**CMSC 420: Short Reference Guide**

This document contains a short summary of information about algorithm analysis and data structures, which may be useful later in the semester.

**Asymptotic Forms:** The following gives both the formal "$c$ and $n_0$" definitions and an equivalent limit definition for the standard asymptotic forms. Assume that $f$ and $g$ are nonnegative functions.

| Asymptotic Form | Relationship | Limit Form | Formal Definition |
|---|---|---|---|
| $f(n) \in \Theta(g(n))$ | $f(n) \equiv g(n)$ | $0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$ | $\exists c_1, c_2, n_0, \forall n \geq n_0,\ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$. |
| $f(n) \in O(g(n))$ | $f(n) \preceq g(n)$ | $\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$ | $\exists c, n_0, \forall n \geq n_0,\ 0 \leq f(n) \leq c g(n)$. |
| $f(n) \in \Omega(g(n))$ | $f(n) \succeq g(n)$ | $\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0$ | $\exists c, n_0, \forall n \geq n_0,\ 0 \leq c g(n) \leq f(n)$. |
| $f(n) \in o(g(n))$ | $f(n) \prec g(n)$ | $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ | $\forall c, \exists n_0, \forall n \geq n_0,\ 0 \leq f(n) \leq c g(n)$. |
| $f(n) \in \omega(g(n))$ | $f(n) \succ g(n)$ | $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$ | $\forall c, \exists n_0, \forall n \geq n_0,\ 0 \leq c g(n) \leq f(n)$. |

**Polylog-Polynomial-Exponential:** For any constants $a$, $b$, and $c$, where $b > 0$ and $c > 1$.

$$\log^a n \prec n^b \prec c^n.$$

**Common Summations:** Let $c$ be any constant, $c \neq 1$, and $n \geq 0$.

| Name of Series | Formula | Closed-Form Solution | Asymptotic |
|---|---|---|---|
| Constant Series | $\sum_{i=a}^{b} 1$ | $= \max(b - a + 1, 0)$ | $\Theta(b - a)$ |
| Arithmetic Series | $\sum_{i=0}^{n} i = 0 + 1 + 2 + \cdots + n$ | $= \frac{n(n+1)}{2}$ | $\Theta(n^2)$ |
| Geometric Series | $\sum_{i=0}^{n} c^i = 1 + c + c^2 + \cdots + c^n$ | $= \frac{c^{n+1} - 1}{c - 1}$ | $\begin{cases} \Theta(c^n)\ (c > 1) \\ \Theta(1)\ (c < 1) \end{cases}$ |
| Quadratic Series | $\sum_{i=0}^{n} i^2 = 1^2 + 2^2 + \cdots + n^2$ | $= \frac{2n^3 + 3n^2 + n}{6}$ | $\Theta(n^3)$ |
| Linear-geom. Series | $\sum_{i=0}^{n-1} ic^i = c + 2c^2 + 3c^3 \cdots + nc^n$ | $= \frac{(n-1)c^{(n+1)} - nc^n + c}{(c-1)^2}$ | $\Theta(nc^n)$ |
| Harmonic Series | $\sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$ | $\approx \ln n$ | $\Theta(\log n)$ |

**Recurrences:** Recursive algorithms (especially those based on divide-and-conquer) can often be analyzed using the so-called *Master Theorem*, which states that given constants $a > 0$, $b > 1$, and $d \geq 0$, the function $T(n) = aT(n/b) + O(n^d)$, has the following asymptotic form:

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a. \end{cases}$$

**Sorting:** The following algorithms sort a set of $n$ keys over a totally ordered domain. Let $[m]$ denote the set $\{0, \ldots, m\}$, and let $[m]^k$ denote the set of ordered $k$-tuples, where each element is taken from $[m]$.

A sorting algorithm is *stable* if it preserves the relative order of equal elements. A sorting algorithm is *in-place* if it uses no additional array storage other than the input array (although $O(\log n)$ additional space is allowed for the recursion stack). The *comparison-based algorithms* (Insertion-, Merge-, Heap-, and QuickSort) operate under the general assumption that there is a *comparator function* $f(x, y)$ that takes two elements $x$ and $y$ and determines whether $x < y$, $x = y$, or $x > y$.

| Algorithm | Domain | Time | Space | Stable | In-place |
|---|---|---|---|---|---|
| CountingSort | Integers $[m]$ | $O(n+m)$ | $O(n+m)$ | Yes | No |
| RadixSort | Integers $[m]^k$ or $[m^k]$ | $O(k(n+m))$ | $O(kn+m)$ | Yes | No |
| InsertionSort | Total order | $O(n^2)$ | $O(n)$ | Yes | Yes |
| MergeSort | Total order | $O(n \log n)$ | $O(n)$ | Yes | No |
| HeapSort | Total order | $O(n \log n)$ | $O(n)$ | No | Yes |
| QuickSort | Total order | $O(n \log n)$ | $O(n)$ | Yes/No* | No/Yes |

*There are two versions of QuickSort, one which is stable but not in-place, and one which is in-place but not stable.

**Order statistics:** For any $k$, $1 \leq k \leq n$, the $k$th smallest element of a set of size $n$ (over a totally ordered domain) can be computed in $O(n)$ time.

**Useful Data Structures:** All these data structures use $O(n)$ space to store $n$ objects.

**Unordered Dictionary:** (by randomized hashing) Insert, delete, and find in $O(1)$ expected time each. (Note that you can find an element exactly, but you cannot quickly find its predecessor or successor.)

**Ordered Dictionary:** (by balanced binary trees or skiplists) Insert, delete, find, predecessor, successor, merge, split in $O(\log n)$ time each. (Merge means combining the contents of two dictionaries, where the elements of one dictionary are all smaller than the elements of the other. Split means splitting a dictionary into two about a given value $x$, where one dictionary contains all the items less than or equal to $x$ and the other contains the items greater than $x$.) Given the location of an item $x$ in the data structure, it is possible to locate a given element $y$ in time $O(\log k)$, where $k$ is the number of elements between $x$ and $y$ (inclusive).

**Priority Queues:** (by binary heaps) Insert, delete, extract-min, union, decrease-key, increase-key in $O(\log n)$ time. Find-min in $O(1)$ time each. Make-heap from $n$ keys in $O(n)$ time.

**Priority Queues:** (by Fibonacci heaps) Any sequence of $n$ insert, extract-min, union, decrease-key can be done in $O(1)$ amortized time each. (That is, the sequence takes $O(n)$ total time.) Extract-min and delete take $O(\log n)$ amortized time. Make-heap from $n$ keys in $O(n)$ time.

**Disjoint Set Union-Find:** (by inverted trees with path compression) Union of two disjoint sets and find the set containing an element in $O(\log n)$ time each. A sequence of $m$ operations can be done in $O(\alpha(m, n))$ amortized time. That is, the entire sequence can be done in $O(m \cdot \alpha(m, n))$ time. ($\alpha$ is the *extremely* slow growing inverse-Ackerman function.)

# Homework 1: Basic Data Structures and Trees

**Problem 1.** (10 points)

(a) (5 points) Consider the rooted tree of Fig. 1(a). Draw a figure showing its representation in the "first-child/next-sibling" form.

(b) (5 points) Consider the rooted tree of Fig. 1(b) represented in the "first-child/next-sibling" form. Draw a figure showing the equivalent rooted tree.

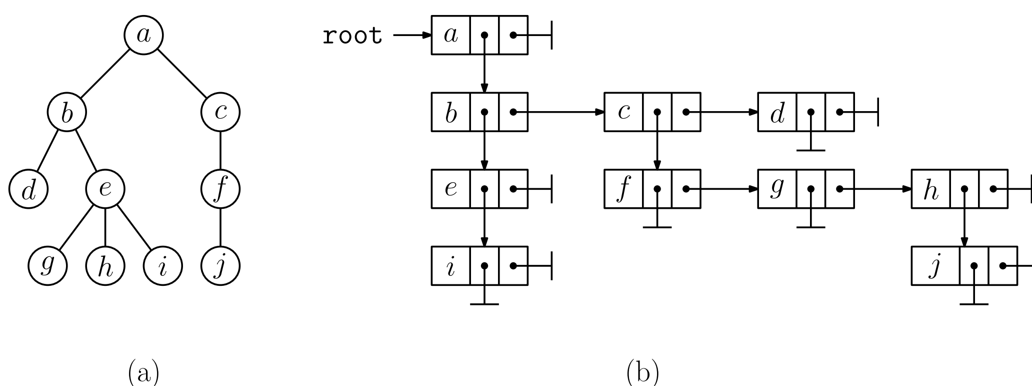



Figure 1: Rooted tree to first-child/next-sibling form and vice versa.

**Problem 2.** (10 points) We can reinterpret the first-child/next-sibling tree representation of a general rooted (multiway) tree as a binary tree. The first-child link is interpreted as the node's left child and the next-sibling link is interpreted as the node's right child. For example, in Fig. 2(a) we show a rooted (multiway) tree $T$, in Fig. 2(b), we show its first-child/next-sibling representation, and in Fig. 2(c), we show the corresponding *binary-equivalent tree*, call it $T'$.

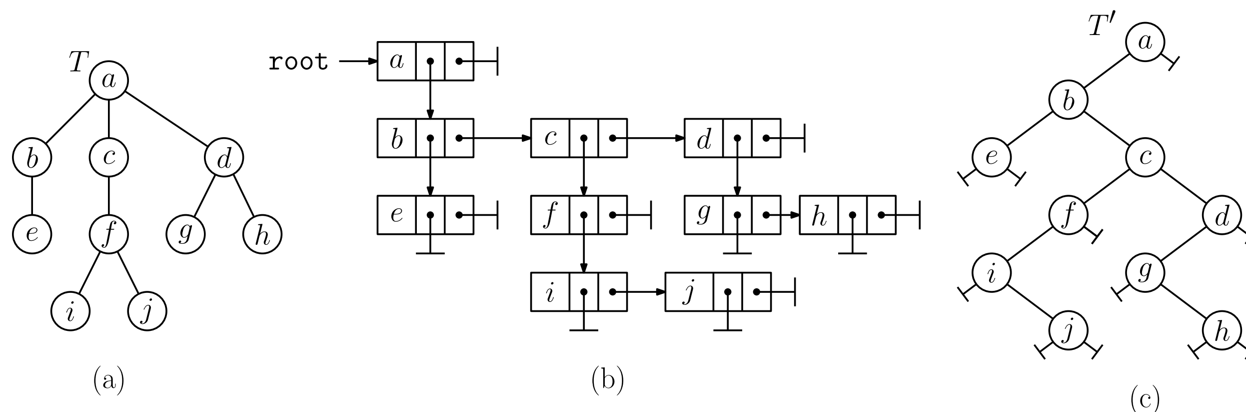



Figure 2: A rooted tree and the binary equivalent.

An interesting question to explore involves the relationships between the original tree and its binary-equivalent tree. Answer each of the following questions. Your answer should hold not only for the above example, but any nonempty rooted tree $T$ and its binary equivalent tree $T'$. *Provide a brief explanation of your answers (perhaps with an adjoining figure). A formal proof is not required.*

(i) (5 points) A *preorder traversal* of the original tree $T$ is the same as:

   (a) a *preorder traversal* of the binary-equivalent tree $T'$
   (b) a *postorder traversal* of the binary-equivalent tree $T'$
   (c) an *inorder traversal* of the binary-equivalent tree $T'$
   (d) none of the above

(ii) (5 points) A *postorder traversal* of the original tree $T$ is the same as:

   (a) a *preorder traversal* of the binary-equivalent tree $T'$
   (b) a *postorder traversal* of the binary-equivalent tree $T'$
   (c) an *inorder traversal* of the binary-equivalent tree $T'$
   (d) none of the above

**Problem 3.** (6 points) Consider the binary tree shown in Fig. 3. Draw a figure showing the inorder threads (analogous to Fig. 7 in Lecture 3).

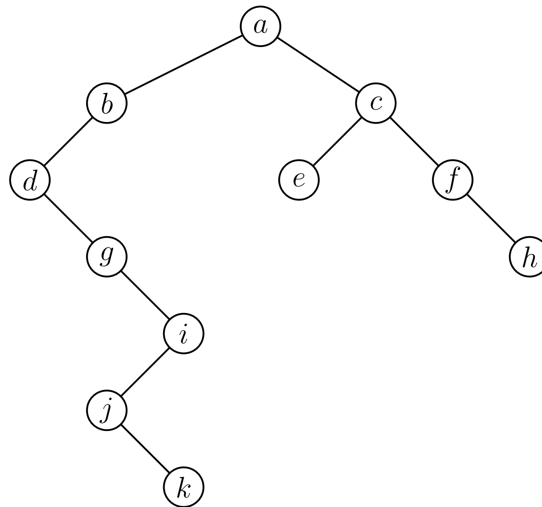



Figure 3: Add inorder threads to this tree.

**Problem 4.** (12 points) In our implementation of binary search trees, we assumed that each node stored a pointer to its left and right children. Suppose that, in addition, we add a pointer to the node's parent in the tree. The new node structure is as follows, and the node constructor takes an additional argument specifying the parent:

```
class BSTNode {
    Key key;
    Value value;
```

```
        BSTNode left;
        BSTNode right;
        BSTNode parent;

        BSTNode(Key x, Value v, BSTNode l, BSTNode r, BSTNode p) {
            // constructor - details omitted
        }
    }
```

Assume that the root of the tree has a parent pointer of `null`.

(a) (6 points) Present pseudocode for an `insert` function that inserts a new key and updates the parent pointers appropriately. (Hint: Modify the `insert` function from Lecture 4.) We will need to change the function's signature, now taking four arguments: the key $x$, the value $v$, the current node $p$, and the current node's parent $q$:

```
BSTNode insert(Key x, Value v, BSTNode p, BSTNode q) {
    // ... fill this in
}
```

In order to insert a key-value pair $(x, v)$ into your tree, the initial call is

```
root = insert(x, v, root, null)
```

Your function should run in time proportional to the height of the tree.

(b) (6 points) Assuming we have parent links, present pseudocode for a function that computes the *inorder successor* of an arbitrary node $p$ in the tree. If $p$ is the last inorder node of the tree, this function returns `null`. The function signature is as follows:

```
BSTNode inorderSuccessor(BSTNode p) {
    // ... fill this in
}
```

Briefly explain how your function works. (*Don't just give code!*) Your function should run in time proportional to the height of the tree.

**Problem 5.** (12 points) In this problem, we will generalize the amortized analysis of the expandable stack from Lecture 2. Recall that we are given a stack that is stored in an array. Initially, the array has 1 element. Each time that a push or pop operation does not cause an overflow, it costs us +1 unit. If a push operation causes the array to overflow, we doubled the array size, growing, say, from $m$ to $2m$, copy the elements over to this new array, and then perform the push. It costs $+2m$ units for the reallocation and an additional +1 unit to perform the actual operation. In class, we showed that the *amortized cost* of such a stack is at most 5. Formally, this means that if we start from an empty stack and an array of size 1, and perform any sequence of $n$ push/pop operations, the total cost is at most $5n$.

Please answer the following problems. (Part (a) is a special case of part (b). If you are confident of your answer, you can answer just part (b), and we will apply the same score to part (a).)

(a) (4 points) Suppose that we modify our expandable stack example so that, whenever the stack overflows, we allocate a new array of *three times* the size, growing, say, from $m$

to $3m$. So, successive overflows would result in arrays of sizes $3, 9, 27, \ldots, 3^k, \ldots$. Let us assume a similar cost model. Each non-overflowing push/pop costs +1 unit. Whenever we overflow the array, we will charge $+2m$ for reallocation and an additional +1 for the actual push itself (see Fig. 4). (Why $+2m$ and not $+3m$? The choice is somewhat arbitrary, but my logic is that I'll charge +1 for reading each item from the old array and +1 for writing it in the new array.)

Using the "token-based" method given in Lecture 2, prove that this version of the data structure has an amortized cost of at most 4. (I believe that this is the best possible upper bound as $n$ becomes large. If you get a smaller number, better check with us, because it is probably wrong.)

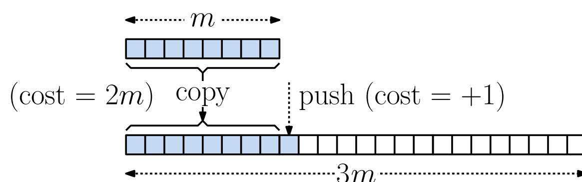
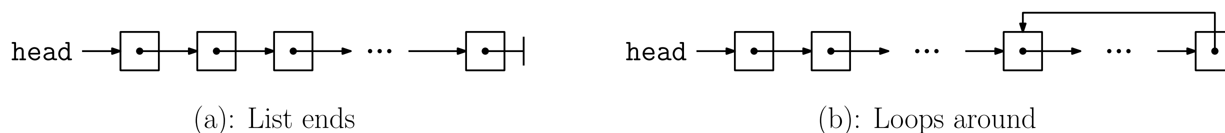



Figure 4: Generalizing the expandable stack (for $c = 3$).

(b) (8 points) Let's generalize this further. Given a positive integer constant $c \geq 2$, our expandable stack allocates a new array of $c$ *times* the size, growing, say, from $m$ to $cm$. So, successive overflows would result in arrays of sizes $c, c^2, c^3, \ldots, c^k, \ldots$. We will charge +1 for each individual operation and $+2m$ for copying the $m$ elements over into the new array (see Fig. 4 for the case $c = 3$). Derive the amortized cost of this version of the stack as a function of $c$. (For full credit your answer should be tight. In particular, for $c = 2$ and $c = 3$, your formula should give you 5 and 4, respectively.)

**Note:** Challenge problems are just for fun. We grade them, but the grade is not used to determine your final grade in the class. Sometimes when assigning cutoffs, I consider whether you attempted some of the challenge problems, and I may "bump-up" a grade that is slightly below a cutoff threshold based on these points. (But there is no formal rule for this.)

