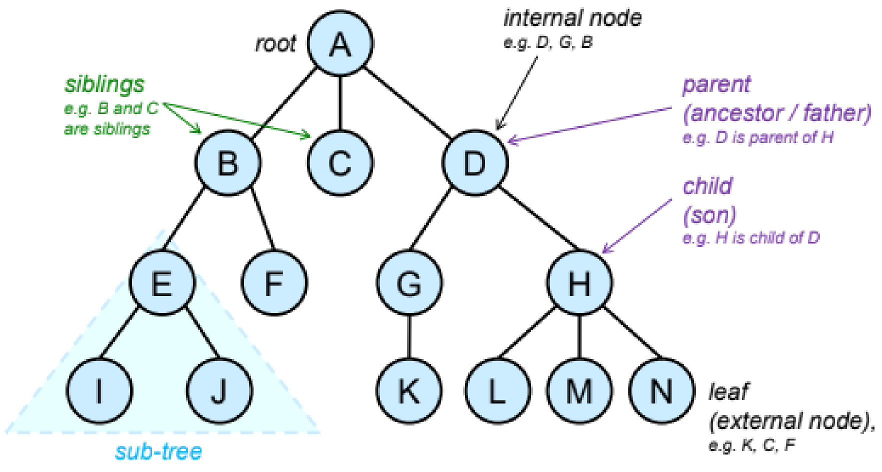
Binary trees

What is a binary tree?

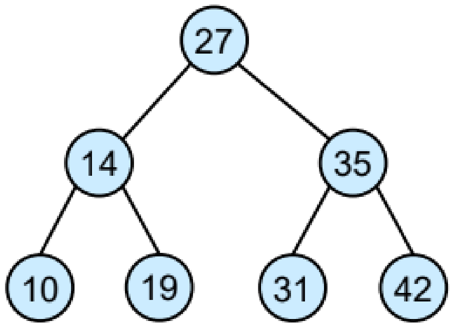
A binary tree, contrary to real trees, grows upside down. The top of the tree is the root. Under the root, you have child nodes. These child nodes further have children. Node D in the following picture is a parent of nodes G and H. At the same time, it is a sibling of nodes B and C. Nodes in the last row are called leaf nodes. If binary tree has height h, maximum number of nodes will be when all levels are completely full. Total number of nodes will be $2^{h+1} - 1$.



How and when is a binary tree used?

- Binary search
- Represent search array as binary tree
- Fast addition and removal of data

Tree walks



- Inorder: [10, 14, 19, 27, 31, 35, 42]
- Preorder: [27, 14, 10, 19, 35, 31, 42]
- Postorder: [10, 19, 14, 31, 42, 35, 27]
- Levelorder: [27, 14, 35, 10, 19, 31, 42

```
// We first go to the left -> middle -> right
void InorderTreeWalk(struct node* p){
    if (p != NULL){
        InorderTreeWalk(p->left);
        VisitNode(p);
        InorderTreeWalk(p->right);
    }
}

void VisitNode(struct node* p){
    printf("%d ", p->value);
}
```

Insertion

```
struct node* TreeInsert(
struct node* p, struct node* r)
{
    struct node* y = NULL;
    struct node* x = r;
    while (x != NULL) {
        y := x;
        if (x->key < p->key) x = x->rgt
        else x = x->lft;
    }
    if (y == NULL) r = p;
    else if (y->key < p->key) y->rgt = p;
    else y->lft = p;
    return r;
}
```

To insert an element into a binary search tree, we first need to know if it's empty or not. With an empty tree, it is easy. We add the first number to the root. In a binary search tree, we have a special rule. This asks us to keep all elements which are smaller than a node, should be on the left side of the node. Everthing that is greater than root should go to the right of the binary search tree. If we already have elements in our BST, we find the correct place for the value and then add it there.

Searching

The task of searching in BST is also easier due to the property of the elements. While searching, if the found element is greater than what is searched, the algorithm should move further left. And on the other hand, when the found element is smaller than what is searched, move further right.

```
void TreeSearchIterative(struct node* p,int value){
    while(t != NULL && p->key != NULL){
        if(v < p->key){
            p = p->left;
        } else {
            p = p->right;
        }
    }
    return p;
}

void TreeSearchRecurisve(struct node* p, int value){
    if(p == NULL || p->key == value){
        return p;
    }

    if(v < p->key){
        return TreeSearchRecursive(p->left, value);
    } else {
        return TreeSearchRecursive(p>right, value);
    }
}
```

