"Trees are the earth's endless effort to speak to the listening heaven."

– Rabindranath Tagore, *Fireflies*, 1928

"Trees are poems that the earth writes upon the sky. We fell them down and turn them into paper, that we may record our emptiness."

– Kahlil Gibran

Alice was walking beside the White Knight in Looking Glass Land.
"You are sad." the Knight said in an anxious tone: "let me sing you a song to comfort you."
"Is it very long?" Alice asked, for she had heard a good deal of poetry that day.
"It's long." said the Knight, "but it's very, very beautiful. Everybody that hears me sing it - either it brings tears to their eyes, or else -"
"Or else what?" said Alice, for the Knight had made a sudden pause.
"Or else it doesn't, you know. The name of the song is called 'Haddocks' Eyes.'"
"Oh, that's the name of the song, is it?" Alice said, trying to feel interested.
"No, you don't understand," the Knight said, looking a little vexed. "That's what the name is called. The name really is 'The Aged, Aged Man.'"
"Then I ought to have said 'That's what the song is called'?" Alice corrected herself.
"No you oughtn't: that's another thing. The song is called 'Ways and Means' but that's only what it's called, you know!"
"Well, what is the song then?" said Alice, who was by this time completely bewildered.
"I was coming to that," the Knight said. "The song really is 'A-sitting On a Gate': and the tune's my own invention."
So saying, he stopped his horse and let the reins fall on its neck: then slowly beating time with one hand, and with a faint smile lighting up his gentle, foolish face, he began...

– Lewis Carroll, *Alice Through the Looking Glass*, 1865

Lecture III
SEARCH TREES

¶1. Anthropologists noted[1] that there is an unusually large number of Eskimo words for snow. The Computer Science equivalent of 'snow' is the 'tree' word:

(a,b)-tree, AVL tree, B-tree, binary search tree, BSP tree, conjugation tree, dynamic weighted tree, finger tree, half-balanced tree, heaps, interval tree, leftist tree, kd-tree, octtree, optimal binary search tree, priority search tree, quadtree, R-trees, randomized search tree, range tree, red-black tree, segment tree, splay tree, suffix tree, treaps, tries, weight-balanced tree, etc.

---

[1]Some called this a "myth" because of issues about how to count word cognates and to account for Eskimo dialects, etc. It is possible that English have even more snow words. Perhaps the real myth consists in not recognizing our own snow words (sleet, slush, powder, drift, etc).

36 I have restricted the above list to trees that are used as search structures. If we include
37 trees arising as data structures in applications (e.g., Huffman tree, DFS/BFS tree, alpha-beta
38 tree), we obtain a much longer list of tree words. The list could be enlarged to include variants
39 of these trees: thus there are subspecies of B-trees called B^+- and B^*-trees, etc. The list will
40 continue to grow as the field progresses.

41 If there is a most important entry in the above list, it has to be *binary search tree* which
42 is abbreviated as **BST**. It is the first non-trivial data structure that students encounter, after
43 linear structures such as arrays, lists, stacks and queues. Trees are useful for implementing
44 a variety of **abstract data types**. We shall see that all the common operations for search
45 structures are easily implemented using binary search trees. Most algorithms on BSTs have a
46 worst-case behavior that is proportional to the height of the tree. Therefore it is vital to bound
47 the height of trees of any given size (i.e., number of nodes).

48 **¶2. Concept of a Balanced Family of Trees.** Suppose $\mathcal{F}$ is a family of trees. By an
49 **admissible pair** of $\mathcal{F}$ we mean a pair of numbers (n, h) such that there exists a $T \in \mathcal{F}$ of
50 size n and height h. We call $\mathcal{F}$ a **balanced family** if (1) for every n, there is an h such that
51 (n, h) is admissible, and (2) there is a fixed constant $C_{\mathcal{F}} > 0$ such that for all admissible (n, h),   *Balance-ness is a*
52 $h \leqslant C_{\mathcal{F}} \lg n$. **Example:** The set of all *perfect* binary trees[2] is not an balanced family because   *family property, not*
53 the sizes of perfect binary trees has the special form $n = 2^h - 1$ for some $h \geqslant 0$. On the other   *of individual trees*
54 hand, the set of all *complete* binary trees is a balanced family. For any admissible pair (n, h)
55 for a family of binary trees, the following inequality holds:

$$n + 1 \leqslant 2^{h+1}. \tag{1}$$

56 (This is easily proved by structural induction on binary trees.) Taking lg's in (1), we conclude
57 that $\lg(n + 1) \leqslant h + 1$, i.e., height is at least logarithmic in n. This implies that our definition
58 of "balanced family" is optimal for binary trees in the sense that for any admissible pair (n, h),
59 it is necessarily the case that $h = \Theta(\log n)$. The implicit constant in the big-Oh notation here
60 depends on the family $\mathcal{F}$, hence we may write $h = O_{\mathcal{F}}(\log n)$ instead of just $O(\log n)$.

61 This chapter is basically about introducing three schemes for constructing balanced families:

62 • **Height-balanced trees:** Assuming binary trees, the absolute difference[3] of the heights
63   of any pair of siblings is at most some constant $\eta \geqslant 1$. If $\eta = 1$, we have AVL trees from
64   Adel'son-Vel'skii and Landis (1962); see §6 below.

65 • **Weight-balanced trees:** Again assuming binary trees, the minimum ratio[4] of the
66   "weights" of any pair of siblings is strictly greater than a constant $\rho > 0$. If the "weight"
67   of a node is the size of its subtree, we basically obtain the bounded-balanced trees of
68   Nievergelt and Reingold (1973); see §7 below.

69 • **Degree-balanced trees:** Degree-balancing only makes sense for non-binary trees: for
70   constants $2 \leqslant a < b$, we say a tree is degree-balanced if the degree of each non-leaf lies in
71   the range $[a, b]$. This still allows too many possibilities, but balanced families emerge if
72   we impose another restriction: *all leaves in the tree have the same depth.* If $b = 2a - 1$,
73   we obtain the class of B-trees from Bayer and McCreight (1972); we shall introduce very
74   general extensions of such trees in §8 below.

---

[2]For definitions related to binary trees, see ¶13 below.
[3]The **absolute difference** of two numbers h, h' is $|h - h'|$. This difference is non-negative.
[4]The **minimum ratio** of two positive numbers w, w' is $\min\left\{\frac{w}{w'}, \frac{w'}{w}\right\}$. This ratio is $\leqslant 1$.

> We emphasize that the above schemes are purely structural constraints on trees. But our trees are meant to be used as **search trees** which store data in discrete units that we call **items**. The set of items stored in the search tree can be retrieved, inserted and deleted. Corresponding to any balanced family $\mathcal{F}$, let $\mathcal{F}^*$ denote the corresponding family of search trees whose shapes belong to $\mathcal{F}$. For instance, if $\mathcal{F}$ are binary trees, then $\mathcal{F}^*$ would be binary search trees (BSTs).
> Retrieving, inserting or deleting on any $T^* \in \mathcal{F}^*$ is considered efficient if it has complexity proportional to the height, which is $\Theta(\log n)$ for balanced families.

This chapter will supply efficient algorithms for the each scheme. By the simple measure of $\Theta(\log n)$ complexity, the three schemes are indistinguishable. So let us discuss some distinguishing features of 3 schemes: height balancing and weight balancing are directly comparable since they both concern binary trees. Height balancing is simpler to analyze than weight balancing. Height balancing maintains the height information[5] of each node, using $\lg \lg n$ bits; weight balancing maintains the size of each node, using $\lg n$ bits. So we see 2 benefits of height balancing. So what is to recommend weight balancing? Despite the difficulty of its analysis, weight balancing algorithms are simpler to implement. It is also more versatile than height balancing because it can be extended to geometric multidimensional searching.[6] Finally, degree-balanced trees have the unique ability to exploit the I/O complexity model in which data is stored in the disk (secondary storage) but we must read data from disk main memory for operations, and then write it out again to disk. The main cost is this data transfer between disk and main memory. This is extremely important in data base applications.

Tarjan [19] gives a brief history of some balancing schemes. Brass [5] is an excellent resource for advanced data structures.

> STUDY GUIDE: all our algorithms for search trees are described in such a way that they can be internalized and so it is possible to carry out hand-simulations on reasonable size examples; we expect students to be able to carry out such hand-simulations on concrete examples. We do not provide any computer code, on the principle that once an algorithm is internalized, most students can implement them in their favorite programming language.

Students: expect to do hand-simulations!

§1. Search Structures with Keys

Search structures store a set of objects, subject to searching ("Lookup" or "Retrieve") and modification ("Insert", "Delete"). Here we will standardize some terminology for search structures.

¶3. Keys and Items. The objects in a search structure are called **items**. Each item is a pair $(Key, Data)$ where the **data** part is arbitrary, but the **key** must come from a totally

---

[5] Actually one bit is enough, e.g., in red-black trees. We do not treat red-black trees because, despite its complicated algorithms, they do not seem to confer any special advantage over other families of balanced binary trees.

[6] Normal searching is based on 1-dimensional search keys (say real numbers). If the search keys are d-dimensional points, then we can design search trees that are recursive on the dimension. Height balancing cannot achieve sublinear algorithms for $d \geqslant 2$ but weight balancing achieves poly-logarithmic complexity, $O(\log^k n)$ where $k \leqslant d$.

ordered set. We organize these items for searching using the ordering of keys. Here are some *item = (key,data)* real life examples.

(i) In an *employee database*, each item is an employee record. The key of an employee record is the social security number (9 digits), with associated data such as name, address, salary, etc. Two items in this database might read:
(111122222, (William Blake, London, £6,000,...))
(123456789, (William Shakespeare, Stratford-upon-Avon, £10,000,...))

(ii) A *dictionary* where each item is a word entry. The key is the word itself, associated with data such as the pronunciation, part-of-speech, meaning, etc. The items for the entry "data" and "key" might read:
(data, (noun. 1. plural of datum, 2. individual facts; statistics. 3.
information. 4....)).

(iii) A *scheduling queue* in a computer operating systems where each item in the queue is a job that is waiting to be executed. The key is the priority of the job, which is an integer.

We refer to such structures as **keyed search structures**. From an algorithmic point of view, the properties of the search structure are solely determined by the keys in items, the associated data playing no role. Hence in illustrations, we normally identify the item with its key, and ignore the associated data (the data is not displayed). This is somewhat paradoxical since, for the users of the search structure, it is the data that is more important. This leads to the jarring thought in examples, when we search a data structure to retrieve a key using the same key! Of course, in reality, we retrieve the item (or data) associated with that key.

Is there any point in searching for keys with no associated data?

¶4. On nodes and attributes.   Search structures can be viewed as a collection of **nodes** that are interconnected by pointers. Abstractly, they can be viewed as directed graphs whose vertices represent these nodes. Each node stores one or more objects which are either items or keys. See Figure 1 for two kinds of nodes used in linked lists and in binary trees.
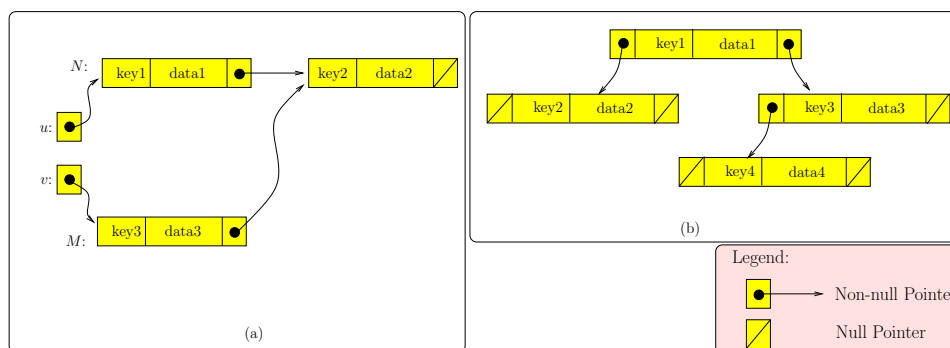

Figure 1: Two Kinds of Nodes: (a) linked lists, (b) binary trees

The following defines the "node type" in the programming languages C and Java, respectively:

```
125   /* C program:  */
          struct Node {
              int key;
              void* data;
              struct Node* next;
          };
```

```
126   // Java program:
          class Node {
              int key;
              String data;
              Node next;
          }
```

127 Note the definition of a *Node* type is recursive because it refers to itself. As a data structure,
128 each node has one or more **attributes** (or **fields**) which have user-given names (e.g., `key,`
129 `data, next`) as well as data types (e.g., `int`, `String`, *Node*). In `Java`, attributes are also
130 called **members**. Each attribute can be assigned a value of the appropriate data type. If u is
131 an instance of *Node*, we can assign values to its attributes as follows:

132 $$u.\texttt{key} \leftarrow 1234; \quad u.\texttt{data} \leftarrow \text{`hello!'}; \quad u.\texttt{next} \leftarrow u.\texttt{next.next};$$

133 If a node has $k \geqslant 1$ attributes, we can think the node as a k-tuple except that we can access
134 its components by attribute names, using the dot-convention $u.attributeName$.

135 **¶* 5. Pointers.** In "typed" modern programming languages such as `C` or `Java`, all data are
136 either of *primitive type* which are built into the language and *reference type* which the user
137 defines. E.g., `int` (machine integer) data is primitive type while the node data (as defined
138 above) is reference type. For each reference type, there is a derived data type called **pointer** to
139 that reference type data. In `C`, you can explicitly define pointer variables and manipulate them;
140 in `Java`, the manipulation of these pointers are implicitly done via the semantics of assignment.
141 In programming language jargon, this amounts to automatic referencing and de-referencing for
142 reference variables.

143 The proper treatment of pointers is beyond our scope, and so in this book, we will be
144 informal in our manipulation of nodes or their pointers. When we say that "u is a node", this u
145 is essentially a node **reference** in the sense of Java. Thus, u will sometimes act like an ordinary
146 *node variable* and other times, act like a *node pointer*. Alternatively, we can follow Lewis and
147 Denenberg [12] by calling u a (node) **locative**.

148 To explore these concepts, it is useful to follow the `C` Programming Language where the
149 node and node-pointers are different types. We illustrate it by this snippet:

150

```
151   Node N; // declare N as a Node
      Node * u; // declare u as a Node pointer
      u = & N; // (& N) is the pointer to N
      N = * u; // (* u) is the node that u points to.
```

152 In `Java`, there is only one type, a reference to `Node`. There is a small but imperceptible price
153 to the user: in the assignment '`N.key = 3`', it is necessary to first fetch the actual location of
154 the node `u` to get the location of memory for storing the value `3`. In `C`, we achieve the same
155 effect directly with '`u->key = 3`'. In this book, we will take the Java approach: we do not
156 distinguish between a node N and its pointer u; the context will tell us which is intended.

157   Besides an item, each node also stores one or more pointers to other nodes. Two simple
158 types of nodes are illustrated in Figure 1: nodes with only one pointer (Figure 1(a)) are used to
159 forming linked lists; nodes with two pointers can be used to form a binary trees (Figure 1(b)),
160 or doubly-linked lists. Nodes with three pointers can be used in binary trees that require parent
161 pointers. First, suppose N is a node variable of the type in Figure 1(a). Thus N has three
162 **fields**, and we may name these fields as `key`, `data`, `next`. Each field has some data type. E.g.
163 `key` is typically integer, `data` can be string, but `next` has to be a pointer to nodes. This field
164 information constitutes the "type" of the node. To access the fields in node N, we write N.`key`,
165 N.`data` or N.`next`. The type of N.`next` is not a node, but pointer to a node. In our figures,
166 we indicate the values of pointers by a directed arrow. Node pointer variables act rather like
167 node variables: if variable u is a pointer to a node, we also[7] write u.`key`, u.`data` and u.`next` to
168 access the fields in the node. There is a special pointer value called[8] the **null pointer** whose
169 value is denoted by `nil`. It points to nothing, but as a figure of speech, we say it points to the
170 **nil node**. Of course the nil node is not a true node since it cannot store any information. In
171 figures (cf. Figure 1) null pointers are indicated by a square box with a slash line.

172   In search queries, we sometimes need to return a set of items. The concept of an iterator
173 captures this in an abstract way: an **iterator** is a special node u that has two fields: u.`value`
174 and u.`next`. Here, u.`next` is a pointer to another iterator node while u.`value` is a pointer to an
175 item node. Thus, by following the `next` pointer until we reach `nil`, we can visit a list of items
176 in some order.

---

[7]View this as a harmless abuse of notation.

[8]The null or nil terminology is often interchangeable in the literature. To distinguish them, I treat "null" as a description of pointers with the special value of "nil".
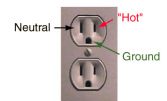
Another convention is for null pointers to point to the "ground symbol" (from electrical engineering)

Informal Programming Semantics: The difference between a node variable N and a node pointer variable u is best seen using the assignment operation. Let us assume that the node type is $(\texttt{key}, \texttt{data}, \texttt{next})$, M is another node variable and v another node pointer variable. In the assignment '$N \leftarrow M$', we copy each of the three fields of M into the corresponding fields of N. But in the assignment '$u \leftarrow v$', we simply make u point to the same node as v. Referring to Figure 1(a), we see that u is initially pointing to N, and v pointing to M. After the assignment $u \leftarrow v$, both pointers would point to M.

But what about '$N \leftarrow u$' and '$u \leftarrow N$'? In the former case, it has the same effect as '$N \leftarrow M$' where u points to M. In the latter case, it has the same effect as '$u \leftarrow v$' where v is any pointer to N (v may not actually exist). In each case, the variable on the left-hand side determines the proper assignment action. Once we understand four assignment possibilities and their intended meaning, there is no real need to distinguish these two types of values (N and u) in assignments. *This is what we meant earlier, when we said that our notion of nodes will variously look like ordinary variable N or like pointer variables u.* Indeed the `Java` language eschews pointers, and introduces an intermediate concept called reference.

The four main players in our story are the two variables u and N, the pointer value of u, and the node that N refers to. This corresponds to the four references in the tale of Alice and the White Knight in the epigraph of this chapter. We may use a simpler example: suppose x is an integer variable whose value is 3. Let $\uparrow x$ be a pointer to x whose value $\&x$ denotes the address of x. We have this correspondence:

English Reference	Value in English	Integer Example	Node Example
Song is Called	'Ways and Means'	x	N
Song is	'A-sitting On a Gate'	3	Node value
Name of Song is Called	'Haddocks' Eyes'	$\uparrow x$	u
Name of Song is	'The Aged, Aged Man'	$\&x$	$\&N$

Four Assignments:
$N \leftarrow M$
$N \leftarrow u$
$u \leftarrow v$
$u \leftarrow N$

The clue from the story of Alice and the White Knight

¶6. Uses of Key. Key values usually come from a totally ordered set. Typically, we use the set of integers for our totally ordered set. Another common choice for key values are character strings ordered by lexicographic ordering. When we speak of the "largest item", or "comparison of two items" we are referring to the item with the largest key, or comparison of the keys in two items, etc.

usually, keys ≡ integers!

By default, we make the **unique key assumption**, that the keys in a keyed search structure are unique. Equivalently, distinct items have the distinct keys. Unique key assumption usually simplify the algorithms, and in most applications, this is assumption is a mild restriction. In the few places where we drop this assumption, it will be stated explicitly.

Keys are called by different names to suggest their function in the structure. For example, a key may variously called:

- **priority**, if there is an operation to select the "largest item" in the search structure (see example (iii) above);

- **identifier**, if the keys are unique (distinct items have different keys) and our operations use only equality tests on the keys, but not its ordering properties (see examples (i) and (ii));

- **cost** or **gain**, depending on whether we have an operation to find the minimum (if cost) or maximum (if gain);

- **weight**, if key values are non-negative.

We may define a **search structure** S as a representation of a set of items that supports the `lookUp` query, among other possible operations. The lookup query on S, on a given key K, returns a node $u \in S$ that contains the key K. If no such node exists, it returns $u = $ `nil`. Next to `lookUp`, perhaps the next most important operation is `insert`.

Since S represents a set of items, two other basic operations we might want to support are inserting an item and deleting an item. If S is subject to both insertions and deletions, we call S a **dynamic set** since its members are evolving over time. In case insertions, but not deletions, are supported, we call S a **semi-dynamic set**. In case both insertion and deletion are not allowed, we call S a **static set**. Thus, the dictionary example (ii) above is a static set from the viewpoint of users, but it is a dynamic set from the viewpoint of the lexicographer.

§2. Abstract Data Types

¶7. Functional versus Self-Modifying Operations.

This section contains a general discussion on abstract data types (ADT's). It may be used as a reference; a light reading is recommended for the first time

Students might be familiar with the concept of **interface** in the programming language Java. In the data structures literature, the general concept is known as **abstract data type** (ADT). Using the terminology of **object-oriented programming languages** (OOPL) such as C++ or Java, we may view a search data structure is an instance of a **container class**. Each instance stores a set of items and have a well-defined set of **members** (i.e., variables) and **methods** (i.e., operations). Thus, a binary tree is just an instance of the "binary tree class". The "methods" of such class support some subset of the following operations listed below.

Java fans: ADT = interface

At this point, we must inject a distinction between mathematical notations and programming notations: in mathematics, there is no built-in concept of of an "object" that persists over time. But the idea of persistence through time is central to programming. Consider a container structure S that stores a set of items. If I is an item that is not in S, we can add it to S by calling a function $\mathtt{insert}(I, S)$ which returns a new structure S' in which I is added. We may denote this as

E.g., mathematical variable vs. programming variables

$$S' \leftarrow \mathtt{insert}(I, S) \quad \text{or} \quad \mathtt{insert}(I, S) \rightarrow S'. \tag{2}$$

Besides the item I, the structure S' also contains copies of all the items in S. This is mathematically very clean, but in programming, it may be a terrible idea to create a new structure S'. We may not want to duplicate the items of S (which may be many) but consider S' as a modified form of S. This is captured by the following form of the insert operation:

$$S.\mathtt{insert}(I) \tag{3}$$

which modifies the structure S to have an additional item I (all previous items are still in S). In OOPL terminology, (3) is calling a method of the class S. We call (2) the **functional form** of the insert operation, and (3) the **self-modifying form**. *Henceforth, when we introduce operations on data structures, we assume that both the functional and self-modifying forms are available. We will freely use whichever form that is more convenient.*

¶8. ADT Operations. We will now list all the main operations found in all the ADT's that we will study. *We emphasize that each ADT will only require a proper subset of these operations. The full set of ADT operations listed here is useful mainly as a reference.* We will organize these operations into four groups (I)-(IV):

	Group	Operation	Meaning
(I)	Initializer and Destroyers	$\texttt{make}()\rightarrow Structure$	creates a structure
		$\texttt{kill}()$	destroys a structure
(II)	Enumeration and Order	$\texttt{list}()\rightarrow Node$	returns an iterator
		$\texttt{succ}(Node)\rightarrow Node$	returns the next node
		$\texttt{pred}(Node)\rightarrow Node$	returns the previous node
		$\texttt{min}()\rightarrow Node$	returns a minimum node
		$\texttt{max}()\rightarrow Node$	returns a maximum node
(III)	Dictionary-like Operations	$\texttt{lookUp}(Key)\rightarrow Node$	returns a node with Key
o		$\texttt{insert}(Item)\rightarrow Node$	returns the inserted node
		$\texttt{delete}(Node)$	deletes a node
		$\texttt{deleteMin}()\rightarrow Item$	deletes a minimum node
(IV)	Set Operations	$\texttt{Split}(Key)\rightarrow Structure$	split a structure into two
		$\texttt{merge}(Structure)$	merges two structures into one

Most applications do not need the full suite of these operations. Below, we will choose various subsets of this list to describe some well-known ADT's. The meaning of these operations are fairly intuitive. We will briefly explain them. Let S, S' be search structures, viewed as instances of a suitable class. E.g., S might be a binary search tree. Let K be a key and u a node. Each of the above operations are invoked from some S: thus, $S.\texttt{make}()$ will initialize the structure S, and $S.\texttt{max}()$ returns the maximum value in S.

When there is a main structure S, we may sometimes suppress the reference to S. E.g., $S.\texttt{merge}(S')$ can be simply written as "$\texttt{merge}(S')$".

Group (I): We need to initialize and dispose of search structures. Thus $\texttt{make}$ (with no arguments) returns a brand new empty instance of the structure. The inverse of $\texttt{make}$ is $\texttt{kill}$, to remove a structure. We view $\texttt{make}$ as a constant time operation, but $\texttt{kill}$ might not be in practice. Nevertheless we can view $\texttt{kill}$ as constant time using amortized analysis.

Group (II): This group of operations are based on some linear ordering of the items stored in the data structure. The operation $\texttt{list}()$ returns a node that is an iterator. This iterator is the beginning of a list that contains all the items in S in *some arbitrary* order. The ordering of keys is not used by the iterators. The remaining operations in this group depend on the ordering properties of keys. The $\texttt{min}()$ and $\texttt{max}()$ operations are obvious. The successor $\texttt{succ}(u)$ (resp., predecessor $\texttt{pred}(u)$) of a node u refers to the node in S whose key has the next larger (resp., smaller) value. This is undefined if u has the largest (resp., smallest) value in S.

Note that $\texttt{list}()$ can be implemented using $\texttt{min}()$ and $\texttt{succ}(u)$ or $\texttt{max}()$ and $\texttt{pred}(u)$. Such a listing has the additional property of sorting the output by key value.

Group (III): The first three operations of this group,

$$\texttt{lookUp}(K) \rightarrow u, \quad \texttt{insert}(K, D) \rightarrow u, \quad \texttt{delete}(u),$$

constitute the "dictionary operations". In many ADT's, these are the central operations.

The node u returned by $\texttt{lookUp}(K)$ has the property that $u.\texttt{key} = K$. In case no such item exists, or it is not unique, some convention should be established. At this abstract level, we purposely leave this under-specified. Each application should further clarify this point.

E.g., in conventional programming languages such as $\texttt{C}$, nodes are usually represented by pointers. In this case, a standard convention is for the $\texttt{lookUp}$ function to return a $\texttt{nil}$ pointer

when there is no item in S with key K. E.g., when the keys are not unique, we may require that `lookUp`(K) returns an iterator that represents the entire set of items with key equal to K.

Both `insert` and `delete` have the obvious basic meaning. In some applications, we may prefer to have deletions that are based on key values. But such a deletion operation can be implemented as '`delete(lookUp(K))`'. In case `lookUp`(K) returns an iterator, we would expect the deletion to be performed over the iterator. Another useful extension is the **replacement operation** `replace`(K, D): replace the data of the item associated with key K with the new data D. This operation would fail if no such item exists. Of course this could be implemented as a deletion followed by an insertion.

The fourth operation $S.$`deleteMin`$()$ in Group (III) is not considered a dictionary operation. The operation returns the minimum item I in S, and simultaneously deletes it from S. Hence, it could be implemented as `delete(min())`. But because of its importance, `deleteMin()` is often directly implemented using special efficient techniques. In most data structures, we can replace `deleteMin` by `deleteMax` without trouble. However, this is not the same as being able to support both `deleteMin` and `deleteMax` simultaneously.

Group (IV): The final group of operations,

$$S.\texttt{Split}(K) \to S', \quad S.Merge(S'),$$

represent manipulation of entire search structures, S and S'. If $S.$`Split`$(K) \to S'$ then all the items in S with keys greater than K are moved into a new structure S'; the remaining items are retained in S. Conversely, the operation $S.$`merge`(S') moves all the items in S' into S, and S' itself becomes empty. This operation assumes that all the keys in S are less than all the items in S'. Thus `Split` and `merge` are inverses of each other.

¶9. Implementation of ADTs using Linked Lists. The basic premise of ADTs is that we should separate specification (given by the ADT) from implementation. We have just given the specifications, so let us now discuss a concrete implementation.

Data structures such as arrays, linked list or binary search trees are called **concrete data types**. Hence ADTs are to be implemented by such concrete data types. We will now discuss a simple implementation of all the ADT operations using linked lists. This humble data structure comes in 8 varieties according to Tarjan [19]. For concreteness, we use the variety that Tarjan calls **endogenous doubly-linked list**. Endogenous means the item is stored in the node itself: thus from a node u, we can directly access $u.$`key` and $u.$`data`. Doubly-linked means u has two pointers $u.$`next` and $u.$`prev`. These two pointers satisfies the invariant $u.$`next` $= v$ iff $v.$`prev` $= u$. We assume students understand linked lists, so the following discussion is a review of linked lists.

Let L be a such a linked list. Conceptually, a linked list is set of nodes organized in some linear order. The linked list has two special nodes, $L.$`head` and $L.$`tail`, corresponding to the first and last node in this linear order. Note that if $L.$`head` $=$ `nil` iff $L.$`tail` $=$ `nil` iff the list is empty. We can visit all the nodes in L using the following routine with a simple while-loop:

List traversal Shell

<div style="border:1px solid red; text-align:center">

List Traversal(L):
$u \leftarrow L.\texttt{head}$
While ($u \neq \texttt{nil}$)
    VISIT(u)
    $u \leftarrow u.\texttt{next}$
CLEANUP()

</div>

302

303 Here, VISIT(u) and CLEANUP() are **macros**, meaning that they stand for pieces of code
304 that will be textually substituted before compiling and executing the program. We will indicate
305 a macro ABC by framing it in a box like ABC . Macros should be contrasted to **subroutines**,
306 which are independent procedures. In most situations, there is no semantic difference between
307 macros and subroutines (except that macros are cheaper to implement). But see the imple-
308 mentation of $\texttt{lookUp}(K)$ next. Note that macros, like subroutines, can take arguments. As a
309 default, the macros do nothing ("no-op") unless we specify otherwise. We call List Traversal   *macros are not*
310 a **shell program** — this theme will be taken up more fully when we discuss tree traversal   *subroutines*
311 below (§4). Since the while-loop (by hypothesis) visits every node in L, there is a unique node
312 u (assume L is non-empty) with $u.\texttt{next} = \texttt{nil}$. This node is $L.\texttt{tail}$.

313   It should be obvious how to implement most of the ADT operations using linked lists. We
314 ask the student to carry this out for the operations in Groups (I) and (II). Here we focus on
315 the dictionary operations:

316   • $\texttt{lookUp}(K)$: We can use the above ListTraversal routine but expand the macro VISIT(u)
317   into the following code fragment:

318
319
<div style="border:1px solid red; text-align:center">

VISIT(u) : if ($u.\texttt{key} = K$) Return(u)

</div>

320   Since **VISIT** is a macro and not a subroutine, the Return in VISIT is *not* a return from
321   **VISIT**, but a return from the $\texttt{lookUp}$ routine! The CLEANUP macro is similarly
322   replaced by

323
324
<div style="border:1px solid red; text-align:center">

CLEANUP() : Return($\texttt{nil}$)

</div>

325   The correctness of this implementation should be obvious.

326   • $\texttt{insert}(K, D)$: We use the ListTraversal shell, but define VISIT(u) as the following
327   macro:

328
329
<div style="border:1px solid red; text-align:center">

VISIT(u) :   if ($u.\texttt{key} = K$) Return($\texttt{nil}$)

</div>

330   Thus, if the key K is found in u, we return $\texttt{nil}$, indicating failure (duplicate key). The
331   CLEANUP() macro creates a new node for the item (K, D) and installs it at the head
332   of the list:

333

$$
\boxed{
\begin{array}{l}
\boxed{\text{CLEANUP}()}: \\
\quad u \leftarrow new(Node) \\
\quad u.\texttt{key} \leftarrow K; \ \ u.\texttt{data} \leftarrow D \\
\quad u.\texttt{next} \leftarrow L.\texttt{head}; \ \ u.\texttt{prev} \leftarrow \texttt{nil} \\
\quad L.\texttt{head.prev} \leftarrow u \quad \lhd \ \textit{No-op if } L.\texttt{head} = \texttt{nil} \\
\quad L.\texttt{head} \leftarrow u \\
\quad \textsf{If } (L.\texttt{tail} = \texttt{nil}) \textsf{ then } L.\texttt{tail} \leftarrow u \\
\quad \textsf{Return}(u)
\end{array}
}
$$

where $new(Node)$ returns a pointer to space on the heap for a node.

