Dynamic indexes

We are searching for a data structure allowing us to: adding new data, searching data, removing data. Idea is to build something called **index**, to manage efficiently those operations.

Binary Search Tree

Data structure meant to provide an index. They are connected acyclic graphs. Consider a node x, we denote by *x.left*, *x.right* its children, *x.parent* its parent, *x.key* the value stored in x (its key). We use *=NIL* to check if some this conditions is empty.

Useful $\Theta(1)$ functions are: IS_ROOT, IS_RIGHT_CHILD (CHILDHOOD_SIDE), SIBLING, GET_CHILD (according to the side we are searching for), SET_CHILD (in a given side), UNCLE, GRANDPARENT, NEW_NODE.

A **binary search tree** is a tree s.t.: * all the key belongs to a totally ordered set wrt $\preceq$; * if x_l is in the left subtree of x, then $x_l.key \preceq x.key$; * if x_r is in the left subtree of x, then $x.key \preceq x_r.key$.

If a node is a leaf, hence it has no children, it has to point to NIL as children (actually leaf are NIL, useful to generalize).

A *inorder visit* for a tree is a procedure which goes through all the nodes in a specific order (first all left subtrees, then the node itself, then right subtree).

Searching for the minimum/maximum, thanks to the BST property is easy:

- the minimum key is in the first node on the leftmost branch that has no left child;
- the maximum key is contained by the first node on the rightmost branch which hasn't any right child.

The complexity of those functions is $O(h_T)$, indeed we are browsing a branch of the BST, hence the worst case scenario is that we go through the longest branch. However at the moment we don't know the height of the tree.

We want to find the *successor of a node* wrt $\preceq$ (node whose key is the next one wrt the total order), due to the BST property we have:

- we can search the minimum among the values of the right subtree (if any);
- if there isn't any right subtree we need to find the nearest right ancestor;

The complexity of this operation is $O(h_T)$.

In order to *search* for a value inside the key, thanks to BST property, we apply a dichotomic approach from the root, complexity is $O(h_T)$.

In order to *insert* a new value is the BST, we need to browse a branch of the tree and insert a new leaf with the value (if not already present), time complexity is

$O(h_T)$ (inserting a node once find the position, takes constant time $\Theta(1)$).

In order to *remove* a value, first of all we need to search for that key ($O(h_T)$), then we have two possible scenarios:

- node with at most one child: delete and replace by its child (if any), by using the procedure SET_CHILD on the parent of the node we want to delete ($\Theta(1)$);
- node with two children: find the minimum among all the node in the right subtree (which will be the successor, and which by definition of BST will have at most one child, right child), assign the key value to the node we want to remove, and replace the removed node with its child (if any).

The complexity of removing a node is $O(h_T)$.

The problem is that the height of a BST h_T can be equal to the number n of nodes (kind of chain, obtained inserting value in order: always the maximum or always the minimum). In that case a BST is worst than a single-linked list, which has a constant insertion cost $\Theta(1)$.

We need to **balance** the BST, so that the height of a BST with n nodes is $O(log(n))$, indeed the minimum is $\lceil \log_2(n) \rceil$.

We can try to balance the tree *globally*, namely after every insertion/deletion. This can be than collecting all the elements in an array and then re-distribute them in a dichotomic way. The asymptotic complexity of this would be $O(n)$.

Or we can try to balance the tree *locally*, we try to fix single unbalanced parts. This is not so easy anyway.

A possible solution is to use a more advanced data structure.

Red-Black Tree

Data structure used to efficiently handle the balancedness of the tree. A **red-black tree** is a BST in which each node is either black or red, this property is used to check if a subtree is balanced or not.

The conditions that a Red-Black tree has to satisfy are the following:

- the root must be black;
- all the leaves are black NIL nodes;
- all the red nodes must have black children;
- for each node x in the tree, all the branches from x will contain exactly the same number of black nodes (balancedness with respect to black nodes).

The *black height* BH(x) is the number of black nodes below x in every branch.

We need to enlarge the data structure BST in order to store the colour, *x.color* will return the colour of the node x, then we can define a function COLOUR of a node ($\Theta(1)$).