**Challenge Problem:** You are given a rather trivial linked list, where each node contains a single member, `next`, which points to the next element in the linked list. The variable `head` points to the head of the linked list. There are two possible forms that the list might take. First, it might *end* in a `null` pointer (see Fig. 5(a)), or second, it might *loop* around on iteself, where for some node `p`, `p.next` points to an earlier node within the list, possibly `p` itself (see Fig. 5(b)).




(a): List ends

(b): Loops around

Figure 5: Generalizing the expandable stack (for $c = 3$).

Give pseudocode for a function that determines whether the list ends in `null` or wraps around on itself. Here is the catch: You cannot modify the nodes of the list, and your function cannot use more than a constant amount of working storage. (This latter requirement implies that you cannot make recursive calls of more than a constant recursion depth, since this would use more than a constant amount of system storage.) Your algorithm must run in time proportional to the total number of elements in the list.

**Homework 2: Search Trees**

**Problem 1.** (5 points) Consider the AVL tree shown in Fig. 1.

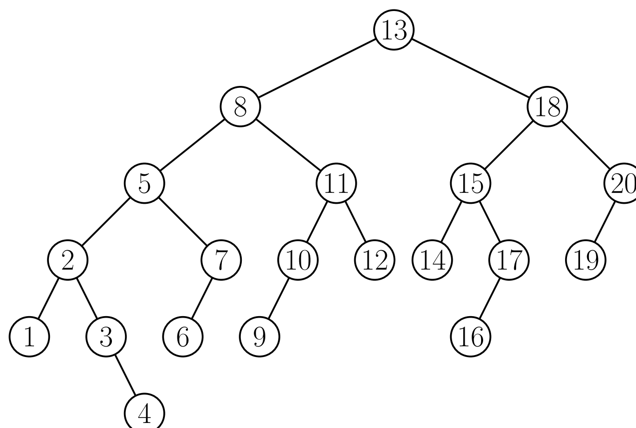



Figure 1: Problem 1: AVL Tree example.

(a) (2 points) Draw the tree again, indicating the balance factors associated with each node.

(b) (3 points) Show the tree that results from the operation `delete(19)`, after all the re-balancing has completed. (We only need the final tree. You can provide intermediate results for partial credit.)

**Problem 2.** (8 points) Consider the AA trees shown in Fig. 2.

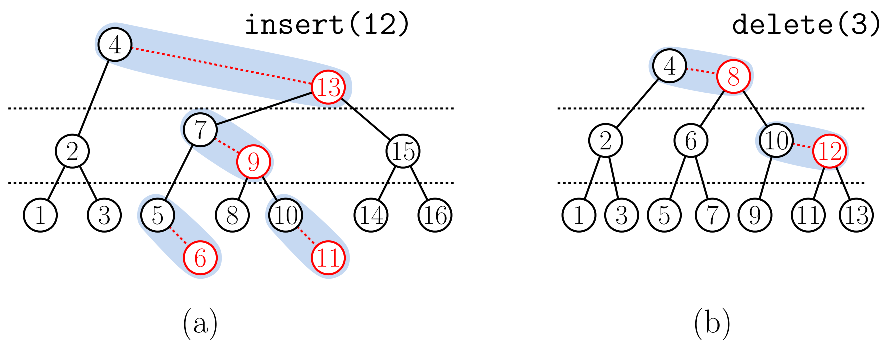



Figure 2: Problem 2. AA Tree example

(a) (4 points) Show the result of performing the operation `insert(12)` into the tree in Fig. 2(a).

(b) (4 points) Show the result of performing the operation `delete(3)` from the tree in Fig. 2(b).

(In both cases, we only need the final tree. You can provide intermediate results for partial credit. Because Gradescope is color-blind, please indicate red nodes using dashed lines, as in the figure.)

**Problem 3.** (8 points – 2 points each) In class, I said "trust me" that the Zig-Zig and Zig-Zag rotations for splay trees produced the results given in the lecture notes. In this problem, you will verify my assertion by working through the rotations one by one.

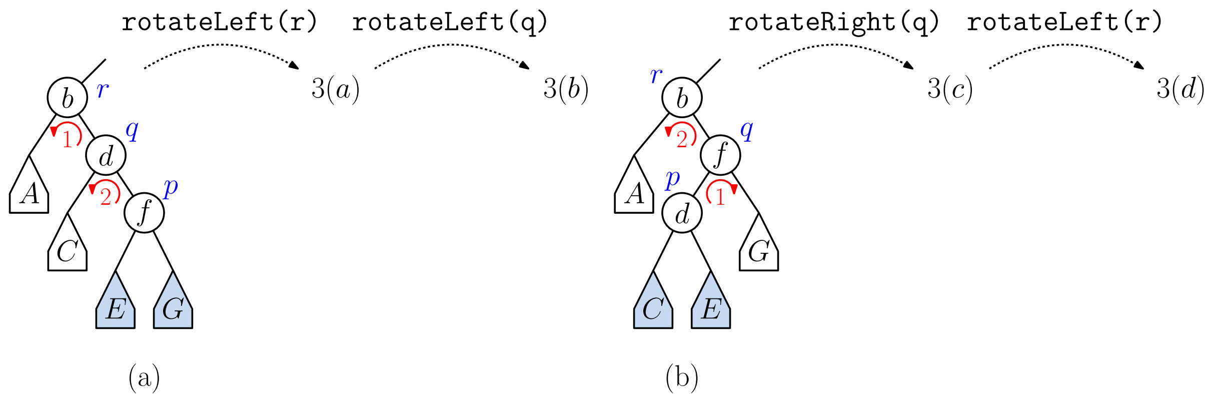



Figure 3: Problem 3: Splay rotations.

(a) Consider the tree shown in Fig. 3(a). Given node `p`, its parent `q`, and its grandparent `r`, show the result after the first rotation of the RR Zig-Zig operation at `p`, namely a left rotation about `r`.

(b) Complete the RR Zig-Zig operation by showing the result after the second rotation, namely a left rotation about `q`.

(c) Consider the tree shown in Fig. 3(b). Given node `p`, its parent `q`, and its grandparent `r`, show the result after the first rotation of the RL Zig-Zag operation at `p`, namely a right rotation about `q`.

(d) Complete the RL Zig-Zag rotation by showing the result after the second rotation, namely a left rotation about `r`.

**Problem 4.** (12 points) In this problem, we will assume that we have a standard (unbalanced) binary search tree. Each node stores a key, value, left-child link and right-child link. Suppose that we add one additional field, `size`, which indicates the number of nodes in the subtree rooted at the current node.

(a) (2 points) Present pseudo-code for an operation `void rotateRight(Node p)`, which performs a right rotation at node `p`, and also updates the `size` values for all nodes whose sizes are affected by the rotation. You may assume that `p.left` is not `null` and all the size values are valid prior to the rotation.

(b) (5 points) Present pseudo-code for an operation `int countSmaller(Key x)`, which returns a count of the number of nodes in the entire tree whose key values are strictly smaller than `x`. (Hint: Write a helper function `countSmaller(Key x, Node p)`, which

returns a count of the number of nodes that are smaller than `x` within the subtree rooted at `p`.) Briefly explain how your function works.

(c) (5 points) Present pseudo-code for an operation `Value getMinK(int k)`, which returns the value associated with the $k$th smallest key in the entire tree. If the tree has fewer than $k$ keys, this should return `null`. (Hint: Write a helper function `getMinK(int k, Node p)`, which returns the value of the $k$th smallest key in the subtree rooted at `p`.) Briefly explain how your function works.

The rotation should run in $O(1)$ time and `getMinK` and `countSmaller` should both run in time proportional to the height of the tree, and neither function should alter the tree's structure.

**Problem 5.** (5 points) It is easy to see that, if you splay twice on the same key in a splay tree (`splay(x); splay(x)`), the tree's structure does not change as a result of the second call.

Is this true when we alternate between two keys? Let $T_0$ be an arbitrary splay tree, and let $x$ and $y$ be two keys that appear within $T_0$. Let:

- $T_1$ be the result of applying `splay(x); splay(y)` to $T_0$.
- $T_2$ be the result of applying `splay(x); splay(y); splay(x); splay(y)` to $T_0$.

**Question:** Irrespective of the initial tree $T_0$ and the choice of $x$ and $y$, is $T_1 = T_2$? (That is, are the two trees structurally identical?) Either state this as a theorem and prove it or provide a counterexample, by giving the tree $T_0$ and two keys $x$ and $y$ for which this fails.

**Problem 6.** (12 points – 4 points each) For this problem, assume that the structure of a node in a skip list is as follows:

```
class SkipNode {
    Key key;            // key
    Value value;        // value
    SkipNode[] next;    // array of next pointers
}
```

The height of a node (that is, the number of levels to which it contributes) is given by the Java operation `p.next.length`.

Often, when dealing with ordered dictionaries, we wish to perform a sequence of searches in sorted order. Suppose that we have two keys, $x < y$, and we have already found the node `p` that contains the key $x$. In order to find $y$, it would be wasteful to start the search at the head of the skip list. Instead, we want to start at `p`. Suppose that there are $k$ nodes between $x$ and $y$ in the skip list. We want the expected search time to be $O(\log k)$, not $O(\log n)$.

(a) Present pseudo-code for an algorithm for a function `Value forwardSearch(p, y)`, which starting a node `p` (whose key is smaller than $y$), finds the node of the skip list containing key $y$ and returns its value. (For simplicity, you may assume that key $y$ appears within the skip list.) In addition to the pseudo-code, briefly explain how your method works.

Show that the expected number of hops made by your algorithm is $O(\log k)$, where $k$ is the number of nodes between $x$ and $y$. The proof involves showing two things:

(b) Prove that the maximum level reached is $O(\log k)$ in expectation (over random coin tosses).

(c) Prove that the number of hops per level is $O(1)$ in expectation.

(Hint: The proofs of both follow the same structure as given in the full lecture notes.)

**Note:** Challenge problems are just for fun. We grade them, but the grade is not used to determine your final grade in the class. Sometimes when assigning cutoffs, I consider whether you attempted some of the challenge problems, and I may "bump-up" a grade that is slightly below a cutoff threshold based on these points. (But there is no formal rule for this.)

**Challenge Problem:** You are given an extended binary search tree with a root note, `root`. For every node `p`, there is a boolean member, `p.isExternal`, which tells you whether this node is external. Also, every internal node has left and right pointers, `p.left` and `p.right`. Your objective is to return a pointer to *any* external node.

(a) Prove that there is a randomized algorithm that, given a pointer to the root of an extended binary tree $T$, returns a pointer to any external node of $T$. This algorithm's expected running time (averaged over all random choices made by the algorithm) must be $O(\log n)$, where $n$ is the number of external nodes in the tree. Note that the algorithm does not know what $n$ is. Prove your algorithm's running time.

(b) Prove that for any deterministic algorithm for the same problem, there exists an extended binary tree $T$ such that the algorithm must visit at least $\Omega(n)$ nodes before finding an external node when run on $T$. (The tree's structure is allowed to depend on the algorithm.)

Your algorithms can only access nodes of the tree by starting at the root and following left-right links. Neither algorithm knows the value of $n$.

## Practice Problems for the Midterm

The exam will be asynchronous and online. It is open-book, open-notes, open-Internet, but it must be done on your own without the aid of other people or software.

**Problem 0.** Expect at least one question of the form "apply operation $X$ to data structure $Y$," where $X$ is a data structure that has been presented in lecture.

**Problem 1.** Short answer questions. Except where noted, explanations are not required, but may be given to help with partial credit.

(a) A binary tree is *full* if every node either has 0 or 2 children. Given a full binary tree with $n$ total nodes, what is the maximum number of leaf nodes? What is the minimum number? Give your answer as a function of $n$ (no explanation needed).

(b) **True or false?** Let $T$ be extended binary search tree (that is, one having internal and external nodes). In an inorder traversal, internal and external nodes are encountered in *alternating order*. (If true, provide a brief explanation. If false, show a counterexample.)

(c) **True or false?** In every extended binary tree having $n$ external nodes, there exists an external node of depth at most $\lceil \lg n \rceil$. **Explain briefly.**

(d) What is the minimum and maximum number of levels in a 2-3 tree with $n$ nodes. (Define the number of levels to be the height of the tree plus one.) Hint: It may help to recall the formula for the geometric series: $\sum_{i=0}^{m-1} c^i = (c^m - 1)/(c - 1)$.

(e) You have an AVL tree containing $n$ keys, and you *insert* a new key. As a function of $n$, what is the maximum number of rotations that might be needed as part of this operation? (A double rotation is counted as two rotations.) Explain briefly.

(f) Repeat (e) in the case of *deletion* from an AVL tree. (You can give your answer as an asymptotic function of $n$.)

(g) You are given a 2-3 tree of height $h$, which you convert to an AA-tree. As a function of $h$, what is the *minimum* number of *red nodes* that might appear on any path from a root to a leaf node in the AA tree? What is the *maximum* number?

(h) Both skip lists and B-trees made use of nodes containing a variable number of elements. (In the skip list, and node has a variable number of pointers, and in a B-tree a node has a variable number of keys/children.) In one case, we allocated nodes of variable size and in the other case, we allocated nodes of the same fixed size. Why did we do things differently in these two cases?

(i) Unbalanced search trees and treaps both support dictionary operations in $O(\log n)$ "expected time." What difference is there (if any) in the meaning of "expected time" in these two contexts?

(j) Splay trees are known to support efficient *finger search queries*. What is a "finger search query"?

(k) Consider a splay tree containing $n$ keys $a_1 < a_2 < \cdots < a_n$. Let $x$, $y$, and $z$ be any three consecutive elements in this sorted sequence. Suppose that we perform splay($x$) followed immediately by splay($z$). What (if anything) can be said about the depth of $y$ at this time?

**Problem 2.** You are given a degenerate binary search tree with $n$ nodes in a left chain as shown in Fig. 3, where $n = 2^k - 1$ for some $k \geq 1$.

(a) Derive an algorithm that, using only single left- and right-rotations, converts this tree into a perfectly balanced complete binary tree (see Fig. 1).

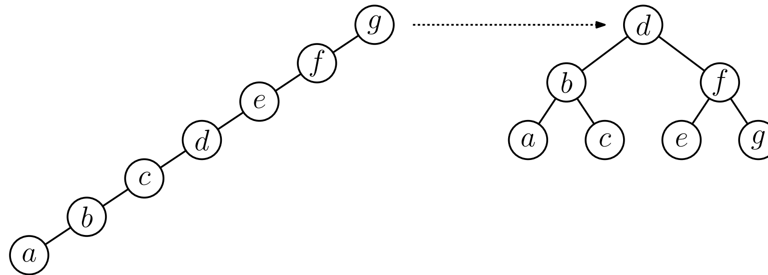



Figure 1: Rotating into balanced form.

(b) As an asymptotic function of $n$, how many rotations are needed to achieve this? $O(\log n)$? $O(n)$? $O(n \log n)$? $O(n^2)$? Briefly justify your answer.

**Problem 3.** You a given a threaded binary search tree $T$ (not necessarily balanced). Recall that each node has additional fields `p.leftIsThread` (resp., `p.rightIsThread`). These indicate whether `p.left` (resp., `p.right`) points to an actual child or it points to the inorder predecessor (resp., successor).

Present pseudocode for each of the following operations. Both operations should run in time proportional to the height of the tree.

(a) `void T.insert(Key x, Value v)`: Insert a new key-value pair $(x, v)$ into $T$ and update the node threads appropriately (see Fig. 2(a)).

(b) `Node preorderSuccessor(Node p)`: Given a non-null pointer to any node $p$ in $T$, return a pointer to its *preorder successor*. (Return `null` if there is no preorder successor.)

**Problem 4.** You are given a binary search tree where, in addition to the usual fields `p.key`, `p.left`, and `p.right`, each node `p` has a *parent link*, `p.parent`. This points to `p`'s parent, and is `null` if `p` is the root. Given such a tree, present pseudo-code for a function

```
Node preorderPred(Node p)
```

which is given a non-null reference `p` to a node of the tree and returns a pointer to `p`'s *preorder predecessor* in the tree (or `null` if `p` has no preorder predecessor). Your function should run in time proportional to the height of the tree. Briefly explain how your function works.

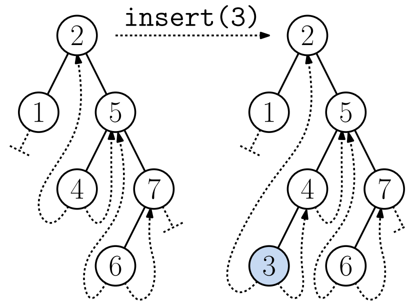



Figure 2: Threaded tree operations.

**Problem 5.** Suppose that in addition to the key-value pair (`p.key` and `p.value`) and pointers to the node's left and right children (`p.left` and `p.right`), each node of a binary search tree stores a sibling pointer, `p.sibling`, which points to `p`'s sibling, or `null` if `p` has no sibling.

Recall the following code for performing a right rotation of a node in a binary search tree:

```
Node rotateRight(Node p) {
    Node q = p.left;
    p.left = q.right;
    q.right = p;
    return q;
}
```

Modify the above code so that, in addition to performing the rotation, the sibling pointers are also updated. (You may *not* assume the existence of other information, such as parent pointers or threads. Your function should run in constant time.)

**Problem 6.** A *zig-zag tree* is defined to be a binary search tree having an odd number of nodes that consists of a single path, alternating between right- and left-child links. An example is shown in Fig. 3, where we have also labeled each node with its *depth*, that is, the length of the path from the root.

(a) Draw the final tree that results from executing `splay("e")` on the tree of the figure below. (Intermediate results can be given for partial credit.)