- delete(u): Since u is a pointer to the node to be deleted, this amounts to the standard deletion of a node from a doubly-linked list:

$$
\boxed{
\begin{array}{l}
u.\texttt{next.prev} \leftarrow u.\texttt{prev} \\
u.\texttt{prev.next} \leftarrow u.\texttt{next} \\
del(u)
\end{array}
}
$$

where $del(u)$ is a standard routine to return a memory to the system heap. This takes time $O(1)$.

¶10. **Complexity Analysis.** Another simple way to implement our ADT operations is to use arrays (Exercise). In subsequent sections, we will discuss how to implement the ADT operations using In order to understand the tradeoffs in these alternative implementations, we now provide a complexity analysis of each implementation. Let us do this for our linked list implementation.

We can provide a worst case time complexity analysis. For this, we need to have a notion of input size: this will be n, the number of nodes in the (current) linked list. Consistent with our principles in Lecture I, we will perform a Θ-order analysis.

The complexity of $\texttt{lookUp}(K)$ is $\Theta(n)$ in the worst case because we have to traverse the entire list in the worst case. $\texttt{insert}(K, D)$ is preceded by a $\texttt{lookUp}$ and is $\Theta(n)$ in the worst case. The $\texttt{delete}$ operation is $\Theta(1)$. Note that such an efficient deletion is possible because we use doubly-linked lists; with singly-linked lists, we would need $\Theta(n)$ time.

More generally, with linked list implementation, all the ADT operations can easily be shown to have time complexity either $\Theta(1)$ or $\Theta(n)$. The principal goal of this chapter is to show that the $\Theta(n)$ can be replaced by $\Theta(\log n)$. This represents an "exponential speedup" from the linked list implementation.

¶11. **Some Abstract Data Types.** The above operations are defined on typed domains (keys, structures, items) with associated semantics. An **abstract data type** (acronym "ADT") is specified by

- one or more "typed" domains of objects (such as integers, multisets, graphs);

- a set of operations on these objects (such as lookup an item, insert an item);

- properties (axioms) satisfied by these operations.

364 These data types are "abstract" because we make no assumption about the actual implemen-
365 tation.

366   It is not practical or necessary to implement a single data structure that has all the op-
367 erations listed above. Instead, we find that certain subsets of these operations work together
368 nicely to solve certain problems. Here are some of these subsets with wide applicability:

369   • **Dictionary ADT**: `lookUp`[, `insert`[, `delete`]].

370   • **Ordered Dictionary ADT**: `lookUp, insert, delete, succ, pred`.

371   • **Priority queue ADT**: `deleteMin, insert`[, `delete`[, `decreaseKey`]].

372   • **Fully mergeable dictionary ADT**: `lookUp, insert, delete, merge`[, `Split`].

373   In practice, there may be stripped-down versions of the above ADT's in which some oper-
374 ations are omitted. For instance, the **dictionary ADT** normally supports the operations of
375 `lookUp, insert, delete`. Some of these operations may be optional, and in the above listing,
376 the optional operations are enclosed in square brackets: [⋯]. Thus, a dictionary ADT without
377 the `delete` operation is called a **semi-dynamic dictionary**, and if it further omits `insert`, it
378 is called a **static dictionary**. So static dictionaries are down to a bare minimum of the `lookUp`
379 operation, and it is popularly known as a ROM ("read-only memory"). If we omit the `Split`
380 operation in fully mergeable dictionary, then we obtain the **mergeable dictionary** ADT.

What do you get if you omit `lookUp`*? A "write-only memory" (WOM)! See Exercise.*

    Alternatively, some ADT's can be enhanced by additional operations. For instance, a prior-
ity queue ADT traditionally supports only `deleteMin` and `insert`. But in some applications,
it must be enhanced with the operation of `delete` and/or `decreaseKey`. The latter operation
can be defined as

$$\texttt{decreaseKey}(K, K') \equiv [u \leftarrow \texttt{lookUp}(K); \texttt{delete}(u); \texttt{insert}(K', u.\texttt{data})]$$

381 with the extra condition that $K' < K$ (assuming a min-queue). In other words, we change the
382 priority of the item u in the queue from K to K'. Since $K' < K$, this amounts to increasing
383 its priority of u in a min-queue.

384   The operations `make` and `kill` (from group (I)) are assumed to be present in every ADT.

385   Our delete operation is normally given the node u to be deleted, but an alternative is to
386 be given the key K to be deleted. If all our accesses to the data are through keys only –
387 Lookup(K), Insert(K, D), Delete(K) – then we may regard the dictionary data structure as
388 a kind of **associative memory**. An interesting area of computer architecture addresses the
389 problem of constructing, in hardware, the ability to arbitrarily associate keys to data (as in our
390 items), and to retrieve the associated data based on key values.

391   Variant interpretations of these operations are possible. For instance, some version
392 of `insert` may wish to return a boolean (to indicate success or failure) or not to return
393 any result (in case the application will never have an insertion failure). Other useful
394 functions can be derived from the above. E.g., it is useful to be able to create a structure
395 S containing just a single item I. This can be reduced to 'S.`make`(); S.`insert`(I)'. The
396 concept of ADT was a major research topic in the 1980's. Many of these ideas found their
397 way into structured programming languages such as Pascal and their modern successors.
398 An interface in Java is a kind of ADT where we capture only the types of operation. Our
399 discussion of ADT is informal, but one way to study them formally is to describe axioms

that these operations satisfy. For instance, if S is a stack, then we can postulate the axiom that pushing an item x on S followed by popping S should return the item x. In our treatment, we will rely on informal understanding of these ADT's to avoid the axiomatic treatment.

¶12. Application to Pseudo-Heapsort   In Chapter I, we introduce the Mergesort Algorithm which was analyzed in Chapter II to have complexity $T(n) = 2T(n/2) + n = \Theta(n \log n)$. We now give another solution to the sorting problem based on the (stripped down) priority queue ADT: in order to sort an array $A[1..n]$ of items, we insert each item $A[i]$ into a priority queue Q, and then remove them from Q using `deleteMin`:

Pseudo-Heapsort(A, n):
   Input: An array A of n items
   Output: The sorted array A
1. $Q \leftarrow \texttt{make}()$
2.   for $i = 1$ to n do
              $Q.\texttt{insert}(A[i])$
3.   for $i = 1$ to n do
              $A[i] \leftarrow Q.\texttt{deleteMin}()$
4.   Return(A)

The correctness of the algorithm is obvious. As each priority queue operation is $O(\log n)$, this gives another $O(n \log n)$ solution to sorting.

Why do we call this "pseudo-heapsort"? The classic Heapsort is actually more sophisticated than this – in particular, it implements its own priority queue using the array A! The idea of classic Heapsort is that a subarray $A[i..j]$ can be viewed as a complete binary tree, and organized as a priority queue Q. We leave this to an Exercise.

---Exercises

Exercise 2.1: Recall our discussion of pointer semantics. Consider the concept of a "pointer to a pointer" (also known as a **handler**).
(a) Let the variable p, q have the type pointer-to-pointer-to-node, while u and N have types pointer-to-node and node (resp.). It is clear what $p \leftarrow q$ means. But what should '$p \leftarrow u$', '$p \leftarrow N$', '$N \leftarrow p$', and '$u \leftarrow p$' mean? Or should they have meaning?
(b) Give some situations where this concept might be useful.   ◇

Exercise 2.2: In ¶9, we provided implementations of the dictionary operations using linked list. Please complete this exercise by implementing the full suite of ADT operations using linked lists. We want you to do this within the shell programming framework.   ◇

Exercise 2.3: A dictionary ADT is normally denoted ADT(`lookUp`, `insert`, `delete`), indicating support for all 3 operations. But we could have a semi-dynamic dictionary, denoted ADT(`lookUp`,`insert`) or a static dictionary denoted ADT(`lookUp`). THe latter is popularly known as a ROM ("read only memory"). But be careful: the notion of ROM would

be meaningless if nobody could write into the ADT initially! Therefore, these stripped down dictionaries only represent the **end-user** viewpoint; there is a class of users called **administrator** that has full access to the 3 dictionary operations. Let us explore more variations. In this question, assume that keys are unique in the ADT.

(a) Consider ADT(`insert`,`delete`) which we will call a WOM ("write only memory"). Discuss some possible uses for a WOM.

(b) Consider ADT(`delete`) which we will call a PWOM ("permanent WOM"). Discuss some possible uses for a PWOM.

(c) Food for thought: Can a WOM or PWOM be implemented more efficiently than a dictionary? ◇

Exercise 2.4: Consider the dictionary ADT.

(a) Describe algorithms to implement this ADT when the concrete data structures are arrays. HINT: A difference from implementation using linked lists is to decide what to do when the array is full. How do you choose the larger size? What is the analogue of the ListTraversal Shell?

(b) Analyze the complexity of your algorithms in (a). Compare this complexity with that of the linked list implementation. ◇

Exercise 2.5: Repeat the previous question for the priority queue ADT. ◇

Exercise 2.6: Suppose D is a dictionary with the dictionary operations of lookup, insert and delete. List a complete set of axioms (properties) for these operations. ◇

_____ End Exercises

§3. Binary Search Trees

We introduce binary search trees and show that such trees can support all the operations described in the previous section on ADT. Our approach will be somewhat unconventional, because we want to reduce all these operations to the single operation of "rotation".

¶13. Binary Trees. The general concept of trees are summarized ¶I.A.10. Here, we are interested in a very special species of trees called **binary trees**. These trees are of paramount importance in computer science. Figure 2 shows two binary trees (small and big) which we will use in our illustrations. For each node u of the tree, we store a value $u.\texttt{key}$ called its key. The keys in Figure 2 are integers, used simply as identifiers for the nodes.

Definition 1 _Formally, we define a **binary tree** T to be a set N of **nodes** where each node u_0 has two pointers, $u_0.\texttt{left}$ and $u_0.\texttt{right}$ subject the following requirements:_

- _(Basis) If N is the empty set, then T is called the empty tree._

- _(Induction) If N is non-empty, then it has a distinguished node u_0 called the **root** of T. The remaining nodes $N \backslash \{u_0\}$ are partitioned into two sets of nodes that, recursively, form_
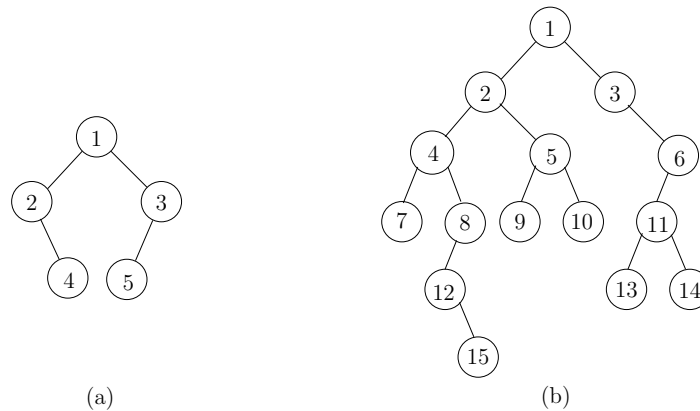
Figure 2: Two binary (not search) trees: (a) small, (b) big

binary trees, T_L and T_R called the **left** and **right subtrees** of T. Moreover, u_0.left
and u_0.right *point to the roots of T_L and T_R (resp.). If any subtree is empty, then the
corresponding pointer is* nil.

The **size** of T is $|N|$. We often identify T with the set of nodes N, and so the size may be
denoted $|T|$, and we may write "$u \in T$" instead of "$u \in N$". Figure 2 illustrates two binary
trees whose node sets are (respectively) $N = \{1, 2, 3, 4, 5\}$ (small tree) and $N = \{1, 2, 3, \ldots, 15\}$
(big tree). The **height** $\mathtt{ht}(T)$ of a binary T as follows:

$$\mathtt{ht}(T) := \begin{cases} -1 & \text{if } T \text{ is empty,} \\ 1 + \max\{\mathtt{ht}(T_L), \mathtt{ht}(T_R)\} & \text{if } T \text{ has subtrees } T_L, T_R. \end{cases} \tag{4}$$

> **Structural Induction for binary trees**: our definition of binary
> trees provides an in-built inductive structure on the set of binary trees.
> Definitions and proofs can exploit this structure. Thus, our definition of
> height in (4) illustrates this. Similarly, "proof by structural induction"
> works analogously like natural induction. *Most properties of binary
> trees are best proved by structural induction.* Likewise, algorithms for
> binary trees are often best described by structural induction.

A node u is a **leaf** if $\mathtt{ht}(u) = 0$; non-leafs are also called **internal nodes**. We also view
T as a directed graph on the node set N and the edges $(u-v)$ iff u.left $= v$ or u.right $= u$.
There is a unique path from the root of T to each $u \in T$, and **depth** of u is the length of this
path. The set of all nodes of a binary tree with the same depth d forms the dth **level**. The dth
level is **complete** if it has the maximum possible size, namely, 2^d. A node in a binary tree is
full if it has two children; a binary tree is said to be **full** if every internal node is full. A simple
result about non-empty full binary trees is that it has exactly one fewer internal node than the
number of leaves. Thus, if it $k \geqslant 1$ leaves iff it has $k - 1 \geqslant 0$ internal nodes. As corollary, a
full binary tree has an odd number of nodes. We define a **complete binary tree** to be one in
which every level except possibly the last one is complete; moreover, the last level is filled from
left-to-right. Further, there is a most one internal node of degree 1, and such a node has a left-
but no right-child. All other internal nodes has degree 2. A complete binary tree whose last
level is also complete is said to be **perfect**. As in real life, perfection is hard to attain: perfect
trees have sizes that are one less than a power of 2. That is, their sizes are $1, 3, 7, 15, \ldots$. On

489   the other hand, for every $n \geqslant 0$, there is a unique complete binary tree of size n. A binary tree
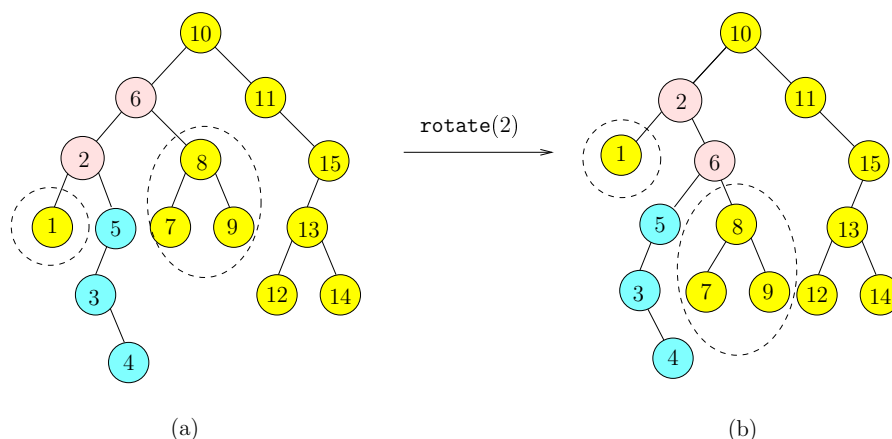490   is said to be **linear** if each node u has at most one child. So it is essentially a linear list.

Figure 3: (a) Binary Search Tree on keys $\{1, 2, 3, 4, \ldots, 14, 15\}$. (b) After `rotate(2)`.

491   The keys of the binary trees in Figure 2 are just used as identifiers. To turn them into a
492   binary *search* tree, we must organize the keys in a particular way. Such a binary search tree is
493   illustrated in Figure 3(a). Structurally, it is the big binary tree from Figure 2(b), but now the
494   keys are no longer just arbitrary identifiers.

BST are binary trees that satisfy the BST property!

495   **¶14. Binary Search Trees.**   A binary tree T is called a **binary search tree** (BST) if each
496   node $u \in T$ has a field $u.\texttt{key}$ that satisfies the **BST property**:

$$u_L.\texttt{key} < u.\texttt{key} \leqslant u_R.\texttt{key}. \tag{5}$$

497   for any left descendant u_L and any right descendant u_R of u. Please verify that the binary
498   search trees in Figure 3 obey (5) at each node u. The "standard mistake" is to replace (5) by
499   $u.\texttt{left.key} < u.\texttt{key} \leqslant u.\texttt{right.key}$. By definition, a left (right) descendant of u is a node in
500   the subtree rooted at the left (right) child of u. The left and right children of u are denoted
501   by $u.\texttt{left}$ and $u.\texttt{right}$. This mistake focuses on a necessary, but not sufficient, condition for
502   a BST. Self-check: construct a counter example to the standard mistake using a binary tree of
503   size 3.

good quiz question...

504   **¶15. Minimum and maximum sizes at a given height**   Let $M(h)$ and $\mu(h)$ (resp.) be
505   the maximum and minimum number of nodes in a binary tree with height h. It is easy to see
506   that

$$\mu(h) = h + 1. \tag{6}$$

507   What about $M(h)$? Clearly, $M(0) = 1$ and $M(1) = 3$. Inductively, we can see that $M(h+1) =$
508   $1 + 2M(h)$. Thus $M(1) = 1 + 2M(0) = 3$, $M(2) = 1 + 2M(1) = 7$, $M(3) = 1 + 2M(2) = 15$.
509   From these numbers, you might guess that

$$M(h) = 2^{h+1} - 1 \tag{7}$$

510   and it is trivial to verify this for all h. Another way to see $M(h)$ is that it is equal to $\sum_{i=0}^{h} 2^i$
511   since there are at most 2^i nodes at level i, and this bound can be achieved at every level. The

512   simple formula (7) tells us a basic fact about the minimum height of binary trees on n nodes:
513   if its height is h, then clearly, $n \leqslant M(h)$ (by definition of $M(h)$). Thus $n \leqslant 2^{h+1} - 1$, leading to

$$h \geqslant \lg(n+1) - 1. \tag{8}$$

514   Thus *the height of a binary tree is at least logarithmic in the size.* This simple relation is critical
515   in understanding complexity of algorithms on binary trees.

516   **¶16. Lookup.**   The lookup algorithm in a binary search tree is almost immediate from the
517   binary search tree property: to look for a key K, we begin at the root (recall the Rule about
518   structural induction above). In general, suppose we are looking for K in some subtree rooted at
519   node u. If $u.\texttt{key} = K$, we are done. Otherwise, either $K < u.\texttt{key}$ or $K > u.\texttt{key}$. In the former
520   case, we recursively search the left subtree of u; otherwise, we recurse in the right subtree of
521   u. In the presence of duplicate keys, what does lookup return? There are two interpretations:
522   (i) We can return the first node u we found to have the given key K. (ii) We may insist on
523   locating all nodes whose key is K.

524   In any case, requirement (ii) can be regarded as an extension of (i), namely, given a node u,
525   find all the other nodes below u with same same key as $u.\texttt{key}$. This subproblem can be solved
526   separately (Exercise). Hence we may assume interpretation (i) in the following.

527   **¶17. Insertion.**   To insert an item, say the key-data pair (K, D), we proceed as in the Lookup
528   algorithm. If we find K in the tree, then the insertion fails (assuming distinct keys). Otherwise,
529   we would have reached a node u that has at most one child. We then create a new node u'
530   containing the item (K, D) and make u' into a child of u. Note that if $K < u.\texttt{key}$, then u'
531   becomes a left child; otherwise a right child. In any case, u' is now a leaf of the tree.

532   **¶18. Rotation.**   Rotation is not an operation in our list of ADT operation (§2), but we view
533   it as the critical operation for binary trees. Roughly, to rotate a node u means to make the
534   parent of u become its child. The set of nodes is unchanged. On the face of it, rotation does
535   nothing essential: it is just redirecting some parent/child pointers. Rotation is an **equivalence**
536   **transformation**, i.e., it transforms a binary search tree into an equivalent one. Two search
537   structures that store exactly the same set of items are said to be **equivalent**. Remarkably, we
538   shall show that rotation can[9] form the basis for all other binary tree operations.

539   The operation $\texttt{rotate}(u)$ is a "no-op" operation (or identity transformation) when u is a
540   root. So assume u is a non-root node in a binary search tree T. Then $\texttt{rotate}(u)$ amounts to
541   the following transformation of T (see Figure 4).

       As seen in Figure 4, $\texttt{rotate}(u)$ amounts to inverting the parent-child relation between u
   and its parent v. The other necessary actions are automatic, given that the result is to remain
   a binary search tree. Let us explain this. Let the subtrees of u and v be A, B, C (any of these
   can be empty) as indicated in Figure 4(left side). Where do they go after rotation? They must
   reattach as children of the rotated u and v. We know that

$$A < u < B < v < C$$

542   in the sense that each key in A is less than u which is less than any key in B, etc. So, the
543   only way to maintain the BST property is to re-attach them as shown in Figure 4(right side).

---

[9]Augmented by the primitive operations of adding or removing a node.
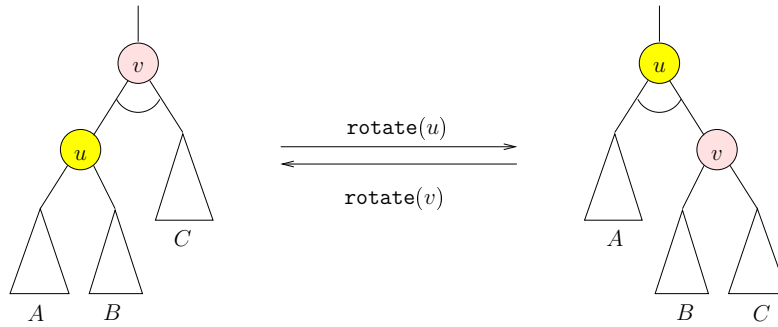
Figure 4: Rotation at u and its inverse.

544 Only the parent of the root of B has switched from u to v; the parents of the root of A and
545 C are unchanged. Furthermore, after `rotate`(u), the former parent of v (not shown) will now
546 have u instead of v as a child. After a rotation at u, the depth of u is decreased by 1. Note
547 that `rotate`(u) followed by `rotate`(v) is the identity or no-op operation; this is indicated by
548 the left-arrow in Figure 4.

549 **¶19. Graphical convention:**   Figure 4 encodes two conventions: consider the figure on the
550 left side of the arrow. First, the edge connecting v to its parent is directed vertically upwards.
551 This indicates that v could be the left- or right-child of its parent. Second, the two edges from v
552 to its children are connected by a circular arc. This is to indicate that u and its sibling could[10]
553 exchange places (i.e., u could be the right-child of v even though we choose to show u as the
554 left-child). Thus Figure 4 is a compact way to represent four distinct situations.

555 **¶* 20. Implementation of rotation.**   Let us discuss how to implement rotation. Until
556 now, when we draw binary trees, we only display child pointers. But we must now explicitly
557 discuss parent pointers.

558   We can classify a node u as one of three **types**: *left*, *right* or *root*. This is defined in the
559 obvious way. E.g., u is a left type iff it is not a root and is a left child. The type of u is easily
560 tested: u is type root iff u.`parent` = `nil`, and u is type left iff u.`parent.left` = u. Clearly,
561 `rotate`(u) is sensitive to the type of u. In particular, if u is a root then `rotate`(u) is the no-op
562 operation. If $T \in \{$`left`, `right`$\}$ denotes a type, its **complementary type** is denoted $\overline{T}$, where
563 $\overline{\texttt{left}} = \texttt{right}$ and $\overline{\texttt{right}} = \texttt{left}$.

564   We are ready to discuss the `rotate`(u) subroutine. We assume that it will return the (same)
565 node u. Assume the type of u is $T \in \{$`left`, `right`$\}$. Let $v = u$.`parent`, $w = v$.`parent` and
566 $x = u.\overline{T}$. Note that w and x might be `nil`. Thus we have up to three child-parent pairs:

x is an inner grandchild of v

$$(x, u), (u, v), (v, w). \tag{9}$$

567 But after rotation, u and v are interchanged, and we have the following child-parent pairs:

$$(x, v), (v, u), (u, w). \tag{10}$$

568 These pairs are illustrated in Figures 5–6 where we explicitly show the parent pointers as
569 well as child pointers. Thus, to implement rotation, we need to reassign 6 pointers (3 parent

---

[10]If this were to happen, the subtrees A, B, C needs to be appropriately relabeled.
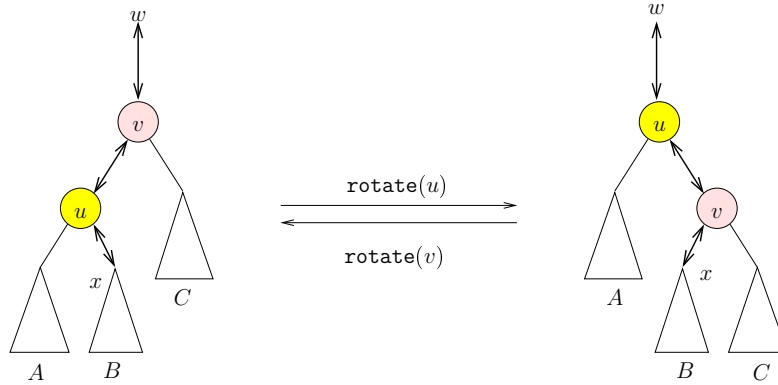
Figure 5: Links that must be fixed in `rotate(u)`.

pointers and 3 child pointers). We show that it is possible to achieve this re-assignment using exactly 6 assignments.
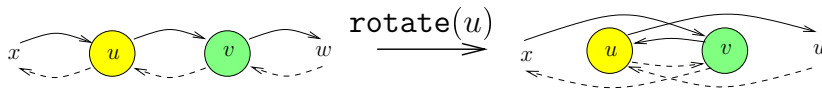
Figure 6: Simplified view of `rotate(u)` as fixing a doubly-linked list (x, u, v, w).

Such re-assignments must be done in the correct order. It is best to see what is needed by thinking of (9) as a doubly-linked list (x, u, v, w) which must be converted into the doubly-linked list (x, v, u, w) in (10). This is illustrated in Figure 6. For simplicity, we use the terminology of doubly-linked list so that u.next and u.prev are the forward and backward pointers of a doubly-linked list. Here is the code:

Rotate(u):
     ▷ *Fix the forward pointers*
1. u.prev.next ← u.next
         ◁ x.next = v
2. u.next ← u.next.next
         ◁ u.next = w
3. u.prev.next.next ← u
         ◁ v.next = u
     ▷ *Fix the backward pointers*
4. u.next.prev.prev ← u.prev
         ◁ v.prev = x
5. u.next.prev ← u
         ◁ w.prev = u
6. u.prev ← u.prev.next
         ◁ u.prev = v

We can now translate this sequence of 6 assignments into the corresponding assignments for binary trees: the u.next pointer may be identified with u.parent pointer. However, u.prev would be $u.T$ where $T \in \{\texttt{left}, \texttt{right}\}$ is the type of x. Moreover, v.prev is $v.\overline{T}$. Also w.prev

583 is $w.T'$ for another type T'. A further complication is that x or/and w may not exist; so these
584 conditions must be tested for, and appropriate modifications taken.

585   If we use temporary variables in doing rotation, the code can be simplified (Exercise).

586 **¶21. Variations on Rotation.**   The above rotation algorithm assumes that for any node
587 u, we can access its parent u' and grandparent u''. This is true if each node has a parent
588 pointer $u.\texttt{parent}$. *This is our default assumption for binary tree algorithms.* But even if we
589 have no parent pointers, we could modify our algorithms to achieve the desired results because
590 our search invariably starts from the root, and we can keep track of the triple (u, u', u'') which
591 is necessary to know when we rotate at u.

   Some authors replace rotation by a pair of variants, called **left-rotation** and **right-rotation**. These can be defined[11] as follows:

$$\texttt{left-rotate}(u) \equiv \texttt{rotate}(u.\texttt{left}), \qquad \texttt{right-rotate}(u) \equiv \texttt{rotate}(u.\texttt{right}).$$

592 The advantage of using these two rotations is that, if we do not maintain parent pointers, then
593 they are slightly easier to implement than our rotate: we only make sure that whenever we are
594 operating on a node u, we also keep track of the parent p of u (however, we do not need to
595 know the parent of p). After we do a $\texttt{left-rotate}(u)$ or $\texttt{right-rotate}(u)$, we need to update
596 one of the child pointers of p.

597 **¶22. Double Rotation.**   Suppose u has a parent v and a grandparent w. Then two successive
598 rotations on u will ensure that v and w are descendants of u. We may denote this operation
599 by $\texttt{rotate}^2(u)$. Up to left-right symmetry, there are two distinct outcomes in $\texttt{rotate}^2(u)$: (i)
600 either v, w are becomes children of u, or (ii) only w becomes a child of u and v a grandchild of
601 u. These depend on whether u is the **outer** or **inner** grandchildren of w. These two cases are
602 illustrated in Figure 7. [As an exercise, we ask the reader to draw the intermediate tree after
603 the first application of $\texttt{rotate}(u)$ in this figure.]

604   It turns out that case (ii) is the more important case. For many purposes, we would like to
605 view the two rotations in this case as one indivisible operation: hence we introduce the term
606 **double rotation** to refer to case (ii) only. For emphasis, we might call the original rotation a
607 **single rotation**.

So "double rotation" is a restricted form of $\texttt{rotate}^2(u)$!

608   Case (i) is known as as the zig-zig or zag-zag cases, and Case (ii) as the zig-zag or zag-zig
609 cases. This terminology comes from viewing a left turn as zig, and a right turn as zag, as we
610 move from up a root path. The Exercise considers how we might implement a double rotation
611 more efficiently than by simply doing two single rotations.

612 **¶23. Rotation Distance Between Trees.**   Given two binary search trees, T and T', we
613 define their **rotation distance** $R(T, T')$ to be the minimum number of rotations to convert T
614 to T'. Of course, if T, T' are not equivalent, then $R(T, T') = \infty$. It is not hard to show that
615 $R(T, T') = O(n^2)$ if T, T' are equivalent. Let $R(n)$ denote the rotation distance between two
616 equivalent full BST's with n internal nodes (hence $n + 1$ external nodes). Sleator, Tarjan and
617 Thurston (1988) has shown that $R(n) \leqslant 2n - 6$ for $n \geqslant 11$, and this bound is tight for n large
618 enough. This is a deep result. There is no known polynomial time to compute $R(T, T')$. Culik

---

[11]Careful: "$\texttt{left-rotate}(T, u)$" in [6] corresponds our $\texttt{right-rotate}(u)$ (and vice-versa).
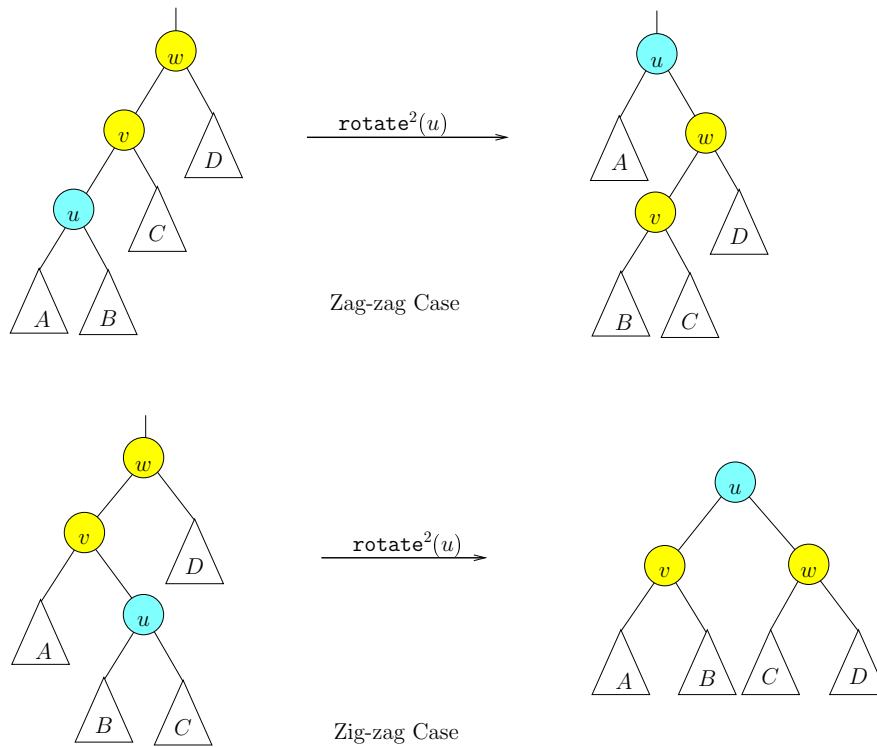
Figure 7: Two outcomes of $\texttt{rotate}^2(u)$

and Wood (1982) has shown that $R(n) \leqslant 2n - 2$. Call a BST T a **left-list** if the right child of each node is null. Let T be a full BST with n internal nodes and let L_n be an equivalent full BST tree that is a left-list. Call L_n the **canonical form** of T. Let us rank the internal nodes from smallest to the largest: the one containing the smallest (largest) key has rank 1 (n).