BST may have has height the number of elements stored in the BST itself, for the red-black tree the maximum height is:

Theorem (Heights of a RB-Tree): Any RBT with n internal nodes has height at most $2\log_2(n+1)$.

proof: by induction we can prove that every subtree rooted in x has at least $2^{BH(x)}-1$ internal nodes. By induction on the height H (not black height) of x:

- base case: $H(x) = 0 \to BH(x) = 0 = 2^{BH(x)} - 1 = 1 - 1$;
- inductive step: assume that $\forall y \;\; s.t. \;\; H(y) < H(x)$ the condition holds. For sure x has $H(x) > 0$, take the subtrees rooted in $x.left$ and $x.right$ (at least one of them exists), and $H(x) = \max\{H(x.left), H(x.right)\}$. By inductive assumption the subtree rooted in $x.left$ and $x.right$ have at most $2^{BH(x.left)} - 1$ and $2^{BH(x.right)} - 1$ internal nodes respectively. So the subtree rooted in x has at most $2^{BH(x.left)} - 1 + 2^{BH(x.right)} - 1 + 1$ internal nodes. Moreover, since we are dealing with a red-black tree, the number of black nodes must be the same in both left and right branch of x (all the paths from x to the leaves must have the same number of black nodes, namely $BH(x.left) = BH(x.right) \leq BH(x)$ depending on the colour of x). If x is black, then $BH(x)$ is equal to $BH(x.left)$ and $BH(x.right)$, if x is red it is greater ($x.left$ and $x.right$ can be themselves black). So the number of internal nodes of the subtree rooted in x is at least $2^{BH(x.right)} + 2^{BH(x.left)} - 1 \leq 2^{BH(x)} - 1$.

We can also prove that $BH(x) \geq H(x)$ for any Moreover $BH(x)$ is at least half of x's height h, so that $n \geq 2^{\frac{h}{2}} - 1$.

This is a nice property, since the complexity of most of the operations in BST have complexity $O(h_T)$, so now this complexity is logarithmic.

Rotation on the tree preserving the BST property (can be applied to every BST): fix a node and rotate the subtree rooted in that node. We apply a right or left rotation: need to reverse the side ($\Theta(1)$), get the child of x ($\Theta(1)$), replace children with the appropriate ones, with TRANSPLANT ($\Theta(1)$) and SET_CHILDREN ($\Theta(1)$), hence rotation takes constant time $\Theta(1)$. If we are dealing with red-black trees, we need to fix the red-black tree property.

Consider **inserting** a new node in a red-black tree (not allowing multiple insertions): we can insert as in a BST (browsing the tree and finding the right position), and red-colouring the newly inserted node (so that the black height has not changed). It may be the case that the parent of the node we are inserting is already a red node, so we are breaking the rule of red-black tree (every red node must have black children). We can look at the uncle y of the new node x:

- Case 1: if the uncle is red (so that the grandparent of x is for sure black): we can black-colour the parent and the uncle and red-colour the grandparent. However we may have pushed up the problem (we repeat this procedure in the necessary subtree);
- Case 2: if the uncle is black the idea is to apply rotation;
- Case 3: if y (with x and the parent of x both red) is black and y and x are on different sides (invert x parent and grandparent colour) and rotate x's grandparent on y's side.

Summarizing:

1. If the parent of the new node (x) is black, nothing to be done (black height is not changed and every red node has black children);
2. Otherwise we need to look at the uncle y of x: we can remove the problem in constant time following the procedure above (eventually pushing the problem toward the root and repeating the fixing procedure).

In the worst case, the algorithm keeps repeating Case 1 steps along the insertion branch, hence the complexity is $O(\log(n))$.

Consider **deleting** a key from a red-black tree: we can proceed as in the BST. First we need to find the node containing the key, if it has at most one child, we remove the node and replace it by its child, if any. If it has both children, need to find the successor, replace the node with its successor and delete the successor. We need to update this procedure to deal with the colour:

- If the node we want to remove is red, everything is fine, indeed we don't destroy the property for which every red node must have black children and we don't change the black height as well.
- If the node we want to remove is black, we are decreasing the black length of that branch by 1. So that $BH(x) = BH(w) - 1$, with x child of the node we want to remove, w is x's sibling. So we have to siblings with different black-height, need to fix this.

0. Case 0: x is red, we black-colour x.
1. Case 1: x is black and x's sibling w is red. Invert colours in x's parent and sibling and rotate x's parent on x's side so that $BH(x)$ does not change;
2. Case 2: x's sibling and nephews are black. We red-colour x's sibling. We changed $BH(x.parent)$ and $BH(x.sibling)$ decreasing by one, we may have pushed forward the problem toward the root.
3. Case 3: only nephew on x's side is red. We can rotate x's sibling on the opposite side wrt x and invert colour in both old and new siblings of x. In this way both $BH(x)$ and $BH(x.parent)$ has not changed.
4. Case 4: x's nephew in the opposite side wrt x is red. Switch the colour of x's parent and sibling, black-colour the x's nephew on the opposite side

wrt x, rotate x's parent on x's side. In this way the black height does not change (this case covers also the situation in which all the nephew are red).

All these cases take constant time $\Theta(1)$. In the worst case scenario, Case 2 is repeated until the problem is pushed up to the root ($O(log(n))$ times), in this case we have decreased the black-height of the tree.