(b) Let $T$ be a zig-zag tree with $n$ nodes, and let $T'$ be the tree that results after performing a splay operation on $T$'s deepest leaf. Consider a node $p$ at level $k$ in $T$, for $0 \leq k \leq n-2$. What is the depth of $p$ in $T'$? Express your answer as a function of $k$ and $n$. Your formula should apply to every node of the tree, *except the node that was splayed.*

(c) Give a short proof justifying the correctness of your formula.

**Problem 7.** You are given a skip list with $n$ nodes in which, rather than promoting each node to the next higher level with probability $1/2$, we promote each node with probability $p$, for $0 < p < 1$.

(a) Given a skip list with $n$ keys, what is the expected number of keys that contribute to the $i$th level. (Recall that the lowest level is level 0.) Briefly explain.

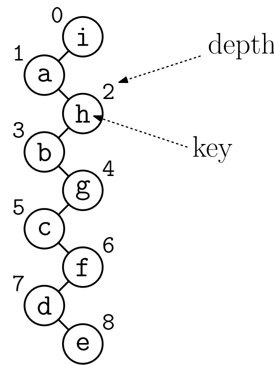



Figure 3: Zig-zag splay tree (with node depths shown).

(b) Show that (excluding the header and sentinel nodes) the total number of links in such a skip list (that is, the total size of all the skip list nodes) is expected to be at most $n/(1-p)$. (Hint: It may be useful to recall the formula for the geometric series from Problem 1(d).)

**Problem 8.** Show that if all nodes in a splay tree are accessed (splayed) in sequential order, the resulting tree consists of a linear chain of left children.

**Problem 9.** The objective of this problem is to design an enhanced stack data structure, called `MinStack`. For concreteness, let's assume that the stack just stores integers. Your stack should support the standard stack operations `void push(int x)`, which pushes `x` on top of the stack, and `int pop()`, which removes the element at the top of the stack and returns its value. It must also support the additional operation, `int getMin()`, which returns the smallest value currently in the stack, *without altering the contents of the stack*. Finally, there is a constructor `MinStack(int n)`, which is given the maximum number $n$ of items that will be stored in the stack.

Present pseudocode for a data structure that implements these operations. All operations should run in $O(1)$ time. (We will give partial credit if algorithm is correct, but your running time is worse than this.) Your answer should include the following things:

- Explain what objects are maintained by your data structure.
- Explain how the data structure is initialized (that is, what does the constructor do?)
- Present pseudocode descriptions of `push(x)`, `pop()`, and `getMin()`.

No error checking is needed. (No more than $n$ elements will be in the stack at any time and no `pop` or `getMin` from an empty stack.)

CMSC 420:Fall 2020 Dave Mount

**Midterm Exam**

The exam is asynchronous and online. It is open-book, open-notes, open-Internet, but it must be done on your own without the aid of other people or software. The total point value is 100 points. Good luck!

**Problem 1.** (10 points)

(1.1) (5 points) Consider the 2-3 tree shown the figure below. Show the **final tree** that results after the operation `insert(6)`. When rebalancing, use only splits, *no adoptions* (key rotations).

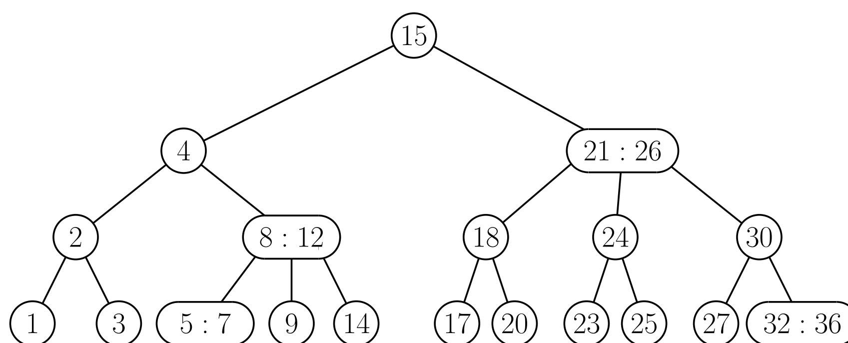



Figure 1: 2-3 tree insertion and deletion.

(1.2) (5 points) Returning to the original tree, show the **final tree** that results after the operation `delete(20)` When rebalancing, you may use *both merge and adoption* (key rotation).

In both cases, you may draw intermediate subtrees to help with partial credit, but don't waste too much time on this.

**Problem 2.** (25 points) Short answer questions. Except where noted, explanations are not required but may be given for partial credit.

(2.1) (4 pts) Suppose that you have a binary tree with $n$ nodes that is *not* full. As a function of $n$, what is the *minimum* and *maximum* number of leaves this tree could have?

(2.2) (5 pts) You have a set of $n$ distinct keys, where $n$ is a large even number. You randomly permute all the keys and insert the first $n/2$ of them into a standard (unbalanced) binary search tree. You then take the remaining $n/2$ keys, sort them in ascending order, and insert them into this tree. The final tree has $n$ nodes. As a function of $n$, what is the expected height of this tree? (Select the best from the choices below.)

(a) $O(\log n)$

(b) $O((\log n)^2)$

(c) $O(\sqrt{n})$

(d) $O(n)$

(2.3) (3 pts) You have a valid AVL tree with $n$ nodes. You insert two keys, one smaller than all the keys in the tree and the other larger than all the keys in the tree, but you do no rebalancing after these insertions. **True or False**: The resulting tree is a valid AVL tree. (Briefly explain.)

(2.4) (4 pts) You have a splay tree containing $n$ nodes for some large value $n$. Let $x$ and $y$ be two keys in the dictionary. You repeat the operations `splay(x)` followed by `splay(y)` $m$ times, where $m$ is much greater than $n$. What is the *worst-case* time complexity for all of these operations? (Select one)

(a) $O(m)$
(b) $O(m \log m)$
(c) $O(m \log n)$
(c) $O(n \log m)$
(d) $O(mn)$
(e) None of the above

(2.5) (3 pts) By mistake, two keys in your treap happen to have the same priority. Which of the following is a possible consequence of this mistake? (Select one)

(a) The `find` algorithm may abort, due to dereferencing a `null` pointer.
(b) The `find` algorithm will not abort, but it may return the wrong result.
(c) The `find` algorithm will return the correct result if it terminates, but it might go into an infinite loop.
(d) The `find` algorithm will terminate and return the correct result, but it may take longer than $O(\log n)$ time (in expectation over all random choices).
(e) There will be no negative consequences. The find algorithm will terminate, return the correct result, and run in $O(\log n)$ time (in expectation over all random choices).

(2.6) (3 pts) A scapegoat tree containing $n$ keys has height $O(\log n)$ . . . (select one):

(a) Always—the height is guaranteed.
(b) In expectation, over the algorithm's random choices.
(c) In expectation, assuming that keys are inserted in random order.
(d) In the amortized sense—the average height will be $O(\log n)$ over a long sequence of operations.
(e) Maybe yes, maybe no—there is just no way of knowing.

(2.7) (3 pts) Given an arbitrary B-tree of order 5 with $n$ keys (for some large value of $n$), what is the smallest number of keys that might be stored in any node of the tree?

(a) 1
(b) 2
(c) 3
(d) 4
(e) 5

**Problem 3.** (20 points) In this problem we will consider an enhanced version of a skip list. As usual, each node `p` stores a key, `p.key`, and an array of next pointers, `p.next[]`. To this we add a parallel array `p.span[]`, which contains as many elements as `p.next[]`. This array is defined as follows. If `p.next[i]` refers to a node `q`, then `p.span[i]` contains the distance (number of nodes) from `p` to `q` (at level 0) of the skip list.

For example, see Fig. 2. Suppose that `p` is third node in the skip list (key value "10"), and `p.next[1]` points to the fifth element of the list (key value "13"), then `p.span[1]` would be $5 - 3 = 2$, as indicated on the edge between these nodes.

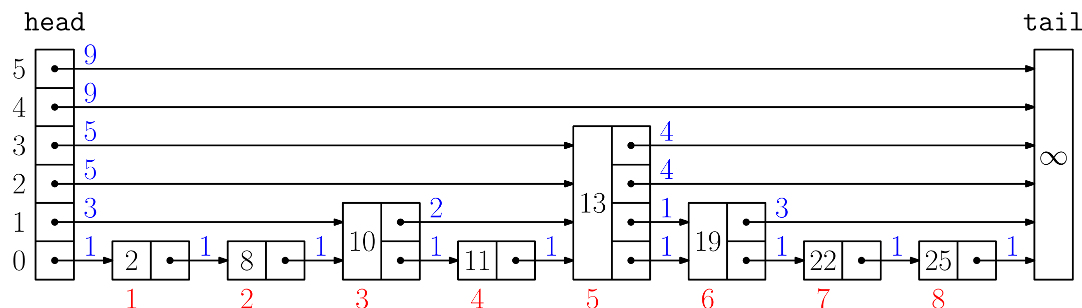



Figure 2: Skip list with span counts (labeled on each edge in blue).

(3.1) (10 points) Present pseudo-code for a function `int countSmaller(Key x)`, which returns a count of the number of nodes in the entire skip list whose key values are strictly smaller than `x`. For example, in Fig. 2, the call `countSmaller(22)` would return 6, since there are six items that are smaller than 22 (namely, 2, 8, 10, 11, 13, and 19).

Your procedure should run in time expected-case time $O(\log n)$ (over all random choices). Briefly explain how your function works.

(3.2) (10 points) Present pseudo-code for a function `Value getMinK(int k)`, which returns the value associated with the $k$th smallest key in the entire skip list. For example, in Fig. 2, the call `getMinK(5)` would return 13, since 13 is the fifth smallest element of the skip list. You may assume that $1 \leq k \leq n$, where $n$ is the total number of nodes in the skip list.

Your procedure should run in time expected-case time $O(\log n)$ (over all random choices). Briefly explain how your function works.

**Problem 4.** (25 points) Consider the following possible node structure for 2-3 trees, where in addition to the keys and children, we add a link to the parent node. The root's parent link is `null`.

```
class Node23 {                     // a node in a 2-3 tree
    int       nChildren            // number of children (2 or 3)
    Node23    child[3]             // our children (2 or 3)
    Key       key[2]               // our keys (1 or 2)
    Node23    parent               // our parent
}
```

Assuming this structure, answer each of the following questions:

(4.1) (5 points) Present pseudocode for a function `Node23 rightSibling(Node23 p)`, which returns a reference to the sibling to the immediate right of node `p`, if it exists. If `p` is the rightmost child of its parent, or if `p` is the root, this function returns `null`. (For example, in Fig. 3, the right sibling of the node containing "2" is the node containing "8:12". Since the node containing "8:12" is the rightmost node of its parent ("4"), it has no right sibling.)

Your function should run in $O(1)$ time.

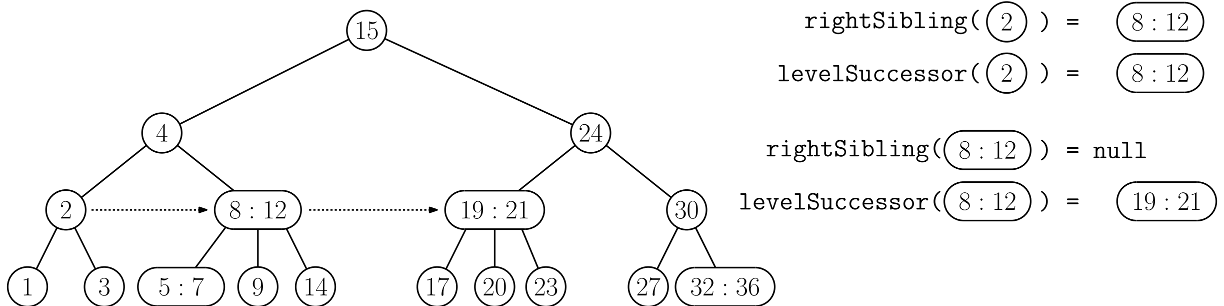



Figure 3: Sibling and level successor in a 2-3 tree.

(4.2) (10 points) For a node `p` in a 2-3 tree, its *level successor* is the node to its immediate right at the same level. Give pseudocode for a function `Node23 levelSuccessor(Node23 p)`, which returns a reference to `p`'s level successor, if it exists. If `p` is the rightmost node on its level (including the case where `p` is the root), this function returns `null`. (For example, in Fig. 3, the level successor of the node containing "2" is the node containing "8:12", and the level successor of "8:12" is the node containing "19:21".)

Your function should run in $O(\log n)$ time. If you like, you may use `rightSibling`.

(4.3) (10 points) Suppose we start at any node `p` in a 2-3 tree with $n$ nodes, and we repeatedly perform `p = levelSuccessor(p)` until `p == null`. What is the (worst-case) total time needed to perform all these operations? (Briefly justify your answer.)

**Problem 5.** (20 points) A *social-distanced bit vector* (SDBV) is an abstract data type that stores bits, but no two 1-bits are allowed to be consecutive. It supports the following operations (see Fig. 4):

- `init(m)`: Creates an empty bit vector `B[0..m-1]`, with all entries initialized to zero.
- `boolean set(i)`: For $0 \leq i \leq m$ (where $m$ is the current size of `B`), this checks whether the bit at positions $i$ and its two neighboring indices, $i - 1$ and $i + 1$, are all zero. If so, it sets the $i$th bit to 1 and returns `true`. Otherwise, it does nothing and returns `false`. (The first entry, `B[0]`, can be set, provided both it and `B[1]` are zero. The same is true symmetrically for the last entry, `B[m-1]`.)

  For example, the operation `set(9)` in Fig. 4 is successful and sets `B[9] = 1`. In contrast, `set(8)` fails because the adjacent entry `B[7]` is nonzero.

There is one additional feature of the SDBV, its ability to *expand*. If we ever come to a situation where it is impossible set any more bits (because every entry of the bit vector is

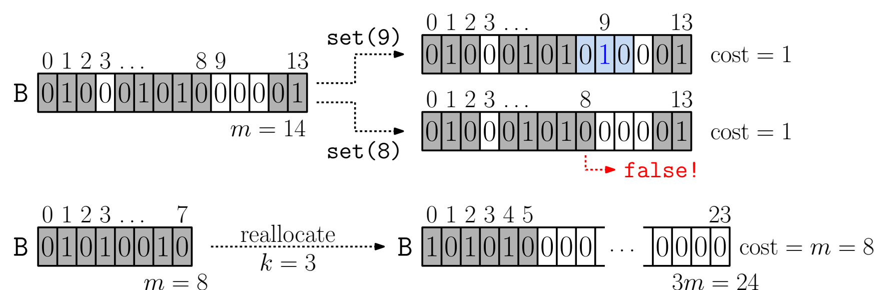



Figure 4: Social-distanced bit vector. (Shaded entries cannot be set to one, due to social-distancing.)

either nonzero or it is adjacent to an entry that is nonzero), we *reallocate* the bit vector to one of three times the current size. In particular, we replace the current array of size $m$ with an array of size $3m$, and we copy all the bits into this new array, compressing them as much as possible. In particular, if $k$ bits of the original vector were nonzero, we set the entries $\{0, 2, 4, \ldots, 2k\}$ to 1, and all others to 0 (see Fig. 4).

The cost of the operation `set` is 1, unless a reallocation takes place. If so, the cost is $m$, where $m$ is the size of the bit vector *before* reallocation.

Our objective is to derive an amortized analysis of this data structure.

(5.1) (4 points) Suppose that we have arrived at a state where we need to reallocate an array of size $m$. As a function of $m$, what is the minimum and maximum number of bits of the SDBV that are set to 1? (Briefly explain.)

(5.2) (4 points) Following the reallocation, what is the minimum number of operations that may be performed on the data structure until the next reallocation event occurs? Express your answer as a function of $m$. (Briefly explain.)

(5.3) (2 points) As a function of $m$, what is the cost of this next reallocation event? (Briefly explain.)

(5.4) (10 points) Derive the amortized cost of the SDBV. (For full credit, we would like a tight constant, as we did in the homework assignment. We will give partial credit for an asymptotically correct answer. Assume the limiting case, as the number of operations is very large and the initial size of the bit vector is small.)

Throughout, if divisions are involved, don't worry about floors and ceilings.

## Homework 3: Hashing, Geometry, Tries

**Problem 1.** In this problem, you will show the result of inserting a sequence of three keys into a hash table, using various open-addressing conflict-resolution methods. In each case, at a minimum you should indicate the following:

- Was the insertion successful? (The insertion fails if the probe sequence loops infinitely without finding an empty slot.)
- Show contents of the hash table after inserting all three keys.
- For each case, give a count of the number of *probes*, that is, the number of entries in the hash table that were accessed in order to find an empty slot in which to perform the insertion. (The initial access counts as a probe, so this number is at least 1. For example, in Fig. 3 in the Lecture 14 LaTeX lecture notes, `insert(z)` makes 1 probe and `insert(t)` makes 4 probes.)

For assigning partial credit, you can illustrate the actual probes that were performed, as we did in Fig. 4 from Lecture 14. But be sure to also list the probe count.

(1.1) Show the results of inserting the keys "X", "Y", and "Z" into the hash table shown in Fig. 1(a), assuming that conflicts are resolved using *linear probing*.

**(a) Linear probing**

```
insert("X")  h("X") = 9
insert("Y")  h("Y") = 1
insert("Z")  h("Z") = 11
```

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| E | H | A | | M | | J | L | | P | C | K | | G | I | W |

**(b) Quadratic probing**

```
insert("M")  h("M") = 4
insert("D")  h("D") = 8
insert("Q")  h("Q") = 3
```

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | F | | | | L | N | P | | | | | | |

Figure 1: Hashing with open addressing.