We describe a recursive procedure to convert T into L_n: Let u_0 be the internal node of rank 1. Clearly, the left child of u_0 is a leaf containing the minimum key in T.

Conversion to Left-List:
1.   Initialize u to be u_0, internal node of rank 1.
2.   While $u \neq root$ ◁ *The left subtree of u is a left-list*
3.   While $(u.\texttt{right} \neq leaf)$
4. $\texttt{rotate}(u.\texttt{right})$ ◁ *Increment $depth(u)$*
5. $u \leftarrow u.\texttt{parent}$ ◁ *Increment $rank(u)$*

Lemma 1 *(Culik and Wood)*
$R(T, L_n) \leqslant n - 1$.

Proof. The left-subtree of u at the beginning of Step 2 is a left-list. Each rotation in Line 4 causes this left-list to increase by 1. Therefore, we make at most $n - 1$ rotations before u

---

633 becomes the root, and the algorithm terminates at Step 2.   **Q.E.D.**

634

635 As corollary, we have *if T' is equivalent to T then $R(T, T') \leqslant 2n - 2$*. To see this, note
636 that rotations are reversible operations, and therefore the lemma implies $R(T', L_n) \leqslant n - 1$.
637 By the lemma, we also have $R(T, L_n) \leqslant n - 1$. Therefore, $R(T, T') \leqslant R(T, L_n) + R(L_n, T') \leqslant$
638 $(n - 1) + (n - 1)$.

639 **¶24. Five Canonical Paths from a node.**   A **path** in a binary tree is a sequence of nodes
640 $(u_0, u_1, \ldots, u_n)$ for some $n \geqslant 0$ such that one of the following holds:

641   • (**downward path**) u_i is a child of u_{i-1} ($\forall i = 1, \ldots, n$).

642   • (**upward path**) u_i is a parent of u_{i-1} ($\forall i = 1, \ldots, n$).

643 The length of this path is n, and u_n is called the **tip** of the path. If $n = 0$, the path is **trivial**.
644 Note that the upward/downward designation is unique for non-trivial paths. This up/down
645 designation comes from the convention of drawing binary trees so that a parent node is always
646 above a child node (and thus the root is at the top of the diagram).

647 Relative to any node u in a binary tree, there are 5 **canonical paths** that starts from u:

648   • The **root-path** of u is the unique upward from u to the root.

5 *paths from a node*

649   • The **left-path** of u is the downward path of maximum length from u such that each
650   successive node is a left child. The **right-path** of u is similarly defined.

     • The **left-spine** of u has this form:

$$(u, \text{right-path}(u.\texttt{left})) = (u, u.\texttt{left}, u.\texttt{left.right}, u.\texttt{left.right}^2, \ldots,).$$
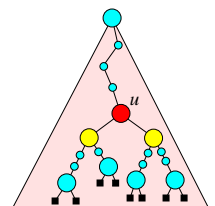
651   The **right-spine** is similarly defined.

OK, it is confusing:
left-spine is based
on *right*-path!

652 Example: consider the binary tree in Figure 2(b). If $u = 4$, then the root-path is $(4, 2, 1)$, the
653 left-path is $(4, 7)$, the right-path is $(4, 8)$, the left-spine is $(4, 7)$ and the right-spine is $(4, 8, 12)$.

654 Any of the 5 canonical paths might be trivial. The root-path of u is trivial iff u is the root;
655 the left-spine is trivial iff u has no left-child. By the **left-tip** of u, we mean the tip of the
656 left-path of u. Similarly for the **right-tip**. Clearly, in a BST, the left-tip and right-tip of u
657 correspond to the minimum and maximum keys in the subtree at u.

658 The tips of the left- and right-spines of u also have natural interpretations as successors and
659 predecessors of u: *Provided the left-spine (right-spine) of u is nontrivial*, the left-tip (right-tip)
660 corresponds to the predecessor (successor) of u. But if the left-spine of u is trivial, then there
661 is a unique node v such that u is the tip of the right-spine of v (and thus the predecessor of u
662 is v). A similar statement holds when the right-spine of u is trivial.

663 We now examine what happens to these five paths of u after a rotation. After performing
664 a left-rotation at u, we reduce the left-spine length of u by one (but the right-spine of u is
665 unchanged). See Figure 8.

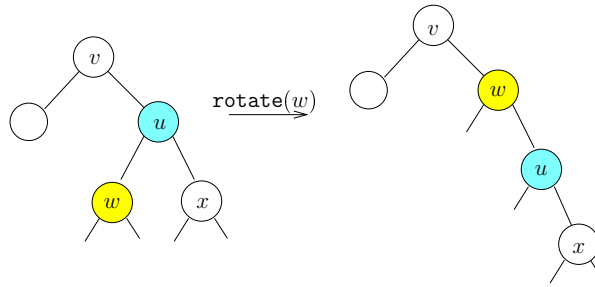*How rotations
affect the 5 paths*

Figure 8: Reduction of the left-spine of u after $\mathtt{rotate}(u.\mathtt{left}) = \mathtt{rotate}(w)$.

Lemma 2 *Let $(u_0, u_1, \ldots, u_k)$ be the left-spine of $u = u_0$ and $k \geqslant 1$. Also let $(v_0, \ldots, v_m)$ be the root-path of $u = v_0$. After performing $\mathtt{rotate}(u.\mathtt{left})$, the left-child of u is transferred from the left-spine to the root-path. More precisely:*

   (i) the left-spine of u becomes $(u_0, u_2, \ldots, u_k)$ of length $k - 1$,

   (ii) the root-path of u becomes $(v_0, u_1, v_1, \ldots, v_m)$ of length $m + 1$, and

   (iii) the right-path and right-spine of u are unchanged.

So repeatedly left-rotations at u will reduce the left-spine of u to length 0. A similar property holds for right-rotations.

¶25. Deletion. Suppose we want to delete a node u. There is nothing to do if u is the root. Otherwise let p be its parent. In case u has at most one child, the deletion of u is easy: we simply redirect p's pointer to u into the child of u, thereby eliminating u. More precisely, let v be the unique child of u (or $v = \mathtt{nil}$ if u is a leaf). Call this procedure $Cut(u, p, v)$ or simply $Cut(u)$ if we assume that p, v is automatically generated from u: *the $Cut(u)$ operation*

$$Cut(u) \equiv [\text{if } (p.\mathtt{left} = u) \text{ then } p.\mathtt{left} \leftarrow v; \text{ else } p.\mathtt{right} \leftarrow v]$$

It is now easy to describe a general algorithm for deleting a node u:

Delete(T, u):
    Input: u is node to be deleted from T.
    Output: T, the tree with u deleted.
       While $u.\mathtt{left} \neq \mathtt{nil}$ do
          $\mathtt{rotate}(u.\mathtt{left})$.
       $Cut(u)$

The overall effect of this algorithm is schematically illustrated in Figure 9.

Note that Delete amounts to moving down the left-spine of the node u to be deleted (see Figure 9(i)). But we could also move down the right-spine. If we maintain information about the left and right spine heights of nodes (Exercise), we could choose to go down the spine that

682 is shorter. To avoid maintaining spine height information, we can also do this: alternately
683 perform left- and right-rotates at u until one of its 2 spines have length 0. This guarantees that
684 the number of rotations is never more than twice the minimal needed.
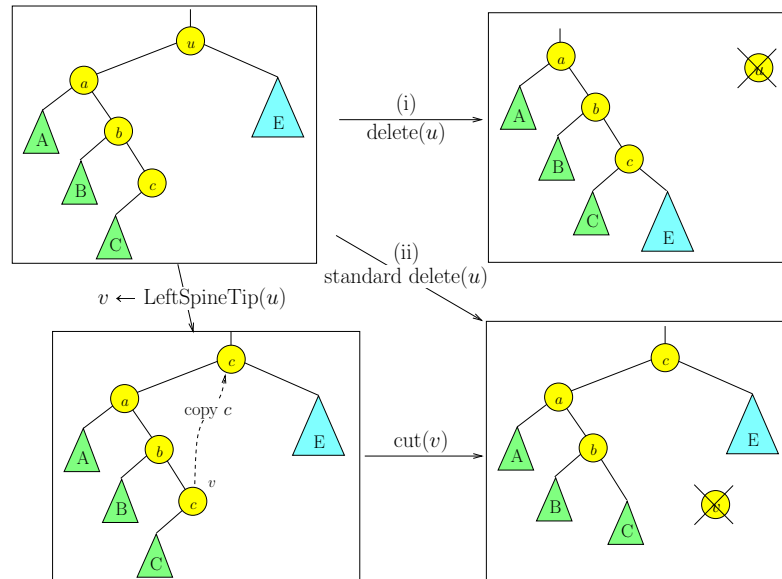

Figure 9: Deletion: (i) Rotation-based, (ii) Standard.

685 We ask the reader to simulate the operations of $\text{Delete}(T, 10)$ where T is the BST of
686 Figure 3.

687 **¶26. Standard Deletion Algorithm.** The preceding deletion algorithm is simple but it is
688 quite non-standard. We now describe the **standard deletion algorithm**:

689

690

691

> Standard Delete(T, u):
>   Input: u is node to be deleted from T.
>   Output: T, the tree with item in u deleted.
>   if u has at most one child, apply $Cut(u)$ and return.
>   else let v be the tip of the left spine of u.
>   Copy the item in v into u (thus erasing the old item in u)
> $Cut(v)$.

692 This process is illustrated in Figure 9. Note that in the else-case, the node u is not physically
693 removed: only the item represented by u is removed. Since v is the tip of the left spine, it has
694 at most one child, and therefore it can be cut. If we have to return a value, it is useful to return
695 the parent of the node v that was cut – this can be used in rebalancing tree (see AVL deletion
696 below). The reader should simulate the operations of $\text{Delete}(T, 10)$ for the tree in Figure 3,
697 and compare the results of standard deletion to the rotation-based deletion.

698 The rotation-based deletion is conceptually simpler, and will be useful for amortized al-
699 gorithms later. However, the rotation-based algorithm seems to be slower as it requires an

700 unbounded number of pointer assignments. To get a definite complexity benefit, we could
701 perform this rotation in the style of splaying (Chapter VI, Amortization).

702 **¶27. Inorder listing of a binary tree.**

703 **Lemma 3** *Let T be a binary tree on n nodes. There is a unique way to assign the keys*
704 $\{1, 2, \ldots, n\}$ *to the nodes of T such that the result is a binary search tree on these keys.*

705   We leave the simple proof to an Exercise. For example, if T is the binary tree in Figure 2(b),
706 then this lemma assigns the keys $\{1, \ldots, 15\}$ to the nodes of T as in Figure 3(a). In general,
707 the node that is assigned key i ($i = 1, \ldots, n$) by Lemma 3 may be known as the i**th node** of
708 T. In particular, we can speak of the **first** ($i = 1$) and **last node** ($i = n$) of T. The unique
709 enumeration of the nodes of T from first to last is called the **in-order listing** of T.

710   See ¶26 under the topic of Tree Traversal for an alternative description of in-order listing.

711 **¶28. Successor and Predecessor.** If u is the ith node of a binary tree T, the **successor** of
712 u refers to the $(i+1)$st node of T. By definition, u is the **predecessor** of v iff v is the successor
713 of u. Let $\mathtt{succ}(u)$ and $\mathtt{pred}(u)$ denotes the successor and predecessor of u. Of course, $\mathtt{succ}(u)$
714 (resp., $\mathtt{pred}(u)$) is undefined if u is the last (resp., first) node in the in-order listing of the tree.

715   We will define a closely related concept, but applied to any key K. Let K be any key, not
716 necessarily occurring in T. Define the **successor** of K in T to be the least key K' in T such
717 that $K < K'$. We similarly define the **predecessor** of K in T to be the greatest K' in T such
718 that $K' < K$. Note that if K occurs in T, say in node u, then the successor/predecessor of K
719 are just the successor/predecessor of u.

720   In some applications of binary trees, we want to maintain pointers to the successor and
721 predecessor of each node. In this case, these pointers may be denoted $u.\mathtt{succ}$ and $u.\mathtt{pred}$. It is
722 an important fact that it worth noting:

723 **Lemma 4** *Successor/predecessor pointers of nodes are unaffected by rotations.*

724
725 *Proof.* The successor/predecessor of a node is completely determined by the in-order listing of
726 the nodes in the tree. More precisely, if $(u_1, \ldots, u_n)$ is the inorder listing, then $u_i.\mathtt{succ} = u_{i+1}$
727 (for $i = 1, \ldots, n-1$) and $u_i.\mathtt{pred} = u_{i-1}$ (for $i = 2, \ldots, n$). Since the in-order listing is
728 preserved by rotation, the pointers are unchanged.   **Q.E.D.**

729

730   *Our default version of binary trees do not include such pointers.* Let us make some simple
731 observations:

732 **Lemma 5** *Let u be a node in a binary tree, but u is not the last node in the in-order traversal*
733 *of the tree. Let $\mathtt{succ}(u) = v$.*
734 *(i) If $u.\mathtt{right} \neq \mathtt{nil}$ then v is the tip of the right-spine of u.*
735 *(ii) If $u.\mathtt{right} = \mathtt{nil}$ then u is the tip of the left-spine of v.*

736   It is easy to derive an algorithm for $\texttt{succ}(u)$ using this lemma:

737

> Succ(u):
> Output: The successor node of u (if it exists) or $\texttt{nil}$.
> 1.   if $u.\texttt{right} \neq \texttt{nil}$ ◁ *return the tip of the right-spine of u*
> 1.1 $v \leftarrow u.\texttt{right}$;
> 1.2   While $v.\texttt{left} \neq \texttt{nil}$, $v \leftarrow v.\texttt{left}$;
> 1.3   Return(v).
> 2.   else   ◁ *return v where u is the tip of the left-spine of v*
> 2.1 $v \leftarrow u.\texttt{parent}$;
> 2.2   While $v \neq \texttt{nil}$ and $u = v.\texttt{right}$,
> 2.3 $(u, v) \leftarrow (v, v.\texttt{parent})$.
> 2.4   Return(v).

739   The algorithm for $\texttt{pred}(u)$ is similar. The Exercise develops a rotation-based version of successor
740   or predecessor.

741   **¶29. Min, Max, DeleteMin.**   This is trivial once we notice that the minimum (maximum)
742   item is in the first (last) node of the binary tree. Moreover, the first (last) node is at the tip of
743   the left-path (right-path) of the root.

744   **¶30. Merge.**   To merge two trees T, T' where all the keys in T are less than all the keys in T',
745   we proceed as follows. Introduce a new node u and form the tree rooted at u, with left subtree
746   T and right subtree T'. Then we repeatedly perform left rotations at u until $u.\texttt{left} = \texttt{nil}$.
747   At this point, we can perform $Cut(u)$ (see ¶25). If you like, you can perform right rotations
748   instead of left rotations.

749   **¶31. Split.**   Suppose we want to split a tree T at a key K. Recall the semantics of split from
750   §2: $T.split(K) \rightarrow T'$. This says that all the keys less than or equal to K is retained in T, and
751   the rest are split off into a new tree T' that is returned.

752   First we do a $\texttt{lookUp}$ of K in T. This leads us to a node u that either contains K or else
753   u is the successor or predecessor of K in T. That is, $u.\texttt{key}$ is either the smallest key in T that
754   is greater or equal to K or the largest key in T that is less than or equal to K. Now we can
755   repeatedly rotate at u until u becomes the root of T. At this point, we can split off either the
756   left-subtree or right-subtree of T, renaming them as T and T' appropriately. This pair (T, T') *work out this little*
757   of trees is the desired result.   *detail*

758   **¶32. Complexity.**   The preceding algorithms show that the worst case complexity all the
759   basic operations on BST is $O(h)$ where h is the bound on the height of the trees.

760   **Theorem 6** *We can do lookup, insert, delete, find successor or predecessor, min or max, merge*
761   *and split of binary search trees with height at most h in worst-case time $O(h)$.*

762   Since $h = \Omega(\log n)$ for a tree of size n, the operations in this theorem are optimal if we can
763   maintain an $O(\log n)$ bound on the height of binary search trees. Of course, after each of the

above operations, we would need to rebalance the resulting BST to regain the $O(\log n)$ bound. This rebalancing depends on the balanced family under consideration.

Our rotation-based algorithms for insertion and deletion may be slower than the "standard" algorithms which perform only a constant number of pointer re-assignments. The attraction of rotation-based algorithms is their simplicity. Other possible benefits of rotation will be explored in Chapter 6 on amortization and splay trees.

_____Exercises

Exercise 3.1: Let T be a left-list (i.e., a BST in which no node has a right-child).
(a) Suppose u is the tip of the left-path of the root. Describe the result of repeated rotation of u until u becomes the root.
(b) Describe the the effect of repeated left-rotate of the root of T (until the root has no left child)? Illustrate your answer to (a) and (b) by drawing the intermediate trees when T has 5 nodes. ◇

Exercise 3.2: Call T_n a **zigzag-list** if it is binary tree comprised of a single path

$$(u_0 - u_1 - \cdots - u_n)$$

from the root u_0 to a unique leaf u_n. Moreover, u_i $(i > 0)$ is a left child if i is odd, and a right child if i is even. Describe the result of performing $\texttt{rotate}^n(u_n)$ (i.e., n repeated rotations at u_n).
(i) Show the transformations on T_n in case $n = 6$.
(ii) For a general n, what are the left- and right-spines of u_n at the end?
(iii) What is the final height of T_n as a function of n? ◇

Exercise 3.3: Consider the BST of Figure 3(a). This calls for hand-simulation of the insertion and deletion algorithms. Show intermediate trees after each rotation, not just the final tree.
(a) Perform the deletion of the key 10 this tree using the rotation-based deletion algorithm.
(b) Repeat part (a), using the standard deletion algorithm. ◇

Exercise 3.4: Suppose the set of keys in a BST are no longer unique, and we want to modify the $\texttt{lookUp}(u, K)$ function to return a linked list containing all the nodes containing key K in a subtree T_u rooted at u. Write the pseudo-code for $LookUpAll(u, K)$. ◇

Exercise 3.5: The function $\text{Verify}(u)$ is supposed to return $\texttt{true}$ iff the binary tree rooted at u is a binary search tree with distinct keys:

```
Verify(Node u)
    if (u = nil) Return(true)
    if ((u.left ≠ nil) and (u.key < u.left.key)) Return(false)
    if ((u.right ≠ nil) and (u.key > u.right.key)) Return(false)
    Return(Verify (u.left) ∧ Verify(u.right))
```

796   Either argue for it's correctness, or give a counter-example showing it is wrong.   ◇

797   **Exercise 3.6:** Recall that a rotation can be implemented with 6 pointer assignments in the
798   worst case. Suppose a binary search tree maintains successor and predecessor links (de-
799   noted $u.\texttt{succ}$ and $u.\texttt{pred}$ in the text). What is the number of pointer assignments needed
800   to do rotation now? (Careful!)   ◇

801   **Exercise 3.7:** (a) Implement the above binary search tree algorithms (rotation, lookup, insert,
802   deletion, etc) in your favorite high level language. Assume the binary trees have parent
803   pointers.
804   (b) Describe the modifications to your algorithms in (a) in case the binary trees do not
805   have parent pointers.
806   (c) Describe the modifications to your algorithms in (a) in case each node maintains a
807   successor/predecessor pointer. [See previous question.]   ◇

808   **Exercise 3.8:** Let T be the binary search tree in Figure 3. You should recall the ADT seman-
809   tics of $T' \leftarrow \texttt{Split}(T, K)$ and $\texttt{merge}(T, T')$ in §2. You only need to show the trees at the
810   end of the operations.
811   (a) Perform the operation $T' \leftarrow \texttt{Split}(T, 5)$. Display T and T' after the split.
812   (b) Now perform $\texttt{insert}(T, 3.5)$ where T is the tree after the operation in (a). Display
813   the tree after insertion.
814   (c) Finally, perform $\texttt{merge}(T, T')$ where T is the tree after the insert in (b) and T' is the
815   tree after the split in (a).   ◇

816   **Exercise 3.9:** Give the code for rotation in your favorite programming language. Use this to
817   implement deletion based on repeated rotation.   ◇

818   **Exercise 3.10:** Instead of minimizing the number of assignments, let us try to minimize the
819   time. To count time, we count each reference to a pointer as taking unit time. For in-
820   stance, the assignment $u.\texttt{next.prev.prev} \leftarrow u.\texttt{prev}$ costs 5 time units because in addition
821   to the assignment, we have to make access 4 pointers.
822   (a) What is the rotation time in our 6 assignment solution in the text?
823   (b) Give a faster rotation algorithm, by using temporary variables.   ◇

824   **Exercise 3.11:** We could implement a double rotation as two successive rotations, and this
825   would take 12 assignment steps.
826   (a) Give a simple proof that 10 assignments are necessary.
827   (b) Show that you could do this with 10 assignment steps.   ◇

828   **Exercise 3.12:** Open-ended: The problem of implementing $\texttt{rotate}(u)$ without using extra
829   storage or in minimum time (previous Exercise) can be generalized. Let G be a directed
830   graph where each edge ("pointer") has a name (e.g., $\texttt{next}, \texttt{prev}, \texttt{left}, \texttt{right}$) taken from
831   a fixed set. Moreover, there is at most one edge with a given name coming out of each
832   node. Suppose we want to transform G to another graph G', just by reassignment of
833   these pointers. Under what conditions can this transformation be achieved with only one
834   variable u (as in $\texttt{rotate}(u)$)? Under what conditions is the transformation achievable at
835   all (using more intermediate variables? We also want to achieve minimum time.   ◇

Exercise 3.13: Suppose T_0 and T_1 are equivalent. This exercise analyzes two methods for transforming T_0 into T_1 by repeated rotations. Assume the keys in each tree are distinct.
Method A: Given T_0 and T_1, Method A recursively transforms T_0 until T_0 is isomorphic to T_1. In other words, T_1 is used as a target but is not modified. Suppose u is the node in T_0 that has the same key as the root of T_1. Repeatedly rotate u until it becomes the root of T_0. Now, Method A recursively transforms the left- and right-subtrees of T_0 so that the are isomorphic to the left- and right-subtrees of T_1. Let $R_A(n)$ be the worst case number of rotations of Method A on trees with n keys. Give a tight bound of $R_A(n)$.
Method B: Given a tree T_0, Method B transforms T_0 into a canonical form. For canonical form, we choose left-lists: a **left-list** is a binary tree in which every node has no right-child. (Of course, we could also use a **right-list** as canonical form.) If every BST can be rotated into an equivalent left-list, it also means that we can rotate a left-list into any BST. Therefore, we rotate from any T_0 to any T_1 by first rotating T_0 into a left-list, and then from the left-list into T_1. Method B operates as follows: recursively, convert the left subtree of T_0 into a left-list, and similarly convert the right subtree of T_0 into a left-list. Now, with at most n rotations, convert the whole tree into a left-list. Let $R_B(n)$ be worst case number of rotations for Method B on trees with n keys. Give a tight analysis of $R_B(n)$. ◇

Exercise 3.14: Prove Lemma 3, that there is a unique way to order the nodes of a binary tree T that is consistent with any binary search tree based on T. HINT: use structural induction on binary trees. ◇

Exercise 3.15: Implement the Cut(u) operation in a high-level informal programming language. Assume that nodes have parent pointers, and your code should work even if u.parent = nil. Your code should explicitly "delete(v)" after you physically remove a node v. If u has two children, then Cut(u) must be a no-op. ◇

Exercise 3.16: Design an algorithm to find both the successor and predecessor of a given key K in a binary search tree. It should be more efficient than just finding the successor and finding the predecessor independently. ◇

Exercise 3.17: Show that if a binary search tree has height h and u is any node, then a sequence of $k \geqslant 1$ repeated executions of the assignment $u \leftarrow successor(u)$ takes time $O(h + k)$. ◇

Exercise 3.18: Suppose we are given a sequence L of keys representing the order we will insert into an initially empty BST. Compute the height of the BST without actually constructing the BST. ◇

Exercise 3.19: Show how to efficiently maintain the heights of the left and right spines of each node. (Use this in the rotation-based deletion algorithm.) ◇

Exercise 3.20: We refine the successor/predecessor relation. Suppose that T^u is obtained from T by pruning all the proper descendants of u (so u is a leaf in T^u). Then the successor and predecessor of u in T^u are called (respectively) the **external successor** and **predecessor** of u in T Next, if T_u is the subtree at u, then the successor and

predecessor of u in T_u are called (respectively) the **internal successor** and **predecessor** of u in T

(a) Explain the concepts of internal and external successors and predecessors in terms of spines.

(b) What is the connection between successors and predecessors to the internal or external versions of these concepts?   ◇

Exercise 3.21: The text gave a conventional algorithm for successor of a node in a BST. Give the rotation-based version of the successor algorithm.   ◇

Exercise 3.22: Suppose that we begin with u pointing at the first node of a binary tree, and continue to apply the rotation-based successor (see previous question) until u is at the last node. Bound the number of rotations made as a function of n (the size of the binary tree).   ◇

Exercise 3.23: Suppose we allow allow duplicate keys. Under (5), we can modify our algorithms suitably so that all the keys with the same value lie in consecutive nodes of some "right-path chain".

(a) Show how to modify lookup on key K so that we list all the items whose key is K.

(b) Discuss how this property can be preserved during rotation, insertion, deletion.

(c) Discuss the effect of duplicate keys on the complexity of rotation, insertion, deletion. Suggest ways to improve the complexity.   ◇

Exercise 3.24: Consider the priority queue ADT.

(a) Describe algorithms to implement this ADT when the concrete data structures are binary search trees.

(b) Analyze the complexity of your algorithms in (a).   ◇

Exercise 3.25: Suppose T is a binary tree storing items at each node with the property that at each internal node u,

$$u_L.\text{key} < u.\text{key} < u_R.\text{key},$$

where u_L and u_R are the left and right children of u (if they exist). So this is weaker than a binary search tree. For simplicity, let us assume that T has exactly $2^h - 1$ nodes and height $h - 1$, so it is a perfect binary tree. Now, among all the nodes at a given depth, we order them from left to right in the natural way. Then, except for the leftmost and rightmost node in a level, every node has a successor and predecessor node in its level. One of them is a sibling. The other is defined to be its **partner**. For completeness, let us define the leftmost and rightmost nodes to be each other's partner. See Figure 10. Now define the following parallel operations. The first operation is $sort1$: at each internal node u at an odd level, we order the keys at u and its children u_L, u_R so that $u_L.\text{key} < u.\text{key} < u_R.\text{key}$. The second is $sort2$, and it is analogous to $sort1$ except that it applies to all internal nodes at an even level. The third operation is $swap$ which order the keys of each pair of partners (by exchanging their keys if necessary). Suppose we repeatedly perform the sequence of operations $(sort1, sort2, swap)$. Will this eventually stabilize? If we start out with a binary search tree then clearly this is a stable state. Will we always end up in a binary search tree?   ◇
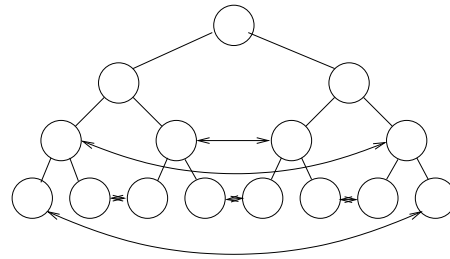
End Exercises

Figure 10: Partners

§4. Tree Traversals and Shell Programs

In this section, we describe systematic methods to visit all the nodes of a binary tree. Such methods are called **tree traversals**. Such traversals provide an "algorithmic skeleton" or **shells** for implementing many useful algorithms. We saw this concept in ¶9, while implementing ADT operations using linked lists.

Unix fans – shell programming is not what you think it is

Shell programs are regular programs that contain subroutines that we call **macros**. What is the difference between a standard subroutine and a macro? Like a subroutine, a macro can take arguments. A typical macro is VISIT(u), but we will indicate its macro nature by enclosing it in a box thus $\boxed{\text{VISIT}(u)}$. We view macros as light-weight subroutines because, unlike subroutines, a macro that does not set up its own variables, have its own execution stack, or return a value. In simple terms, a macro call is just textual substitution. E.g., suppose we define

$$\boxed{\text{VISIT}(u)} \equiv \texttt{count++}; S.\texttt{push}(u)$$

then we just replace any occurence of $\boxed{\text{VISIT}(x)}$ by '$\texttt{count++}; S.\texttt{push}(x)$'. Thus macros communicate via "global variables" such as the $\texttt{count}$ variable in $\boxed{\text{VISIT}(u)}$. Most programming languages provide some form of macro facility.

Because we envision the shell program to have many multiple applications, we need to design a sufficient set of macros to cover these applications. For any specific application, some macros will not be needed. Then we simply set these to be empty macros, i.e., a No-Op.

¶33. In-order Traversal. There are three systematic ways to visit all the nodes in a binary tree: they are all defined recursively. Perhaps the most important is the **in-order** or **symmetric traversal**. To do in-order traversal of a binary tree rooted at u, you recursively do in-order traversal of $u.\texttt{left}$, then you visit u, then recursive do in-order traversal of $u.\texttt{right}$. Here is the shell for this traversal:

Structural Induction on binary trees!

> In-Order(u):
> Input: u is root of binary tree T to be traversed.
> Output: The in-order listing of the nodes in T.
>   0.   BASE(u) .
>   1.   In-Order(u.left).
>   2.   VISIT(u) .
>   3.   In-Order(u.right).

941   This recursive shell uses two macros called BASE and VISIT. We can define the BASE as

943
$$\boxed{\text{BASE}(u)}:$$
$$\text{If } u = \texttt{nil}, \text{Return.}$$

944   We regard BASE to be a **macro call** (or an "inline" or textual substitution). Thus the
945   Return statement in BASE is meant to return from the In-Order routine, and not to just return
946   from a "BASE subroutine" call! For traversals, the VISIT(u) macro can simply be:

948
$$\boxed{\text{VISIT}(u)}:$$
$$\text{Print } u.\texttt{key.}$$

949   In illustration, consider the two binary trees in Figure 2. The numbers on the nodes are keys,
950   but they are not organized into a binary search tree. They simply serve as identifiers.

