1. A standard method to improve the cache performance of search trees is to pack more search keys and subtrees into each node. A ***B-tree*** is a rooted tree in which each internal node stores up to B keys and pointers to up to $B+1$ children, each the root of a smaller B-tree. Specifically each node v stores three fields:

   - a positive integer $v.d \le B$,
   - a *sorted* array $v.key[1\,..\,v.d]$, and
   - an array $v.child[0\,..\,v.d]$ of child pointers.

   In particular, the number of child pointers is always exactly one more than the number of keys.

   Each pointer $v.child[i]$ is either Null or a pointer to the root of a B-tree whose keys are all larger than $v.key[i]$ and smaller than $v.key[i+1]$. In particular, all keys in the leftmost subtree $v.child[0]$ are smaller than $v.key[1]$, and all keys in the rightmost subtree $v.child[v.d]$ are larger than $v.key[v.d]$.

   Intuitively, you should have the following picture in mind:

$$\begin{tikzcd}
{[} & \bullet \arrow[d, dotted] & < key[1] < & \bullet \arrow[d, dotted] & < key[2] < & \bullet \arrow[d, dotted] & \cdots & \bullet \arrow[d, dotted] & < key[d] < & \bullet \arrow[d, dotted] & {]} \\
 & T_0 & & T_1 & & T_2 & \cdots & T_{d-1} & & T_d &
\end{tikzcd}$$

   Here T_i is the subtree pointed to by $child[i]$.

   The ***cost*** of searching for a key x in a B-tree is the number of nodes in the path from the root to the node containing x as one of its keys. A 1-tree is just a standard binary search tree.

   Fix an arbitrary positive integer $B > 0$. (I suggest $B = 8$.) Suppose we are given a sorted array $A[1, \ldots, n]$ of search keys and a corresponding array $F[1, \ldots, n]$ of frequency counts, where $F[i]$ is the number of times that we will search for $A[i]$.

   Describe and analyze an efficient algorithm to find a B-tree that minimizes the total cost of searching for n keys with a given array of frequencies.

   - For 5 points, describe a polynomial-time algorithm for the special case $B = 2$.
   - For 10 points, describe an algorithm for arbitrary B that runs in $O(n^{B+c})$ time for some fixed integer c.
   - For 15 points, describe an algorithm for arbitrary B that runs in $O(n^c)$ time for some fixed integer c that does *not* depend on B.

   Like all other homework problems, 10 points is full credit; any points above 10 will be awarded as extra credit.

   A few comments about *B*-trees. Normally, B-trees are required to satisfy two additional constraints, which guarantee a worst-case search cost of $O(\log_B n)$: Every leaf must have exactly the same depth, and every node except possibly the root must contain at least $B/2$ keys. However, in this problem, we are not interested in optimizing the *worst-case* search cost, but rather the *total* cost of a sequence of searches, so we will not impose these additional constraints.

   In most large database systems, the parameter B is chosen so that each node exactly fits in a cache line. Since the entire cache line is loaded into cache anyway, and the cost of loading a cache line exceeds the cost of searching within the cache, the running time is dominated by the number of cache faults. This effect is even more noticeable if the data is too big to lie in RAM at all; then the cost is dominated by the number of page faults, and B should be roughly the size of a page.

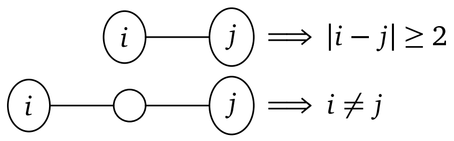
Finally, don't worry about the cache/disk performance in your homework solutions; just analyze the CPU time as usual. Designing algorithms with few cache misses or page faults is a interesting pastime; simultaneously optimizing CPU time *and* cache misses *and* page faults is even more interesting. But this kind of design and analysis requires tools we won't see in this class.

2. ***Extra credit***, because we screwed up the first version.

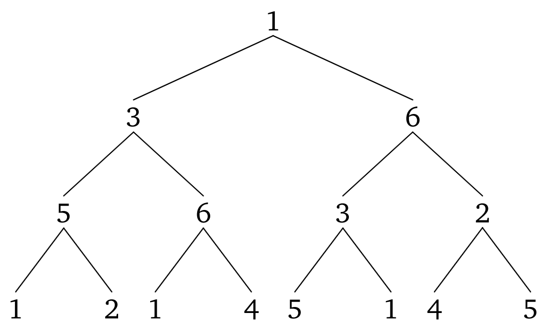
A ***k*-2 coloring** of a tree assigns each vertex a value from the set $\{1, 2, \ldots, k\}$, called its *color*, such that the following constraints are satisfied:

- A node and its parent cannot have the same color or adjacent colors.
- A node and its grandparent cannot have the same color.
- Two nodes with the same parent cannot have the same color.

The last two rules can be written more simply as "Two nodes that are two edges apart cannot have the same color." Diagrammatically, if we write the names of the colors inside the vertices,



For example, here is a valid 6-2 coloring of the complete binary tree with depth 3:



(a) Describe and analyze an algorithm that computes a 6-2 coloring of a given binary tree. The existence of such an algorithm proves that every binary tree has a 6-2 coloring.

(b) Prove that not every binary tree has a 5-2 coloring.

(c) A *ternary* tree is a rooted tree where every node has at most *three* children. What is the smallest integer k such that every ternary tree has a k-2 coloring? Prove your answer is correct.