(1.2) Show the results of inserting the keys "M", "D", and "Q" into the hash table shown in Fig. 1(b) using *quadratic probing*. (Hint: If you are unsure whether quadratic probing has gone into an infinite loop, you may find it helpful to delve into the topic of quadratic residues.)

(1.3) Show the results of inserting the keys "R", "T", and "D" into the hash table shown in Fig. 2(c) using *double hashing*. (The second hash function $g$ is shown in the figure.)

(Intermediate results are not required, but may be given to help assigning partial credit.)

**Problem 2.** You are given a 2-dimensional point kd-tree, as described in Lectures 15 and 16. The operation `float findUp(float x0)` is given a scalar $x_0$, and it returns the smallest $x$-coordinate in the tree that is greater than or equal to $x_0$. If every point has $x$-coordinate strictly smaller than $x_0$, the function returns the special value `Float.MAX_VALUE`.

For example, given the points in Fig. 3, the result of `findUp(x1)` would be $b$'s $x$-coordinate, `findUp(x2))` would be $k$'s $x$-coordinate, and `findUp(x3)` would be `Float.MAX_VALUE`.

**(c) Double hashing**

```
insert("R")  h("R") = 7; g("R") = 3
insert("T")  h("T") = 0; g("T") = 4
insert("D")  h("D") = 8; g("D") = 7
```

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| E | | A | F | M | B | J | L | N | P | | K | U | G | I | W |

Figure 2: Hashing with open addressing.

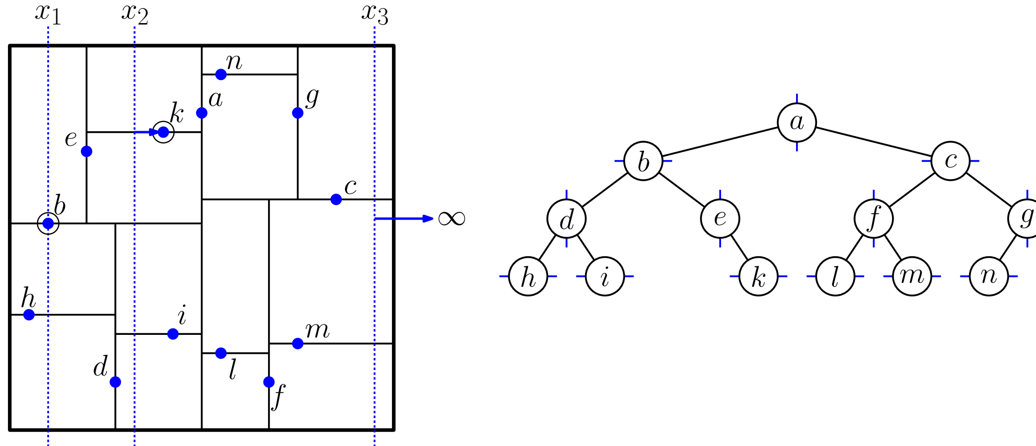



Figure 3: FindUp operation in a point kd-tree.

(2.1) Present pseudocode for this function. (Hint: Use a recursive helper function `float findUp(float x0, KDNode p)`.) Assuming that the tree is balanced and has $n$ points, your function should run in time $O(\sqrt{n})$. (See part 4.2 below.)

(2.2) Derive the running time of your procedure, under the assumption that there are $n$ points and the tree is balanced. (Hint: It may help to recall the analysis of the orthogonal range-search algorithm from Lecture 16. As we did in class, you may make the idealized assumption that the left and right subtrees of each node have equal numbers of points.)

**Problem 3.** Consider the point kd-tree shown in Fig. 4. As in class, assume that the cutting dimensions alternate between $x$ and $y$. Suppose that we perform a range query involving the red rectangle $R$ shown in the figure. Indicate (e.g., by drawing the tree and circling them) which nodes of the tree are visited by the orthogonal-range search algorithm given in class. (A node $p$ is "visited" if a call `rangeCount(R, p, cell)` is made.)

**Problem 4.** In this problem, we will consider how to use/modify range trees to answer two related queries. Given a set $P$ of $n$ points in $d$-dimensional space, recall that a range tree storing these points uses a total of $O(n \log^{d-1} n)$ storage. Given an axis-aligned rectangle $R$ in $d$-dimensional space, in $O(\log^d n)$ time it is possible to identify a set of $O(\log^d n)$ subtrees in the range tree, such that the points lying within these subtrees form a partition of $P \cap R$. (This is the only fact about range trees that you need to solve this problem.)

The input set $P$ is in the $x, y$-plane in all three cases. However, the tree that you construct

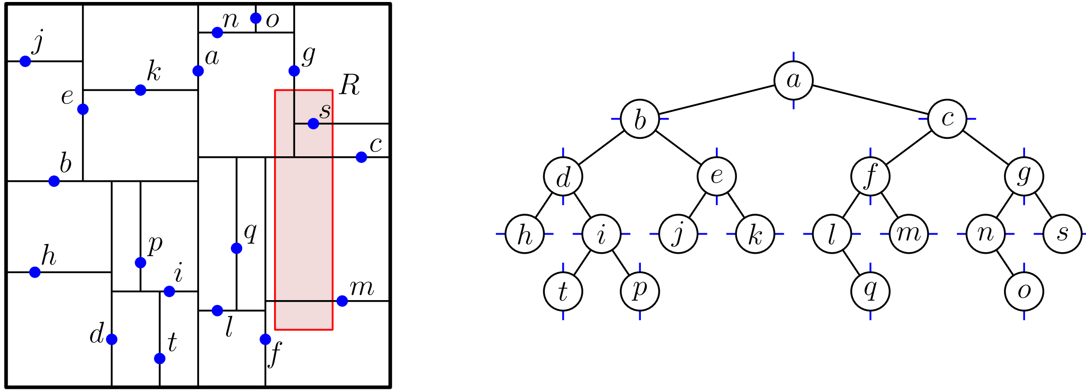



Figure 4: kd-tree operations.

might be for a different set of points, and it may even be in a different dimension. In each case, describe the points that are stored in the range tree and how the search process works. Justify your algorithm's correctness and derive its running time.

(4.1) In a *3-sided min query*, you are given two $x$-coordinates, $x_0$ and $x_1$, where $x_0 < x_1$, and one $y$-coordinate, $y_0$. Among all the points of $P$ whose $x$-coordinates lie within the interval $[x_0, x_1]$ and whose $y$-coordinate is greater than or equal to $y_0$, the answer is the point of $P$ with the smallest $y$-coordinate. If there are no points that satisfy these conditions, the query returns the value `null`. If there are multiple points that satisfy the $x$-conditions and have the same $y$-coordinates, any of them may be returned. (In Fig. 5(a), the query returns point $s$.)

Your data structure should use $O(n \log n)$ storage and answer queries in $O(\log^2 n)$ time.

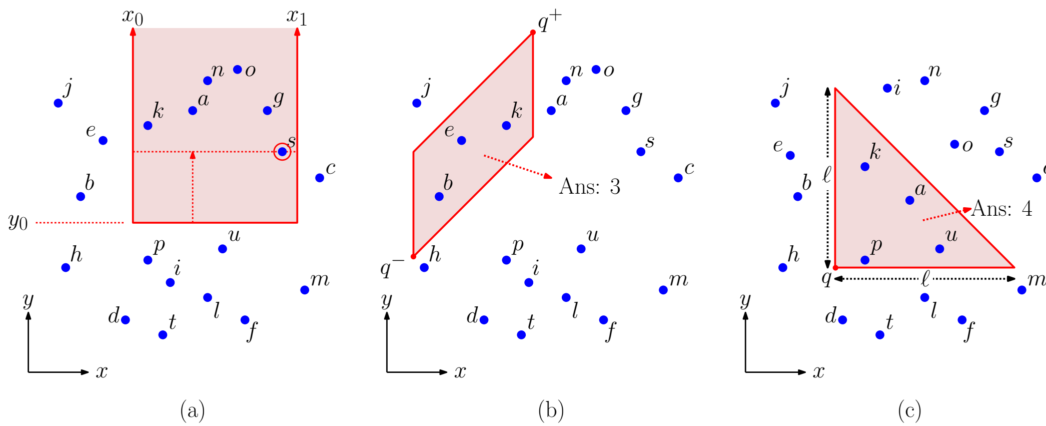



Figure 5: Using range trees to answer various queries. (Part (c) is for the Challenge Problem.)

(4.2) A *skewed rectangle* is defined by two points $q^- = (x^-, y^-)$ and $q^+ = (x^+, y^+)$. The range shape is a parallelogram that has two vertical sides and two sides with a slope of $+1$. The lower left corner is $q^-$ and the upper right corner is $q^+$. The answer to the query is the number of points of $P$ that lie within the parallelogram. (In Fig. 5(b), the answer

to the given query is 3.)

Your data structure should use $O(n \log n)$ storage and answer queries in $O(\log^2 n)$ time.

(Hint: Apply a transformation to space so that any skewed rectangle is mapped to an axis-aligned rectangle. Transform the points of $P$ accordingly and build a range tree with the transformed points.)

**Problem 5.** In this problem we will build a suffix tree for the string $S =$ `babaabaabbabaab$`.

(5.1) List the 16 substring identifiers for the 16 suffixes of $S$. For the sake of uniformity, list them from back to front, so the first substring identifier will be "`$`", and the last will be the substring identifier for the entire string.

(5.2) Draw a picture of the suffix tree. For the sake of uniformity, when drawing your tree, use the conventions of Fig. 7 in the Lecture 19 LaTeX lecture notes. In particular, label edges of the final tree with substrings, index the suffixes from 0 to 15, and order subtrees in ascending lexicographical order ($\texttt{a} < \texttt{b} < \texttt{\$}$).

**Challenge Problem:** Consider the same set-up as in Problem 4. Apply this to the following query.

A *NE right-triangle query* is defined by a point $q = (q_x, q_y) \in \mathbb{R}^2$ and a scalar $\ell > 0$. The NE right triangle has horizontal and vertical sides of length $\ell$ that share a vertex at $q$, and the triangle lies in the northeast quadrant relative to $q$. The answer to the query is the number of points of $P$ that lie within this triangle. (In Fig. 5(c), the answer to the given query is 4.)

Your data structure should use $O(n \log^2 n)$ storage and answer queries in $O(\log^3 n)$ time.

(Hint: Map the problem into a 3-dimensional orthogonal range query. The $z$-coordinate is a function of the $x$ and $y$ coordinates, and it is used to enforce the constraint that points lie to the lower left of the diagonal edge of the triangle.)

CMSC 420:Fall 2020 Dave Mount

**Practice Problems for the Final Exam**

The exam will be asynchronous and online. It is open-book, open-notes, open-Internet, but it must be done on your own without the aid of other people or software.

**Problem 0.** Since the exam is comprehensive, please look back over the previous homework assignments, the midterm exam, and the midterm practice problems. You should expect at least one problem that involves tracing through an algorithm or construction given in class.

**Problem 1.** Short answer questions. Except where noted, explanations are not required but may be given for partial credit.

(a) Let $T$ be extended binary search tree (that is, one having internal and external nodes). You visit the nodes of $T$ according to one of the standard traversals (preorder, postorder, or inorder). Which of the following statements is necessarily true? (Select all that apply.)

(i) In a *postorder traversal*, all the external nodes appear in the order *before* any of the internal nodes

(ii) In a *preorder traversal*, all the internal nodes appear in the order *after* any of the external nodes

(iii) In an *inorder traversal*, internal and external node *alternate* with each other

(iv) None of the above is true

(b) You have an AVL tree containing $n$ keys, and you insert a new key. As a function of $n$, what is the maximum number of rotations that might be needed as part of this operation? (A double rotation is counted as two rotations.) Explain briefly.

(c) Repeat (b) in the case of deletion. (Give your answer as an asymptotic function of $n$.)

(d) Suppose you know that a very small fraction of the keys in a data structure are to be accessed most of the time, but you do not know which these keys are. Among the data structures we have seen this semester, which would be best for this situation? Explain briefly.

(e) In class, we mentioned that when using double hashing, it is important that the second hash function $g(x)$ should not share any common divisors with the table size $m$. What might go wrong if this were not the case?

(f) What is the maximum number of points that can be stored in a 3-dimensional point quadtree of height $h$? Express your answer as an exact (not asymptotic) function of $h$. (Hint: It may be useful to recall the formula for any $c > 1$, $\sum_{i=0}^{m} c^i = (c^{m+1})-1)/(c-1)$.)

(g) We have $n$ uniformly distributed points in the unit square, with no duplicate $x$- or $y$-coordinates. Suppose we insert these points into a kd-tree in *random* order (see the left side of Fig. 1). As in class, we assume that the cutting dimension alternates between $x$ and $y$. As a function of $n$ what is the expected height of the tree? (No explanation needed.)

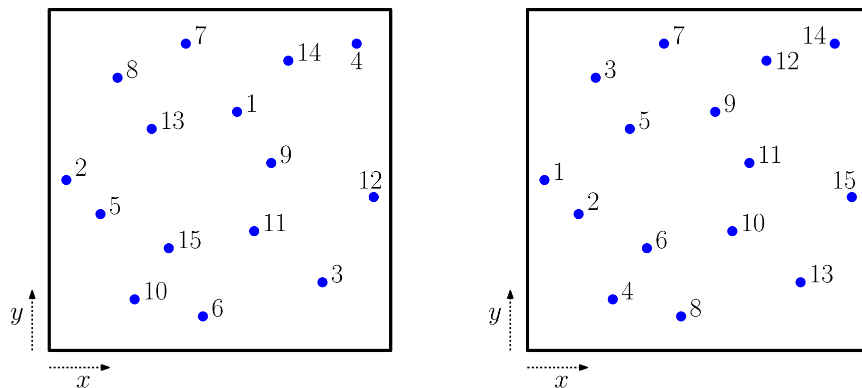



Figure 1: Height of kd-tree.

(h) Same as the previous problem, but suppose that we insert points in *ascending* order of $x$-coordinates, but the $y$-coordinates are *random* (see the right side of Fig. 1). What is the expected height of the tree? (No explanation needed.)

(i) Between the classical dynamic storage allocation algorithm (with arbitrary-sized blocks) or the buddy system (with blocks of size power of 2) which is more susceptible to *internal fragmentation*? Explain briefly.

**Problem 2.** In our simple binary-search tree implementations, we assumed that each node stores just a key-value pair (`p.key` and `p.value`) and pointers to the node's left and right children (`p.left` and `p.right`). In practice, it is often useful to store additional information including the following:

- `p.parent`: $p$'s parent, or `null` if $p$ is the root
- `p.min`: The smallest key in the subtree rooted at $p$
- `p.max`: The largest key in the subtree rooted at $p$
- `p.size`: The total number of nodes (including $p$) in $p$'s subtree

Modify the *right rotation* pseudo-code so that (in addition to the rotation) all of the above associated values are updated. (**Note**: Remember that the return value is significant and should remain $q$.)

**Problem 3.** Define a new treap operation, `expose(Key x)`. It finds the key `x` in the tree (throwing an exception if not found), sets its priority to $-\infty$ (or more practically `Integer.MIN_VALUE`), and then restores the treap's priority order through rotations. (Observe that the node containing `x` will be rotated to the root of the tree.) Present pseudo-code for this operation.

**Problem 4.** In scapegoat trees, we showed that if `size(u.child)/size(u)` $\leq \frac{2}{3}$ for every node of a tree, then the tree's height is at most $\log_{3/2} n$. In this problem, we will generalize this condition to:

$$\frac{\texttt{size(u.child)}}{\texttt{size(u)}} \leq \alpha, \qquad (*)$$

for some constant $\alpha$.

(a) Why does it **not** make sense to set $\alpha$ larger than 1 or smaller than $\frac{1}{2}$?

(b) If every node of an $n$-node tree satisfies condition $(*)$ above, what can be said about the height of the tree as a function of $n$ and $\alpha$? Briefly justify your answer.

**Problem 5.** We say that an extended binary search tree is *geometrically-balanced* if the splitter value stored in each internal node `p` is midway between the smallest and largest keys of its external nodes. More formally, if the smallest external node in the subtree rooted at `p` has the value $x_{\min}$ and the largest external node has the value $x_{\max}$, then `p`'s splitter is $(x_{\min}+x_{\max})/2$ (see Fig. 2).

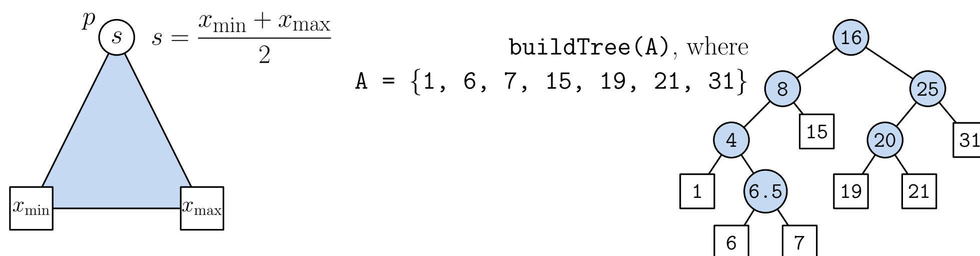



Figure 2: Geometrically balanced tree.

Given a sorted array $A[0 \ldots n-1]$ containing $n \geq 1$ numeric keys, present pseudo-code for a function that builds a geometrically-balanced extended binary search tree, whose external nodes are the elements of $A$. **Convention:** If a key is *equal* to an internal node's splitter value, then the key is stored in the *left subtree*.

Briefly explain any assumptions you make about underlying primitive operations (e.g., constructors for your internals and external nodes). Any running time is okay.