An in-order traversal of the small tree in Figure 2 will produce $(2, 4, 1, 5, 3)$. For a more
substantial example, consider the output of an in-order traversal of the big tree:

$$(7, 4, 12, 15, 8, 2, 9, 5, 10, 1, 3, 13, 11, 14, 6)$$

951   Basic fact: *if we list the keys of a BST using an inorder traversal, then the keys will be*
952   *sorted.*

For instance, the in-order traversal of the BST in Figure 3 will produce the sequence

$$(1, 2, 3, 4, 5, \ldots, 12, 13, 14, 15).$$

953   This leads to an application for sorting: *sorting a set S of numbers can be reduced to constructing*
954   *a binary search tree on a set of nodes with S as their keys.* Once we have such a BST, we can
955   do an in-order traversal to output the keys in sorted order:

956

957

```
        Sort(A, B):
          Input: A an array of n numbers to be sorted
          Output: B, the sorted output array
          Initialize a BST T with no items.
          For i = 1, . . . , n
              Insert A[i] into T
          i ← 1
          In-Order(T) with the macro  VISIT(u) ≡ B[i++] ← u.key.
```

¶34. Pre-order Traversal.   We can re-write the above In-Order routine succinctly as:

$$IN(u) \equiv \left[\; \boxed{\text{BASE}(u)} ; IN(u.\texttt{left}); \boxed{\text{VISIT}(u)} ; IN(u.\texttt{right}) \; \right]$$

Changing the order of Steps 1, 2 and 3 in the In-Order procedure (but always doing Step 1 before Step 3), we obtain two other methods of tree traversal. Thus, if we perform Step 2 before Steps 1 and 3, the result is called the **pre-order traversal** of the tree:

$$PRE(u) \equiv \left[\; \boxed{\text{BASE}(u)} ; \boxed{\text{VISIT}(u)} ; PRE(u.\texttt{left}); PRE(u.\texttt{right}) \; \right]$$

Applied to the small tree in Figure 2, we obtain $(1, 2, 4, 3, 5)$. The big tree produces

$$(1, 2, 4, 7, 8, 12, 15, 5, 9, 10, 3, 6, 11, 13, 14).$$

¶35. Post-order Traversal.   If we perform Step 2 after Steps 1 and 3, the result is called the **post-order traversal** of the tree:

$$POST(u) \equiv \left[\; \boxed{\text{BASE}(u)} ; POST(u.\texttt{left}); POST(u.\texttt{right}); \boxed{\text{VISIT}(u)} \; \right]$$

Using the trees of Figure 2, we obtain the output sequences $(4, 2, 5, 3, 1)$ and

$$(7, 15, 12, 8, 4, 9, 10, 5, 2, 13, 14, 11, 6, 3, 1).$$

¶36. Applications of Tree Traversal: Shell Programming   Tree traversals may not appear interesting on their own right. However, they serve as shells for solving many interesting problems. That is, many algorithms can be programmed by taking a tree traversal shell, and replacing the named macros by appropriate code: for tree traversals, we have two such macros, called $\boxed{\text{BASE}}$ and $\boxed{\text{VISIT}}$.

To illustrate shell programming, suppose we want to compute the height of each node of a BST. Assume that each node u has a variable $u.H$ that is to store the height of node u. If we have recursive computed the values of $u.\texttt{left}.H$ and $u.\texttt{right}.H$, then we see that the height of u can be computed as

$$u.H = 1 + \max\left\{u.\texttt{left}.H, u.\texttt{right}.H\right\}.$$

This suggests the use of post-order shell to solve the height problem: We keep the no-op BASE subroutine, but modify $\boxed{\text{VISIT}(u)}$ to the following task:

computing height in post-order

$$\boxed{\text{VISIT}(u)}$$
$$\text{if } (u.\texttt{left} = \texttt{nil}) \text{ then } L \leftarrow -1.$$
$$\text{else } L \leftarrow u.\texttt{left}.H.$$
$$\text{if } (u.\texttt{right} = \texttt{nil}) \text{ then } R \leftarrow -1.$$
$$\text{else } R \leftarrow u.\texttt{right}.H.$$
$$u.H \leftarrow 1 + \max\{L, R\}.$$

On the other hand, suppose we want to compute the depth of each node. Again, assume each node u stores a variable $u.D$ to record its depth. Then, assuming that $u.D$ has been computed, then we could easily compute the depths of the children of u using

computing depth in pre-order

$$u.\texttt{left}.D = u.\texttt{right}.D = 1 + u.D.$$

This suggests that we use the pre-order shell for computing depth.

¶37. Return Shells. For some applications, we want a version of the above traversal routines that return some value. Call them "return shells" and these can in turn call "return macros". We illustrate this by modifying the previous postorder shell $\boxed{\text{POST}(u)}$ to a new version $\boxed{\text{RPOST}(u)}$ which returns a value. It calls the return macros $\boxed{\text{RBASE}(u)}$ and $\boxed{\text{RVISIT}(u, L, R)}$:

$$\text{RPOST}(u)$$
$$\boxed{\text{RBASE(u)}}.$$
$$L \leftarrow rPOST(u.\texttt{left}).$$
$$R \leftarrow rPOST(u.\texttt{right}).$$
$$\boxed{\text{RVISIT}(u, L, R)}.$$

As an application of this RPOST routine, consider our previous solution for computing the height of binary trees. Previously we assume that every node u has an extra field called $u.H$ that is used to store the height of u. We can avoid this extra field by defining the return shell $\boxed{\text{RPOST}(u)}$ to compute the height of u. Thus we define

$$\boxed{\text{RBASE}(u)}:$$
$$\text{if } (u = \texttt{nil}) \text{ Return}(-1).$$

and

$$\boxed{\text{RVISIT}(u, L, R)}$$
$$\text{Return}(1 + \max\{L, R\}).$$

The reader can readily check that $\boxed{\text{RPOST}}$ solves the height problem elegantly. As another application of return shell, suppose we want to check if a binary tree is a binary search tree. This is explored in Exercises below.

The motif of using shell programs to perform node traversals, augmented by a small set of macros such as BASE and VISIT will be further elaborated when we study graph traversals in the next Chapter.   Indeed, graph traversal is a generalization of tree traversal. Shell programs unify many programming aspects of traversal algorithms: we cannot over emphasize this point.

Attention!
Attention! I love
attention
– Lucy in Peanuts

─── Exercises

Exercise 4.1: Joe said that in a post-order listing of the keys in a BST, we must begin with the smallest key in the tree. Is he right?   ◇

Exercise 4.2: Give the in-order, pre-order and post-order listing of the nodes in the binary tree in Figure 17.   ◇

Exercise 4.3: Binary tree reconstruction from node-listings in tree traversals.
Note: a traversal of a BST produces a listing of the nodes in the order they are VISITed. The nodes are given (symbolic) names, but you cannot compare them as they are not keys. We want to reconstruct the binary tree from these node-listings.
(a) Let the in-order and pre-order traversal of a binary tree T with 10 nodes be $(a, b, c, d, e, f, g, h, i, j)$ and $(f, d, b, a, c, e, h, g, j, i)$, respectively. Draw the tree T.
(b) Prove that if we have the pre-order and in-order listing of the nodes in a binary tree, we can reconstruct the tree.
(c) Consider the other two possibilities: (c.1) pre-order and post-order, and (c.2) in-order and post-order. State in each case whether or not they have the same reconstruction property as in (b). If so, prove it. If not, show a counter example.
(d) Redo part(c) for full binary trees. Recall that in a full binary tree, each node either has no children or 2 children.   ◇

Exercise 4.4: BST reconstruction from key-listings in tree traversals. *Careful: in the previous question, we list nodes of a binary tree; in this question, we list the keys in the nodes of a BST!* In the previous problem, we want to reconstruct a binary from the list of nodes from two traversals. In this question, we only need one list of keys for reconstruction.
(a) Here is the list of keys from post-order traversal of a BST:

$$2, 1, 3, 7, 10, 8, 5, 13, 15, 14, 12$$

Draw this binary search tree.
(b) Describe the general algorithm to reconstruct a BST from its post-order traversal.
                    ◇

Exercise 4.5: Here is the inorder and preorder listing of nodes in a binary tree: $(a, b, c, d, e, f, g, h, i)$ and $(f, b, a, e, c, d, h, g, i)$, respectively. Please draw the binary tree. REMARK: unlike the previous problem, this is not a listing of keys in the tree. Indeed, this problem is about binary trees, not BSTs.   ◇

Exercise 4.6: Use shell programming to compute the size of each node u of a binary tree. The size of u is the size of the subtree rooted at u. Give two versions: (a) using a return shell (e.g., , and (b) using a version where the size of node u is recorded in a field $u.size$.   ◇

───

Exercise 4.7: Let $size(u)$ be the number of nodes in the tree rooted at u. Say that node u is **size-balanced** if

$$1/2 \leqslant size(u.\texttt{left})/size(u.\texttt{right}) \leqslant 2$$

where a leaf node is size-balanced by definition.

(a) Use shell programming to compute the routine $B(u)$ which returns $size(u)$ if each node in the subtree at u is balanced, and $B(u) = -1$ otherwise. Do not assume any additional fields in the nodes or that the size information is available.

(b) Suppose you know that $u.\texttt{left}$ and $u.\texttt{right}$ are size-balanced. Give a routine called $REBALANCE(u)$ that uses rotations to make u balanced. Assume each node v has an extra field $u.SIZE$ whose value is $size(u)$ (you must update this field as you rotate). ◇

Exercise 4.8: Show how to use the pre-order shell to compute the depth of each node in a binary tree. Assume that each node u has a depth field, $u.D$. ◇

Exercise 4.9: (a) Give a non-recursive routine called $isBST(u)$ which returns **true** if the node u represents (the root of) a BST; otherwise it returns **false**. Although $isBST(u)$ is non-recursive, it calls a *recursive* routine which we denote by $R(u)$. HINT: The subroutine $R(u)$ returns a pair of values.

(b) Please specify clearly what is returned by $R(u)$, and give a brief proof that your routine for $R(u)$ is correct.

Notes. Assume that a node u is either **nil**, or else it has three fields: $u.\texttt{key}, u.\texttt{left}, u.\texttt{right}$. The keys k in a BST are, by definition finite, i.e., $-\infty < k < +\infty$. ◇

Exercise 4.10: This exercise is similar to the previous exercise. However, we explore a different approach to checking if a node u represents a BST. Design a recursive subroutine called $inBST(u, min, max)$ that returns true iff the keys in the BST u keys lies in the half-open range $[min, max)$. CLEARLY, we may define $isBST(u)$ to be $inBST(u, -\infty, +\infty)$. ◇

Exercise 4.11: A student proposed a different approach to the previous question. Let $minBST(u)$ and $maxBST(u)$ compute the minimum and maximum keys in T_u, respectively. These subroutines are easily computed in the obvious way. For simplicity, assume all keys are distinct and $u \neq$ **nil** in these arguments. The recursive subroutine is given as follows:

CheckBST(u)
▷ *Returns largest key in T_u if T_u is BST*
▷ *Returns $+\infty$ if not BST*
▷ *Assume u is not* **nil**
    If $(u.\texttt{left} \neq$ **nil**$)$
        $L \leftarrow maxBST(u.\texttt{left})$
        If $(L > u.\texttt{key}$ or $L = \infty)$ return(∞)
    If $(u.\texttt{right} \neq$ **nil**$)$
        $R \leftarrow minBST(u.\texttt{right})$
        If $(R < u.\texttt{key}$ or $R = \infty)$ return(∞)
    Return $(CheckBST(u.\texttt{left}) \wedge (CheckBST(u.\texttt{right}))$

Is this program correct? Bound its complexity. HINT: Let the "root-path length" of a node be the length of its path to the root. The "root-path length" of a binary tree T_u is the sum of the root-path lengths of all its nodes. The complexity is related to this number. ◇

Exercise 4.12: Like the previous problem, we want to check if a binary tree is a BST. Write a recursive algorithm called $SlowBST(u)$ which solves the problem, except that the running time of your solution must be provably exponential-time. Your overall algorithm must achieve this exponential complexity without any trivial redundancies. E.g., there should be no redundant code, or plain iteration as:

> $SlowBST(u)$
> Compute the number n of nodes in T_u
> Do for 2^n times:
> $FastBST(u)$

◇

_____End Exercises

§5. Variant Binary Search Trees: External BST

¶38. Non-Standard BSTs   This section consider variations on the concept of binary search trees. The BSTs we have seen so far will be designated **standard BSTs**. Here are some examples of non-standard BSTs:

Augmentation of Standard BST We can augment a standard BST in two ways:
(A) Auxiliary data in nodes: we may maintain additional information in each node: height, depth or size or range of keys in the subtree. We may also maintain auxiliary pointers: parent pointers, successor/predecessor pointers, and "level pointers". We must then modify our insertion/deletion algorithms to maintain these information. Most of these will be explored in the Exercises.
(B) Auxiliary links: we may maintain additional pointers such as level links or successor/predecessor links. An interesting example of such augmentation is the **finger tree**.

External BST Note that in standard BST's we store items[12] (i.e., key-data pairs) in every node of the tree. In contrast, **external BST** stores items only in their leaves; the internal nodes only have keys (without associated data) that serve as "guides" to lead to the items at the leaves.

Implicit Key BST Imagine a BST in which each node u stores a "key object", $u.keyObj$. Given a query object q, we want return a suitable key object from the BST. Ww conduct this search starting from the root, and at each node, we perform some computation on on $(u.keyObj, q)$ in order to decide whether we should return $u.keyObj$, or continue the search in the left or right child of u. Thus the search keys are implicitly encoded by objects. In Chapter VI, we see an example where the objects are points in $\mathbb{R}^2$. Another class of implicit key search trees falls under the **parametric search** paradigm.

---

[12]This remark might not be obvious because in our typical examples, we only show the key stored in each node (since the data part has no influence on the algorithm). But we confirm our remark by noting that on LookUp's, we stop as soon as we find a node with the key we are searching for – presumably because the associated data is thereby found.

¹⁰⁸⁹ In the following, we will treat external BST's. Before doing this, we take a short detour to
¹⁰⁹⁰ discuss the related concepts of *extended binary trees* and *exogenous search structures*.

¹⁰⁹¹ **¶39. Extended binary trees.**   Underlying a BST is a purely structural object, the binary
¹⁰⁹² tree. Before looking into non-standard BST's, we first give an alternative account of these
¹⁰⁹³ binary trees. It will provide the vocabulary for discussing some non-standard BST's. Following
¹⁰⁹⁴ Knuth [10, p. 399], the alternative view of binary trees is called **extended binary trees**. In
¹⁰⁹⁵ an extended binary tree, every node has 0 or 2 children; nodes with no children are called[13] **nil**
¹⁰⁹⁶ **nodes** while the other nodes are called **non-nil nodes**. For emphasis, our original version of
¹⁰⁹⁷ binary trees will be called **standard binary trees**.

¹⁰⁹⁸ See Figure 11(a) for a standard binary tree and Figure 11(b) for the corresponding extended
¹⁰⁹⁹ version. In this figure, we see a common convention (following Knuth) of representing nil nodes
¹¹⁰⁰ by black squares.
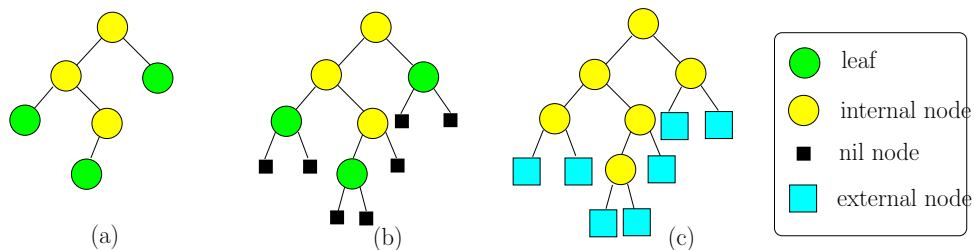
Figure 11: Three versions of binary trees: (a) standard, (b) extended, (c) external

¹¹⁰¹ The bijection between extended and standard binary trees is given as follows:

¹¹⁰² *1. By deleting all nil nodes of an extended binary tree, we obtain a standard binary*
¹¹⁰³ *tree.*
¹¹⁰⁴ *2. Conversely, from a standard binary tree, if every leaf is given two nil nodes as*
¹¹⁰⁵ *children, and every internal node with one child is given a nil node as child, then*
¹¹⁰⁶ *we obtain a corresponding extended binary tree.*

¹¹⁰⁷ In view of this correspondence, we could switch between the two viewpoints depending on
¹¹⁰⁸ which is more convenient. Generally, we avoid drawing the nil nodes since they just double
¹¹⁰⁹ the number of nodes without conveying new information. In fact, nil nodes cannot store   *why care about nil*
¹¹¹⁰ data or items. One reason we explicitly introduce them is that it simplifies the description of   *nodes?*
¹¹¹¹ some algorithms (e.g., red-black tree algorithms). They serve as sentinels in an iterative loop.
¹¹¹² The "nil node" terminology may be better appreciated when we realize that in conventional
¹¹¹³ realization of binary trees, we allocate two pointers to every node, regardless of whether the
¹¹¹⁴ node has two children or not. The lack of a child is indicated by the presence of a `nil` pointer.
¹¹¹⁵ In standard binary trees, we just say that the `nil` pointer points to a null node.

¹¹¹⁶ The concept of a "leaf" of an extended binary tree is apt to cause some confusion. We shall
¹¹¹⁷ use the "leaf" terminology as follows: a node of an extended binary tree is called a **leaf** if it is
¹¹¹⁸ the leaf of the corresponding standard binary tree. Thus a node in an extended binary tree is a

---

[13]A binary tree in which every node has 2 or 0 children is said to be "full". Knuth calls the nil nodes "external
nodes". A path that ends in an external node is called an "external path". But we will avoid using the term
"external" when discussing extended binary trees. This will be re-introduced next under the topic of external
binary trees.

1119 leaf iff the node has two nil nodes as children. *Thus a nil node is never a leaf.* In Figure 11(a)
1120 and (b), the leaves are colored green.

1121 **¶40. External Binary Search Tree.** To turn extended binary trees into search structures,
1122 we could just store keys in nodes satisfying the BST Property. The resulting "extended BST"
1123 is not very different from the standard BST. All we get from the nil nodes in extended BST is
1124 some book-keeping convenience (e.g., as sentinels in algorithms). For a more substantial trans-
1125 formation, we now modify extended BST by replacing nil nodes with a new type of node that
1126 can store items, calling them **external nodes**. Also the original nodes are now called **internal**
1127 **nodes**. The result is known as an **external binary tree**, and illustrated in Figure 11(c).

*Now I see why
'internal nodes' are
so-called in
ordinary BST...*

1128 We turn this external binary trees into a exogenous search structure by storing items in ex-
1129 ternal nodes only. We regard items as pairs in the form $(key, data)$ or the $(key, pointer\text{-}to\text{-}data)$.
1130 The internal nodes store only keys. We may call the keys in internal nodes **search keys** as
1131 opposed to the **item keys** found in items. For this purpose, the keys (whether in internal or
1132 external nodes) must satisfy the usual BST Property. Recall that BST Property says that at
1133 any internal node u,

$$u_L.\texttt{key} < u.\texttt{key} \leqslant u_R.\texttt{key} \tag{11}$$

1134 where u_L (resp., u_R) is any node in the left (resp., right) subtree at u. Note that u_L and u_R
1135 may be internal or external nodes. The new search structure is called, naturally, an **external**
1136 **BST**. Note that (11) allows $u.\texttt{key}$ to be equal to $u_R.\texttt{key}$ (but not to $u_L.\texttt{key}$).

1137 The item keys arise naturally from the application. But the search keys are artifacts of
1138 external BSTs. Indeed, they can be generated out of thin-air as long as the satisfy (11). In
1139 practice, they are copies of item keys, created after insertions. Suppose the external BST has
1140 n external nodes storing items with distinct item keys, say $k_1 < k_2 < \cdots < k_n$. So there must
1141 be exactly $n - 1$ search keys, say $k'_1 \leqslant \cdots \leqslant k'_{n-1}$. By structural induction (¶13), we may
1142 conclude that two sets of keys are interleaved as follows:

$$k_1 < k'_1 \leqslant k_2 < k'_2 \leqslant \cdots < k'_{n-1} \leqslant k_n. \tag{12}$$

1143 In fact, it is possible to ensure that $k'_i = k_{i+1}$ for all i. This is called the **partner property**:

Partner Property: If e is an external node and it has a predecessor $e.\texttt{pred}$,
then

$$e.\texttt{key} = e.\texttt{pred.key}. \tag{13}$$

We say e and $e.\texttt{pred}$ are **partners** in this case. The only external node
without a partner is the leftmost leaf e (storing the minimum key in the
tree).

1144

We will see application of external BST for dynamic Huffman codes (Chapter V) and in
dynamic string compression (Chapter VI). In these applications, a standard BST will not do.
The external BST shown in the right margin comes from dynamic Huffman code application:
the leaves in this tree store the seven items (represented by their keys only)

$$\texttt{* < e < h < l < o < r < w.}$$

1145 Note that this external BST satisfies the Partner Property.

1146 The preceding description of external BST's is sufficient for Lookups, but in order to support
1147 insertion and deletion, we must answer this question: *where do search keys in internal nodes*
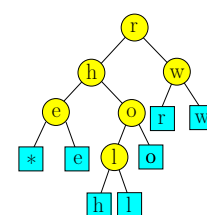
An external BST

come from? When the user inserts an item, a new internal node is created. What is the key in that internal node? It is simplest to give our insertion algorithm the ability to duplicate item key. With this ability, we can now do insertions. By the same token, whenever we do deletions, we must delete an item and also an internal node containing a search key.

¶41. **Exogenous versus Endogenous Search Structures**   Tarjan [19], calls *an arrangement of nodes* **endogenous** *if the pointers forming the "skeleton" of the structure are contained in the nodes themselves, and* **exogenous** *if the skeleton is outside the nodes.* What he calls "nodes" are really items or data in our terminology. Figure 12 illustrates his interpretation of this definition for linked lists.
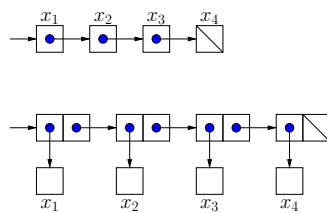
Figure 12: Endogenous/Exogenous

Let us apply his classification to BST's. As noted above, a standard BST stores a set of items in each of its nodes. In Tarjan's terminology, the entire BST forms a skeleton that is intermingled with the data (what he calls "nodes"). So the standard BST is an endogeneous search structure. In contrast, an external BST stores data only in the leaves of the BST. Thus the internal nodes form a binary tree that is a "skeleton" that is outside the "nodes" (i.e., data). So Tarjan would call the external BST an exogenous search structure.

What are the relative advantages of the exogenous or endogenous forms? The external BST need more space than a standard BST: with n items, the external BST has $2n - 1$ nodes while the standard BST has n nodes. So the standard BST has half the size of a corresponding external BST. But exogenous structures allow more flexibility since the actual items can be freely re-organized independent of the search skeleton. In databases, this freedom is important, and the exogenous search structures are called "indexes". The database user can freely create and destroy such indexes for a given set of items. Each index amounts to searching the database on a different criterion. For instance, in a personnel database, each data contains information about an individual such as social security number, salary, date of employment, address, etc. We might want to set up an index to search on social security number, and another one to search on date of employment, etc.

¶42. **The Dictionary Operations of an External BST.**   The dictionary operations are lookup, insertion and deletions. We shall see that deletion for external BST is simpler than for standard BST.

For simplicity, we do not maintain the Partner Property; we leave as an Exercise to ensure the partner property. First, consider the lookup operation:

Lookup(K)
Output: Returns an external node e.
    $u \leftarrow root$
    While (u is an internal node)
        if ($u.Key \leqslant K$), $u \leftarrow u.\texttt{right}$
        else $u \leftarrow u.\texttt{left}$
    Return u

Note that when $u.Key = K$, we move to the right child of u. It follows that that $\mathrm{Lookup}(K)$ returns an external node e where $e.Key \leqslant K$. Next, consider the insertion of an item with
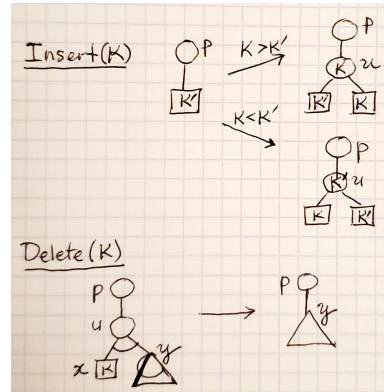

Figure 13: Insertion and Deletion from external BST

key K, as illustrated in Figure 13. We first do a lookup on K, and arrive at an external node whose key is K' and parent is p. If $K' = K$, the insertion fails. Otherwise, we create a new external node to store K, and create a new internal node u whose children are K and K'. If K' was previously the left child of p, we let $p.\texttt{left} \leftarrow u$. Otherwise $p.\texttt{right} \leftarrow u$. If $K' < K$, we set $u.\texttt{key} \leftarrow K$; otherwise $u.\texttt{key} \leftarrow K'$. Remark: if we have the Partner Property, then the case $K < K'$ does not arise.

Consider the deletion of the item with key K, again refering to Figure 13. We first do a Lookup on K, and arrive at an external node x. If $x.\texttt{key} \neq K$, then the deletion fails. Otherwise, let the sibling of x be y (y may be an external or internal node). Let $u = x.\texttt{parent}$. If u is the root, the resulting tree is now rooted in y. If u is not the root, let $p = u.\texttt{parent}$. In that case, let u is either the left- or right-child of p. Accordingly, we let y be the left- or right-child of p.

Remarks: Observe that deletion in external BSTs is simpler than in standard BSTs. Moreover, insertion and deletion in external BSTs are more symmetrical as both actions originate in external nodes.

¶43. **Auxiliary Information.** In some applications, additional information must be maintained at each node of the binary search tree. We already mentioned the predecessor and successor links. Another information is the the size of the subtree at a node. Some of this information is independent, while other is dependent or **derived**. Maintaining the derived information under the various operations is usually straightforward. In all our examples, the derived information is **local** in the following sense that *the derived information at a node u can only depend on the information stored in the subtree at u.* We will say that derived information is **strongly local** if it depends only on the independent information at node u, together with all the information at its children (whether derived or independent).

¶44. **Duplicate keys.** We normally assume that the keys in a BST are distinct. Let us now relax this requirement by allowing duplicate keys. We continue to assume that items are stored in internal nodes as well as in leaves of the BST. Let $\texttt{lookUp}(K)$ returns the first node u of the BST with $u.Key = K$. This is implemented in the usual way. It is not hard to see

that any other node w with $w.Key = K$ must lie in the subtree rooted at u. We can extend this by defining a recursive procedure $FindAll(u)$ to return a list *containing all the nodes in the subtree at u that has the same key as $u.Key$.*

¶**45. Implicit Data Structures.**   Pointers take up space, and we would like to avoid them when we can. For example, parent pointers are not essential in BST operations that use keys as arguments (e.g., $T.\texttt{delete}(K)$) because such algorithms typically begin operation from the root of the tree, and we can adjust our algorithms to remember the parents of nodes we visit. It would seem that the child pointers can never be eliminated in binary search trees. To some extent, we can achieve this by switching to another framework for data organization, based on arrays. Say we want to store n keys in an array $A[1..N]$ (with $n \leqslant N$). To store the data associated with these keys, we can use another array $B[1..N]$ so that each item is the pair $(A[i], B[i])$. But as usual, we will ignore $B[1..N]$. The idea of "implicit pointers" by specifying rules to restrict the location of keys in A based on the key values. Here are two examples:
(Eg1) Sorted Array: we require the keys to satisfy $A[i] < A[i+1]$ for all i. Then we can perform Lookup in logarithmic time, but insertion and deletion in the sorted array is now $O(n)$ time.
(Eg2) **Binary heap**: we require the keys to be stored in "heap order":

$$A[i] \leqslant \min \{A[2i], A[2i+1]\}. \tag{14}$$

Array A can be viewed as a binary tree where node i has children $2i$ and $2i+1$. Binary heaps can support deleteMin and insertion operations in $O(\log n)$ time, leading to the $O(n \log n)$ algorithm for heapsort.

These restrictions on how keys are stored in A can be intepreted as encoding a standard data structure based on pointers: (Eg1) represents a linked list in which the successor of node i is node $i+1$. Likewise, (Eg2) represents as a binary search tree where the children of node i is $2i$ and $2i+1$. In lieu of pointers, we do index arithmetic to simulate these pointers. Munro and Suwanda (1980) extended the binary heap to "beap" (biparental heap) that supports insertion, deletion and lookup in $O(\sqrt{n})$ time. More generally, they introduce the concept of **implicit data structures** based on such array representations. Although no pointers are needed, the above examples (sorted array, binary heap, beap) need extra storage of $\log n$ bits to remember the value n, the number of keys in the array.

――――――――――――――――――――――――――――――――――――Exercises

Exercise 5.1: Consider search trees with duplicate keys. First recall the BST Property.
     (a) Draw all the possible BST's with keys $1, 3, 3, 3, 4$, assuming the root is 3.
     (b) Suppose we want to design AVL trees (next Section) with duplicate keys. Say why and how the BST property must be modified.   ◇

Exercise 5.2: (a) Show the result of inserting the following letters, `h, e, l, o, w, r, d` (in this order) into an external BST that initially contains the letter `*`. Assume the letters have the usual alphabetic sorting order but `*` is smaller than any other letter. Show the external BST after each insertion. No rebalancing is required.
     (b) Starting with the final tree in part(a), now delete `w` and then delete `h`.   ◇

Exercise 5.3: Repeat the previous question using an external AVL tree instead: First insert `h, e, l, o, w, r, d` , then delete `w, h`.   ◇

Exercise 5.4: Using your favorite program language, code up the Lookup/Insert/Delete algorithms for External BST.   ◇

Exercise 5.5: Let $\mu(h)$ be the minimum size of an external AVL tree of height h. Thus $\mu(0) = 1$ and $\mu(1) = 3$. Give tight upper and lower bounds on $\mu(h)$.   ◇

Exercise 5.6: Consider an alternative organization of external BST. Suppose that each internal node, instead of storing a key, stores a pointer to its successor which is necessarily an external node. If u is an internal node, let $u.Key$ be the pointer to the successor of u, otherwise $u.Key$ is the actual key. With successor pointers, we no longer need predecessor pointers from external nodes.
(a) Spell out the modifications to the Lookup, Insert, Delete Algorithms.
(b) What are the pros and cons of this approach?   ◇

Exercise 5.7: Describe the changes needed in our binary search tree algorithms if we do not maintain parent pointers. Consider Lookup, Insert and Delete. Do both versions of Delete (standard as well as rotation version).   ◇

Exercise 5.8: Prove that the $FindAll(u)$ procedure in the text is correct.   ◇

Exercise 5.9: Consider the usual binary search trees in which we no longer assume that keys in the items are unique. State suitable conventions for what the various operations mean in this setting. E.g., `lookUp`(K) means find any item whose key is K or find all items whose keys are equal to K. Describe the corresponding algorithms.   ◇

Exercise 5.10: Implement the algorithms for insert, delete, etc, on binary search trees that maintains the size of subtree at each node.   ◇

Exercise 5.11: Repeat the previous question, but where we maintain the average value of the keys in each subtree.   ◇

Exercise 5.12: Recall the concept of heaps in the text. Let $A[1..n]$ be an array of real numbers. We call A an **almost-heap at** i there exists a number such that if $A[i]$ is replaced by this number, then A becomes a heap. Of course, a heap is automatically an almost-heat at any i.
(i) Suppose A is an almost-heap at i. Show how to convert A into a heap be pairwise-exchange of array elements. Your algorithm should take no more than $\lg n$ exchanges. Call this the $Heapify(A, i)$ subroutine.
(ii) Suppose $A[1..n]$ is a heap. Show how to delete the minimum element of the heap, so that the remaining keys in $A[1..n-1]$ form a heap of size $n-1$. Again, you must make no more than $\lg n$ exchanges. Call this the $DeleteMin(A)$ subroutine.
(iii) Show how you can use the above subroutines to sort an array in-place in $O(n \log n)$ time.   ◇

Exercise 5.13: Normally, each node u in a binary search tree maintains two fields, a key value and perhaps some balance information, denoted u.KEY and u.BALANCE, respectively.