Deletion

- Deletion case 1: Node t has no children -> remove t
- Deletion case 2: Node t has one child x -> let parent of t point to x and then remove t
- Deletion case 3: Node t has two children -> find largest (smallest) child s in left (right) subtree of t, then replace $p \to key$ with $s.key$ and remove s (often this is done by changing pointers (makes it independent of node content))

```
/* Node x is a pointer to the node to be deleted */
struct node* delete(struct node* root, struct node* x) {
    u = root;
    v = NULL;

    //we search x and the node above
    while (u != x) {
        v = u;
        if (x->key < u->key) u = u->lft;
        else u = u->rgt;
    }

    if (u->rgt == NULL) {
        if (v == NULL) root = u->lft;
        else if (v->lft == u) v->lft = u->lft; else v->rgt = u->lft;
    } else if (u->lft == NULL) {
        if (v == NULL) root = u->rgt;
        else if (v->lft == u) v->lft = u->rgt;
        else v->rgt = u->rgt;
    } else {
        p = x->lft;
        q = p;
        while (p->rgt != NULL) {
            q = p;
            p = p->rgt;
        }

        if (v == NULL) root = p;
        else if (v->lft == u) v->lft = p;
        else v->rgt = p;

        p->rgt = u->rgt;
        if (q != p) {
            q->rgt = p->lft;
            p->lft = u->lft;
        }
    }
    return root
```

Printing

Maximum and minimum

Finding the minimum

- Finds the minimum key in a tree rooted at p
- Running time is proportional to height of tree
- We do not have to compare keys.

```
void TreeMin(struct node* p){
    while(p->left != NULL){
        p = p->left;
    }
    return p;
}
```

Same principle for maximum, just on the other side:

```
void TreeMin(struct node* p){
    while(p->right != NULL){
        p = p->right;
    }
    return p;
}
```

Distance to root

```
int heightBT(struct node* p){
    if (p === NULL) return -1;
    return max(heightBT(p->left), heightBT(p->right)) + 1;
}
```

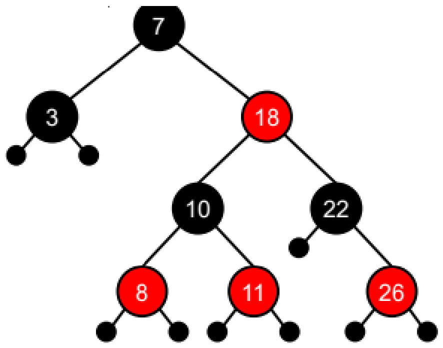
Red-black trees

What is a red-black tree?

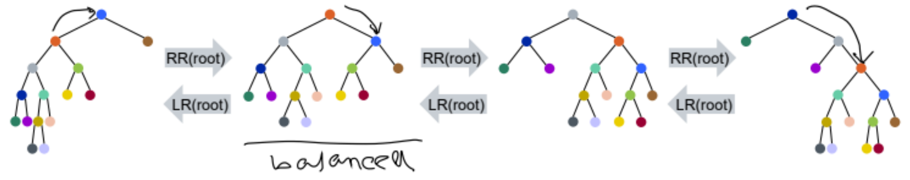
Following conditions need to fulfilled for a valid red-black tree:

- Every node is either red or black (needs extra bit to store this information)
- All leaves are black and have no values, so called nil pointers
- Root is always black
- A red node has black children, there can not be two consecutive red nodes
- Depth of black pointers has to be same over all paths of the tree
- The tree needs to be a valid BST

This is an example of a valid red-black tree

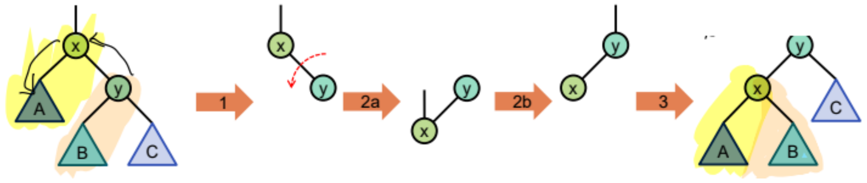


Balancing a red-black tree

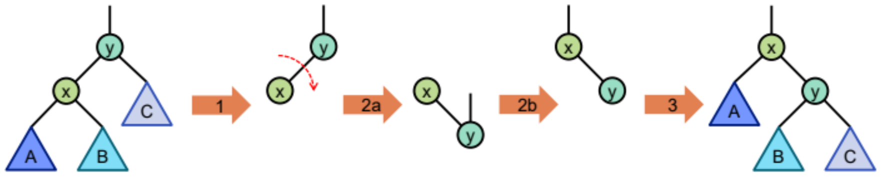


When balancing a red-black tree, we have 2 options. Either a left-rotation or a right-rotation.