**Problem 6.** Given a set $P$ of $n$ points in the real plane, a *partial-range max query* is given two $x$-coordinates $x_1$ and $x_2$, and the problem is to find the point $p \in P$ that lies in the vertical strip bounded by $x_1$ and $x_2$ (that is, $x_1 \leq p.x \leq x_2$) and has the maximum $y$-coordinate (see Fig. 3).

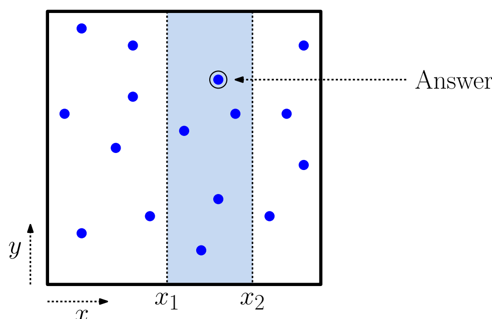



Figure 3: Partial-range max query.

Present pseudo-code for an efficient algorithm to solve partial-range max queries, assuming that the points are stored in a kd-tree. You may make use of any primitive operations on

points and rectangles (but please explain them). You may assume that there are no duplicate coordinate values, and no coordinates are equal to $x_1$ and $x_2$. If you solve the problem recursively, indicate what the initial call is from the root level. If the tree is balanced, your algorithm should run in time $O(\sqrt{n})$.

**Problem 7.** In class we demonstrated a simple idea for deleting keys from a hash table with open addressing. Namely, whenever a key is deleted, we stored a special value "`deleted`" in this cell of the table. It indicates that this cell contained a deleted key. The cell may be used for future insertions, but unlike "`empty`" cells, when the probe sequence searching for a key encounters such a location, it should continue the search.

Suppose that we are using *linear probing* in our hashing system. Describe an alternative approach, which does not use the "`deleted`" value. Instead it moves the table entries around to fill any holes caused by a deleted items.

In addition to explaining your new method, justify that dictionary operations are still performed correctly. (For example, you have not accidentally moved any key to a cell where it cannot be found!)

**Problem 8.** In class we showed that for a balanced kd-tree with $n$ points in the real plane (that is, in 2-dimensional space), any *axis-parallel line* intersects at most $O(\sqrt{n})$ cells of the tree.

The purpose of this problem is to show that does not apply to lines that are not axis-parallel. Show that for every $n$, there exists a set of points $P$ in the real plane, a kd-tree of height $O(\log n)$ storing the points of $P$, and a line $\ell$, such that *every* cell of the kd-tree intersects this line.

**Problem 9.** In applications where there is a trade-off to be faced, a common query involves a set called the *Pareto maxima*.[1] Given a set of 2-dimensional data, we say that a point $q$ *dominates* another point $q'$ if $q_x > q'_x$ and $q_y > q'_y$. The set of points of $P$ that are not dominated by any other point of $P$ are called the *Pareto maxima* (the highlighted points of Fig. 4(a)). As seen in the figure, these points naturally define a "staircase" shape.

Given a 2-dimensional point set $P$ and a query point $q = (q_x, q_y)$ define $q$'s *Pareto predecessor* to the point $(x, y) \in P$ such that $x \leq q_x$, $y \geq q_y$, and among all such points, $x$ is maximum. An more visual way of think about the Pareto predecessor is as the rightmost point in the subset of $P$ lying in $q$'s northwest quadrant (see Fig. 4(b)).

Assuming that the points of $P$ are stored in a kd-tree $T$, present pseudo-code for a function `T.paretoPred(Point q)`, which returns the Pareto predecessor of a query point $q$.

Hint: The recursive function to compute the predecessor has the following structure:

```
Point paretoPred(Point q, KDNode p, Rectangle cell, Point best),
```

[1] To motivate this, suppose that you are a policy maker and you have set of energy technologies to chose from a (coal, nuclear, wind, solar) where each has an associated cost of deployment and environmental impact. Some alternatives are inexpensive to deploy but have a high negative impact on the environment, and others are more expensive to deploy but have a lower impact on the environment. Clearly, we are not interested in any technology that is "dominated" by another technology that is both less expensive and has a lower environmental impact.

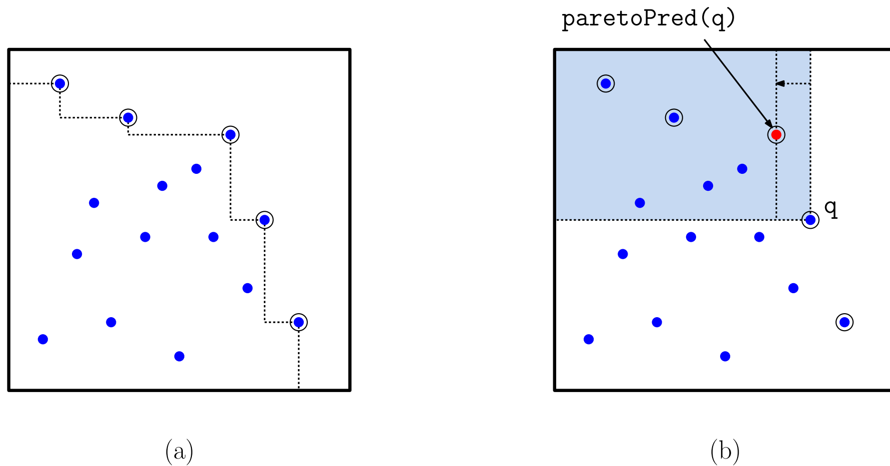



Figure 4: (a) the Pareto maxima and (b) the Pareto predecessor.

where `q` is the query point, `p` is the current node of the kd-tree being visited, `cell` is the rectangular cell associated with the current node, and `best` is the rightmost point encountered so far in the search that satisfies the Pareto criteria.

**Problem 10.** In this problem we will see how to use kd-trees to answer a common geometric query, called *ray shooting*. You are given a collection of vertical line segments in 2D space, each starts at the $x$-axis and goes up to a point in the positive quadrant. Let $P = \{p_1, \ldots, p_n\}$ denote the upper endpoints of these segments (see Fig. 5). You may assume that both the $x$- and $y$-coordinates of all the points of $P$ are strictly positive real numbers.

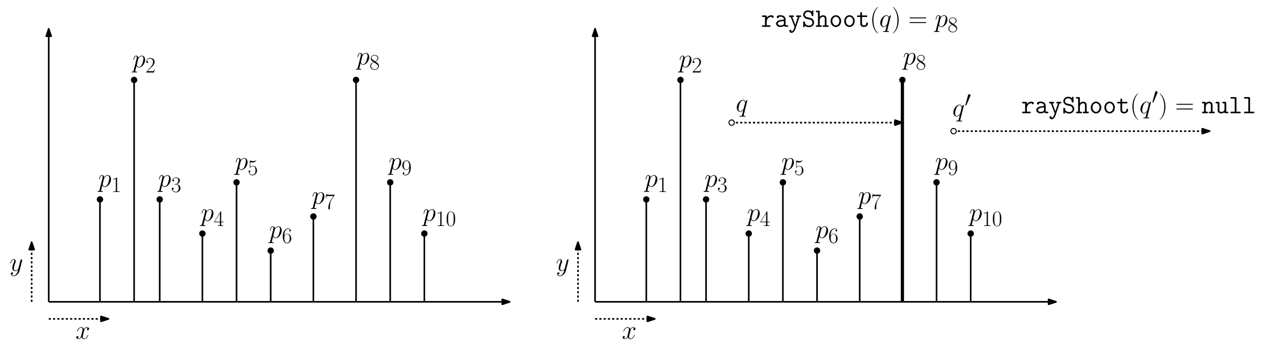



Figure 5: Ray shooting in a kd-tree.

Given a point $q$, we shoot a horizontal ray emanating from $q$ to the right. This ray travels until it hits one of these segments (or perhaps misses them all). For example, in the figure above, the ray shot from $q$ hits the segment with upper endpoint $p_8$. The ray shot from $q'$ hits nothing.

In this problem we will show how to answer such queries using a standard point kd-tree for the point set $P$. A query is given the point $q = (q_x, q_y)$, and it returns the upper endpoint $p_i \in P$ of the segment the ray first hits, or `null` if the ray misses all the segments.

Suppose you are given a kd-tree of height $O(\log n)$ storing the points of $P$. (It does *not* store the segments, just the points.) Present pseudo-code for an efficient algorithm, `rayShoot(q)`, which returns an answer to the horizontal ray-shooting query (see the figure above, right).

You may assume the kd-tree structure given in class, where each node stores a point `p.point`, a cutting dimension `p.cutDim`, and left and right child pointers `p.left` and `p.right`, respectively. You may make use of any primitive operations on points and rectangles (but please explain them). You may assume that there are no duplicate coordinate values among the points of $P$ or the query point.

**Hint:** `rayShoot(q)` will invoke a recursive helper function. Here is a suggested form, which you are *not* required to use:

```
Point rayShoot(Point2D q, KDNode p, Rectangle cell, Point best),
```

Be sure to indicate how `rayShoot(q)` makes its initial call to the helper function.

**Problem 11.** Recall the buddy system of allocating blocks of memory (see Fig. 6). Throughout this problem you may use the following standard bit-wise operators:

| | | | |
|---|---|---|---|
| `&` | bit-wise "and" | `\|` | bit-wise "or" |
| `^` | bit-wise "exclusive-or" | `~` | bit-wise "complement" |
| `<<` | left shift (filling with zeros) | `>>` | right shift (filling with zeros) |

You may also assume that you have access to a function `bitMask(k)`, which returns a binary number whose $k$ lowest-order bits are all 1's. For example `bitMask(3)` $= 111_2 = 7$.

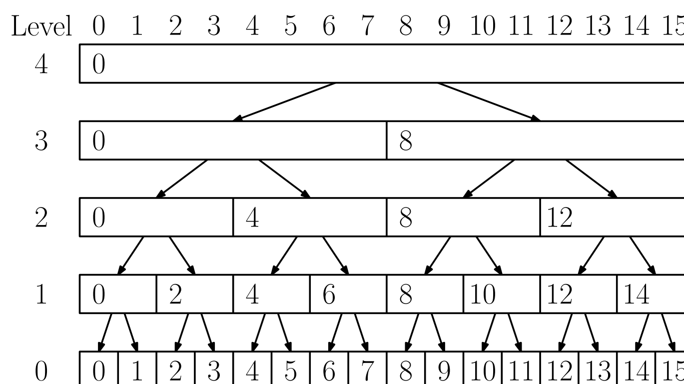



Figure 6: Buddy relatives.

Present a short (one-line) expression for each of the following functions in terms of the above bit-wise functions:

(a) `boolean isValid(int k, int x)`: True if and only if $x \geq 0$ a valid starting address for a buddy block at level $k \geq 0$.

(b) `int sibling(int k, int x)`: Given a valid buddy block of level $k \geq 0$ starting at address $x$, returns the starting address of its *sibling*.

(c) `int parent(int k, int x)`: Given a valid buddy block of level $k \geq 0$ starting at address $x$, returns the starting address of its *parent* at level $k+1$.

(d) `int left(int k, int x)`: Given a valid buddy block of level $k \geq 1$ starting at address $x$, returns the starting address of its *left child* at level $k-1$.

(e) `int right(int k, int x)`: Given a valid buddy block of level $k \geq 1$ starting at address $x$, returns the starting address of its *right child* at level $k-1$.

**Problem 12.** Suppose you have a large span of memory, which starts at some address `start` and ends at address `end-1` (see Fig. 7). (The variables `start` and `end` are generic pointers of type `void*`.) As the dynamic memory allocation method of Lecture 15, this span is subdivided into blocks. The block starting at address `p` is associated with the following information:

- `p.inUse` is 1 if this block is in-use (allocated) and 0 otherwise (available)
- `p.prevInUse` is 1 if the block immediately preceeding this block in memory is in-use. (It should be 1 for the first block.)
- `p.size` is the number of words in this block (including all header fields)
- `p.size2` each available block has a copy of the size stored in its last word, which is located at address `p + p.size - 1`.

(For this problem, we will ignore the available-list pointers `p.prev` and `p.next`.)

In class, we said that in real memory-allocation systems, blocks cannot be moved, because they may contain pointers. Suppose, however, that the blocks are movable. Present pseudo-code for a function that compacts memory by copying all the allocated blocks to a single contiguous span of blocks at the start of the memory span (see Fig. 7). Your function `compress(void* start, void* end)` should return a pointer to the head of the available block at the end. Following these blocks is a single available block that covers the rest of the memory's span.

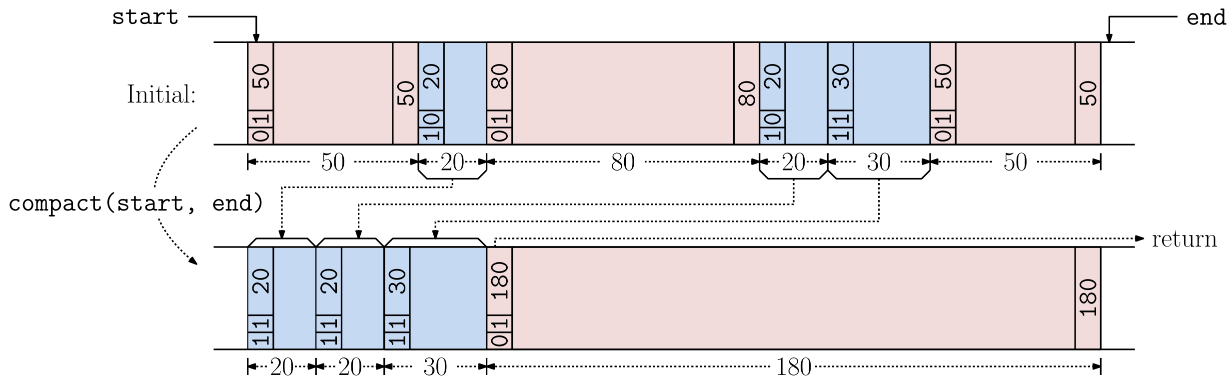



Figure 7: Memory compactor.

To help copy blocks of memory around, you may assume that you have access to a function `void* memcpy(void* dest, void* source, int num)`, which copies `num` words of memory from the address `source` to the address `dest`.

CMSC 420:Fall 2020 Dave Mount

**Final Exam**

The exam is asynchronous and online. It is open-book, open-notes, open-Internet, but it must be done on your own without the aid of other people or software. The total point value is 120 points. Good luck!

**Problem 1.** (35 points) Short answer questions. Except where noted, explanations are not required but may be given for partial credit.

(1.1) (4 pts) In 2-3 tree deletion, under what circumstances do you perform a merge as opposed to adoption (key rotation)?

(1.2) (4 pts) Although AA trees are a variant of red-black trees, nodes are not colored. What is the condition that determines whether a node in an AA tree is red or black?

(1.3) (4 pts) You access an element in a splay tree, and then a few operations later you access the very same element. Which property of splay trees best explains why sequences of operations like this are processed efficiently? (Choose from: static optimality, static finger theorem, dynamic finger theorem, working set theorem, scanning theorem)

(1.4) (4 pts) What is the purpose of the *next-leaf* pointer in B+ trees?

(1.5) (4 pts) We claimed that scapegoat trees are efficient in the amortized sense, but `find` operations in scapegoat trees are efficient even in the worst case. Why is this?

(1.6) (10 pts) You are using hashing with open addressing. Suppose that the table has just one empty slot in it. In which of the following cases are you *guaranteed* to succeed in finding the empty slot? (Select all that apply.)

(a) Linear probing (under any circumstances)

(b) Quadratic probing (under any circumstances)

(c) Quadratic probing, where the table size $m$ is a prime number

(d) Double hashing (under any circumstances)

(e) Double hashing, where the table size $m$ and hash function $h(x)$ are relatively prime

(f) Double hashing, where the table size $m$ and secondary hash function $g(x)$ are relatively prime

(1.7) (5 pts) In the unstructured memory management system described in Lecture 20, what was the purpose of the `size2` field at the end of each free block? (Why was it needed?)

**Problem 2.** (20 points) This problem involves the tree shown in Fig. 1.

(2.1) (4 points) List the nodes according to a *preorder traversal*

(2.2) (4 points) List the nodes according to an *inorder traversal*

(2.3) (4 points) List the nodes according to a *postorder traversal*

(2.4) (8 points) Show the final result of applying the operation `splay(8)` on this tree. (You need only show the final result. Intermediate results can be shown to help with partial credit.)

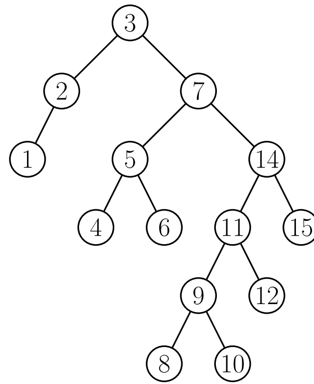



Figure 1: Tree traversal and splaying.

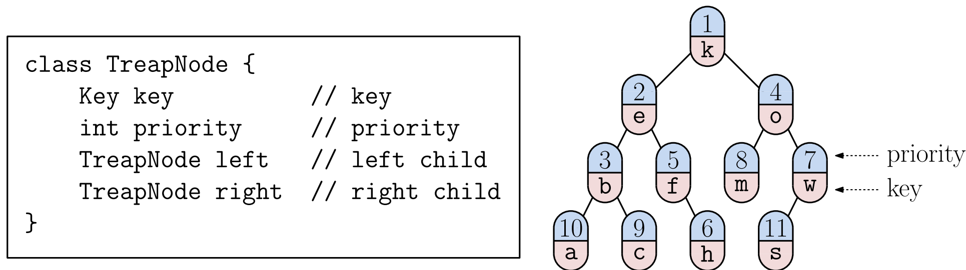



Figure 2: Treap node structure and an example.