Suppose we now wish to "augment" our tree T by maintaining two additional fields called u.PRIORITY and u.MAX. Here, u.PRIORITY is an integer which the user arbitrarily associates with this node, but u.MAX is a pointer to a node v in the subtree at u such that v.PRIORITY is maximum among all the priorities in the subtree at u. (Note: it is possible that $u = v$.) Show that rotation in such augmented trees can still be performed in constant time.

$\Diamond$

_____ End Exercises

§6. AVL Trees

AVL trees is the first known family of balanced trees, and is named after its two Russian inventors G.M. Adel'son-Vel'skii and E.M. Landis (1962). By definition, an AVL tree is a binary search tree in which the left subtree and right subtree at each node differ by at most 1 in height. They have relatively simple insertion/deletion algorithms compared to other balancing schemes for BSTs.

More generally, define the **balance** of any node u of a binary tree to be the height of the left subtree minus the height of the right subtree:

$$balance(u) = u.\texttt{left.ht} - u.\texttt{right.ht}$$

The node is **perfectly balanced** if the balance is 0. It is **AVL-balanced** if the balance is $-1, 0$ or $+1$. Our insertion and deletion algorithms will need to know this balance information at each node. Thus we need to store at each AVL node a 3-valued variable. Theoretically, this space requirement amounts to $\lg 3 < 1.585$ bits per node. Of course, in practice, AVL trees will reserve 2 bits per node for the balance information (but see Exercise).

We are going to prove that the family of AVL trees is a balanced family. Re-using some notations from binary trees (see (6) and (7)), we now define $M(h)$ and $\mu(h)$ to be the maximum and minimum number of nodes in any AVL tree with height h. It is not hard to see that $M(h) = 2^{h+1} - 1$, as for binary trees. It is more interesting to determine $\mu(h)$: its first few values are

$$\mu(-1) = 0, \qquad \mu(0) = 1, \qquad \mu(1) = 2, \qquad \mu(2) = 4.$$

It seems clear that $\mu(0) = 1$ since there is a unique tree with height 0. The other values are not entirely obvious. To see that $\mu(1) = 2$, _we must define the height of the empty tree to be_ -1. This explains why $\mu(-1) = 0$. We can verify $\mu(2) = 4$ by case analysis.

> For instance, if we define the height of the empty tree to be $-\infty$, then $\mu(1) = 3, \mu(2) = 5$. This definition of AVL trees could certainly be supported. See Exercise for an exploration of such ideas.

Consider an AVL tree T_h of height h and of size $\mu(h)$ (i.e., it has $\mu(h)$ nodes). Clearly, among all AVL trees of height h, T_h has the minimum size. For this reason, we call T_h a **min-size AVL tree** (for height h). Figure 14 shows the first few min-size AVL trees. Of course, we can interchange the roles of any pair of siblings of such a tree to get another min-size AVL

tree. Using this fact, we could compute the number of non-isomorphic min-sized AVL trees of a given height. Among these min-sized AVL trees, we define the **canonical min-size AVL trees** to be the ones in which the balance of each non-leaf node is $+1$. Note that we only drew such canonical trees in Figure 14.
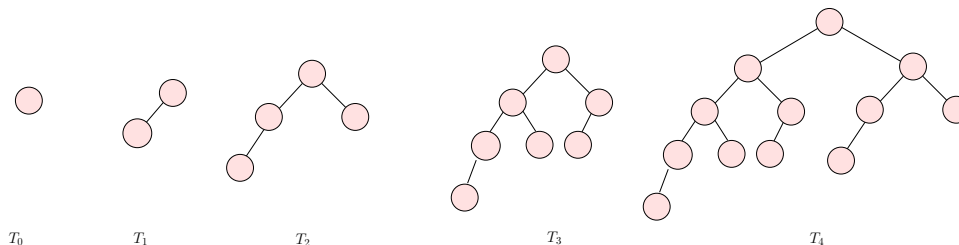

Figure 14: Canonical min-size AVL trees of heights 0, 1, 2, 3 and 4.

In general, $\mu(h)$ is seen to satisfy the recurrence

$$\mu(h) = 1 + \mu(h-1) + \mu(h-2), \qquad (h \geqslant 1). \tag{15}$$

This equation says that the min-size tree of height h having two subtrees which are min-size trees of heights $h-1$ and $h-2$, respectively. For instance, $\mu(2) = 1+\mu(1)+\mu(0) = 1+2+1 = 4$, as we found by case analysis above. We similarly check that the recurrence (15) holds for $h = 1$. The recurrence for $\mu(h)$ is very similar to the Fibonacci recurrence; the next lemma shows that $\mu(h)$ is also Θ-order of the Fibonacci numbers F_h.

so binary trees of height h with size $\mu(h)$ are sometimes called Fibonacci trees

From (15), we have $\mu(h) > 2\mu(h-2)$ for $h \geqslant 1$. It is then easy to see by induction that $\mu(h) > 2^{h/2}$ for all $h \geqslant 1$. The base cases are $\mu(1) > 2^{1/2}$ and $\mu(2) > 2^1$. Writing $C = \sqrt{2} = 1.4142\ldots$, we have thus shown

$$\mu(h) > C^h, \qquad (h \geqslant 1).$$

To sharpen this lower bound, we show that C can be replaced by the golden ratio $\phi > 1.6180$. Moreover, it is tight up to a multiplicative constant. Recall that $\phi = \frac{1+\sqrt{5}}{2} > 1.6180$ and $\widehat{\phi} = \frac{1-\sqrt{5}}{2} < -0.6180$ are the two roots of the equation $x^2 - x - 1 = 0$. Hence, $\phi^2 = \phi + 1$.

Fun arithmetic with ϕ: to square ϕ, just add 1. To take reciprocal? Subtract 1!

Lemma 7 *For $h \geqslant 0$, we have*

$$\phi^{h+1} + \widehat{\phi} \leqslant \mu(h) < 2\phi^h. \tag{16}$$

The first inequality is strict for $h \geqslant 2$.

Proof. First we prove $\mu(h) \geqslant \phi^{h+1} + \widehat{\phi}$: $\mu(0) = 1 = \phi^1 + \widehat{\phi}$ and $\mu(1) = 2 = 1 + \phi + \widehat{\phi} = \phi^2 + \widehat{\phi}$. For $h \geqslant 2$, we have

$$\mu(h) \geqslant 1 + \mu(h-1) + \mu(h-2) \geqslant 1 + (\phi^h + \widehat{\phi}) + (\phi^{h-1} + \widehat{\phi}) > \phi^{h-1}(\phi+1) + \widehat{\phi} = \phi^{h+1} + \widehat{\phi}.$$

Next, to prove $\mu(h) < 2\phi^h$, we will strengthen our hypothesis to $\mu(h) \leqslant 2\phi^h - 1$. Clearly, $\mu(0) = 1 \leqslant 2\phi^0 - 1$ and $\mu(1) = 2 \leqslant 2\phi^1 - 1$. Then for $h \geqslant 2$, we have

$$\mu(h) = 1 + \mu(h-1) + \mu(h-2) \leqslant 1 + (2\phi^{h-1} - 1) + (2\phi^{h-2} - 1) = 2(\phi+1)\phi^{h-2} - 1 = 2\phi^h - 1.$$

Q.E.D.

1337

This lemma tells us that $\mu(h)+1 = A\phi^h$ where $\phi \leqslant A < 2$. We can obtain the exact value of A by taking into account the "+1" term that was ignored in the above proof — See Exercises. It is the lower bound on $\mu(h)$ that is more important for us. For, if an AVL tree has n nodes and height h then

$$\mu(h) \leqslant n$$

by definition of $\mu(h)$. The lower bound in (16) then implies $\phi^h \leqslant n$ (for $h \geqslant 1$). Taking logs, we obtain

$$h \leqslant \log_\phi(n) = (\log_\phi 2)\lg n < 1.4404\lg n.$$

1338   This constant of 1.44 is clearly tight in view of lemma 7. Thus the height of AVL trees are at
1339   most 44% more than the absolute minimum.

1340   Recall ¶1, a family F of binary trees is said to be **balanced** if every tree in F of size n has
1341   height $h = \Theta(\log n)$. The implicit constants in the Θ-notation depends on F.

1342   **Corollary 8** *The family of AVL trees is balanced.*

1343

1344   *Proof.* Suppose T is an AVL tree of size n and height h. We must give show that $h = \Theta(\lg n)$.
1345   Upper bound on height: We had just shown that $h < 1.5\lg n = O(\lg n)$.
1346   Lower bound on height: We also know that for any binary tree, $h = \Omega(\lg n)$. The argument
1347   goes as follows: level i of a binary tree has at most 2^i nodes. Summing over all levels,
1348 $n \leqslant \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$. So $\lg(n+1) \leqslant h+1$ or $h \geqslant \lg(n+1) - 1$.   **Q.E.D.**

1349

1350   **¶46. Insertion and Deletion Algorithms.**   These algorithms for AVL trees are relatively
1351   simple, as far as balanced trees go. In either case there are two phases:

1352   **UPDATE PHASE:** Insert or delete as we would in a binary search tree. It is important that
1353   we use the *standard* deletion algorithm, not its rotational variant. It follows that the node
1354   containing the deleted key and the node which was *cut* may be different.

Must use standard deletion!

1355   **REBALANCE PHASE:** Let p be parent of the node x that was just inserted, or just *cut*
1356   during deletion, in the UPDATE PHASE. The path from p to the root will be called the
1357   **rebalance path**. We usually write this path in reverse order, starting from the root:

$$(u_0, u_1, \ldots, u_k) \tag{17}$$

1358   where $u_k = p$ and u_0 is the root. In rebalancing, we traverse (17) in reverse order, by
1359   following parent pointers starting from u_k.

1360   It remains to give details for the REBALANCE PHASE. If every node along the rebalance
1361   path is AVL-balanced, then there is nothing to do in the REBALANCE PHASE. Otherwise,
1362   let u be the first unbalanced node we encounter. It is clear that u has a balance of ± 2. In
1363   general, we fix the balance at the "current" unbalanced node and continue searching upwards
1364   along the rebalance path for the next unbalanced node. By symmetry, we may suppose that u

$u = \underline{u}$nbalanced
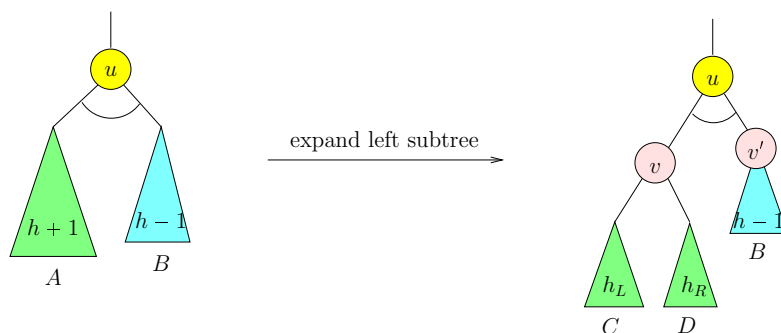
Figure 15: Node u is unbalanced after insertion or deletion.

has balance 2. Suppose its left child is node v with height $h+1$. Then its right child v' has height $h-1$. This situation is illustrated in Figure 15.

Inductively, it is assumed that all the proper descendants of u are balanced. The current height of u is $h+2$. In any case, let the current heights of the children of v be h_L and h_R, respectively.

¶47. Insertion Rebalancing. Suppose that this imbalance came about because of an insertion. What was the heights of u, v and v' before the insertion? It is easy to see that the previous heights are (respectively)

$$h+1, \quad h, \quad h-1. \tag{18}$$

The inserted node x must be in the subtree rooted at v. Clearly, the heights h_L, h_R of the children of v satisfy $\max(h_L, h_R) = h$. Since v is currently balanced, we know that $\min(h_L, h_R) = h$ or $h-1$. But in fact, we claim that $\min(h_L, h_R) = h-1$. To see this, note that if $\min(h_L, h_R) = h$ then the height of v *before* the insertion was also $h+1$ and this contradicts the initial AVL property at u. Therefore, we have to address the following two cases, as illustrated in Figure 16.

CASE (I.a): $h_L = h$ and $h_R = h-1$. This means that the inserted node is in the left subtree of v. In this case, if we rotate v, the result would be balanced. Moreover, the height of v is now $h+1$. We call this the "single-rotation" case, in contrast to the next case.

CASE (I.b): $h_L = h-1$ and $h_R = h$. This means the inserted node is in the right subtree of v. In this case let us expand the subtree D and let w be its root. Since the height of w increased as a result of an insertion, the two children of w must have different heights, say, $h-\delta$ and $h-\delta'$ where $\{\delta, \delta'\} = \{1, 2\}$. Now a double-rotation at w results in a balanced tree of height $h+1$ rooted at w.

In both cases (I.a) and (I.b), the resulting subtree has height $h+1$. Since this was height before the insertion (see (18)), there are no unbalanced nodes further up the path to the root. Thus the insertion algorithm terminates with at most two rotations.

For example, suppose we begin with the AVL tree in Figure 17, and we insert the key 9.5. This yields the unbalanced tree on the left-hand side of Figure 18. Following the rebalance path up to the root, we find the first unbalanced node is at the root, 12. Comparing the heights of nodes 3 and 8 in the left-hand side of Figure 18, we conclude that this is case (I.b). Performing a double rotation at 8 yields the final AVL tree on the right-hand side of Figure 18.
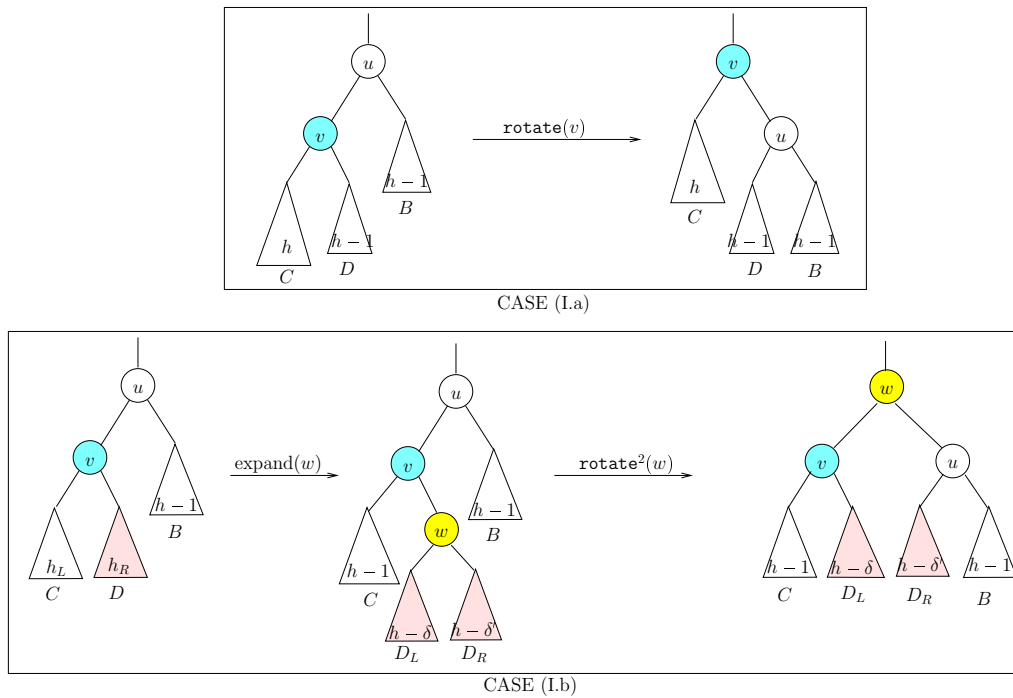
CASE (I.a)

CASE (I.b)

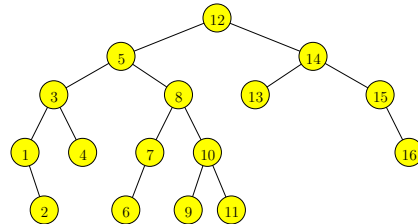Figure 16: AVL Insertion: CASE (I.a) and CASE (I.b)

Figure 17: An AVL tree

¶48. Deletion Rebalancing. Suppose the imbalance in Figure 15 comes from a deletion. The previous heights of u, v, v' must have been

$$h + 2, h + 1, h$$

and the deleted node x must be in the subtree rooted at v'. We now have three cases to consider:

CASE (D.a): $h_L = h$ and $h_R = h - 1$. This is like case (I.a) and treated in the same way, namely by performing a single rotation at v. Now u is replaced by v after this rotation, and the new height of v is $h + 1$. Now u is AVL balanced. However, since the original height is $h + 2$, there may be unbalanced node further up the rebalance path. Thus, this is a non-terminal case (i.e., we have to continue checking for balance further up the root-path).

CASE (D.b): $h_L = h - 1$ and $h_R = h$. This is like case (I.b) and treated the same way, by performing a double rotation at w. Again, this is a non-terminal case. But there is a slight difference: As in CASE (I.b), the children of w have heights $h - \delta$ and $h - \delta'$ where $\delta, \delta' \in \{1, 2\}$. However, this time, it is possible that $\delta = \delta' = 1$. CASE (D.c): $h_L = h_R = h$. This case is
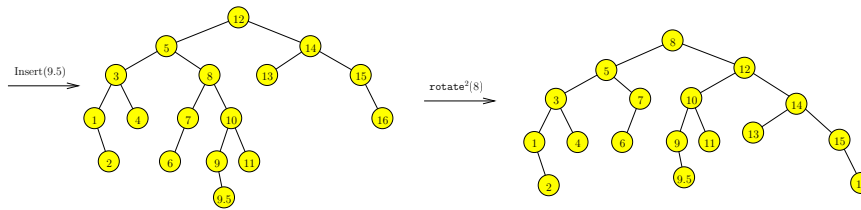
Figure 18: Inserting 9.5 into an AVL tree
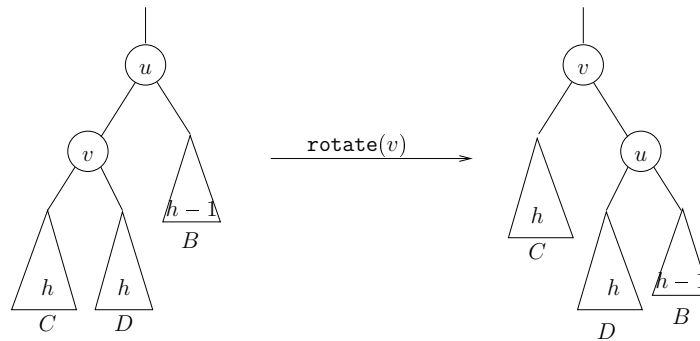
Figure 19: CASE (D.c): `rotate(v)`

new, and is illustrated in Figure 19. We simply rotate at v. We check that v is balanced and has height $h + 2$. Since v is in the place of u which has height $h + 2$ originally, we can safely terminate the rebalancing process.

This completes the description the insertion and deletion algorithms for AVL trees. In illustration, suppose we delete key 13 from Figure 17. After deleting 13, the node 14 is unbalanced. This is case (D.a) and balance is restored by a single rotation at 15. The result is seen in the left-hand side of Figure 20. Now, the root containing 12 is unbalanced. This is case (D.c), and balance is restored by a single rotation at 5. The final result is seen in the right-hand side of Figure 20.
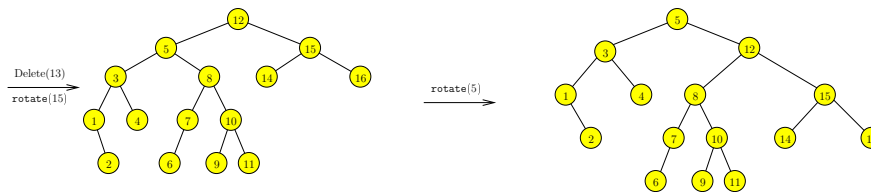
Figure 20: Deleting 13 from the AVL tree in Figure 17

Both insertion and deletion take $O(\log n)$ time. In case of deletion, we may have to do $O(\log n)$ rotations but a single or double rotation suffices for insertion.

¶49. Maintaining Balance Information. In order to carry out the rebalancing algorithm, we need to check the balance condition at each node u. If node u stores the height of u in some field, $u.H$ then we can do this check. If the AVL tree has n nodes, $u.H$ may need $\Theta(\lg \lg n)$ bits to represent the height. However, it is possible to get away with just 2 bits: we just need

Student: I thought you need $\Theta(\lg n)$ bits

to indicate three possible balance states $((0)/(+)/(-1))$ for each node u. These are just the signs of the value $u.\texttt{left.ht} - u.\texttt{right.ht}$. In implementations, we could just use an integer to represent this information, or two bits. Since two bits can encode 4 values, we are using only "3/4 of the allocated space". Theoretically, we only need $\lg 3 = 1.58\ldots$ bits per node. We explore this in the Exercises.

　　The following table tells us how to recognize the cases based on the balance information alone, and how to update the balance information. Assume that the unbalance is at u, and furthermore, $v = u.\texttt{left}$ and $w = v.\texttt{right}$. Refer to Figure 16 for Cases a and b, and Figure 19 for Case c.

	Operation	Old balance	New balance
Case (I.a):	$\texttt{rotate}(v)$	$bal(u,v) = (+^2, +)$	$bal(u,v) = (0,0)$
Case (I.b):	$\texttt{rotate}^2(w)$	$bal(u,v,w) = (+^2, -, b)$	$bal(u,v,w) = \begin{cases} (-,0,0) & \text{if } b = (+), \\ (0,+,0) & \text{if } b = (-). \end{cases}$
Case (D.a):	$\texttt{rotate}(v)$	$bal(u,v) = (+^2, +)$	$bal(u,v) = (0^*, 0)$
Case (D.b):	$\texttt{rotate}^2(w)$	$bal(u,v,w) = (+^2, -, b)$	$bal(u,v,w) = \begin{cases} (0,0,0^*) & \text{if } b = 0, \\ (-,0,0^*) & \text{if } b = (+), \\ (0,+,0^*) & \text{if } b = (-). \end{cases}$
Case (D.c):	$\texttt{rotate}(v)$	$bal(u,v) = (+^2, 0)$	$bal(u,v) = (+, -)$

　　We omit another table that is symmetric to the above table, except that we interchange the left/right relationships and signs $+/-$ in the above table. These tables have the AVL balance values $\{(0), (+), (-)\}$ as well as the unbalanced values $\{(+^2), (-^2)\}$. In addition, we have the asterisk'd value (0^*) indicating that although the node has balance (0), its height been changed and this must be communicated to its parent. Interestingly, we do not encounter $(+^*)$ or $(-^*)$.

¶50. Global Correctness: Almost-AVL Trees.　　The global correctness argument for our rebalance phase is best understood as follows: a binary tree is **almost-AVL** if it is obtained from another AVL tree T by a **surgical operation at a node** u of T. This means we replace the subtree T_u rooted at u by another AVL tree T' where $|h(T') - h(T_u)| = 1$. If $h(T') = h(T_u) + 1$, then it is an **insertion surgery**; else a **deletion surgery**. If u has depth $k \geqslant 0$, we say that the surgery is **at depth** k. If $k = 0$, then $u = u_0$, then the surgery amounts to replacing T by T'. If $k \geqslant 1$, we have a unique path from the root u_0 of T to the parent u_{k-1} of u:

$$(u_0 - u_1 - \cdots - u_{k-1}).$$

There are 3 possibilities:

(a) u_{k-1} remains AVL-balanced after the surgery, and the height of its subtree is unchanged. This is the terminal case of rebalancing: T is already an AVL-tree. E.g., original balance of u_{k-1} is $(+)$ and we have an insertion surgery at $u = u_{k-1}.\texttt{right}$. E.g., original balance of u_{k-1} is (0) and we have a deletion surgery at a child of u.

(b) u_{k-1} remains AVL-balanced, but its height has changed by ± 1. We can view T as the result of doing surgery at depth $k - 1$.

(c) u_{k-1} is no longer AVL-balanced (its balance is $+2$ or -2).

The AVL algorithms for insertion and deletion show that by applying a rotation to a child v of u_{k-1} or a double-rotation to an inner grandchild w of u_{k-1}, we can obtain an AVL-tree T''

1446 rooted at v or at w. *The height of T'' is equal to that of the original subtree at u_k, or is one less*
1447 *(if deletion) or one more (in insertion). If $k = 1$, T'' is the new AVL tree; otherwise the parent*
1448 *of the root of T'' is u_{k-2};* Thus, we can view T as the result as another insertion or deletion
1449 surgery at depth $k - 1$. The process halts when $k = 0$.

1450 **¶51. Merging AVL Trees.**   We now address two new kinds of operations on AVL trees:
1451 merging two AVL trees into one, and (in the next paragraph) splitting one AVL tree into two.

1452   Suppose T_L and T_R are two AVL trees, with $T_L < T_R$ in the sense that every key in T_L is
1453 less than the keys in T_R. We want to merge them into an AVL tree $Merge(T_L, T_R)$ in time
1454 $O(\log n)$ where the size of T_L and T_R are $O(n)$. The steps are illustrated in Figure 21.
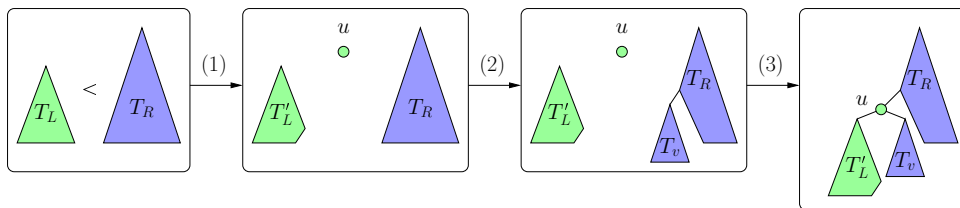
Figure 21: Steps in merging T_L and T_R.

1455   (1) Assume the height of T_L is at most the height of T_R. We first delete the tip of the
1456 right-path of T_L. Let the resulting tree be T_L' and the deleted node be u. Let h' be the height
1457 of T_L'. If T_L' is empty, $h' = -1$. This first step takes $O(1 + h_L)$ where $h_L = \mathtt{ht}(T_L)$.

1458   (2) Next, we need to merge the triple (T_L', u, T_R) into an AVL tree denoted $Merge(T_L', u, T_R)$.
1459 To do this, we first walk down the left-path of T_R until you find the first node v whose height is
1460 $\leqslant h'$ or v is tip of the left-path. Note that the heights of consecutive nodes along the left-path
1461 differs by 1 or 2. Thus there are logically three possibilities:

1462   • $\mathtt{ht}(v) = h'$.

1463   • $\mathtt{ht}(v) = h' - 1$.

1464   • $\mathtt{ht}(v) > h'$ and v is the tip of the left-path.

1465 We claim that the last case cannot happen. Otherwise, the tip v has height $\geqslant 2$ and has no
1466 left-child. This implies v is not AVL-balanced, contradiction. It takes $O(1 + h_R - h_L)$ time to
1467 find v.

1468   (3) Finally, construct a tree T_u rooted at u with left-subtree equal to T_L' and right subtree
1469 equal to T_v (i.e., the subtree rooted at v). Note that T_u is AVL of height $1 + h'$. Now, we
1470 "graft" T_u into T_R by replacing the subtree T_v by T_u. Now T_R is AVL balanced except that
1471 the nodes along the path from u to the root might have balance of $+2$. We rebalance along the
1472 root-path of u as in the usual AVL insert rebalancing. This takes time $O(1 + h_R - h_L)$. The
1473 result is the desired $Merge(T_L', u, T_R) = Merge(T_L, T_R)$

1474 **Lemma 9** *Suppose T_L and T_R are AVL trees of heights h_L and h_R.*
1475 *(a) We can compute $Merge(T_L, T_R)$ in time $O(\max\{h_L, h_R\})$*
1476 *(b) If $T_L < u < T_R$, we compute $Merge(T_L, u, T_R)$ in time $O(1 + |h_L - h_R|)$.*

Proof. Assuming $h_R > h_L$, then part(a) has complexity of steps (1), (2) and (3),

$$O(1 + h_L) + O(1 + h_R - h_L) + O(1 + h_R - h_L) = O(1 + h_R).$$

But part(b) only has steps (2) and (3), with complexity $O(1 + h_R - h_L)$.   **Q.E.D.**

¶52. Splitting AVL Trees. We can splitting AVL trees in $O(\log n)$ time, but it is slightly more involved. Suppose we want to split a tree T at a key k. We want to produce two trees, T_L (resp., T_R) comprising those keys in T that are $< k$ (resp., $\geq k$). There are two phases: the first "decomposition phase" splits T into two lists of trees, followed by the "composition phase" which repeatedly merge the trees in one list to produce T_L, and similarly for T_R with respect to other list.

Decomposition Phase: first perform a `lookUp` on k, using $O(\log n)$ time. This leads us down a path $(x_0, x_1, \ldots, x_t)$ to a node x_t with key k' that is equal to k if k is in the tree; otherwise k' is equal to the predecessor or successor of k in T. See Figure 22 for an illustration with $t = 5$.
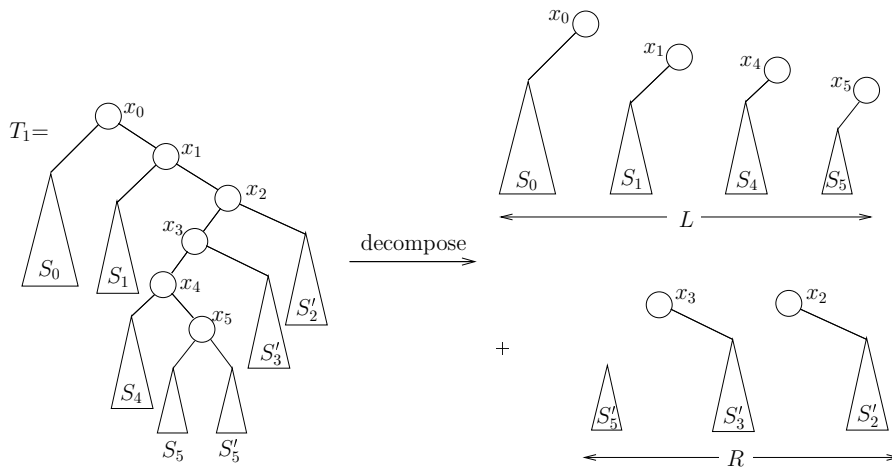

Figure 22: Split: decomposition of T.

Let S_i and S_i' denote the left and right subtrees of x_i. We form two list L and R of AVL trees: for each i, if the key in x_i is less than k, we put x_i (viewed as an AVL tree with one node) and S_i into L. Otherwise, we put x_i and S_i' into R. This rule places every node of T into L or R except possibly for S_t or S_t': note that x_t itself is in L or R. If x_t is in L then we put S_t' into R, and otherwise put S_t into R. Observation: *the trees in L and R are sorted and moreover, $L < R$.* In particular, the trees in L are sorted in the form

$$S_{i_1} < x_{i_1} < S_{i_2} < x_{i_2} < \cdots \tag{19}$$

for some $1 \leq i_1 < i_2 < \cdots$. Similarly the trees in R are

$$S_{j_1} > x_{j_1} > S_{j_2} > x_{j_2} > \cdots$$

for some $t \geq j_1 > i_2 > \cdots$. In Figure 22, we display the x_i's and $S_i's$ as one BST x_i as root; a similar remark holds for S_j's and x_j's in R.