Following condition must hold: nodes in A ≤ x ≤ nodes in B ≤ y ≤ nodes in C (see BST condition)



What we do in left-rotation is as first step, identify our elements x and y. We also identify subtrees A, B and C. Then we rotate$x$ and y, send y to the root and add the B subtree to our x.



What we do in right-rotation is as first step, identify our elements x and y. We also identify subtrees A, B and C. Then we rotate$y$ and x, send x to the root and add the B subtree to our y.

Searching

Searching in a red-black tree is exactly the same as in an elementary binary search tree (just ignore colour). Read-only operations are also identical (but they benefit from the fact that red-black trees are much better balanced).

Insertion

Case 0: Black parent If the parent of the new node is Black and the tree was a valid RB-tree before insertion, no condition can be violated and there is nothing to do. This is a terminal case.

Case 1: Red uncle

1. Recolor Parent and Uncle in Black.
2. Recolor the Grandparent in Red.
3. If the Grandparent's Parent is Red, there could be another violation of the red property. Therefore, propagate upwards in the tree (while treating the Grandparent as the newly inserted node, possibly until reaching the root)(in the end change root to black if necessary).
4. Check all cases again.

Case 2: Black uncle line Red property violated and the Uncle of the new node is Black; Grandparent, Parent and new node form a Triangle.

1. Apply a Rotation on the Parent of the new node. The rotation should get Grandparent, Parent and new node into a Line configuration

- Left Rotation is applied when the Triangle points to the Left.
- Right Rotation is applied when the Triangle points to the Right.

2. Consider the former Parent as the newly inserted node.(Do not swap them, but just shift the status of "new" to the former Parent.)
3. Continue with Case 3

Case 3: Black uncle triangle Red property violated and the Uncle of the new node is Black ; Grandparent, Parent and new node form a Straight Line.

1. Recolor the Parent Black and recolor the Grandparent Red.
2. Apply a Rotation on the Grandparent of the new node towards the opposite site the tree is currently leaning.

- Left Rotation is applied when the new node is a Right child.(MIRROR)
- Right Rotation is applied if the new node is a Left child.

This is a terminal case.

Deletion

Before deletion find node to be deleted and copy to it the value from either the in order predecessor or successor. This successor or predecessor is now the new node to be deleted. If node to do be deleted is a leaf node. Don't worry, consider it the node to be deleted.

REMEMBER! The standard case assumes the node to be deleted is a left child. If it is a right child, consider it a mirror case.

Case 0: Red Node or Red Child In this case if the node to be deleted is red and has only null children, simply remove it. If it has a single child that is red, then recolour the child black and delete the node to be deleted. This is terminal case.

Case 1: Red Brother In this case the depth property is violated and the node to be deleted's brother is red. You must recolour the brother black and parent red. Then apply left rotation on parent. This is not a teminal case and you must continue to case 2, 3 or 4.

Mirror case 1 This applies if the node to be deleted is a right child. In this case perform the same recolouring and apply a right rotation on parent. This is not a teminal case and you must continue to case 2, 3 or 4.

Case 2: All Black In this case the depth property is violated and the brother of the node to be deleted is black and has two black children. If parent also black: Make brother red and CONSIDER the parent as the new node to be deleted (i.e. search for cases again from the top!)

Note: If new (considered) node to be deleted is the root and black, you are finished. If parent red: Exchange colours between parent and brother. This is a terminal case, you are done.

Case 3: Black Brother, only! left nephew red In this case the node to be deleted has a black brother whose left child is red. You must recolour the left nephew black and the brother red. Then apply a right rotation to the brother. You must now proceed with case 4.

Mirror case 3 If the node to be deleted is a right child this case is applicable when the brother is black and it's right child is red. You follow the same recolouring scheme but apply a left rotation to the brother instead. You must now proceed with case 4.

Case 4: Black Brother, right or both nephews red In this case the node to be deleted has a black brother which has atleast a red right child. (The other child or parent's colour is not important). You must recolour the brother in the colour of the parent and recolour the parent and right nephew black. Then perform a left rotation on the parent. Case 4 is a terminal case.

Mirror case 4 If the node to be delected is a right child this case is applicable when the brother is black and at least the left nephew is red. The recolouring procedure remains the same as in the standard case but instead we perfrom a right rotation on the parent. Case 4 is a terminal case.