**Problem 3.** (10 points) Suppose that you are given a treap data structure storing $n$ keys. The node structure is shown in Fig. 2. You may assume that *all keys and all priorities are distinct.*

(3.1) (7 points) Present pseudocode for the operation `int minPriority(Key x0, Key x1)`, which is given two keys $x_0$ and $x_1$ (which may or may not be in the treap), and returns the lowest priority among all nodes whose keys $x$ lie in the range $x_0 \leq x \leq x_1$. If the treap has no keys in this range, the function returns `Integer.MAX_VALUE`. Briefly explain why your function is correct.

For example, in Fig. 2 the query `minPriority("c", "g")` would return 2 from node `"e"`, since it is the lowest priority among all keys $x$ where `"c"` $\leq x \leq$ `"g"`.

(3.2) (3 points) Assuming that the treap stores $n$ keys and has height $O(\log n)$, what is the running time of your algorithm? (Briefly justify your answer.)

**Problem 4.** (15 points) In this problem we will build a suffix tree for $S =$ `aabbabaabbbaaba$`.

(4.1) (4 points) List the 16 substring identifiers for the 16 suffixes of $S$. For the sake of uniformity, list them in order (either back to front or front to back). For example, you could start with "`$`" and end with the substring identifier for the entire string.

(4.2) (8 points) Draw a picture of the suffix tree. For the sake of uniformity, when drawing your tree, use the convention of Fig. 7 in the Lecture 19 LaTeX lecture notes. In particular, label edges of the final tree with substrings, index the suffixes from 0 to 15, and order subtrees in ascending lexicographical order (`a` $<$ `b` $<$ `$`).

(4.3) (3 points) Draw the root and the first few levels of your suffix tree using the convention of Fig. 5 in the Lecture 19 LaTeX lecture notes. In particular, label each node with the index field and label each edge with a single character. (It is not necessary to show the entire tree, but you may. It suffices to show the root, its children, and its grandchildren.)

**Problem 5.** (15 points) Throughout this problem we are given a set $P = \{p_1, \ldots, p_n\}$ of $n$ points in 2D space stored in a point kd-tree (see Fig. 3(a)).

(5.1) (10 points) In a *segment sliding*, you are given a vertical line segment, specified by its lower endpoint $q$ and its height $h$ (see Fig. 3(b)). The query returns the first point $p_i \in P$ that is first hit if we slide the segment to its right. If no point of $P$ are hit, the query returns `null`.

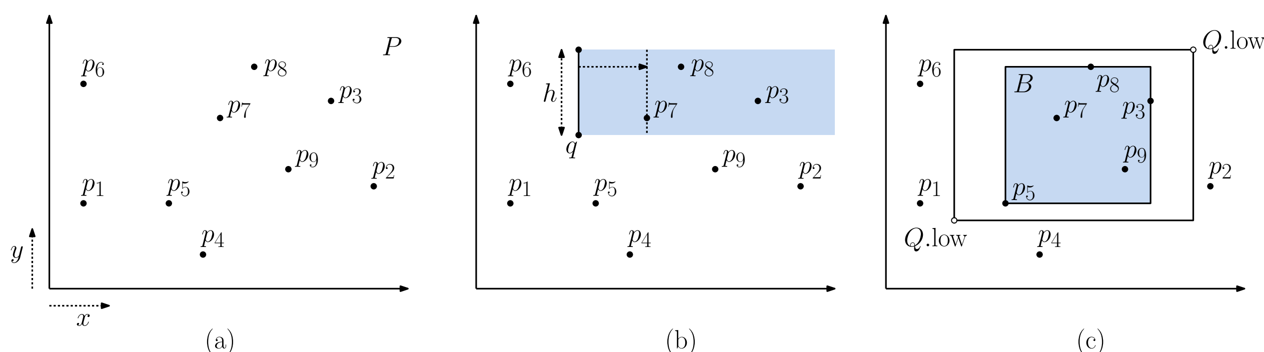



Figure 3: Segment-sliding and minimum box queries.

Give pseudo-code for an efficient algorithm, `Point segSlideRight(Point q, float h)`, which given the coordinates of the line segment, returns the answer to the segment-sliding query.

You may assume the standard kd-tree structure given in class, where each node stores a point `p.point`, a cutting dimension `p.cutDim`, and left and right child pointers `p.left` and `p.right`, respectively. You may make use of any primitive operations on points and rectangles. You may assume that there are no duplicate coordinate values among the points of $P$ or the query point.

(5.2) (5 points) In a *minimum box query*, you are given an axis-aligned rectangle $Q$ as input (given, say, by its corner points $Q$.low and $Q$.high), and the output is the smallest axis-aligned rectangle $B$ that contains all the points of $P$ that lie within $Q$ (see Fig. 3(c)). If there are no points of $P$ in $Q$, the query returns `null`. Present pseudo-code for an efficient algorithm, `Rectangle minBox(Rectangle Q)`, which given the query rectangle $Q$, returns the answer to the minimum box query. (You may reuse or modify your solution to (5.1).)

**Problem 6.** (25 points) In this problem, we will consider how to use/modify range trees to answer related queries. Given a set $P$ of $n$ points in $d$-dimensional space, recall that a range tree storing these points uses a total of $O(n \log^{d-1} n)$ storage. Given an axis-aligned rectangle $R$ in $d$-dimensional space, in $O(\log^d n)$ time it is possible to identify a set of $O(\log^d n)$ subtrees in the range tree, such that the points lying within these subtrees form a partition of $P \cap R$.

In all cases, the input set $P$ is a set of $n$ points with positive coordinates. Note that the tree that you construct might be for a different set of points, and it may even be in a different dimension. In each case, describe the points that are stored in the range tree and how the search process works. Justify your algorithm's correctness and derive its running time.

(6.1) (10 points) In a *skewed interval min query* (SIM query), you are given two $y$-coordinates, $y_0$ and $y_1$, where $y_0 < y_1$. Among all the points of $P$ whose $y$-coordinate lies within the interval $[y_0, y_1]$, the answer is the point $p_i = (x_i, y_i)$ that minimizes $x_i + y_i$. More visually, among all the points of $P$ that lie within the horizontal strip $y_0 \leq y \leq y_1$, find the point that is first hit by a line of slope $-1$ sweeping up from the origin. If there are no points in the strip, the query returns the value `null`. If there are multiple points that satisfy the $x$-conditions and have the same $y$-coordinates, any of them may be returned. (In Fig. 4(a), the query returns point $b$.)

Your data structure should use $O(n \log n)$ storage and answer queries in $O(\log^2 n)$ time.

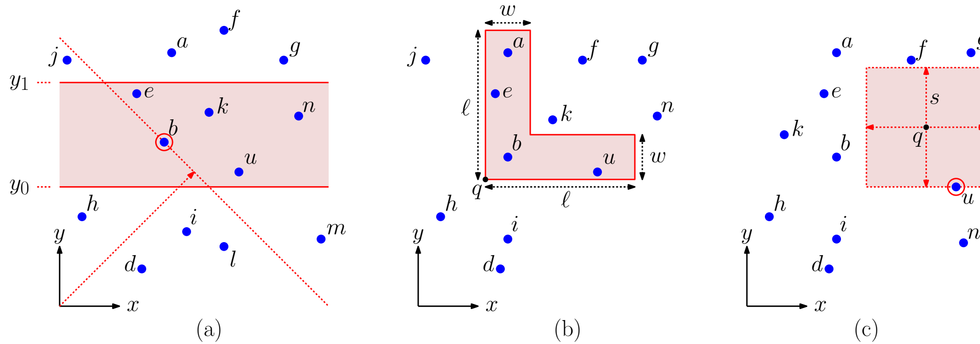



Figure 4: Using range trees to answer various queries.

(6.2) (5 points) An *L-shaped counting query* involves counting all the points lying within an L-shaped region. The query is described by three parameters, the lower-left corner $q$ of the L-shape, the length $\ell$ which is the total height and width of the shape, and the width $w$ of the segments of the L-shape (see Fig. 4(b)). The query returns a count of all the points that lie within this region.

Your data structure should use $O(n \log n)$ storage and answer queries in $O(\log^2 n)$ time.

(6.3) (10 points) A *maximum empty square query* is given a query point $q$, and it returns half the side length of the largest box centered at $q$ that contains no point $P$ in its interior. If $q$ coincides with a point of $P$, the query returns 0. (In Fig. 4(c), the query returns the value $s$.)

Your data structure should use $O(n \log^2 n)$ storage and answer queries in $O(\log^3 n)$ time.

**Programming Assignment 1: Extended Binary Search Trees**

**Overview:** In this assignment you will implement an extended version of a standard (unbalanced) binary search tree. (This data structure is discussed in class on 09/24.) Recall that an extended binary tree is one in which all internal nodes have two non-null children, and all the leaves are called external nodes. An extended binary search tree (which we will call *XBSTree* for short), is an alternative implementation of an ordered dictionary, storing key-value pairs. It differs from the traditional binary search tree in that the internal nodes do not store values, only keys. We refer to these as *splitters*. These splitters do not necessarily correspond to keys in the dictionary, they are merely used as an *index* to help us locate the external node that contains a given key-value pair. Separating the indexing (internal nodes) from the key-value storage (external nodes) has a number of practical advantages, which we will mention later this semester.

**Defintion:** Given an internal node storing a splitter value $x$, we make the convention that the key-value pairs whose keys are strictly smaller than $x$ are stored its left subtree and those whose keys are greater than or equal to $x$ are stored in its right subtree (see Fig. 1(a)).

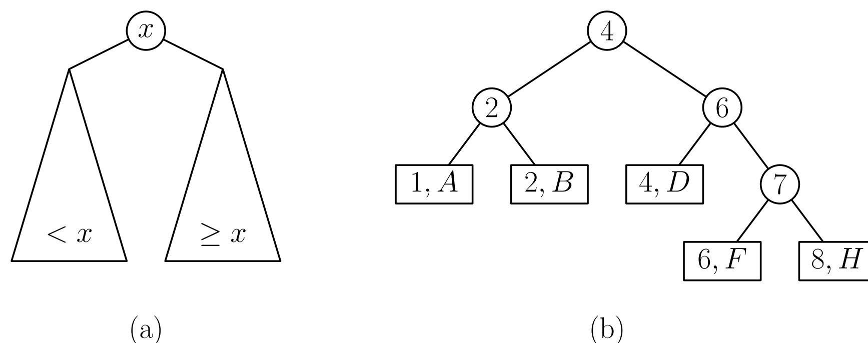



Figure 1: (a) Convention for internal nodes, and (b) a possible extended binary search tree storing the key-value pairs $\{(1, A), (2, B), (4, D), (6, F), (8, H)\}$.

An example of such a tree is shown in Fig. 1(b). Observe that the splitter values may or may not correspond to keys that appear in the leaf nodes. It is important to note that splitter values that do not arise as an actual key value in an external node (such as 7 in Fig. 1(b)) are not present in the dictionary. Thus, the operation `find(7)` would return `null`, and an attempt to insert a key-value pair with the same key value, such as $(7, G)$, would be allowed.

**Operations:** The operations for extended binary search trees are very similar to the standard trees we have seen. Here is a formal definition of how the operations work.

**Initialization:** The initial tree consists of a `root` pointer whose value is `null`. If the tree consists of a single key-value pair, the root points directly to an external node containing this pair. Once the tree has two or more entries, the root will point to an internal node.

`find(x)`: Starting at the root, we traverse the path to a leaf in the natural manner for a binary search tree. (If the root is `null`, then the search fails immediately.) Given an internal node with key $y$, we recurse on the left subtree if $x < y$ and on the right subtree if $x \geq y$. On arriving at an external node, we report success if $x$ matches the key there and failure otherwise. If we succeed, we return the value associated with this pair, and otherwise we return `null`.

`insert(x, v)`: If the root is `null` (meaning that the tree is empty), we create a single external node containing the pair $(x, v)$ and set the root to point to this node. Otherwise, we apply the same process as in `find` to determine the closest leaf node, say $(y, w)$. If $x = y$, then we generate a duplicate-key error. Otherwise, we create an new external node containing the pair $(x, v)$. There are two cases for how to connect it to the tree. If $x < y$, we create a new internal node in which we store the splitter value $y$. We make $(x, v)$ its left child, $(y, w)$ its right child, and this new internal node replaces $(y, w)$ as the child of its parent (see Fig. 1(a)). On the other hand, if $x > y$, we create a new internal node in which we store the splitter value $x$. We make $(y, w)$ its left child, $(x, v)$ its right child, and this new internal node replaces $(y, w)$ as the child of its parent.

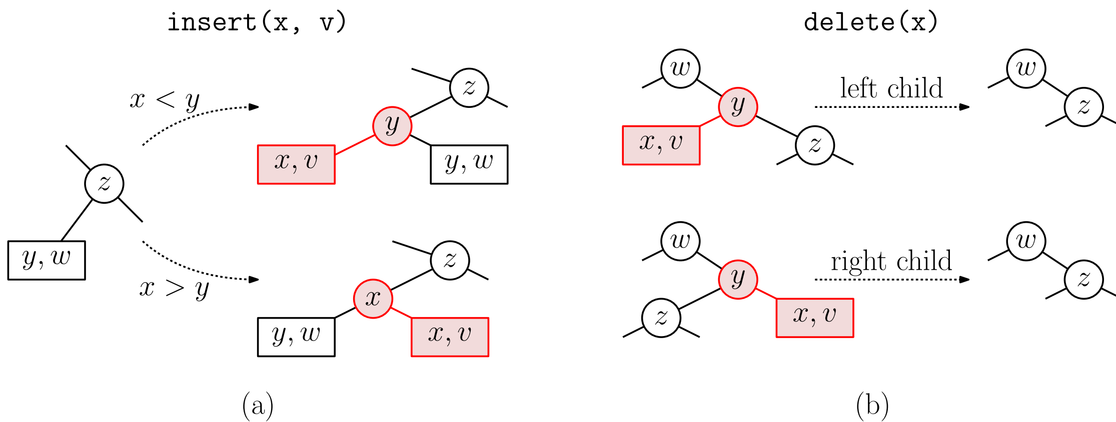



Figure 2: Insertion and deletion.

Observe that both cases are consistent with our convention that the key-value pairs in the left subtree are strictly smaller than the splitter value and those in the right subtree are greater than or equal to the splitter.

`delete(x)`: As usual, we apply the same process as in `find` to locate the leaf containing $x$. If $x$ does not appear in this leaf, we generate a nonexistent-key error. If this external node has no parent (implying that the tree has only one entry), we delete this node and set the root to `null`. Otherwise, let $y$ denote its parent, and let $z$ denote the other child of $y$. (Note that $y$ must be an internal node, so by definition of extended trees, it has two non-null children.) We delete both the nodes $(x, v)$ and $y$, and make $y$'s parent point to $z$, instead of $y$ (see Fig. 2(b)).

`getPreorderList()`: This operation generates a preorder enumeration of the nodes in the tree. This is represented as a Java `ArrayList` of type `String`. We assume that both keys and values support a `toString` method. For each internal node storing a key $x$, the

`ArrayList` entry is of the form `"(" + x.toString() + ")"`. For each external node storing the pair $(x, v)$, the entry is `"[" + x.toString() + " " + v.toString + "]"`. (Our testing programs are based on Java `String` equality. Note the use of parentheses for internal nodes and square brackets for external nodes. Also, note the single space between the key and value.)

For example, given the tree of Fig. 1(b), the elements of the `ArrayList` would consist of the following strings:

```
(4)
(2)
[1 A]
[2 B]
(6)
[4 D]
(7)
[6 F]
[8 H]
```

`getMin()` / `getMax()`: These two functions return the values associated with the smallest and largest key values of the dictionary. For example, given the tree of Fig. 1(b), these would return the values `A` and `H`, respectively. If the dictionary is empty, these return `null`.

`findDown(x)` / `findUp(x)`: Recall that the `find` function returns `null` if there is no entry with key $x$. These functions behave exactly like `find(x)` if there is a key-value pair whose key is $x$. If there is not such key-value pair, the `findDown` function returns the value associated with the *next smaller key* and `findUp` returns the value associated with the *next larger key*. If there is no such key (e.g., if $x$ is smaller than the smallest key in the case of `findDown` or $x$ is larger than the largest key in the case of `findUp`), the function returns `null`. Here are some examples for the tree of Fig. 1(b),

| | | | |
|---|---|---|---|
| findDown(0) = | null | findUp(0) = | A |
| findDown(2) = | B | findUp(2) = | B |
| findDown(3) = | B | findUp(3) = | D |
| findDown(7) = | F | findUp(7) = | H |
| findDown(8) = | H | findUp(8) = | H |
| findDown(10) = | H | findUp(10) = | null |

`clear()`: This removes all the entries from the dictionary, resulting in an empty tree.

**Running-time Requirements:** All the dictionary operations (`insert`, `delete`, `find`, `getMin`, `getMax`, `findUp`, and `findDown`) should run in time proportional to the height of the tree. The operation `getPreorderList` should run in time proportional to the number of nodes in the tree. We will determine this by inspection of your submitted code.

**Program structure:** We will provide a driver program that will input a set of commands. You need only implement the data structure and the functions listed above. In particular, you will implement a class called `XBSTree`, which has the following public interface:

```
package cmsc420_f20;
```

```
public class XBSTree<Key extends Comparable<Key>, Value> {

    public XBSTree() { ... } // constructs an empty tree
    public Value find(Key x) { ... }
    public void insert(Key x, Value v) throws Exception { ... }
    public void delete(Key x) throws Exception { ... }
    public ArrayList<String> getPreorderList() { ... }
    public Value getMin() { ... }
    public Value getMax() { ... }
    public Value findDown(Key x) { ... }
    public Value findUp(Key x) { ... }
    public void clear() { ... }

}
```

Observe that the `XBSTree` class is parameterized with two types, `Key` and `Value`. We assume that both of these objects implement a `toString` method. The `Key` type implements `Comparable<Key>`, which means that it defines a comparator function, `compareTo`. To determine whether key `x` is less than key `y`, use `x.compareTo(y) < 0`.