Composition Phase: We must merge the trees in L into T_L, and similarly combine the trees in R into T_R. We may focus on L (R is similarly treated). Clearly there are $O(\log n)$ trees in L and so we can easily merge them into one tree in $O(\log^2 n)$ time. But we show that $O(\log n)$ time suffices. We rename and reindex the trees in (19) as

$$L_1 > y_1 > L_2 > y_2 > \cdots > L_\ell > y_\ell > L_{\ell+1} \tag{20}$$

where

$$\mathtt{ht}(L_1) \geqslant \mathtt{ht}(L_2) \geqslant \cdots \geqslant \mathtt{ht}(L_{\ell+1}) \geqslant -1. \tag{21}$$

Note that $L_{\ell+1}$ could be empty. See Figure 23.
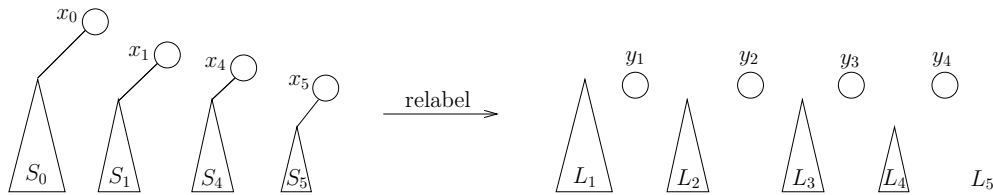
Figure 23: Relabeling the trees in the set L.

The basic idea is to merge the trees in (20) from right to left: initially, we merge $L_\ell, y_\ell, L_{\ell+1}$ to obtain $L'_{\ell-1} \leftarrow Merge(L_\ell, y_\ell, L_{\ell+1})$. Inductively, let

$$L'_{i-1} \leftarrow Merge(L_i, y_i, L'_i) \tag{22}$$

for $i = \ell - 1$ down to $i = 1$. From Lemma 9(b), we see that Step (22) takes time $O(1 + \mathtt{ht}(L_i) - \mathtt{ht}(L_{i+1}))$. Finally, we obtain one AVL tree L'_0 with total cost

$$\sum_{i=1}^{\ell} O(1 + \mathtt{ht}(L_i) - \mathtt{ht}(L_{i+1})) = O(\ell + \mathtt{ht}(L_1) - \mathtt{ht}(L_{\ell+1}))$$

Since $\ell = O(\log n)$ and $\mathtt{ht}(L_1) = O(\log n)$, this proves

Lemma 10 *We can split an AVL tree of size n in time $O(\log n)$.*

¶53. Relaxed Balancing. Larsen [11] shows that we can decouple the rebalancing of AVL trees from the updating of the maintained set. In the semi-dynamic case, the number of rebalancing operations is constant in an amortized sense (amortization is treated in Chapter 5).

¶54. Generalized AVL Trees. For any $k \geqslant 1$, we can define the family $AVL[k]$ of trees (Exercises) in which siblings differ in height by at most k. So AVL trees are just $AVL[1]$ trees. Foster [7] investigated empirically such trees for $k \leqslant 5$, showing that for modest increase in lookup time, the amount of restructuring is decreased by a factor of 10.

—— EXERCISES

Exercise 6.1: Beginning with an empty AVL tree, and insert the following sequence of keys into the tree in succession:

$$1, 2, 3, 4, 5, 6, 7$$

Please show the sequence of AVL trees obtained. Based on this, state any conjectures you have about repeated insertions of the keys $1, 2, 3, \ldots, n-1, n$ for any n. ◇

Exercise 6.2: This calls for hand-simulation of the insertion and deletion algorithms. Show intermediate trees after each rotation, not just the final tree.
(a) Insert the key 10.5 into the final AVL tree in Figure 20.
(b) Delete the key 4 from the final AVL tree in Figure 20. NOTE: part(b) is independent of part(a).

$\diamondsuit$

Exercise 6.3: Give an algorithm to check if a binary search tree T is an AVL tree. Your algorithm should take time $O(|T|)$. You must use shell programming. $\diamondsuit$

Exercise 6.4: We want to do a hand simulation of rebalancing a path after insertion/deletion. Moreover, all we know at each node u is its original balance information: $bal(u) \in \{(0), (+), (-)\}$. Suppose the rebalance path (see ¶46) is

$$(u_0, u_1, u_2, \ldots, u_5) \tag{23}$$

where u_0 is the root and u_5 is the parent of the inserted leaf (if insertion) or the parent of the cut node (if deletion). Depending on whether u_{i+1} is a left- or right-child of u_i for each i, we have different rebalancing scenarios. See Figure 24 for two possibilities. The u_i's are colored pink in Figure 24, and the sibling of each u_i is the blue node v_i (with the indicated balance information).
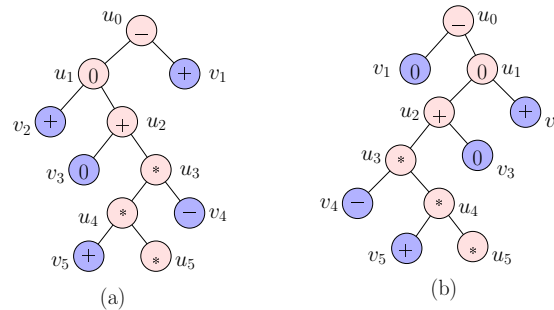

Figure 24: Two rebalance paths

As we move up the rebalance path from u_5, we update their balance information. In Figure 24, we have reached u_3 and for the first time, we need to do rotations. The updated balance information in u_5, u_4, u_3 are not shown explicitly. Although u_5, u_4 are now balanced, their heights has changed. **NOTE**: *For full credit, please tell us how the balance of nodes are updated after each rotation.*

(i) AVL Insertion for Figure 24(a): suppose there is an unbalance $(+2/-2)$ at u_3 in the Rebalance Phase. Explain what will happen in the rest of the Phase.

(ii) Repeat part (i) but using Figure 24(b).

(iii) AVL Deletion for Figure 24(a): suppose there is an unbalance at u_3 in the Rebalancing Phase. Explain what will happen in the rest of the Phase.

(iv) Repeat part (iii) but using Figure 24(b).

$\diamondsuit$

Exercise 6.5: Draw an AVL tree with 12 nodes such that, by deleting one node, you achieve these effects (respectively):

1548   (a) Rebalancing requires two double-rotations.

1549   (b) Rebalancing requires one single rotation and one double-rotation.

1550   (c) Rebalancing requires two single rotations.

1551   Note that you can have a different tree for each of the three parts. You must draw the
1552   tree after each double-rotation. HINT: It is unnecessary to assign keys to the nodes: just
1553   show the tree shape, and label some nodes to clarify the operations.   ◇

1554   **Exercise 6.6:** (a) What is the minimum number of nodes in an AVL tree of height 10?

1555   (b) Suppose we define an AVL[2]-tree to be one in which each node u satisfies $|bal(u)| \leqslant$
1556   2. In other words, $bal(u) := u.\texttt{left}.Ht - u.\texttt{right}.Ht \in \{-2, -1, 0, 1, 2\}$. What is the
1557   minimum number of nodes in an AVL[2]-tree of height 10?

1558   ◇

Exercise 6.7: We want to develop an exact formula for $\mu(h)$, the min-size AVL tree of height
h. This satisfies the recurrence

$$\mu(h) = 1 + \mu(h-1) + \mu(h-2)$$

for $h \geqslant 2$, with $\mu(0) = 1, \mu(1) = 2$.
(a) First consider a general linear recurrence of the form

$$T(n) = C + \sum_{i=1}^{k} a_i T(n-i)$$

1559   where $C, a_1, \ldots, a_k$ are constants. We say the recurrence is **homogeneous** iff $C = 0$, else
1560   **non-homogeneous**. If $T(n)$ is non-homogeneous, we transform it into a homogeneous
1561   linear recurrence,

$$t(n) = \sum_{i=1}^{k} a_i t(n-i). \tag{24}$$

by[14] choosing $t(n) = T(n) + c$ for some constant c. What is the limitation of this
transformation?
(b) Give an exact solution for $\mu(h)$ of the form

$$\mu(h) = c + A\phi^h + B\widehat{\phi}^h, \qquad (h \geqslant 0)$$

1562   where $\phi, \widehat{\phi}$ are the roots of $x^2 - x - 1$ and A, B are $\frac{2}{\sqrt{5}} \pm 1 = 1.8944, -0.1056$.

1563   (c) (Yuejie Wang, f2020) The transformation of part(a) has limitations. Show that the
1564   homogeneous linear recurrence (24) can always be achieved by $t(n) = T(n) + cn^s$ for some
1565   constant c and some $0 \leqslant s < k$.   ◇

Exercise 6.8: (Lixuan Zhu, f'2013) Show that the exact solution for the non-homogeneous
Fibonacci recurrence

$$F(n) = n + F(n-1) + F(n-2)$$

1566   is

$$\begin{aligned}
F(n) &= \frac{\phi^2}{(\phi-1)^2(\phi-\widehat{\phi})}\phi^n + \frac{\widehat{\phi}^2}{(\widehat{\phi}-1)^2(\widehat{\phi}-\phi)}\widehat{\phi}^n \\
&\quad + \frac{\phi}{(\phi-1)^2(\widehat{\phi}-1)} + \frac{\widehat{\phi}}{(\widehat{\phi}-1)^2(\phi-1)} + \frac{n}{(\phi-1)(\widehat{\phi}-1)}
\end{aligned}$$

---

[14]Note that we could such a homogeneous recurrence by defining $t(n) := T(n) - T(n-1)$. But it would be
painful to recover $T(n)$ from $t(n)$.

---

where $F(0) = 0, F(1) = 1$ and, as usual, ϕ is the golden ratio and $\widehat{\phi} = 1 - \phi$. For instance, $F[2, 3, 4, 5] = [3, 7, 14, 26]$. ◇

Exercise 6.9: My pocket calculator tells me that $\log_\phi 100 = 9.5699$ and $\log_\phi 2 = 1.4404$.
(a) What does this tell you about the maximum height achievable by an AVL tree of size 100?
(b) What is exact range of heights for an AVL tree of size 100?
NOTE: we told you the pocket calculator values so that you don't have to consult your own calculators. We want you to use the bound $\phi^{h+1} < \mu(h) + 1 \leqslant 2\phi^h$. ◇

Exercise 6.10: We know that a single insertion can always be fixed by a single rebalancing act. By a **rebalancing act**, we mean either a single rotation or a double rotation as performed during our Rebalance Phase. But how many rebalancing acts can be caused by a deletion? Let $m(k)$ be the minimum size AVL trees such that a single deletion will cause k rebalancing acts.
(a) Determine $m(1), m(2), m(3)$. You must draw the AVL tree for each $m(k)$, and mark a node whose deletion will cause k rebalancing acts.
(b) Prove that $m(k) = \mu(2k)$.
(c) Can you design the AVL tree for $m(k)$ in part(b) to be any combination of k single (S) or double (D) rotations? E.g., $k = 2$ has four combinations: SS, SD, DS, DD.
(d) In an AVL tree of size n, what is the most number of rebalancing acts you can get after a deletion. Note that $O(\log n)$ is a trivial answer, so we really what a non-asymptotic bound. ◇

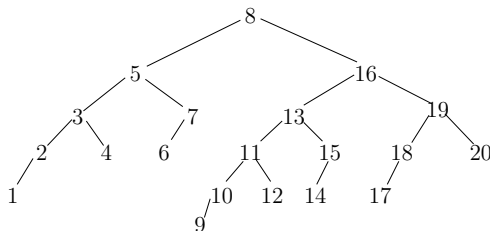**Exercise 6.11:** Consider the AVL tree in Figure 25.

Figure 25: An AVL Tree for deletion

(a) Please delete Key 6 from the tree, and draw the intermediate AVL trees after each rebalancing act. NOTE: a double-rotation counts as one act.
(b) Find the set S of keys that each deletion of a $K \in S$ from the AVL tree in Figure 25 requires requires two rebalancing acts. Be careful: the answer may depends on some assumptions.
(c) Among the keys in part (b), which deletion has a double rotation among its rebalancing acts? ◇

Exercise 6.12: Please re-insert 6 back into the tree obtained in part(a) of the previous exercise. Do you get back the original tree of Figure 25? ◇

Exercise 6.13: (a) Draw two AVL of the same size n but with different heights. Minimize the value of n.

(b) Starting with an empty AVL tree, insert the following keys, in this order:

$$1, 2, 3; 7, 8, 9; 4, 5, 6.$$

Please show the AVL tree immediately after insertion and after each rebalancing act (either single or double rotation). Call the final tree T_9. Please do this accurately as T_9 will be used in the next part.

(c) There are 3 cases in deletion: list the possible keys whose deletion from T_9 would result in Case D(c). Show the final tree after such a deletion.

$\diamondsuit$

Exercise 6.14: Let T be an AVL tree with n nodes. We consider the possible heights for T.
(a) What are the possible heights of T if $n = 15$?
(b) What if T has $n = 16$ or $n = 20$ nodes?
(c) Are there arbitrarily large n such that all AVL trees with n nodes have unique height?

$\diamondsuit$

Exercise 6.15: Let $H(n)$ denote the number of distinct heights attained by AVL trees of size n. In the previous exercise, we show that $H(n) > 1$ eventually. We now want to prove that $H(n) \to \infty$ as $n \to \infty$. $\diamondsuit$

Exercise 6.16: (a) What is the maximum height of an AVL tree of size 52?
(b) Write a program that, given any n, computes the maximum possible height of an AVL tree with n nodes. Make your program as efficient as possible. Goal: linear time and constant space solution. $\diamondsuit$

Exercise 6.17: Draw the AVL trees after you insert each of the following keys into an initially empty tree: $1, 2, 3, 4, 5, 6, 7, 8, 9$ and then $19, 18, 17, 16, 15, 14, 13, 12, 11$. $\diamondsuit$

Exercise 6.18: Insert into an initially empty AVL tree the following sequence of keys: $1, 2, 3, \ldots, 14, 15$.
(a) Draw the trees at the end of each insertion as well as after each rotation or double-rotation. [View double-rotation as an indivisible operation].
(b) Prove the following: if we continue in this manner, we will have a perfect binary tree at the end of inserting key $2^n - 1$ for all $n \geqslant 1$. $\diamondsuit$

Exercise 6.19: Consider the range of possible heights for an AVL tree with n nodes. For this problem, it is useful to recall the functions $M(h)$ in (7) and $\mu(h)$ in (15).
(a) For instance if $n = 3$, the height is necessarily 1, but if $n = 7$, the height can be 2 or 3. What is the range when $n = 15$? $n = 16$? $n = 19$?
(b) Suppose that the height h^* of an AVL trees is uniquely determined by its number n^* of nodes. Give the exact relation between n^* and h^* in order for this to be the case. HINT: use the functions $M(h)$ and $\mu(h)$.
(c) Is it true that there are arbitrarily large n such that AVL trees with n nodes has a unique height? $\diamondsuit$

Exercise 6.20: Show that for each n in the range $[\mu(h), M(h)]$ we can find an AVL tree of height h of size n. $\diamondsuit$

Exercise 6.21: Starting with an empty tree, insert the following keys in the given order: $13, 18, 19, 12, 17, 14, 15, 16$. Now delete 18. Show the tree after each insertion and deletion. If there are rotations, show the tree just after the rotation.   ◇

Exercise 6.22: Draw two AVL trees, with n keys each: the two trees must have different heights. Make n as small as you can.   ◇

Exercise 6.23: TRUE or FALSE: In CASE (D.c) of AVL deletion, we performed a single rotation at node v. This is analogous to CASE (D.a). Could we have also have performed a double rotation at w, in analogy to CASE (D.b)?   ◇

Exercise 6.24: Let $\overline{\mu}(h)$ be the number of *non*-isomorphic *min*-size AVL trees of height h. Give a recurrence for $\overline{\mu}(h)$. How many non-isomorphic min-size AVL trees are there of heights 3 and 4? Provide sharp bounds on $\overline{\mu}(h)$.   ◇

Exercise 6.25: Improve the lower bound $\mu(h) \geqslant \phi^h$ by taking into consideration the effects of "+1" in the recurrence $\mu(h) = 1 + \mu(h-1) + \mu(h-2)$.
(a) Show that $\mu(h) \geqslant F(h-1) + \phi^h$ where $F(h)$ is the h-th Fibonacci number. Recall that $F(h) = h$ for $h = 0, 1$ and $F(h) = F(h-1) + F(h-2)$ for $h \geqslant 2$.
(b) Further improve (a).   ◇

Exercise 6.26: Prove the following connection between ϕ (golden ratio) and F_n (the Fibonacci numbers):
$$\phi^n = \phi F_n + F_{n-1}, \qquad (n \geqslant 1)$$
Note that we ignore the case $n = 0$.   ◇

Exercise 6.27: Recall that at each node u of the AVL tree, we can represent its balance state using a 2-bit field called $u.BAL$ where $u.BAL \in \{00, 01, 10\}$.
(a) Show how to maintain these fields during an insertion.
(b) Show how to maintain these fields during a deletion.   ◇

Exercise 6.28: Allocating one bit per AVL node is sufficient if we exploit the fact that leaf nodes are always balanced allow their bits to be used by the internal nodes. Work out the details for how to do this.   ◇

Exercise 6.29: Implement the deletion for AVL trees. In particular, assume that after cutting a node, we need a "Rebalance(x)" procedure. Remember that this procedure needs to check the balance of each node along the re-balanced path.   ◇

Exercise 6.30: It is even possible to allocate no bits to the nodes of a binary search tree. The idea is to exploit the fact that in implementations of AVL trees, the space allocated to each node is constant. In particular, the leaves have two null pointers which are basically unused space. We can use this space to store balance information for the internal nodes. Figure out an AVL-like balance scheme that uses no extra storage bits.   ◇

Exercise 6.31: For simplicity, $AVL[2]$ trees are called "relaxed AVL trees". So the heights of
siblings differ by at most 2.
(a) Draw the smallest size relaxed AVL trees of heights $h = 0, 1, 2, 3, 4, 5$.
(b) Derive an upper bound on the height of a relaxed AVL tree on n nodes. You may
imitate arguments in the text; try to give the sharpest bounds you can.
(c) Give an insertion algorithm for relaxed AVL trees. Try to imitate the AVL algorithms,
pointing out differences.
(d) Give the deletion algorithm for relaxed AVL trees.   ◇

Exercise 6.32: Prove that the family $AVL[k]$ (for any $k \geqslant 1$) is a balanced family.   ◇

Exercise 6.33: For the standard AVL trees (i.e., AVL[1]-trees), the height of tree of size n is
not more than 44% greater than the optimal (i.e., $\lg(1+n)$). Prove that the corresponding
percentages for AVL[2] and AVL[3] are 81% and 115%, respectively.   ◇

Exercise 6.34: Give tight bounds on the height of $AVL[k]$ ($k \geqslant 1$) trees using the positive
root ϕ_k of the equation $x^{k+1} - x^k - 1$. For instance,

$$\phi_2 = \frac{1}{3} + \frac{(116 + 12\sqrt{93})^{1/3}}{6} + \frac{2}{3(116 + 12\sqrt{93})^{1/3}} \simeq 1.465571232$$

◇

Exercise 6.35: To implement AVL trees, we normally reserve 2 bits of storage per node to
represent the balance information. This is a slight waste because we only use 3 of the 4
possible values afforded by 2 bits. Consider the family of "biased-AVL trees" in which
the balance of each node is one of the values $b = -1, 0, 1, 2$.
(a) In analogy to AVL trees, define $\mu(h)$ for biased-AVL trees. Give the general recurrence
formula and conclude that such trees form a balanced family.
(b) Is it possible to give an $O(\log n)$ time insertion algorithm for biased-AVL trees? What
can be achieved?   ◇

Exercise 6.36: We introduce a new notion of "height" of an AVL tree based on the following
base case: if u has no children, $h'(u) := 0$ (as before), and if node u is `nil`, $h'(u) := -2$
(this is new!). Recursively, $h'(u) := 1 + \max\{h'(u_L), h'(u_R)\}$ as before. Let 'AVL' (AVL
in quotes) trees refer be those trees that are AVL-balanced using h' as our new notion of
height. We compare the original AVL trees with 'AVL' trees.
(a) TRUE or FALSE: every 'AVL' tree is an AVL tree.
(b) Let $\mu'(h)$ be defined (similar to $\mu(h)$ in text) as the minimum number of nodes in an
'AVL' tree of height h. Determine $\mu'(h)$ for all $h \leqslant 5$.
(c) Prove the relationship $\mu'(h) = \mu(h) + F(h)$ where $F(h)$ is the standard Fibonacci
numbers.
(d) Give a good upper bound on $\mu'(h)$.
(e) What is one difficulty of trying to use the family of 'AVL' trees as a general search
structure?   ◇

Exercise 6.37: A node in a binary tree is said to be **full** if it has exactly two children. A **full
binary tree** is one where all internal nodes are full.
(a) Show that full binary tree have an odd number of nodes.
(b) Show that 'AVL' trees as defined in the previous question are full binary trees.   ◇

1707 **Exercise 6.38:** Define an "asymmetric AVL tree" as one where each node u satisfies $bal(u) \in$
1708 $\{-1, 0, 1, 2\}$. It is asymmetric because we allow an unbalance of 2, but not an unbalance
1709   of -2. Define insertion/deletion algorithms to maintain such asymmetric AVL trees.   ◇

1710 **Exercise 6.39:** (Open-ended) Let a node u of height $h+1$ have four grandchildren with heights
1711 $\{h_0, h_1, h_2, h_3\}$. In an AVL tree, we may assume $h_0 = h - 1$. What about the other 3
1712   heights? We can have $\{h_1, h_2, h_3\} = \{h - 1 - \delta, h - 2, h - 2 - \delta'\}$ where δ, δ' are $0, 1$
1713   values. Suppose we now enforce the possible values for $\{h_1, h_2, h_3\}$. Explore such trees
1714   and algorithms.   ◇

1715 _____End Exercises

§7. Ratio Balanced Trees
1716

1717 **¶55.**   We consider another form of balance in binary trees. Instead of balancing the heights
1718 of the left and right subtrees, we now balance their sizes. Typically, we want the ratio of the
1719 left and right sizes to be within a fixed range $(\rho, 1/\rho)$ for some $0 < \rho < 1/2$. These are called
1720 **weight balanced trees** and were first introduced by Nievergelt and Reingold (1973). We will
1721 see that their insertion/deletion algorithms are simpler than in AVL trees, but the analysis
1722 is more involved. It turns out that weight balancing is more "robust" than height balancing
1723 in a certain sense that is crucial for applications such as multidimensional range search trees
1724 [13]. We will also look at an interesting combination of weight and height balancing called
1725 **logarithmic BST** from Roura [18].

¶56. On ρ-Ratio Balanced Trees.   Recall that the size of a binary tree T is the number
of nodes in T, which we denote by $\texttt{size}(T)$. Instead of size, we will now use a related quantity
$\texttt{esize}(T) := 1 + \texttt{size}(T)$. We can call $\texttt{esize}(T)$ the **extended size** of T, and it can be
interpreted as the number of $\texttt{nil}$ nodes in the extended binary tree corresponding to T (see
¶39). Figure 26(a) shows a binary tree and Figure 26(b) is the corresponding extended binary
tree. If T is rooted at a node u, we may write $\texttt{esize}(u)$ for $\texttt{esize}(T)$. Thus,

$$\texttt{esize}(u) = \begin{cases} 1 & \text{if } u = \texttt{nil} \\ \texttt{esize}(u.\texttt{left}) + \texttt{esize}(u.\texttt{right}) & \text{if } u \neq \texttt{nil} \end{cases}$$

The key definition is the **ratio** of u, defined as

$$\text{ratio}(u) := \frac{\texttt{esize}(u.\texttt{left})}{\texttt{esize}(u.\texttt{right})}.$$

1726 Let us see why we prefer $\texttt{esize}$ to $\texttt{size}$ in defining ratio(u). The recursive equations for $\texttt{esize}$
1727 and $\texttt{size}$ are thus:

$$\begin{aligned} \texttt{esize}(u) &= \texttt{esize}(u.\texttt{left}) + \texttt{esize}(u.\texttt{right}) \\ \texttt{size}(u) &= 1 + \texttt{size}(u.\texttt{left}) + \texttt{size}(u.\texttt{right}). \end{aligned}$$

1728 The "$1 + \cdots$" term in the definition of $\texttt{size}(u)$ is inconvenient when computing ratios of sizes.
1729 Moreover, when taking ratio of two sizes, we might divide by a zero size. In contrast, $\texttt{esize}$ is
1730 at least 1, and their ratios are always well-defined.

Suppose $\rho \in (0, \frac{1}{2})$. We say u has ρ-**ratio balance** if

$$\rho < \text{ratio}(u) < 1/\rho.$$

Note that both inequalities here are strict. This detail will simplify our development; in particular, it allows us to focus on simple constants such as $\rho = 1/3$ for balancing.
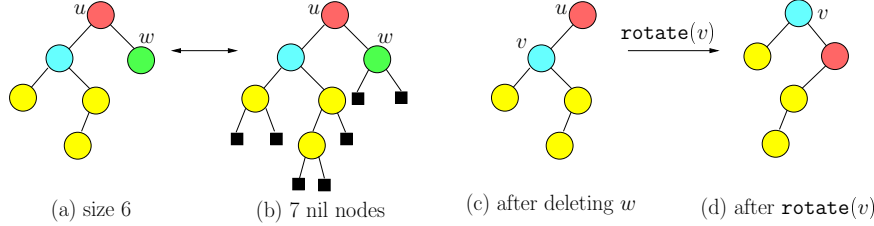

(a) size 6   (b) 7 nil nodes   (c) after deleting w (d) after `rotate(v)`

Figure 26: (a) Binary tree with ratio$(u) = 5/2$. (b) Corresponding extended binary trees. (c) ratio$(u) = 5/1$ after deletion. (d) ratio$(v) = 2/4$ after rotation; but we need one more rotation at $u.\texttt{left}$ to get a RBT.

¶57. **Relation to β-Bounded Balanced Trees.** Let $\beta \in (0, \frac{1}{2})$. Nievergelt and Reingold defined the **root balance** of u to be

$$B(u) := \frac{\texttt{esize}(u.\texttt{left})}{\texttt{esize}(u)} = \frac{\texttt{esize}(u.\texttt{left})}{\texttt{esize}(u.\texttt{left}) + \texttt{esize}(u.\texttt{right})}. \tag{25}$$

For instance, the root u in Figure 26(a) has ratio $5/2$ but root balance $5/7$. A binary tree T is said to have β-**bounded balance** if[15]

$$\beta < B(u) < 1 - \beta$$

holds for each node u. There is a simple connection between bounded balance and ratio balance:

Lemma 11 *For $\beta, \rho \in (0, \frac{1}{2})$, let*

$$\overline{\beta} := 1 - \beta, \qquad \tilde{\rho} := 1 + \rho. \tag{26}$$

(a) Then

$$\rho = \beta/\overline{\beta} \quad \Longleftrightarrow \quad \beta = \rho/\tilde{\rho}$$

(b) If $\rho = \beta/\overline{\beta}$ (equivalently $\beta = \rho/\tilde{\rho}$) then T is β-bounded balanced iff T ρ-ratio balanced.

Proof. We omit the the elementary calculations of (a). For (b), let $L = \texttt{esize}(u.\texttt{left})$ and $R = \texttt{esize}(u.\texttt{right})$ for any node u in T. Then ratio$(u) = L/R > \rho$ iff

$$\begin{aligned}
L/R > \beta/\overline{\beta} \quad &\Leftrightarrow \quad (1 - \beta)L > R\beta \\
&\Leftrightarrow \quad L > (L + R)\beta \\
&\Leftrightarrow \quad L/(L + R) > \beta.
\end{aligned}$$

I.e., $B(u) > \beta$. A similar argument shows that ratio$(u) < 1/\rho$ iff $B(u) < 1 - \beta$.   **Q.E.D.**

---

[15]The usual definition uses non-strict inequality, $\beta \leqslant B(u) \leqslant 1 - \beta$. But we use strict inequality to be consistent with our notion of ρ-ratio balance.

1742 In view of this, we can freely go back and forth between the two concepts. For example, the
1743 bounded balance of $\beta = 1/4$ is equivalent to ratio balance $\rho = 1/3$ since $\beta/\overline{\beta} = (1/4)/(3/4) =$
1744 $1/3$. We prefer to use the ratio balance concepts because it offers several[16] pedagogical advan-
1745 tage.

Weight balanced trees is another common name for bounded balanced
trees. Nievergelt and Reingold defined the class "$BB[\beta]$" comprising the
β-bounded balanced trees. Similarly, let the class "$RB[\rho]$" comprise the
ρ-ratio balanced trees. E.g., if $\beta = 1/4$ then $\widetilde{\beta} = 3/4$ and therefore $\rho =$
$(1/4)/(3/4) = 1/3$. Thus, $BB[1/4] = RB[1/3]$.

1746 Consider the possible ratios of small sized trees. If the (ordinary) size of
T_u is 1, then ratio$(u) = 1$. If the size of T_u is 2, then ratio$(u) = 1/2$ if
u.left = nil, and ratio$(u) = 2$ if u.right = nil. This shows that if T
is ρ-ratio balanced and $\rho \geqslant 1/2$, then T cannot have any subtree of size 2.
This is a severe restriction, as it implies the size of T must be odd. For this
reason, we only consider the class of $RB[\rho]$ trees for $\rho < 1/2$. If the size of
T_u is 3, then ratio$(u) \in \{1/3, 1, 3\}$.

Just as "AVL tree" usually refers only to the most important case of AVL[1]
tree (see ¶54), we will similarly let "RB tree" (or RBT) refer to the case
RB[1/3] when there is no ambiguity. In these notes, we will give the first
complete analysis of $RB[1/3]$ (equivalently, $BB[1/4]$).

1747   It can be verified that for trees of size up to 5, the AVL trees and the $RB[1/3]$ trees are
1748 co-extensive: a tree is AVL iff it is $RB[1/3]$. But Figure 26(a) shows a binary tree of size 6
1749 that is $BB[1/3]$ but not AVL. This tree is not AVL because we know that the smallest AVL
1750 tree of height 3 has size $\mu(3) = 7$.

1751   Weight balanced trees were introduced by Nievergelt and Reingold [17], with corrected
1752 analysis by Blum and Mehlhorn [4]. Nievergelt and Reingold realized that β must not get too
1753 close to $\frac{1}{2}$: in fact, $\beta < 1 - \frac{1}{\sqrt{2}}$ is necessary. Blum and Mehlhorn observed another issue: the
1754 rebalance rule also run into problems when the tree size is too small, and their solution is to
1755 assume β is bounded away from 0. It is intuitively clear why this is necessary: any simple
1756 rebalance rule applied to a small tree could introduce a large discrete change that violates
1757 $BB[\beta]$ for small β. Thus, the rebalancing theorem of Blum and Mehlhorn requires an upper
1758 and lower bound on β: $2/11 < \beta < 1 - \frac{1}{\sqrt{2}}$. We will provide a simplified and complete analysis
1759 for $BB[1/4] = RB[1/3]$ trees.

¶58. On Minimum Size RBTs: the function $\mu(h)$ To bound the size of $RB[\rho]$ trees, we
follow our earlier approach for AVL trees by defining $\mu(h)$ to be the minimal esize of a $RB[\rho]$
tree of height h. For example, if $\rho = 1/3$, we have

$$\mu(0) = 2, \quad \mu(1) = 3, \quad \mu(2) = 5, \quad \mu(3) = 7, \quad \mu(4) = 10, \quad \mu(5) = 15, \dots$$

1760 as illustrated in Figure 27.

