Lecture 11: Binary search tree and red-black tree

Please review Chapter 12 if you forget what binary search tree (BST) is. The main property of BST is that the left descendants of a node must be no bigger than the value at the node, and the right descendants must be no smaller than it. Also, the running time is takes to major operations (e.g. searching for an item) takes $O(h)$ time, where $h$ is the height of the BST. This motivates the definition of balanced BST, where $h = O(\log n)$ ($n$ is the number of values stored in the BST).

Red-black tree (RBT) is one type of balanced BST. RBT is a BST where nodes receive colors, with several additional properties. The important ones are:

- The descendants of a red node must be colored as black. This means that on any path from a node going downwards, we will not see two consecutive red nodes.
- The black-height property. For each node, the number of black nodes (excluding the node itself) are visited along each path from the node to any leaf. This number is called the black height. Note I did not say it clearly in class that the black height does not include the node itself, and this may fix some confusion caused in class.

To get an intuition why RBT is balanced, we claim that lengths of any two paths $P_1$ and $P_2$ from the root to leaves can not differ by a factor of 2. To see this, suppose $P_1$ is shorter and contains $k$ nodes. So the black nodes on $P_1$ is at most $k$. Note that $P_2$ must also contain the same number of black nodes as $P_1$, which is also at most $k$. Since there are two consecutive red nodes on $P_2$, we know that the number of red nodes on $P_2$ is no more than the number of black nodes. Thus, the total number of nodes of $P_2$ is at most $2k$.

Lecture 12: Red-black tree and greedy algorithm

In class, we take one more step forward to formally prove that a RBT with $n$ internal nodes has height of $O(n \log n)$. This is proved by induction. The textbook contains a good explanation. If you forget, check the textbook. This means that when we have a RBT, we can perform searching in $O(\log n)$ time. This will be useful for future applications.

Then we went over the operations done in RBT: how to insert new elements into RBT while still maintaining the RBT properties. This will involve several cases. In each case, the key is to ensure two things: all children of a red node must be black and the black-height property is maintained. For example, when we insert an item, we start by coloring the new node as red. This will need no more change if its parent is black, since adding a new red node does not impact the black-heights. But we must change the RBT if its parent is red. This involves several cases, which depends on the color settings of its parent and its uncle. Read Chapter 13 if you do not quite get how this is done.

We then analyze the running time for insertion, which is $O(\log n)$. This is because the case 1 operations can be repeated but we will always moving up the tree, and the other two cases can only be applied once. In each case, it takes $O(1)$ to perform tree modification. We did not cover how deletion is done, which is slightly more complicated.

Now, we start Chapter 16, the Greedy algorithms. Greedy algorithms make locally optimal decisions, which sometimes lead to globally optimal solution. Our first example is the coin change problem. You are given unlimited supplies of several coins (say quarters, dimes, nickels and pennies), you want to give out $n$ cents using the fewest number of coins. The common greedy strategy of using largest coins is sometimes globally optimal, as in the US coins. However, if we slightly change the coin types, the situation will change. Suppose we have quarters, dimes and pennies but no nickels, the greedy strategy does not work. This can be seen by $n = 30$ example.

Our next example is the activity selection problem in the textbook. In class, several strategies were proposed:

1. First come, first serve.

2. Picking shortest requests.
3. Picking the requests overlapping the fewest other intervals.

We showed counter-examples that the first two strategies are not globally optimal. The third strategy will be part of next homework.

Lecture 13: Greedy algorithm

I claim that choosing the requests with earliest finish time is optimal. In class, I showed how to prove this strategy is optimal. This is done in two steps. This needs considering an optimal solution $O$, where $O = \{o_1, \ldots, o_k, o_{k+1}, \ldots\}$. We denote the solution found by greedy algorithm is $A = \{a_1, \ldots, a_k\}$. Note that if $O$ and $A$ contain the same number of requests, we are done. Otherwise, we assume $o_{k+1}$ exists.

First, I show that $f(a_i) \leq f(o_i)$ for each $1 \leq i \leq k$. This is proved by induction. When $i = 1$, it is true because the greedy algorithm picks the one with smallest finish time. Then we assume the claim holds for all indices less than $i$. We now show $f(a_i) \leq f(o_i)$. The key to this is that if this is not true, the greedy algorithm would have selected $o_i$ instead of $a_i$ since $o_i$ is compatible with the previous $a_{i-1}$ (why? Make sure you understand this).

Then, I show that $o_{k+1}$ can not exist. Otherwise, $f(a_k) \leq f(o_k)$, which will suggest the greedy algorithm will not stop picking after $a_k$.

Then, I explained two aspects of greedy strategy mentioned in the textbook.

- Optimal-choice property. This says the greedy strategy will make locally optimal move, that is also globally optimal.
- Optimal sub-structure. This means when we have a new subproblem, we can just seek the optimal solution for the sub-problem and what we did before has no impact on the new subproblem.

The main example of this class is the classic Knapsack problem. We are given $n$ items, each with $w_i$ weight and $v_i$ value. We want to pack the most valued goods with a knapsack that can hold no more than $W$ pounds. There are two variations: fractional knapsack (where we can take portion of an item) and 0-1 knapsack (where we take or not take an item as a whole). The greedy strategy of picking the highest value per unit weight is optimal for the fractional (which we will prove now) but not for 0-1 knapsack.

Now I claim that the fractional knapsack problem is solved optimally by the above greedy algorithm. We let the greedy solution being $A = a_1, \ldots, a_k$, where each $a_i = (x, p(x))$ ($x$ is the type of items to pick, and $p(x)$ is how much we take). For comparison, we consider an optimal solution $O = o_1, \ldots, o'_k$. For the ease of exposition, we assume items with distinct value per unit weight (this can be removed fairly easily by merging the items with same value per unit weight). We re-arrange $O$ so that its items are also listed according to the sorted list of items by value per unit weight. We consider the first position where $A$ and $O$ are different, say at $p$. That is, $a_p \neq o_p$. There can be two cases: (a) $o_p = (p, w_1)$ and $a_p = (p, w_2)$ are taking the same item $p$, but $a_p$ just take fewer than $o_p$ (i.e. $w_1 \leq w_2$; also note the greedy strategy will ensure $a_p$ can not take fewer). In this case, we swap $w_2 - w_1$ of other later items in $O$ with $w_2 - w_1$ item $p$ (which is available and also more valuable). The changed solution will be more valuable than the original $O$, which is a contradiction to the assumption that $O$ is optimal. (b) $a_p = (p_1, w_1)$ and $o_p = (p_2, w_2)$ are taking different items. Then if $w_1 \leq w_2$, we will simply swap in $O$ the $w_1$ of item $p_1$ for item $p_2$ (which is available and more valuable). This again leads to a contradiction. If instead $w_1 > w_2$, we would swap $w_1 - w_2$ of $p_1$ for the same weight of $p_2$. This again leads to contradiction. In any case, we know the greedy strategy is optimal.