**Node structure:** The most natural way to represent nodes is through inheritance (and we will check for this in grading). There should be a parent class, called say `Node`, from which we derive two subclasses, one for each type of node. Let's call them `InternalNode` and `ExternalNode` (but you can call them whatever you like). Only internal and external nodes will ever be generated (that is, the parent class is abstract). Thus, the `Node` methods are all declared to be "`abstract`," meaning that you don't supply a function body for them.

```
public class XBSTree<Key extends Comparable<Key>, Value> {

    private abstract class Node { // generic node (purely abstract)
        Key key;
        abstract Value find(Key x); // no function body for these!
        abstract Node insert(Key x, Value v) throws Exception;
        abstract Node delete(Key x) throws Exception;
    }

    private class InternalNode extends Node { // internal node
        Node left;
        Node right;
        Value find(Key x) { ... }
        Node insert(Key x, Value v) throws Exception { ... }
        Node delete(Key x) throws Exception { ... }
        // ... other functions omitted
    }

    private class ExternalNode extends Node { // external node
        Value value;
        Value find(Key x) { ... }
        Node insert(Key x, Value v) throws Exception { ... }
```

```
            Node delete(Key x) throws Exception { ... }
            // ... other functions omitted
        }

        // ... public functions (see above)
    }
```

These are just examples. You are allowed to augment the above node structure by adding additional data fields, modifying the function signatures, or adding additional methods. Because recursive methods such as insert, find, and delete will depend on the type of node, the recursive utility functions should be member functions associated with each node type, rather than passing a reference to the node as an argument. For example, in class, when inserting on the right child of a node `p`, we used the command `p.left = insert(x, v, p.left)`. Using the above structure, the `insert` method associated with the `InternalNode` would have the following command instead: `left = left.insert(x, v)`.

**Actual Test Data?** What are the actual key and value types that we will use in our testing. The short answer is the you should not know and you should not care. Your program should work correctly, subject to the above assumptions and nothing else.

For this part of the assignment, our tests will involve a data set drawn from a database of airports. The keys will be three-letter codes (so called, IATA codes), such as "`DCA`", "`IAD`", "`LAX`", and such. The values will be stored in a class that contains related information, such as the airport's name, its city, country, and latitude and longitude. The only attributes that you will observe in our test output files are the IATA code and the airport's city. However, none of these will affect your implementation.

**Skeleton Code:** We will provide you with some skeleton code to start with. This consists of the following:

`XBSTree.java`: A skeletal version of the main class for the extended binary search tree. **This is the only file you need modify.**

`Airport.java`: A class that stores information about airports

`Point2D.java`: A small utility class for storing $(x, y)$ coordinates. Used for each airport's latitude and longitude.

`XBSTreeTester.java`: Main program for testing your implementation. It inputs commands either from a file or standard input and sends output to another file or standard output.

`CommandHandler.java`: A class that processes commands that are read from the input file and produces the appropriate function calls to the member functions of your `XBSTree` class. This also contains a function that converts the output of the `getPreorderList` operation into an indented inorder traversal of the tree, which is a bit easier to read. Here is an example of the output for the above tree:

```
Tree structure:
  | | [1 A]
  | (2)
  | | [2 B]
  (4)
```

```
| | [4 D]
| (6)
| | | [6 F]
| | (7)
| | | [8 H]
```

You may create additional files as well. Other than `XBSTree.java` avoid modifying or reusing any of the above files, since we overwrite them with our own when testing your program. Use the package "`cmsc420_f20`" for all your source files.

**Programming Assignment 2: Weight-Balanced Jackhammer Tree**

**Overview:** In this assignment you will implement a weight-balanced variant of an extended binary search tree. As in standard ordered dictionaries, the entries stored in your data structure will consist of key-value pairs. Following the same structure of the first programming assignment, all the key-value pairs are stored in the external nodes (leaves) of the tree, and the internal nodes just contain keys as splitters. Given an internal node with key value $x$, the left subtree contains key-value pairs whose keys are strictly smaller than $x$ and the right subtree contains key-value pairs whose keys are greater than or equal to $x$.

The new twist here is that each key-value entry will have an associated positive numeric *weight*, which for us will be represented by a Java `float`. So, we can think of each entry in the tree as consisting of three things $(x, v, w)$, where $x$ is its key, $v$ is its associated value, and $w > 0$ is its associated weight. We think of the weight as an indication of the importance of a node, and entries of higher weight (relative to the total weight) should reside closer to the root.

**Weight-Balanced Trees:** One way to implement this is through the use of *weight-balanced trees*. This generalizes the notion of height balance with standard binary search trees. We assume that we are given a real parameter $\alpha$, where $1/2 < \alpha < 1$. Given an extended binary search tree $T$ and any internal node $p$ of $T$, we define $p$'s *weight*, denoted `weight`($p$) to be the sum of the weights of the external nodes in the subtree rooted at $p$. (Note that the internal nodes do not contribute to the weight. They are just there to help us find external nodes.) Define $p$'s *balance ratio*, denoted balance($p$) to be

$$\text{balance}(p) \;=\; \frac{\max(\text{weight}(p.\text{left}), \text{weight}(p.\text{right}))}{\text{weight}(p)}.$$

We say that an extended binary search tree $T$ is $\alpha$-*balanced* if for all internal nodes $p$ in $T$, balance($p$) $\leq \alpha$. Note that as $\alpha$ gets closer to $1/2$, such a tree must be nearly perfectly balanced in the sense that its two subtrees have nearly half of the total weight, and as $\alpha$ approaches 1, the tree can be completely arbitrary, since either subtree can have an arbitrarily large fraction of the total weight.

Unfortunately, when nodes are associated with arbitrary weights, an $\alpha$-balanced tree may not even exist. For example, if we have just two keys, with weights $w_1 = 0.1$ and $w_2 = 0.9$, the only possible tree will have a balance ratio of 0.9. For this reason, we introduce an additional rule. Given any node, define its *maximum weight*, denoted maxWt($p$) to be the largest weight of any single entry in $p$'s subtree. (If the node is an external node, this is just the weight of the associated entry.) Define the *max ratio* of an internal node $p$, denoted max($p$) to be

$$\max(p) \;=\; \frac{\text{maxWt}(p))}{\text{weight}(p)}.$$

Given a numeric parameter $0 \leq \beta \leq 1$, we say that a node $p$ is $\beta$-*exempt* if $\max(p) > \beta$. We say that a tree is $(\alpha, \beta)$-*balanced* if every internal node $p$ is either $\beta$-exempt or is $\alpha$-balanced.

If we chose $\alpha$ and $\beta$ properly, it can be shown that, for any collection of weighted dictionary entries, there exists an $(\alpha, \beta)$-balanced extended binary search containing these keys. In particular, it suffices to choose $\alpha$ and $\beta$ such that $1/2 < \alpha < 1$ and $\beta < 2\alpha - 1$.[1] In our implementation, we will use the values $\alpha = 0.66667 \approx 2/3$ and $\beta = 1/4 = 0.25$. An example of such a tree is shown in Fig. 1. The tree is valid because all nodes that are not $\alpha$-balanced are $\beta$-exempt.

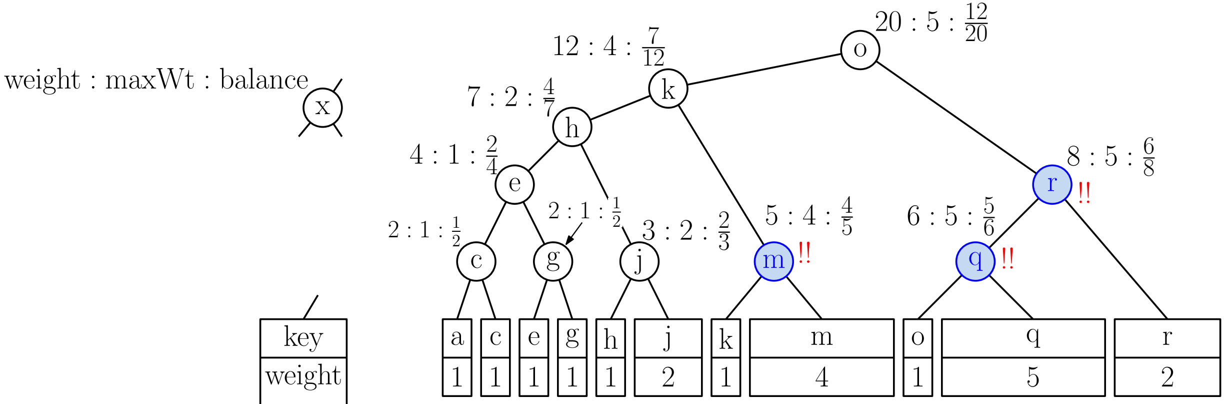



Figure 1: An $(\alpha, \beta)$-balanced (extended) search tree, for $\alpha = 2/3$ and $\beta = 1/4$. We have omitted the values, showing just the keys. Each internal node $p$ is labeled with weight($p$) : max($p$) : balance(p). Nodes have balance ratios larger than $\alpha$ are flagged with "!!" and $\beta$-exempt nodes are shaded in blue.

**Operations:** Below we outline the specifications of our weight-balanced trees, which we call *balanced jackhammer trees* or *BJ trees*, for short. The operations for extended binary search trees are very similar to the standard trees we have seen. Here is a formal definition of how the operations work.

**Initialization:** The initial tree consists of a `root` pointer whose value is `null`. If the tree consists of a single key-value pair, the root points directly to an external node containing this pair. Once the tree has two or more entries, the root will point to an internal node.

`Value find(Key x):` (Same as in Programming Assignment 1)

`void insert(Key x, Value v, float w):` Inserts the pair $(x, v)$ of weight $w$. The insertion process starts out exactly as in Programming Assignment 1, but as we are returning from the recursive calls, we update the weights and max-weights associated with each node along the search path. Next, starting at the root, we retrace the search path, walking down from the root. At the first instance (if any) that we encounter a node $p$ that violates the $(\alpha, \beta)$-balance condition (that is, balance(p) $> \alpha$ and $\max(p) \leq \beta$), we

[1] Why is this? Suppose that a node $p$ is not $\alpha$-balanced. Let us normalize the weights of $p$'s subtree so they sum to 1. (Otherwise, just divide all the following computations through by $p$'s total weight.) In order to be able to create a subtree whose root node is $\alpha$-balanced, it must be possible to find a splitter that partitions $p$'s weight somewhere within the interval $(1-\alpha, \alpha)$. If this were not possible, there must be an entry that spans this entire interval, meaning that its weight must be at least $\alpha - (1 - \alpha) = 2\alpha - 1$. If there were such an entry, however, its max ratio would be $2\alpha - 1 > \beta$, and so this node would be exempt from the weight-balance condition.

apply a procedure (described in the function `buildTree` of Lecture 13) that rebuilds the tree in the most balanced manner possible, and replaces the subtree at $p$ with this new tree.

For the sake of consistency, you should observe the following conventions:

- Use the float value $\alpha$ = `0.66667f` (rather than exactly 2/3) when checking the balance condition.
- Since there may be multiple nodes on the search path that violate the $(\alpha, \beta)$-balance conditions, choose the one that is *closest* to the root. (Note that this is the opposite of what Scapegoat trees do.)
- As is done in `buildTree` given in Lecture 13, if there are two splitting indices $i$ that achieve identical weight differences, favor the one that places more weight in the *right subtree*.

`void delete(Key x)`: Again, the process starts the same as in Programming Assignment 1, and as we are returning from the recursive calls, we update the weights and max-weights associated with each node along the search path. When we return to the root, we apply the same rebalancing process as for insertion, by walking down the search path until first finding an instance (if any) of a node $p$ that violates the $(\alpha, \beta)$-balance condition. We apply the same rebuilding process to this node as in insertion.

For example, in Fig. 2, we consider the deletion of key "m" from the tree in Fig. 1. On left, we show result of the standard deletion algorithm (from Programming Assignment 1), and we update the node weights as we return up the search path. As we descend the search path, we find that node "k" is not $\alpha$-balanced (its balance ratio is $7/8 > 2/3$), and it is not $\beta$-exempt (its max ratio is $2/8 \leq 1/4$), so we rebuild this subtree. The result is shown on the right side.

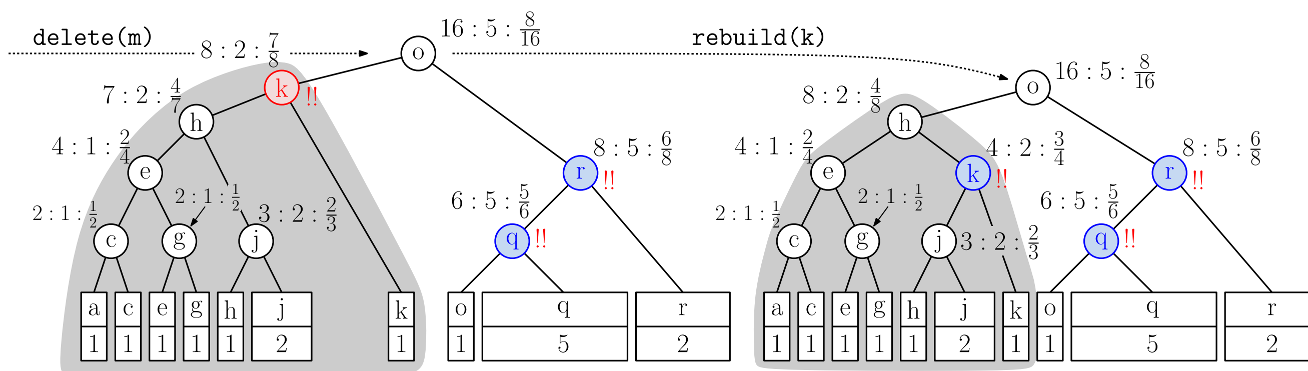



Figure 2: Deletion from a BJ tree.

`ArrayList<String> getPreorderList()`: This is the same as in Programming Assignment 1, but we include the weight associated with each node.

- Internal node $p$ storing a key $x$ and having total weight `p.weight`:
```
"(" + x.toString() + ") wt: " + p.weight
```
- External node $p$ storing the pair $(x, v)$ with weight `p.weight`:
```
"[" + x.toString() + " " + v.toString + "] wt: " + p.weight
```

Our testing programs are based on Java `String` equality. Note the use of parentheses for internal nodes and square brackets for external nodes. Also, note the single space between the key and value.

For example, given the tree of Fig. 1, the elements of the `ArrayList` would consist of the following strings:

```
(...Finish this...)
```

**Other operations:** The remaining operations `getMin()`, `getMax()`, `findDown(x)`, `findUp(x)`, and `clear()` are the same as in Programming Assignment 1.

**Running-time Requirements:** If no rebuilding takes place, all the dictionary operations (`insert`, `delete`, `find`, `getMin`, `getMax`, `findUp`, and `findDown`) should run in time proportional to the height of the tree. (The find operation should run in time proportional to the length of the search path to the key being sought.) If rebuilding is necessary, the rebuilding time should be $O(k \log k)$ for a subtree with $k$ external nodes and weights are uniform. (The algorithm presented in class achieves the bound.) The operation `getPreorderList` should run in time proportional to the number of nodes in the tree. We will determine this by inspection of your submitted code.

**Program structure:** We will provide a driver program that will input a set of commands. You need only implement the data structure and the functions listed above. In particular, you will implement a class called `BJTree`, which has the following public interface. (We have included the values of $\alpha$ and $\beta$ as constants.)

```
package cmsc420_f20;

public class BJTree<Key extends Comparable<Key>, Value> {

    private final float ALPHA = 0.66667f; // maximum allowed balance ratio
    private final float BETA = 0.25f; // maximum weight exemption

    public BJSTree() { ... } // constructs an empty tree
    public Value find(Key x) { ... }
    public void insert(Key x, Value v, float w) throws Exception { ... }
    public void delete(Key x) throws Exception { ... }
    public ArrayList<String> getPreorderList() { ... }
    public Value getMin() { ... }
    public Value getMax() { ... }
    public Value findDown(Key x) { ... }
    public Value findUp(Key x) { ... }
    public void clear() { ... }

}
```

As in Programming Assignment 1, we assume that `Key` and `Value` support the `toString()` method, and the type `Key` implements `Comparable<Key>`.

**Node structure:** As in Programming Assignment 1, you should use inheritance in your node structure. The notable difference is that, since all nodes will store a weight value, this can

be added to the parent class. For external nodes, this is the weight of the entry. For internal nodes, this is the total weight of the entire subtree rooted at this node (that is, `p.weight = p.left.weight + p.right.weight`). Also, for each internal node, the max-weight entry should be cached at this node, since computing it each time we visit the node would take too much time. So, we add an entry `float maxWeight` to each internal node. (If you prefer, you could place it instead in the parent class `Node`. While this is not ideal, it will make your coding simpler to assume that all keys have this field.)

```
public class BJTree<Key extends Comparable<Key>, Value> {

    private abstract class Node { // generic node (purely abstract)
        Key key;
        float weight; // total weight of all entries in this subtree
        abstract Value find(Key x); // no function body for these!
        abstract Node insert(Key x, Value v, float w) throws Exception;
        abstract Node delete(Key x) throws Exception;
        // ... other functions omitted
    }

    private class InternalNode extends Node { // internal node
        Node left;
        Node right;
        float maxWeight; // maximum weight of any single entry

        Value find(Key x) { ... }
        Node insert(Key x, Value v) throws Exception { ... }
        Node delete(Key x) throws Exception { ... }
        // ... other functions omitted
    }

    private class ExternalNode extends Node { // external node
        Value value;
        Value find(Key x) { ... }
        Node insert(Key x, Value v) throws Exception { ... }
        Node delete(Key x) throws Exception { ... }
        // ... other functions omitted
    }

    // ... public functions (see above)
}
```

**Programming Assignment 3: Persistent Weight-Balanced Jackhammer Tree**

**Overview:** A dynamic data structure is *persistent*, if it is possible to perform queries not only to the current version of the data structure but to earlier versions as well. In this assignment you will further extend your implementation of the weight-balanced jackhammer tree (BJ tree) to include persistence. To keep matters simple, we will restrict modifications of the data structure to the insert operation (thus, no delete and no clear). We call these *Persistent weight-Balanced Jackhammer trees* or *PBJ trees*, for short.