1761   We also draw the min-size RBT's T_h of height h for $h \leqslant 5$ in Figure 27. Each T_h is made
1762 **canonical** by requiring that the ratio of each node $u \in T_h$ is $\geqslant 1$ (i.e., the size of the right-
1763 subtree of u is not less than that of its left-subtree). To justify the data in this figure, we use
1764 the following lemma:

---

[16]We can work with ratios that are rational numbes, e.g., ratio$(u) = 1/3$. The "critical ratio" β_0 for bounded
balance is not an algebraic integer; but the corresponding critical ratio ρ_0 for ratio balance is an algebraic
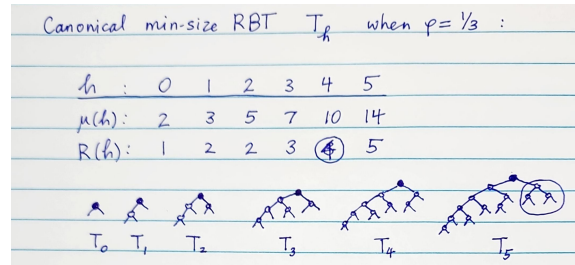integer. See Ex.III.7.13.

Figure 27: T_h is the canonical RBT of height h and `esize` $\mu(h)$. We may extend the table with the values $\mu(-1) = 1$ and $R(-1) = 1$.

Lemma 12 *Let $\rho \in (0, \frac{1}{2})$ and $h \geq 1$. Then $\mu(1 + h) = \mu(h) + R(h)$ where*

$$R(h) = R_\rho(h) := 1 + \lfloor \rho \cdot \mu(h) \rfloor.$$

In case ρ is irrational, we can simply write $R(h) = \lceil \rho \cdot \mu(h) \rceil$.

Before proving the lemma, we will use it to verify the data in Figure 27. E.g., assuming that $\mu(3) = 7$, our lemma tells us that $\mu(4) = \mu(3) + R(3) = 7 + 3 = 10$. Thus, we can inductively use this lemma to successive values of $\mu(h)$, but it is short of a simple formula for $\mu(h)$ (unlike the formula for AVL trees). Note that all the listed values of $R(h)$ have the form $\mu(h')$ for some $h' < h$, with the exception of $R(4) = 4$ (this "anomalous" value and the corresponding subtree are circled).

Proof. Let T_{h+1} denote a minimum size tree of height $h + 1$. Wlog, its left-subtree has height h. Then its `esize` must be equal to $\mu(h)$. If the right-subtree has `esize` of R, then by definition of ρ-ratio balance, $R > \rho \cdot \mu(h)$. But

$$R > \rho \cdot \mu(h) \quad \Longleftrightarrow \quad R \geq 1 + \lfloor \rho \cdot \mu(h) \rfloor.$$

This is an elementary fact.[17] By minimality of $\mu(h + 1)$, we must have $R(h) = 1 + \lfloor \rho \cdot \mu(h) \rfloor$. In case ρ is irrational, it is easier to simpler to write $R(h) = \lceil \rho \cdot \mu(h) \rceil$.   **Q.E.D.**

¶59. On Maximum Height RBTs: the function $H(n)$ To show that $RB[\rho]$ trees form a balanced family, let $H(n)$ denote the maximum height of a $RB[\rho]$ tree of `esize` $\leq n$. For instance, the first three values of H are given by $H(1, 2, 3) = (-1, 0, 1)$ (independent of ρ) but $H(4)$ could be 1 or 2, depending on ρ.

Note if n is not an integer, then $H(n) = H(\lfloor n \rfloor)$

Lemma 13 *The function $H(n)$ satisfies the recurrence*

$$H(n) = 1 + H(\beta(n)) \tag{27}$$

where $\beta(n) := \left\lceil \frac{n}{1+\rho} \right\rceil - 1$.

---

[17]It is also an application of the mixed integer inequalities discussed in ¶67 in §8 on generalized B-trees.

Proof. Let u be the root of an $RB[\rho]$ tree with esize n. Then $n = a+b$ where $\texttt{esize}(u.\texttt{left}) = a$ and $\texttt{esize}(u.\texttt{right}) = b$. Without loss of generality, let $a \leqslant b$ and we have the inequality $\rho < a/b$. Thus $b\rho < a = n - b$, or $b(1 + \rho) < n$ or $b < \frac{n}{1+\rho}$. Since b is an integer, the mixed integer inequalities (see ¶67 in §8) tells us that it is equivalent to

$$b \leqslant \left\lceil \frac{n}{1 + \rho} \right\rceil - 1$$

The lemma follows since $H(n)$ is maximimimized when b is maximized. **Q.E.D.**

Theorem 14 (Height Bound)

$$H(n) = \left\lfloor \log_{1+\rho}(n) \right\rfloor - C(n)$$

where $C(n) \geqslant 0$ is upper bounded by a constant $K_0 = K_0(\rho) > 0$ that depends only on ρ.

Proof. Note that $\beta(n) < \frac{n}{1+\rho}$ because $\lceil x \rceil - 1 < x$ for any real $x > 0$ (and choose $x = n/(1+\rho)$). Consider a related recurrence

$$\overline{H}(n) = 1 + \overline{H}\left(\frac{n}{1+\rho}\right). \tag{28}$$

Comparing (28) with (27), we conclude for all n, $\overline{H}(n) \geqslant H(n)$ if we choose the Real Basis of $\overline{H}(n)$ to be $\overline{H}(n) = H(n)$ for $n < 1 + \rho$. ("by DIC"). Then

$$
\begin{aligned}
\overline{H}(n) &= 1 + \overline{H}\left(\frac{n}{(1+\rho)}\right) \\
&= i + \overline{H}\left(\frac{n}{(1+\rho)^i}\right) && \text{(by induction on } i \geqslant 0) \\
&= \left\lfloor \log_{(1+\rho)} n \right\rfloor + \overline{H}(x) && \left(\text{let } i = \left\lfloor \log_{(1+\rho)} n \right\rfloor, \text{ and so } x < 1 + \rho\right) \\
&\leqslant \left\lfloor \log_{(1+\rho)} n \right\rfloor && \left(\text{since } \overline{H}(x) = H(x) \leqslant H(1+\rho) \leqslant H(3/2) = 0\right).
\end{aligned}
$$

This proves that $RB[\rho]$ is a balanced family.

To prove a lower bound on $H(n)$, consider the recurrence

$$\underline{H}(n) = 1 + \underline{H}\left(\frac{n}{1+\rho} - 1\right). \tag{29}$$

Again, we see that $H(n) \geqslant \underline{H}(n)$ for all n provided we choose the Real Basis to be $\underline{H}(n) = H(n)$ for $n < (1+\rho)^{h_0}$ for some $h_0 \geqslant 1$ to be chosen:

$$
\begin{aligned}
\underline{H}(n) &= 1 + \underline{H}\left(\frac{n}{(1+\rho)} - 1\right) \\
&= 2 + \underline{H}\left(\frac{n}{(1+\rho)^2} - (1 + (1+\rho)^{-1})\right) \\
&= i + \underline{H}\left(\frac{n}{(1+\rho)^i} - \left(\sum_{j=0}^{i-1}(1+\rho)^{-j}\right)\right) && \text{(by induction on } i) \\
&\geqslant \left\lfloor \log_{(1+\rho)} n \right\rfloor - h_0 + \underline{H}\left(x - \sum_{j \geqslant 0} y^j\right) && \left(\text{let } i = -h_0 + \left\lfloor \log_{(1+\rho)} n \right\rfloor, \text{ and } y := (1+\rho)^{-1}\right) \\
&= \left\lfloor \log_{(1+\rho)} n \right\rfloor - h_0 + \underline{H}\left(x - (1-y)^{-1}\right) && \text{(by the geometric series for } y) \\
&\geqslant \left\lfloor \log_{(1+\rho)} n \right\rfloor - h_0 && \left(\text{provided } 2 \leqslant x - (1-y)^{-1}\right).
\end{aligned}
$$

Note that the last proviso implies that $\overline{H}(x - (1-y)^{-1}) = H(x - (1-y)^{-1}) \geqslant H(2) \geqslant 0$, thus justifying its removal from the expression. But why is the proviso true? Our choice of $i = -h_0 + \left\lfloor \log_{(1+\rho)} n \right\rfloor$ implies that $x \geqslant (\log_{(1+\rho)})^{h_0}$. Since $y = (1+\rho)^{-1}$ is a constant that depends only on ρ, we can choose h_0 just large enough to ensure that $2 + (1-y)^{-1} \leqslant (1+\rho)^{h_0} \leqslant x$.

1797   **Q.E.D.**

1798

1799   This theorem proves that $RB[\rho]$ is a balanced family. We next show that $RB[\rho]$ trees
1800 satisfies a new kind of "height balance" (to be contrasted with the height balance of AVL$[k]$
1801 trees):

1802 **Lemma 15 (Ratio of Heights)** *If u is a node in a $RB[\rho]$ tree, then the ratio of the heights*
1803 *of u.left and u.right can be arbitrarily close to $\lg(1 + \rho)$. Moreover, it is at most $\lg(1 + \rho)$.*

1804

1805 *Proof.* Let u be the root of tree in $RB[\rho]$. Suppose $\texttt{esize}(u.\texttt{right}) = s$. Then the height
1806 of u.right is at most $H(s) \leqslant \log_{1+\rho} s$ as in the previous lemma. Also $\texttt{esize}(u.\texttt{left}) > s\rho$.
1807 Hence the height of u.left is at least $\lg(s\rho)$. The ratio of these heights is at least $\frac{\lg(s\rho-1)}{\log_{1+\rho} s}$
1808 which approaches $\lg(1 + \rho)$ for large s. It is also easy to see that the height ratio cannot be
1809 more than $\lg(1 + \rho)$.   **Q.E.D.**

if $\rho = 1/3$, the height ratios approach $\lg(1 + \rho) = (2 - \lg 3) \sim 0.415$.

1810

1811   This lemma shows that $RB[\rho]$ trees contain non-AVL trees. Conversely, we claim that the
1812 family of AVL trees is not contained in $RB[\rho]$ for any $\rho > 0$.

1813   To see this, for each $h \geqslant 0$, there is an AVL tree T_h of height h and size $\mu(h) \leqslant 2\phi^h - 1$
1814 (see (16)). Let S_h denote the perfect binary tree of height h and size $2^{h+1} - 1$. Consider the
1815 AVL tree whose left subtree is T_h and right subtree is S_h. The ratio balance of the root is
1816 $\leqslant 2\phi^h/2^{h+1}$, which is asymptotically $(\phi/2)^h$. Since $(\phi/2)^h \to 0$ as $h \to \infty$, the ratio balance of
1817 AVL trees can be arbitrarily close to 0. This proves our claim.

1818   Algorithms for ratio balanced trees will require $\texttt{esize}(u)$ to be stored at each node u. If
1819 there are n nodes in the tree, we may need up to $\lg n$ space to store this information. This is
1820 inferior to the $O(\log \log n)$ space needed for general height balanced schemes, or the 2-bits for
1821 AVL trees.

1822 **¶60. Box Queries: Motivation for Ratio Balancing.** Maintaining size information
1823 for weight balancing takes $\log n$ space per node, as compared to $O(\lg \lg n)$ space for height
1824 balancing. However, this size information allows you to locate an item in the BST based on
1825 rank in $O(\log n)$ time; this feat is beyond height information. Let us now discuss some more
1826 significant applications that also benefit from balancing by weight. Consider a fundamental
1827 scenario:

1828   (S) *Starting from an initially empty BST tree, perform a n sequence of insertions and*
1829 *deletions.*

1830   What is the total cost of the n operations in (S)? With any balanced family, this cost is
1831 $O(n \log n)$. Alternatively, we say the "amortized cost" of each operation in (S) is $O(\log n)$. But
1832 suppose we introduce a new cost model for rotation:

1833   (C) *The cost of $\texttt{rotate}(u)$ is $size(u) + size(u.\texttt{parent})$ where $size(u)$ denotes the size of the*
1834 *subtree at u.*

Now, each insertion or deletion can have linear cost $\Omega(n)$. With such a cost model, does each operation in scenario (S) still have amortized cost of $O(\log n)$? As we shall see, this is impossible with AVL trees (or any height balancing schemes). But with weight balanced trees, the amortized cost remains $O(\log n)$.

How does the cost model (C) arise? We consider a problem of computational geometry, to maintain a set $S = \{p_1, \ldots, p_n\}$ of points $p_i \in \mathbb{R}^2$ in the plane, subject to insertions and deletions and queries. Each query is specified by a box $B = [a, b] \times [c, d] \subseteq \mathbb{R}^2$. The response to the query is the set $B \cap S$ (or perhaps the cardinality of $B \cap S$). Such queries are also called **range queries**. E.g., S represents an employee data base where each employee record stores (among other things) the employee date-of-birth (x) and salary (y). We can view the record as point $p = (x, y)$ so we can query for all employees whose date-of-birth lies in some range $[a, b]$ and salary lies in some range $[c, d]$. A data structure for S could be an external BST using date-of-birth's as search key. Let $T(S)$ denote such a BST. Each internal node u in $T(S)$ would store the range (i.e., minimum and maximum) of the date-of-births in its subtree T_u. To allow a search based on salaries, we associate with u an **auxiliary BST** $u.\text{aux}$ which stores all the external nodes in T_u using salary as search key. Note that each external node in $T(S)$ is duplicated in $O(\log n)$ auxiliary BST's and so the space requirement of this data structure is $\Theta(n \log n)$, not linear.

Suppose we perform a rotation at the node $v = u.\text{left}$, as in Figure 30(i). After rotation, the auxiliary structure $v.\text{aux}$ can be taken from the previous value of $u.\text{aux}$. However, the new value of $\text{esize}(u)$ is now $b + c$, and $u.\text{aux}$ must be a new auxiliary BST tree obtained as a merge of the original $v.\text{right.aux}$ with the original $u.\text{right.aux}$. The overall operation takes time $\Theta(b + c)$ since we must first make new copies of these trees before merging. However, it is not hard to show that the worst case $\Theta(b + c)$ cost is really amortized $O(1)$ cost because after an expensive operation needs $\Theta(b + c)$ inserts/deletes to build up to. You will learn such amortizations arguments in Chapter VI.

height rebalancing is fragile; ratio rebalancing is robust!

With an AVL tree or 2-3 trees (say), this update complexity cannot be[18] amortized. This claim is proved using the scenario in Figure 28, where we start with an AVL tree T_h with 3 subtrees of height $h - 1$ and size $2^h - 1$. Let a (b) be smaller (larger) than any element in T_h. Now we repeat this cycle of four operations: insert and delete a, then insert and delete b. After each cycle, we return to the original tree T_h, but each cycle has two rotations, each costing $\Theta(2^h)$. Clearly, there is no amortized $O(1)$ cost possible.

The intuitive reason is this: although rotations can restore height balance temporarily, a subsequent insertion or deletion can bring back the same height unbalance condition. Rotations in ratio balanced trees are not susceptible to this behavior. Anderson [1] introduced a class of trees called **general balanced trees** that can provide an alternative to bounded balanced trees in the present application. General balanced trees exploit the partial rebuilding of trees in order to achieve optimal amortized complexity.

¶61. Insertion and Deletion Algorithms. As promised, we now focus on the family of $RB[1/3]$ trees (equivalently, $BB[1/4]$ trees). Note that $\rho = 1/3$ is perhaps the "simplest" ratio we could use, since we noted that the family $RB[1/2]$ is not tenable. Focusing on a single ratio $1/3$ is rather like our treatment of AVL trees, where we focus on height balance of 1 (and not height balance of some arbitrary k). Henceforth, we simply say **RB trees** (or RBT) to refer to $RB[1/3]$ trees. A node is said to be **RB balanced** if its ratio is in the range $(1/3, 3)$.

---

[18]See Theorem 3.1, [20, p.608].
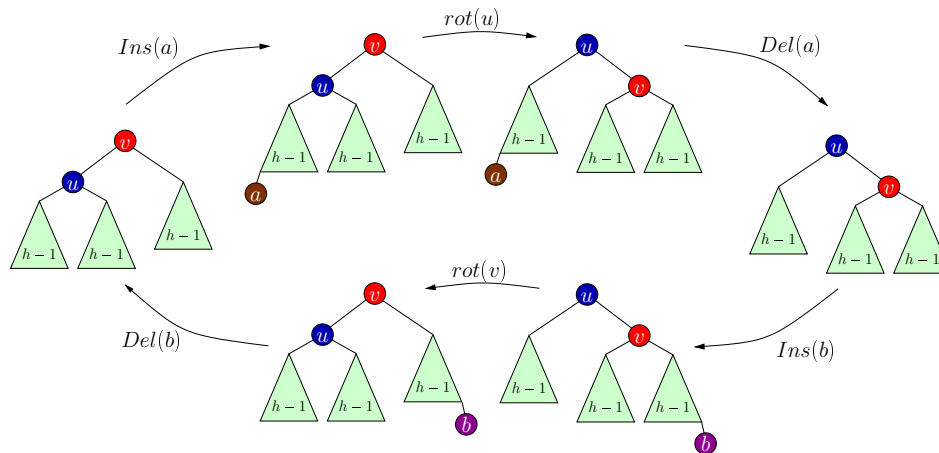
Figure 28: Cycle: $(Ins(a) \to Del(a) \to Ins(b) \to Del(b))$

In practice, it is seldom the case that we need to use more than one choice of ρ. Nevertheless see [8] in the following box.

> *Implementation and Correctness.* Yoichi Hirai and Kazuhiko Yamamoto [8] described their quest to find a rigorous correctness proof for insertion and deletion into β-bounded balance trees (for various β). Given a β, they considered valid pairs (β, γ) where γ is the constant used for deciding whether to do single and double rotation. It turns out that many implementations use invalid pairs of (β, γ). They reduced the validity problem to checking 14 inequalities in five parameter zones, and used the automated proof assistant `Coq` from Bertot & Casteran (2004). Experimental results show how the choice of these constants affect the speed performance.

Suppose T is an RB tree. To insert or delete a given item, we carry out the standard BST insertion or deletion algorithms. If insertion, this results in a new leaf x in T. If deletion, we will physically "cut" a node x of degree $\leqslant 1$. In either case, let u be the parent of x. Clearly, every node that is not RB balanced must lie on the root-path of u, known as the "rebalance path" (see ¶46 for AVL insertion/deletion). We therefore move up this path, rebalancing at each such node. Observe that before an insertion (deletion) rebalancing, the `esize` of each node on the rebalance paths is one less (more) than its true value. To distinguish these two cases, we pass a boolean variable `deleteFlag` that is true iff this is deletion rebalancing.

The following Rebalance procedure accomplishes this:

```
Rebalance(u, deleteFlag)
1.   Do
2.       deleteFlag? u.esize−− : u.esize++
3.       Rebalance_Step(u, deleteFlag)
4.       If (u ≠ u.parent)
5.           u ← u.parent
6.       else
7.           return.
```

1893 Thus we do the Rebalance-Step on each node of the rebalance path. Unlike AVL trees, *this*
1894 *loop cannot stop until the root is rebalanced!* The reason is the `esize` and possibly balance of
1895 *every node* in the rebalance path has been changed by the insertion or deletion. No rebalancing
1896 in any subtree alters this fact! At the end of this loop, T would be balanced. Line 1 uses a
1897 programming construct from `Java` called the "ternary conditional": if the `deleteFlag=true`,
1898 then u.`esize` is decremented, otherwise u.`esize` is incremented. Lines 2 and 4 use the conven-
1899 tion that u is the root iff $u = u$.`parent`. In Line 3, we call the following Rebalance-Step(u)
1900 subroutine to ensure that the subtree rooted at u is RB balanced.

1901

1902

> Rebalance-Step$(u$, `insertFlag`$)$
> 0.   If (`deleteFlag` $\wedge$ `size`$(u) = 5$)
>   Return(SPECIAL(u))
> 1.   If (ratio$(u) \geqslant 3$)
> 2. $v \leftarrow u$.`left`
> 3.   If (ratio$(v) \geqslant 3/5$),
> 4.   Return(`rotate`(v))   ◁ *See Figure 30(i)*
> 5.   else   ◁ ratio$(v) < 3/5$
> 6.   Return(`rotate`$^2(v$.`right`$)$)   ◁ *See Figure 30(ii)*
> 7.   elif (ratio$(u) \leqslant 1/3$)
>   ▷ *By symmetry:*
> 8. $v \leftarrow u$.`right`
> 9.   If(ratio$(v) \leqslant 5/3$)
> 10.   Return(`rotate`(v))
> 11.   else   ◁ ratio$(v) > 5/3$
> 12.   Return(`rotate`$^2(v$.`left`$)$)
> 13. Return(u) ◁ *u is now RB balanced*

1903

> Line 0 of Rebalance-Step(u) only applies when `size`$(u) = 5$ after dele-
> tion! We will discuss this anomaly under "deletion of small trees" below.
> The flow of control depends on the value of ratio(w) where w is either
> on the rebalance path or the left or right child of nodes on the rebal-
> ance path. Since ratio$(w) = w$.`left`.`esize`$/w$.`right`.`esize`, we know
> that the `esize` of w.`left` and w.`right` have already been updated if
> they lie on the rebalance path. Thus the ratio(w) is correctly computed.
> Also note that, by definition, after performing `rotate`(w), the node w
> is returned.

1904

1905 A simple illustration of this rebalancing is given in Figure 29.

> **The Critical Constant** $3/5$. Note that our algorithm compares
> ratio(v) to this critical constant in Line 3 to decide whether to do single
> or double rotation. We have judiciously chosen this to be a "simple"
> rational number. The original paper of Nievergelt and Reingold used
> $\sqrt{2}$ as the critical constant. Clearly, an implementation based on irra-
> tional critical constants require some extra care (see Hirai-Yamamoto
> [8]). Roura [18] has argued against the use of irrational critical con-
> stants.

1906
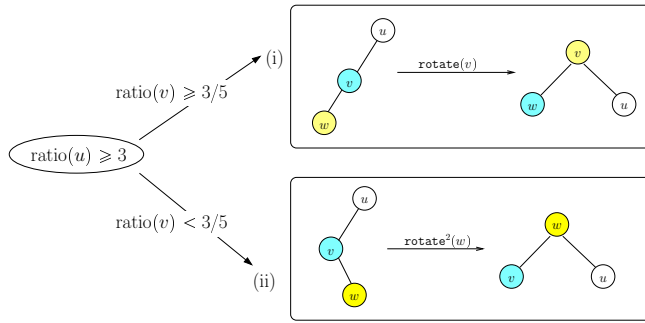
Figure 29: Simple example of Rebalance-Step(u)

Both Lines 4 and 6 return the node that is being rotated. The codes in Lines 8-12 are just a copy of Lines 3-6, except that we interchange `left` and `right`, and replace the condition "ratio(v) ⩾ 3/5" by "ratio(v) ⩽ 5/3". The decisions of Lines 2-6 are illustrated in Figure 30.
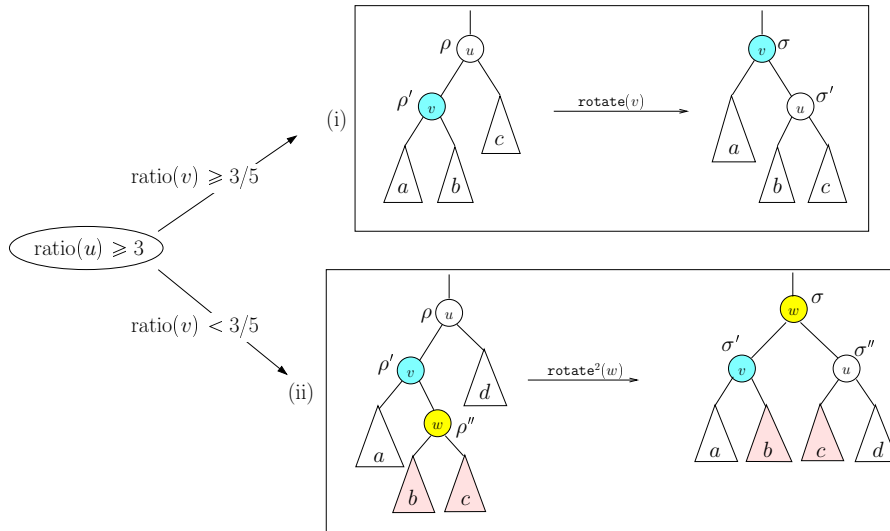

Figure 30: Rebalance-Step: (i) Single Rotate (ii) Double Rotate

The above insertion/deletion algorithm is easily turned into a top-down algorithm as follows: while we are doing a LookUp with the aim of inserting/deleting, we can preemptively perform the re-balancing as we go down the tree. In other words, to insert into (resp., delete from) the subtree at u, we first increment (resp., decrement) the size stored at u. By symmetry, assume we are about to insert into the subtree at u.`left`, or about to delete from the subtree at u.`right`. These operations will increase the ratio of u. We check whether this would make ratio(u) ⩾ 3. If so, we carry out the Rebalance-Step(u) pre-emptively. We continue the insertion/deletion after this rebalancing (if any). There is one other detail: our insertion or deletion might fail (we will not know until the end of operation). When this happens, our pre-emptive re-balancing is in vain. But nevertheless, there is no need to undo these re-balancing steps. We do need, however, to do another top-down pass from the root to the leaf in order to decrement (resp. increment) one from the weights since our insertion (resp. deletion) did not succeed. Although this is a 2-pass algorithm, both passes do not need a stack (or parent painter).

¶**62. Analysis of Ratio-Balanced Trees.**   Although the algorithms for RBT are simple, their analysis is somewhat[19] intricate. The follow is a new analysis similar to the approach of Willard and Leuker [20].

The critical analysis of ratio balancing is given by the scenario illustrated in Figure 30: we have a BST T_u rooted at u with $v = u.\texttt{left}$ and $w = v.\texttt{right}$. Assume $\text{ratio}(u) = \rho$, $\text{ratio}(v) = \rho'$ and $\text{ratio}(w) = \rho''$. It is assumed that every node in T_u is RB balanced, with the possible exception of u. Moreover, assume that $\text{ratio}(u) \in [\frac{1}{4}, 4]$, i.e., if it is not RB balanced, then $\text{ratio}(u) \in [\frac{1}{4}, \frac{1}{3}] \cup [3, 4]$. There are three tasks before us:

Task(a) After we perform $\texttt{rotate}(v)$, let the new ratios of v and u be σ and σ', as shown Figure 30(i). We must express σ and σ' in terms of ρ and ρ'.

Similarly, if we perform $\texttt{rotate}^2(v)$, let the new ratios of w, v, u be σ, σ' and σ'', as shown Figure 30(ii). We must express $\sigma, \sigma', \sigma''$ in terms of ρ, ρ', ρ''.

Task(b) Suppose, with the exception of u, each node v in T_u is 1/3-ratio balanced. As for u, we assume that $\text{ratio}(u)$ lies in the range $[3, 4]$ or $[1/4, 1/3]$. We want to prove the application of Rebalance-Step(u) results in a RB tree. This analysis will depend on the expressions for $\sigma, \sigma', \sigma''$ from Task(a).

Task(c) Conclude that our insertion/deletion algorithm is correct. We must justify the assumption that the ratio of u lies in the range $[\frac{1}{4}, 4]$. This calls for a special treatment of small size trees.

We now proceed with these 3 tasks.

¶**63. Task (a):**   This has two parts, depending on whether we do single or double rotations. For a single rotation, we first express the original ratios ρ, ρ' in terms of the $\texttt{esize}$'s a, b, c shown in Figure 30(i):

$$\rho = \frac{a+b}{c}, \qquad \rho' = \frac{a}{b}. \tag{30}$$

where a, b, c are the extended sizes of the subtrees in Figure 30(i). Expressing a and b in terms of c, we get

$$b = c\frac{\rho}{\widetilde{\rho'}}, \qquad a = c\frac{\rho\rho'}{\widetilde{\rho'}}$$

where (cf. (26)) $\widetilde{\rho'} = 1 + \rho' = (a+b)/b$. The transformed ratios σ, σ' can likewise be given in terms of a, b, c. Next, plugging in the ρ's we get:

$$\left.\begin{array}{lll} \sigma' & = \frac{b}{c} & = \frac{\rho}{\widetilde{\rho'}} \\[2mm] \sigma & = \frac{a}{b+c} & = \frac{\rho\rho'}{\rho+\widetilde{\rho'}} \end{array}\right\} \tag{31}$$

where $\widetilde{\rho''} := 1 + \rho'' = \frac{b+c}{c}$. The expressions (31) are what we seek. An Exercise shows that no exact relation like (31) are possible if we use $\texttt{size}$ instead of $\texttt{esize}$.

---

[19] The book of Mehlhorn [15, p. 193] declines a proof, because it would be "very tedious and inelegant".

We now turn to double rotation. First express the original ratios ρ, ρ', ρ'' in terms of the `esize`'s a, b, c, d shown in Figure 30(ii):

$$\rho = \frac{a+b+c}{d}, \qquad \rho' = \frac{a}{b+c}, \qquad \rho'' = \frac{b}{c}. \tag{32}$$

The transformed ratios σ, σ', σ' can now be calculated from the ρ, ρ', ρ'' as before:

$$\left.\begin{array}{rclcl}
\sigma' & = & \frac{a}{b} & = & \frac{\rho'\widetilde{\rho''}}{\rho''} \\[2mm]
\sigma'' & = & \frac{c}{d} & = & \frac{\rho}{\widetilde{\rho'}\widetilde{\rho''}} \\[2mm]
\sigma & = & \frac{a+b}{c+d} & = & \frac{\rho(\rho''+\rho'\widetilde{\rho''})}{\rho+\rho'\widetilde{\rho''}}.
\end{array}\right\} \tag{33}$$

where $\widetilde{\rho''} := 1 + \rho''$.

The expressions in (33) are what we seek.

¶64. Task (b): We now want to show that the Rebalance-Step results in a RB tree. The summary of our computation is shown in Table 1. The key observation is that the σ's are monotonic (increasing or decreasing) as a function of the ρ's. These σ's correspond to the rows of Table 1 (rows R1 and R2 for single-rotation, and rows D1, D2 and D3 for double-rotation). The "New Ratio" column is a summary of the computation of Task(a), giving the expressions of the σ's in terms of the ρ's.

	New Ratio	(ρ, ρ', ρ'')	$S_{\min}$	$\frac{S_{\min}}{-\frac{1}{3}}$	$S_{\max}$	$\frac{3-}{S_{\max}}$
R1	$\sigma = \frac{\rho\rho'}{\rho+\widetilde{\rho'}}$	$(+,+,0)$	$\frac{3\cdot(3/5)}{3+(1+(3/5))} = 9/23$	$\frac{4}{69}$	$\frac{4\cdot 3}{4+(1+3)} = 3/2$	$\frac{3}{2}$
R2	$\sigma' = \frac{\rho}{\rho'}$	$(+,-,0)$	$\frac{3}{1+3} = 3/4$	$\frac{5}{12}$	$\frac{4}{1+(3/5)} = 5/2$	$\frac{1}{2}$
D1	$\sigma = \frac{\rho(\rho''+\rho'\widetilde{\rho''})}{\rho+\rho'\widetilde{\rho''}}$	$(+,+,+)$	$\frac{3(\frac{1}{3}+\frac{1}{3}(1+1/3))}{1+3+(1/3)+\frac{1}{3}(1+1/3)} = \frac{21}{43}$	$\frac{22}{129}$	$\frac{4(3+(3/5)(1+3))}{1+4+3+(3/5)(1+3)} = \frac{27}{13}$	$\frac{12}{13}$
D2	$\sigma' = \frac{\rho'\widetilde{\rho''}}{\rho''}$	$(0,+,-)$	$\frac{(1/3)(1+3)}{3} = 4/9$	$\frac{1}{9}$	$\frac{(3/5)(1+(1/3))}{1/3} = 12/5$	$\frac{3}{5}$
D3	$\sigma'' = \frac{\rho}{\widetilde{\rho'}\widetilde{\rho''}}$	$(+,-,-)$	$\frac{3}{(3/5)(1+3)+3+1} = 15/32$	$\frac{17}{32}$	$\frac{4}{(1/3)(1+1/3)+1/3+1} = 9/4$	$\frac{3}{4}$

Table 1: Columns $S_{\min}$ and $S_{\max}$ show extremal values for $\sigma, \sigma', \sigma''$.