We assume that each insertion operation is tagged with a nonnegative integer *timestamp*, which indicates the time at which this operation is performed. We assume that these time-stamps increase strictly monotonically. For example, suppose that we insert the following keys at times 01, 03, 05, and 07, respectively:

```
01:insert:IAD ...
03:insert:BWI ...
05:insert:SFO ...
07:insert:ATL ...
```

Each query operation (find, find-up, get-min, etc.) is also associated with a nonnegative timestamp $t$. The query is applied to the tree as it existed at time $t$. More formally, let $t'$ denote the largest insertion time such that $t' \leq t$. The query is applied to the tree that results as a result of the insertion at time $t'$. If $t$ is smaller than the timestamp of the earliest insertion, then it is applied to the initial (that is, empty) search tree. For example, given the above insertions, consider the following queries and the associated results:

```
00:find:IAD  ----> fails (prior to the first insertion and tree is empty)
01:find:IAD  ----> succeeds (IAD is inserted at time 01)
08:find:IAD  ----> succeeds (IAD is still present at all times greater than 01)
01:get-min   ----> returns IAD (IAD is the only, hence smallest at time 01)
06:get-min   ----> returns BWI (smallest among {IAD, BWI, SFO} at time 06)
00:get-min   ----> returns null (prior to first insertion and tree is empty)
```

There are many ways to implement persistence in search trees. Our approach will be to introduce a new node type in our tree, called a *temporal node* (in addition to *internal* and *external*). Each temporal node `q` stores a *timestamp* `q.time` and has two children, `q.pre` and `q.post` (see Fig. 1(a)). The subtree referenced by `q.pre` contains the contents of the tree strictly prior to time `q.time` and the subtree referenced by `q.post` contains the contents of the tree on or after this time. When processing a query at some given time `t`, if we arrive at a temporal node `q`, if `t < q.time`, we recurse on `q.pre` and otherwise we recurse on `q.post`.

For the sake of consistency with the other nodes of the jackhammer tree, it is useful to associate each temporal node with `weight` and `maxWt` values. Because weights are only of use when rebalancing the tree, which can only in the latest version, for a given temporal node `q`, we extract these values from the post subtree:

$$\texttt{q.weight} \leftarrow \texttt{q.post.weight} \quad \text{and} \quad \texttt{q.maxWt} \leftarrow \texttt{q.post.maxWt}$$

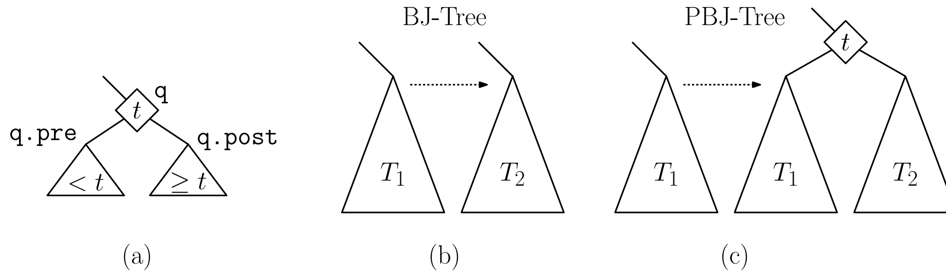



Figure 1: Temporal node and rebuilding subtrees.

(Note that the weight values of `q.pre` have *no effect* on `q`'s weights.)

When and how are temporal nodes generated? It is easiest to see this in the context of when a subtree is rebuilt. Suppose that a subtree $T_1$ rooted at some node `p` is rebuilt at time $t$, due to the jackhammer balance conditions (as described in Part 2 of the assignment). After rebuilding, we now have a new subtree $T_2$. In the standard BJ tree, we would just unlink the old subtree and link in the new subtree (see Fig. 1(b)). In PBJ tree, we will create a new temporal node `q`, and we will set `q.pre` to point to the old subtree $T_1$ and set `q.post` to point to the new subtree $T_2$ (see Fig. 1(c)). Queries for times smaller than $t$ are applied to $T_1$, and otherwise they are applied to $T_2$.

**Technical Specifications:** These are the minimum requirements. You may, of course, add additional members and functions.

**Data members:** The tree has three (private) data members, which are initialized as follows:

`Node root:` The tree's root. (Initial value `null`)

`int firstUpdateTime:` Time of the first insertion. (Initial value $-1$)

`int lastUpdateTime:` Time of the most recent insertion. (Initial value $-1$)

`void insert(Key x, Value v, float w, int t):` This inserts the key-value pair $(x, v)$ with weight $w$ in the tree at time $t$. If the root is `null`, then this is the first insertion into the tree. We create a new external node containing the pair $(x, v)$ and point the root to this node. We also set `firstUpdateTime = lastUpdateTime = t`.

Otherwise, the tree is nonempty. We first check that `t > lastUpdateTime`. (Because insertion times must be monotonically increasing.) If not, we throw an exception ("Update time must exceed prior update time"). We begin by invoking the standard extended binary tree insertion process. Through the use of a recursive helper function, we descend the tree searching for $x$. (For internal nodes, we use the standard insertion form, `left = left.insert(...)` or `right = right.insert(...)`. For temporal nodes we always recurse on the post side, `post = post.insert(...)`.) Eventually, we reach an external node. Let $y$ denote the key in this node. If $x == y$, we signal a duplicate-key error. Otherwise, the standard insertion process replaces node $y$ by a three-node subtree, consisting of an internal node containing $\max(x, y)$, and two external nodes containing $x$ and $y$.

In the persistent setting, we create a new temporal node, labeled with $t$. Its pre child is the original external node $y$, and its post child is the three-node subtree described above

(see Fig. 2).

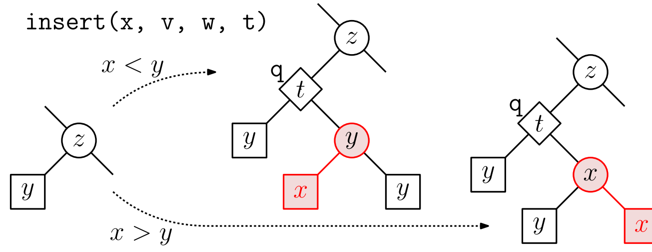



Figure 2: Temporal node and leaf-level insertion.

For example, suppose we insert the key `"D"` with weight 2 into the PBJ tree shown in Fig. 3 at time 7. We follow the search path (post of 3, left of `"I"`) until reaching the external node `"B"`. Since `"D" > "B"`, we create the three-node subtree with `"D"` as its root and make it the post child of a temporal node with timestamp $t = 7$. The original external node `"B"` is its pre child.

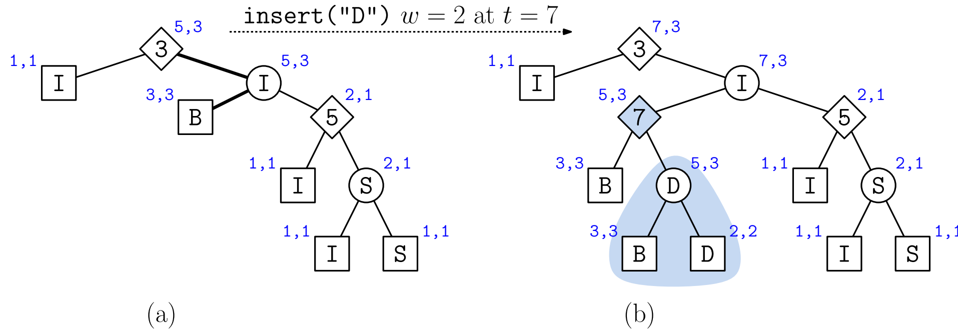



Figure 3: Inserting `"D"` with weight 2 at time 7. The blue numbers by each node are its `weight` and `maxWt` values.

**Weight-Based Rebalancing:** After performing the insertion, we apply the rebalancing process from Part 2. First, as we are returning from the recursive calls following the insertion, we update the `weight` and `maxWt` values at each node along the search path.

On returning to the root, we initiate a rebalancing process by walking down the tree along the search path. (This is analogous to insertion. We create a recursive helper function, `rebalance(Key x, int t)`. For internal nodes, we do `left = left.rebalance(...)` or `right = right.rebalance(...)`. For temporal nodes we always recurse on the post side, `post = post.rebalance(...)`.) We compute the balance ratio and the max ratio (see Part 2) at each *internal node* (ignore temporal nodes). If/when we first encounter an internal node that violates the $(\alpha, \beta)$-balance conditions (using the same $\alpha$ and $\beta$ values from Part 2), we rebuild this subtree. To do this, we collect all the external nodes that exist in the tree *at the time of the insertion*. (In other words, you should ignore external nodes in the pre subtrees of any temporal nodes.)

We then apply the `buildTree` function from Part 2 on the resulting list to obtain an ideally balanced subtree. Finally, we create a temporal node whose timestamp is the

insert time, and set its pre link to the original subtree and set its post link to the newly created subtree. The `weight` and `maxWt` values of this node are the same as those of the its post subtree. (There is one more step, described below under Temporal Rebalancing.) This is illustrated in Fig. 4. Observe that only the external nodes that are active at time $t = 8$ are included in the list. Also, observe that the newly rebalanced tree (shaded in blue) is free of any temporal nodes.

Figure 4: Rebuilding a subtree. (This is *not* a real instance because the subtree is $\beta$-exempt.)

**Shallow or deep?** When copying external nodes, it is always good to be mindful as to whether you are performing *shallow copies* (just copying a reference to a node) or *deep copies* (cloning a duplicate copy of the node). In our figures, we show the result of deep copying. (For example, in Fig. 3, each instance of the external node "B" is a distinct instantiation of an external node object.) We tried it both ways, and it appears that it does not matter. The reason is that, once created, the contents of an external node never change. Thus, there appears to be no harm in having many references to the same object in memory. In general, however, using shallow copying in data structure design is a risky practice.

**Temporal Rebalancing:** Weight-based rebalancing implies that the tree is balanced with respect to weights, but what about the temporal nodes. To keep the data structure simple, we are only going to perform only a very simple form of rebalancing for temporal nodes. To see what the issue is, consider Fig. 4(b), and observe that we have just created a new temporal node 8 as the right child of the temporal node 3. This is not good. If we were to perform a sequence of updates, it is possible to create a long post-chain sequence of temporal nodes with incresing time values. As a result, queries to the current version of the tree (which tend to be the most common) may take a long time.

To remedy this, we want to convert post chains of temporal nodes into pre chains. (While this is not necessarily more balanced, it means that we suffer the performance degradation for searches in the past, not for searches in the current structure.) We implement this as follows. Whenever we invoke `rebalance(x,t)` from a temporal node, recall that we always invoke the function recursively on its post side. (In the example, consider the invocation of `post.rebalance("D", 7)` from the temporal node 3.) When we return from this function, we check whether our post child is a temporal node. (In Java, this can be done using `TemporalNode.class.isInstance(post)`.) If so, we apply

a left rotation at this node (see Fig. 4(c)). After doing so, we should also adjust the weights of the two rotated nodes.

`ArrayList<String> getPreorderList(int t)`: As in the previous parts, this returns a Java `ArrayList` of strings containing a preorder traversal of the tree contents at time $t$. Note that because the time is specified, this list will contain only internal and external nodes (not temporal nodes).

The procedure is almost the same as in Part 2, except we do *not* include the weights of the internal nodes. (Why not? The weight stored in each internal node reflects the weight of the node in the current version of the tree, and hence is not a persistent quantity.) More formally, on encountering a temporal node, we branch to the appropriate child depending on $t$. On encountering an internal node with key `key`, we generate the string:

```
"(" + key.toString() + ")"
```

and on encountering an external node storing the pair `key` and `value` with weight `weight`, we generate the string:

```
"[" + key.toString() + " " + value.toString() + "] wt: " + weight
```

For example, here is the result of the call `getPreorderList(7)`, for the tree of Fig. 4(a). The weights of the external nodes are shown, but not the internal nodes. Observe that no temporal nodes are given. When we arrive at a temporal node, we visit the appropriate subtree based on the value of $t$.

```
Preorder list at time = 7:
  (I)
  (D)
  [B Baltimore] wt: 3.0
  [D Washington] wt: 2.0
  (S)
  [I Washington] wt: 1.0
  [S San Francisco] wt: 1.0
```

`ArrayList<String> getFullPreorderList()`: In contrast to the previous command, this command produces a preorder listing of *all* the nodes of the tree, including the temporal nodes. For each temporal node containing a time value `time`, we generate the string:

```
"<" + time + ">"
```

For example, here is the result of the call `getFullPreorderList()`, for the tree of Fig. 4(a).

```
Full-Preorder list at time = 7:
  <3>
  [I Washington] wt: 1.0
  (I)
  <7>
  [B Baltimore] wt: 3.0
  (D)
  [B Baltimore] wt: 3.0
  [D Washington] wt: 2.0
  <5>
  [I Washington] wt: 1.0
  (S)
```

```
[I Washington] wt: 1.0
[S San Francisco] wt: 1.0
```

The structure of this function is almost identical to `getPreorderList`, except that when we visit a temporal node, we add its string to the list and we recurse on both of its children, pre then post.

`Value find(Key x, int t)`: Searches for $x$ in the tree at time $t$. If the tree is empty (that is, the root is `null`) or $t$ is smaller than `firstUpdateTime`, this returns `null`. Otherwise, a search is performed for $x$ at time $t$ (that is, at each internal node we branch left-right depending on $x$ and at each temporal node we branch pre-post depending on $t$). On arriving at an external node, we check whether the keys match, and if so we return the associated value. Otherwise, we return `null`.

**Other operations:** The other query operations `getMin(t)`, `getMax(t)`, `findDown(x,t)`, `findUp(x,t)` are straightforward generalizations of their Part 2 versions. They are given a time value $t$, and they are applied to the version of the tree at time $t$. If the tree is empty (that is, the root is `null`) or $t$ is smaller than `firstUpdateTime`, they all return `null`. Otherwise, they apply the search procedure in the same manner as in Part 2, except that on arriving at a temporal node, they branch based on the value of $t$.

**Program structure:** As before, we will provide a driver program that will input a set of commands. You need only implement the data structure and the functions listed above. In particular, you will implement a class called `PBJTree`. As in Programming Assignment 1, we assume that `Key` and `Value` support the `toString()` method, and the type `Key` implements `Comparable<Key>`.

**Node structure:** As in the first two parts, you should use inheritance in your node structure. There is a new `TemporalNode` node type. Because temporal nodes do not store keys, we have reorganized the node contents slightly. First, we store keys explicitly in the internal and external nodes. Second, we store both the weight and max-weight in the parent node class. (It just makes things a bit more convenient, since all nodes need to access these values.)

```
public class PBJTree<Key extends Comparable<Key>, Value> {

    private final float ALPHA = 0.66667f; // maximum allowed balance ratio
    private final float BETA = 0.25f; // max ratio filter

    private abstract class Node { // generic node type
        float weight; // the weight associated with this node
        float maxWt; // the max weight in this subtree

        abstract Value find(Key x, int t); // no function body for these!
        abstract Node insert(Key x, Value v, float w, int t) throws Exception;
        // ... other functions omitted
    }

    private class TemporalNode extends Node { // temporal node
        int time; // timestamp
        Node pre; // subtree prior to timestamp
```

```
        Node post; // subtree on or after timestamp

        Value find(Key x, int t) { ... }
        Node insert(Key x, Value v, float w, int t) throws Exception { ... }
        // ... other functions omitted
    }

    private class InternalNode extends Node {
        Key key; // the key
        Node left // subtree smaller than key;
        Node right; // subtree greater or equal to key

        Value find(Key x, int t) { ... }
        Node insert(Key x, Value v, float w, int t) throws Exception { ... }
        // ... other functions omitted
    }

    private class ExternalNode extends Node {
        Key key; // the key
        Value value; // the associated value object

        Value find(Key x, int t) { ... }
        Node insert(Key x, Value v, float w, int t) throws Exception { ... }
        // ... other functions omitted
    }

    // ... public functions (see above)
}
```

**Is this efficient?** How much space do we sacrifice for the sake of persistence? It may seem that by rebuilding subtrees, we are wasting lots of space. But this is not so. By the amortized analysis of scapegoat trees, we know that the overall time spent in rebuilding subtrees for $m$ update operations is $O(m \log m)$. It follows that the total space needed is also $O(m \log m)$. Persistence demands that we use at least $O(m)$ space to store these updates. Thus, we pay at most an additional $O(\log n)$ factor to achieve persistence.

Another possible concern is that we do not do any rebalancing with respect to temporal nodes, and indeed, searches in the past may be slow. (We'll leave fixing this as an open problem!) But because of our temporal rebalancing, we know that there can be no two consecutive temporal nodes along the search path for queries in the current version of the tree. Because of weight rebalancing, we know that the internal-node height is $O(\log n)$, and hence the overall height for the current version of the tree is $O(\log n)$. Thus, queries to the current state of the data structure are guaranteed to be efficient.

Skeleton code, test data, and the submission process will be essentially the same as in the previous two parts. We will provide a link to the final test data on the class Projects page.