The next column headed by (ρ, ρ', ρ'') tells us how the σ's vary with respect to the ρ's. For instance, in row R2, we have $\sigma' = \frac{\rho}{\rho'}$. Elementary calculus tells that σ is increasing ("+") with ρ, decreasing with ρ' ("−") but independent ("0") of ρ''. Hence the corresponding entry in row R2 shows the triple $(+,-,0)$. To see this we partially differentiate σ with with respect to ρ and ρ':

$$\frac{\partial\sigma}{\partial\rho} = \frac{1}{\rho'} > 0, \qquad \frac{\partial\sigma}{\partial\rho'} = \frac{-\rho}{(\rho')^2} < 0.$$

Using the data from the (ρ, ρ', ρ'')-column, we can now compute the minimum $\sigma_{\min}$ and maximum $\sigma_{\max}$ achieved by σ as ρ, ρ', ρ'' vary within prescribed ranges of values: just plug in one of extremal values for the ρ's. Similarly, we can determine $\sigma'_{\min}, \sigma'_{\max}$, and $\sigma''_{\min}, \sigma''_{\max}$. For

1968 instance, in row R1, we know that $\rho \in [\frac{1}{4}, 4]$ and $\rho' \in [3/5, 5/3]$. To get $\sigma_{\min}$, we substitute
1969 $\rho = \frac{1}{4}$ and $\rho' = 3/5$:

1970 Table 1 records this key observation about how the σ's vary with the ρ's. The rows corre-
1971 spond to the different σ's. The columns headed by the ρ's display the signs '+', '$-$' or '0': a
1972 '+' sign indicates the σ expression in that row is increasing with the particular ρ. Likewise, a
1973 '$-$' sign indicates decreasing with, and a '0' sign indicates independence from the ρ.

Consider Table 1. The "New Ratio" column records the results from Task(a) which express
the σ's in terms ρ's. The next column records how the σ's vary with the ρ's. E.g., , row
R1 contains the triple $(+, +, 0)$ which tells us that σ is increasing with ρ and with ρ' but
independent of ρ''. To show that σ increases with ρ, we have to show that $\frac{\partial \sigma}{\partial \rho} > 0$. The sign
of these partial derivatives in Rows R1, R2 and D2 are easily seen by inspection. For Row D1,
let us see how σ vary with ρ' (equivalently, with $\widetilde{\rho'} = 1 + \rho'$):

$$\frac{\partial \sigma}{\partial \rho'} = \frac{\rho \widetilde{\rho''}(\rho + \widetilde{\rho'} \widetilde{\rho''}) - \widetilde{\rho''} \rho(\rho'' + \rho' \widetilde{\rho''})}{D} = \frac{\widetilde{\rho''} \rho(1 + \rho)}{D}$$

1974 where the denominator D is positive. Since the numerator $\widetilde{\rho''} \rho(1 + \rho)$ is positive, it proves
1975 that σ also increases with ρ'. Similarly, the numerator of $\frac{\partial \sigma}{\partial \rho''}$ is is positive. The remaining
1976 monotonicity properties in Rows D1 and D3 are left to the reader.

1977 With these monotonicity properties, it is now easy to plug in the extremal values of ρ, ρ', ρ''
1978 that will maximize or minimize the expression $S \in \{\sigma, \sigma', \sigma''\}$. By symmetry, let us assume that
1979 $\rho \in [3, 4]$. This implies $\rho' \geqslant 3/5$ in rows R1 and R2, and $\rho' < 3/5$ in rows D1, D2, D3. Let $S_{\min}$
1980 and $S_{\max}$ denote these maximas. The computed entries for $S_{\min}$ and $S_{\max}$ are shown in their
1981 respective columns in Table 1. For instance, row R1 shows that, in the case of single rotation,
1982 $\sigma_{\min} = 9/23$ and $\sigma_{\max} = 3/2$.

1983 Here is more information about calculating $S_{\min}, S_{\max}$. By assumption, $\rho \in [3, 4]$, so the
1984 the extreme values for ρ are 3 and 4. For single rotation, the extremal values for ρ' are 3/5
1985 and 3. For double rotation, the extremal values for ρ' are 1/3 and 3/5. The extremal values
1986 for ρ'' are 1/3 and 3, regardless of single or double rotation. From the table, we see that each
1987 $S \in \{\sigma, \sigma', \sigma''\}$ is monotonic increasing with ρ. Hence we always pick the extremal value $\rho = 3$
1988 to achieve $S_{\min}$, and pick $\rho = 4$ to achieve $S_{\max}$. The other choices can be similarly made. It
1989 should be emphasized that we do not claim that ρ, ρ', ρ'' are independent (clearly not). This
1990 means is that the values $S_{\min}$ and $S_{\max}$ are only lower and upper bounds on S; and these
1991 bounds may not be reachable.

In conclusion, the entries in Table 1 show that

$$1/3 < S_{\min} \leqslant \mathrm{ratio}(x) \leqslant S_{\max} < 3$$

1992 holds for each node $x \in \{u, v, w\}$ after a single or double rotation. Since the ratios of the other
1993 nodes are unchanged, this proves our key result:

1994 **Lemma 16** *Assume every node x in T_u is RB balanced if $x \neq u$. Moreover,* $\mathrm{ratio}(u) \leqslant [\frac{1}{4}, 4]$. *If*
1995 T_u *is not a RB tree, it would become a RB tree after performing a single* Rebalance-Step(u).

1996 **¶65. Task (c):** We wish to reduce the correctness of the Rebalance procedure to the
1997 correctness of Rebalance-Step procedure (Lemma 16). This amounts to ensuring that after

1998   insertion or deletion in an RB tree, each node u still satisfies

$$\frac{1}{4} \leqslant \text{ratio}(u) \leqslant 4. \tag{34}$$

1999   Let ρ (resp., σ) be the ratio of u after (resp., before) the operation. If $\frac{1}{3} < \rho < 3$, then
2000   there is nothing to show. Otherwise, we may assume that $\rho \geqslant 3$ (the case $\rho \leqslant \frac{1}{3}$ is treated
2001   symmetrically).

2002   Insertion is easier to deal with, so let us dispose of it in the following lemma:

2003   **Lemma 17 (Correctness of Insertion)** *Suppose u is RB balanced. After an insertion into*
2004 T_u, $\text{ratio}(u) \in [\frac{1}{3}, 3]$.

2005   In other words, if u is no longer RB balanced after insertion, this lemma says it is only mildly
2006   unbalanced: $\text{ratio}(u) \in \left\{ \frac{1}{3}, 3 \right\}$ and it certainly confirms (34).
2007   *Proof.* Let n be the size of T_u before the insertion. We may write $\sigma = a/b$ where $a + b = n + 1$
2008   with $\texttt{esize}(u.\texttt{left}) = a$ and $\texttt{esize}(u.\texttt{right}) = b$. Note that a/b may be a non-reduced fraction
2009   like $2/4$. Since $a/b = \sigma < 3$, we immediately have $a < 3b$.

2010   The correctness of insertion is very simple: After insertion, we have $\rho = (a + 1)/b \geqslant 3$ or
2011 $a + 1 \geqslant 3b$. Thus $a < 3b \leqslant a + 1$. We conclude $a + 1 = 3b$ or $\rho = 3b/b = 3$.   **Q.E.D.**

2012

2013   **¶* 66. Task (c) when deleting in small trees.**   Deletion is a bit trickier especially when
2014   the size n of the tree is small. Before proving the correctness of deletion, we look at maximum
2015   value of σ attainable after a deletion. This value can be expected to lie outside $(\frac{1}{3}, 3)$ but our
2016   above analysis further assumes that it lies inside $[\frac{1}{4}, 4]$. We must verify this. Let us define
2017 $\max \rho(n)$ and $\max \sigma(n)$ as follows:

2018   • $\max \rho(n)$ is the maximum ratio of the root of a tree of size n.

2019   • $\max \sigma(n)$ is the maximum ratio of the root of a tree of size $n - 1$, after a deletion from
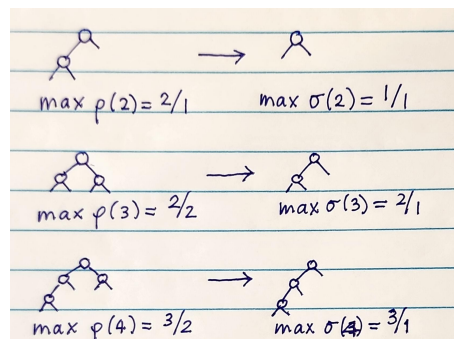2020   the tree of previous size n, but before rebalancing.

Figure 31: $\max \rho(n)$ and $\max \sigma(n)$ for $n = 2, 3, 4$

2021   See Figure 31 for the RBT trees of sizes $n = 2, 3, 4$ that achieve these maximum values.
2022   Observe that we write these ratios in the form a/b as *non-reduced fractions*. In other words,
2023 $4/2$ is not the same as $2/1$. A tabulation of these values for n up to 9 is given in Table 2:

$n = \texttt{size}(u)$	$\max \rho(n)$	$\max \sigma(n)$
2	2/1	1/1
3	2/2	2/1
4	3/2	3/1
5	4/2	4/1
6	5/2	5/1
7	5/3	5/2
8	6/3	6/2
9	7/3	7/2

Table 2: Maximum Ratios for small trees

Let us now see the recursive formulas for $\max \rho(n)$ and $\max \sigma(n)$ for $n \geqslant 2$:

$$n = a + b - 1, \ \max \rho(n) = a/b \quad \Longrightarrow \quad \begin{aligned} \max \rho(n+1) &= \begin{cases} (a+1)/b & \text{if } (a+1)/b < 3, \\ a/(b+1) & \text{else.} \end{cases} \\[2mm] \max \sigma(n) &= \begin{cases} 1/1 & \text{if } n = 2, \\ a/(b-1) & \text{else.} \end{cases} \end{aligned}$$

$$(35)$$

These formulas are easily verified. It may be checked that Table 2 follows these formulas.

Numerically, the $\max \sigma(n)$ is a rational number (i.e., 4/2 and 2/1 are the same value). Our goal is to show that (numerically) $\max \sigma(n) \leqslant 4$ for all n. We make a simple remark for use later: *The denominator b of the non-reduced $\max \sigma(n) = a/b$ is non-decreasing in n.* Since the denominator in $\max \sigma(7) = 5/3$ is 3, we conclude:

$$\text{For } n = \texttt{size}(u) \geqslant 7, \text{ the non-reduced denominator is at least 3.} \qquad (36)$$

The most interesting row in Table 2 is when $n = 6$: Just before deletion, the ratio is equal to $\max \sigma(6) = 5/2$; the corresponding binary tree is illustrated in Figure 26(a) After deletion, the maximum ratio is $\max \rho(6) = 5/1$, a clear violation of (34). Such a bad case is illustrated in Figure 26(c). For all other values of n, the maximum ratio after deletion will be less than 4, the case which had fully analyzed.

Lemma 18 *If $n \neq 6$, the $\max \rho(n) \leqslant 4$.*

Hence it remains to show $\rho \leqslant 4$.

CORRECTNESS OF DELETION: We are ready to analyze rebalancing after a deletion. As before, let $\sigma = a/b$ in non-reduced form. Then $\sigma < 3$ implies $a < 3b$. After deletion, $\rho = a/(b-1)$. Assuming $\rho \geqslant 3$ we have $a \geqslant 3b - 3$. Thus $3b - 3 \leqslant a < 3b$. Therefore $a = 3b - r$ for some $r = 1, 2$ or 3. The ratio $\rho = a/(b-1) = (3b-r)/(b-1)$ is maximized with $r = 1$ and so $\rho = (3b-1)/(b-1) = 3 + 2/(b-1)$. Therefore, if $b \geqslant 3$, we will have $\rho \leqslant 4$, proving (34). According to (36), $b \geqslant 3$ for all $n \geqslant 7$.

It remains to consider the case $n \leqslant 6$. By Table 2, the only situation where ρ can lie outside the range $[\frac{1}{4}, 4]$ is when the $n = 6$. This very special situation is illustrated in Figure 32. In the Rebalancing-Step above, this special situation is called $\texttt{SPECIAL}(u)$.

Suppose that after a deletion, we have a tree of size 5 with root u. All our analysis depends on the assumption that the ratio$(u) \in [\frac{1}{4}, 4]$. *The sole exception is this case, when* ratio$(u) = 5/1$
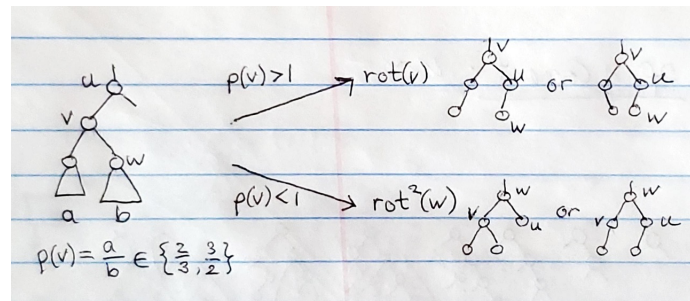

Figure 32: Special Rebalancing rule in deleting from a tree of size $n = 6$: after cutting x with ratio$(u) = 5/1$: if ratio$(v) > 1$, do `rotate`(v); else do `rotate`$^2(w)$.

(or $1/5$). We handle this directly. The fix is simple: in the Rebalance-Step, we compare the ratio of node v to 1 instead of $5/3$ (or $3/5$). If $v = u$.`left`, we consider two cases as illustrated in Figure 32.

FINAL DETAIL: in case ratio$(v) < 3/5$ we want to ensure that $w = v$.`right` is not equal to `nil` (so that we can carry out double rotation). Say ratio$(v) = c/d$ where $c = $ `esize`$(v$.`left`$)$ and $d = $ `esize`$(v$.`right`$)$. So $c/d < 3/5$ or $5c < 3d$. Since c and d are at least 1, this shows that $d \geqslant 2$ and so v.`right` $\neq$ `nil`.

¶67. A Hybrid Scheme – Logarithmic BST. Roura [18] introduced class of balanced trees that combines height balancing and weight balancing. Define the **logarithmic height** (log-height) of a node u as $\ell(u) := \lfloor \lg size(u) \rfloor$ if u is not nil; otherwise $\ell(\texttt{nil}) = -1$. A **logarithmic BST** is one in which any two siblings have logarithmic height that differs by at most one. Such a family represents an interesting hybrid between AVL trees and weight balanced trees. Like the latter, we must use $\lg n$ space per node to store the size of the subtree at the node. But the advantages is that we can now answer rank queries in $O(\log n)$ time. Moreover, it has the $O(1)$ amortized time for rotations in multidimensional range trees.

It is easy to see if $\ell(u) = h$ then $2^{h-1} < size(u) \leqslant 2^h$. Moreover, it is possible that the log-height of an node u is equal to the log-height of its parent v. For instance, if $size(u) = 8$ and $size(v) = 13$. However, this "anomaly" cannot be repeated: the log-height of the grandparent w of u must be greater than that of v. In proof, suppose $\ell(u) = \ell(v) = h$. Therefore the siblings of v and u have logarithmic height at least $h - 1$, and therefore sizes $\geqslant 1 + 2^{h-2}$. Therefore size of w is at least $2 + size(u) + 2(1 + 2^{h-2}) \geqslant 2 + (2^{h-1} + 1) + (2 + 2^{h-1}) = 5 + 2^h$. Then the logarithmic height of w is at least $h + 1$, as claimed. From this we conclude that the height of a Log BST of size n is at most

$$2\ell(n) = 2 \lfloor \lg n \rfloor.$$

To see this, take any path from the root to a leaf. The root has logarithmic $\ell(n)$. Every two steps results in a dropped log-height. So after $2\ell(n)$ steps, the log-height must drop to 0, i.e., we must move past a leaf.

To analyze the height of Log BST's we define two notions of min-size trees:

- $\mu(h)$ is the minimum size of a Log BST tree of height h. Trees with these sizes for small h are shown in Figure 33.

2069   • $\overline{\mu}(h)$ is the minimum size of a Log BST tree of log-height h. For instance, the first few

2070   values are given by $\overline{\mu}(-1, 0, 1, 2, 3, 4) = (0, 1, 2, 4, 8, 16)$.

2071   The formula for $\overline{\mu}(h)$ is quite easy: to achieve a height of h, the size s satisfies $h = \lfloor \lg s \rfloor$ or

2072 $s \geqslant 2^h$. Therefore $s = 2^h$ is the minimum size of a BST of log-height h. This proves

$$\overline{\mu}(h) = 2^h. \tag{37}$$
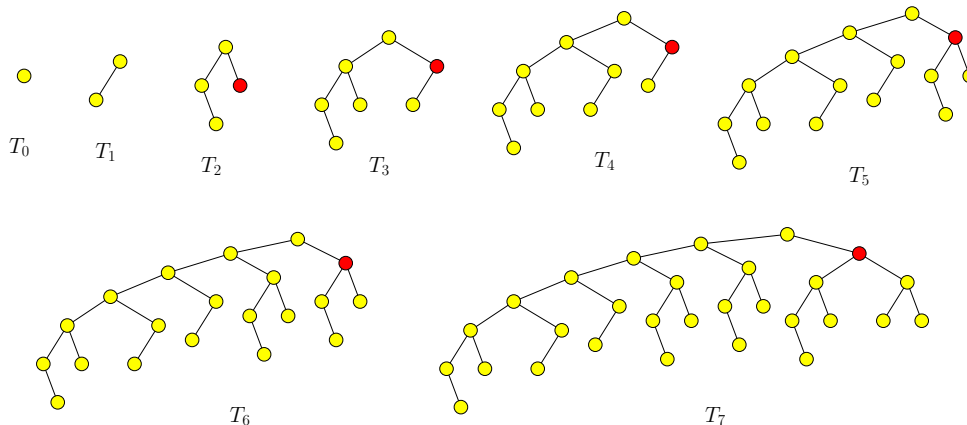
Figure 33: Min-size Logarithmic BST's

2073   I have changed the definition of $\ell(n)$, so the following must be re-derived...

2074   Next we derive a recurrence for $\mu(h)$. We must not be misled by the AVL recurrence, and

2075 simply claim that $\mu(h+1) = 1 + \mu(h) + \mu(h-1)$. The correct recurrence is

$$\mu(h+1) = 1 + \mu(h) + 2^{\lfloor \lg \mu(h) \rfloor - 1}. \tag{38}$$

2076 To see this, let T_{h+1} be a min-size tree of height $h+1$. Wlog, assume that its left subtree has

2077 height h and therefore size $\mu(h)$. But what is the size of its right subtree? Well, the log-height

2078 of the left subtree is $\ell(\mu(h)) = 1 + \lfloor \lg \mu(h) \rfloor$. Therefore the log-height of the left subtree must be

2079 $\lfloor \lg \mu(h) \rfloor$. By (37), the minimum size tree with this log-height $2^{\lfloor \lg \mu(h) \rfloor - 1}$. This is precisely the

2080 the quantity we incorporate into the recurrence (38). Using (38), we can easily verify Table 3:

h	$n = \mu(h)$	$\ell(n)$
0	1	1
1	2	2
2	4	3
3	7	3
4	10	4
5	15	4
6	20	5
7	20	5

Table 3: Minimum Size for small Logarithmic BST

2081   We make two observations about the entries in this table: the case $h = 4$ marks the first

2082 time when $\mu(h)$ is strictly less than $1 + \mu(h-1) + \mu(h-2)$. Therefore, the min-size AVL tree

of height 4 is 12, but the min-size Log BST tree of height 4 is 10. Therefore the trees T_h in Figure 33 is non-AVL for all $h \geqslant 4$. The case $h = 7$ marks the first time when $\mu(h+1)$ is not of the form $1 + \mu(h) + \mu(h-k)$ for some k. In Figure 33, we can see that right subtrees (rooted at the red nodes) all have size that are a power of 2, as required by the recurrence (38). In the Exercises, we develop further properties of Log BST's.

———————————————————————————————————————Exercises

Exercise 7.1: Consider the ratio balanced tree ($\rho = 1/3$) in Figure 34.
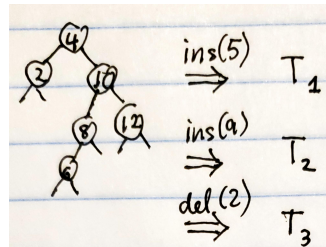
Figure 34: RBT with keys $2, 4, 6, 8, 10, 12$

      (i) Insert key 5 and draw the result T_1.
      (ii) Insert key 9 and draw the result T_2.
      (iii) Delete key 2 and draw the result T_3, using the general rebalancing rule.
      (iv) Delete key 2 and draw the result T_4, but using the special rule shown in Figure 32.

Note that these 4 parts are independent of each other!

$\diamondsuit$

Exercise 7.2: Hand-simulation of insertion and deletions for Ratio Balanced Trees (RBT or $RB[1/3]$). Start with the binary search tree in Fig.18 (p.48). You should verify that it is also an RBT. Consider the following sequence of RBT insertions/deletions:

$$(a) Del(14), \quad (b) Del(1), \quad (c) Ins(6.5), \quad (d) Ins(7.5), \quad (e) Del(13).$$

Draw the RBT after each operations. Show the intermediate tree after each rebalancing act. A rebalancing act can be a single- or a double-rotation.
NOTE: starting from the initial RBT, you have 10 distinct trees to draw! $\diamondsuit$

Exercise 7.3: In Figure 27, we gave some initial values of $\mu(h)$, defined as the minimum size of an RB[1/3] tree of height h. Please compute $\mu(8)$. $\diamondsuit$

Exercise 7.4: The ratio bound ρ in the ratio balanced class $RB[\rho]$ is normally restricted to $(0, \frac{1}{2})$. Can we extend to $\rho \in [\frac{1}{2}, 1)$?
(a) What is wrong with the class $RB[\frac{2}{3}]$?
(b) Which other values of $\rho \in [\frac{1}{2}, 1)$ pose problem similar to part(a)?
(c) Suppose we *really* want to allow values of ρ that lie in the range $(\frac{1}{2}, 1)$. How can we overcome the above objections? $\diamondsuit$

Exercise 7.5: The threshold constant 3/5 in Rebalance Step requires an exception when `size(u) = 5` (after a deletion). Find another simple rational constant that works without exception. ◇

Exercise 7.6: Prove that $\max \sigma(n) \leqslant 5$. ◇

Exercise 7.7: Let $\mu(h)$ denote the minimum size (not extended size) of a RB-tree of height h. Give an exact recursive formula for $\mu(h)$. Use your formula to determine the maximum height of a RB-tree of size 100. ◇

Exercise 7.8: In (27), we gave a recursive equation for the maximum height $H(n)$ of a ρ-ratio balanced tree of size n. Suppose that $H'(n)$ is the maximum height of a ρ-ratio balanced tree of $n-1$ (i.e., `esize = n`). Prove the recurrence:

$$H'(n) = 1 + H'(\lceil n/\tilde{\rho}\rceil + 1).$$

◇

Exercise 7.9: (Michael Mathieu, 2012) Instead of `esize(u)`, suppose we want to use `size(u)` to formulate the ratio inequalities. We seek the analogue of the relations (31) that express the σ's in terms of ρ's after a rotation. Show that there are no similar "simple" relations in this case. In particular, show that the σ's are not continuous functions of ρ, ρ'. ◇

Exercise 7.10: The text gives the correctness proof for the insertion/deletion algorithms for RB trees. Without modifying th algorithms or proofs, determine the range $[\rho_{\min}, \rho_{\max}]$ such that our algorithm maintains the balance of $RB[\rho]$ trees, for all $\rho \in [\rho_{\min}, \rho_{\max}]$. ◇

Exercise 7.11: Suppose T_u is an RB tree. Show that the ratio of the heights of $u.\texttt{left}$ and $u.\texttt{right}$ is bounded by a constant. ◇

Exercise 7.12: Can the inequality

$$1/3 < S_{\min} < S_{\max} < 3$$

be maintained in Table 1 if we only assume that $\rho \in [3,5]$? Where does it break down? ◇

Exercise 7.13: Suppose we want to maintain the class of $RB[5/12]$ trees. Develop the analogue of our algorithms for RB trees.
NOTE: $5/12 > 1/3$, so such trees are presumably more balanced than RB trees. ◇

Exercise 7.14: Let $\beta_0 = 1 - \frac{1}{\sqrt{2}}$ and $\rho_0 = \sqrt{2} - 1$.
(a) Show that $BB[\beta_0]$ is equivalent to $RB[\rho_0]$.
(b) Show that ρ_0 is an algebraic integer but β_0 is an algebraic non-integer. Note: An algebraic number is the root of a polynomial with integer coefficients. It is an algebraic integer if we can find a polynomial with leading coefficient of 1. Otherwise it is an algebraic non-integer.
(c) The inequality $\beta < \beta_0$ is critical in the analysis of Nievergelt and Reingold. Show how this inequality arise. ◇

Exercise 7.15: We sketched the ideas of a range queries in 2-dimensions.
    (a) Work out the details of such a data structure using ratio balanced trees. What is the space complexity $S(n)$ of your data structure for an input set of n points in $\mathbb{R}^2$?
    (b) Describe the algorithms for querying the data structure. What is the query complexity $Q(n)$?
    (c) Describe an algorithms for inserting into the data structure. What is the insertion complexity $I(n)$?
    (d) Describe an algorithms for deleting from the data structure. What is the insertion complexity $D(n)$? $\diamondsuit$

Exercise 7.16: Generalize the data structure in the previous question to range queries in $\mathbb{R}^d$ ($d \geqslant 3$). $\diamondsuit$

Exercise 7.17: Suppose the range query data structure in the previous questions is implemented using AVL trees instead of ratio balanced trees. For any n, show a sequence of n insert/delete requests into an initially empty tree that would cause $\Omega(n)$ rotations. $\diamondsuit$

Exercise 7.18: Roura actually defined the **logarithmic height** of a node u to be $\ell(u) := 1 + \lfloor \lg size(u) \rfloor$ if u is not nil; otherwise $\ell(\texttt{nil}) = 0$.
    (a) Re-derive our formula for $\mu(h)$ using this definition of $\ell(u)$.
    (b) One of the fundamental properties of AVL is this: Say a binary tree is **almost-AVL** if every node has an unbalance of at most 1, with one exceptional node that have an unbalance of 2. Then by doing a single or double rotation, we convert it an AVL tree or another almost-AVL tree where the depth of the exceptional node is reduced by 1. Show a similar property for **almost-Log BST**. $\diamondsuit$

─────────────────────────────────────── End Exercises

§8. Generalized B-Trees

We now consider a class of trees that is very important in practice, especially in database applications. These are external search trees (see ¶40). They are also no longer binary trees, but the degree of its nodes parametrized by two integers,

$$2 \leqslant a < b. \tag{39}$$

An (a, b)-**tree** is a rooted, ordered[20] tree with the following structural constraints:

- DEPTH PROPERTY: All leaves are at the same depth.

- DEGREE BOUND: Let m be the number of children of an internal node u. This is also known as the **degree** of u. In general, we have the bounds

$$a \leqslant m \leqslant b. \tag{40}$$

---

[20] "ordered" means that the children of each node has a specified total ordering. If a node in an ordered tree that has only one child, then that ordering is unique. Although binary trees are ordered trees, but they are more than just ordered because, when a node has only one child, we could specify that child to be a left- or a right-child. We might say binary trees have labeled children (labels are either LEFT or RIGHT).

2171   The root is an exception, with the bound $2 \leqslant m \leqslant b$.

2172   Perhaps the most important case is when $b = 2a - 1$: these will be called *B*-**trees**, and
2173   is from McCreight and Bayer [3]. Hence we regard (a, b)-trees as **generalized** *B*-**trees**.
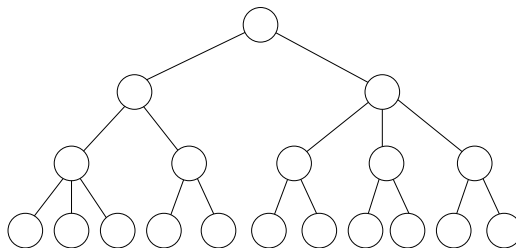

Figure 35: A $(2, 3)$-tree.

2174   Figure 35 illustrates an (a, b)-tree for $(a, b) = (2, 3)$. To see the intuition behind these
2175 conditions, compare with binary trees. In binary trees, the leaves do not have to be at the
2176 same depth. To re-introduce some flexibility into trees when all leaves have the same depth, we
2177 allow the degree of an internal node to vary over a range $[a, b]$. Moreover, in order to ensure
2178 logarithmic height, we require $a \geqslant 2$. This means that if there are n leaves, the height is at
2179 most $\log_a(n) + \mathcal{O}(1)$. *Therefore, (a, b)-trees constitute a balanced family of trees.* Notice that
2180 an (a, b)-tree is also (c, d)-tree if c, d satisfy

$$2 \leqslant c \leqslant a < b \leqslant d. \tag{41}$$

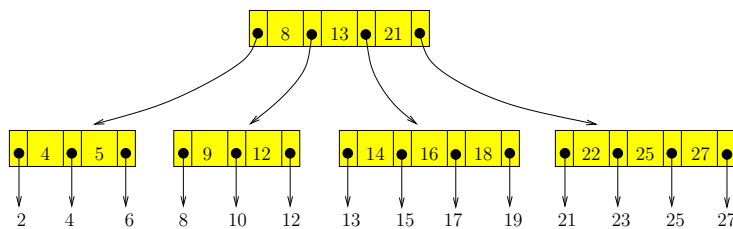2181 E.g., Figure 35 could have represented an $(2, 10)$-tree but not a $(3, 4)$-tree.

Figure 36: A (3,4)-search tree on 14 items

2182   **¶68.** (a, b)-**Tree as External Search Tree.** The definition of (a, b)-trees imposes purely
2183 structural requirements. To use such trees as search structures, we need to store keys and items
2184 in the tree nodes. These keys and items must be suitably organized. Before giving these details,
2185 we provide some intuition for such search trees by looking at the example in Figure 36. We
2186 assume that this is a $(3, 4)$-tree. Note that you cannot know the parameters $(a, b) = (3, 4)$ just
2187 by looking at the tree! These parameters are explicitly associated with the tree. For example,
2188 this *could* have been an (a, b)-tree for any $2 \leqslant a \leqslant 3$ and $b \geqslant 4$. It has 14 leaves, each storing
2189 a single item. The keys of these items are $2, 4, 6, 8, \ldots, 23, 25, 27$. Recall that an item is a
2190 (`key`, `data`) pair, but as usual, we do not display the associated data in items. *We continue*
2191 *our default assumption that items have unique keys.* But we see in Figure 36 that the key 13
2192 appears in the root as well as in a leaf. In other words, although keys are unique in the leaves,
2193 they might be duplicated (once) in the internal nodes (e.g., key 27). Conversely, the keys in
2194 the internal nodes (e.g. key 5) *need not* correspond to keys of items.

2195   We define an (a, b)-**search tree** to be an (a, b)-tree whose nodes are organized as an **exter-**
2196 **nal search tree**: as in external BST (see §5), it means that items are only stored in the leaves.

In recognition of this application of (a, b)-trees, the[21] leaves are called **external nodes** while non-leaves are called **internal nodes**. The different organization of the internal and external nodes are illustrated in Figure 37. The external node organization is controlled by an independent pair of parameters a', b' that satisfy the inequalities $1 \leqslant a' \leqslant b'$. They function just like a, b in controlling the minimum and maximum number of items in the leaves. We now specify the organization in the nodes:

$$1 \leqslant a' \leqslant m \leqslant b' \qquad\qquad\qquad 2 \leqslant a \leqslant m \leqslant b$$

(i) Leaf Node Organization   (ii) Internal Node Organization
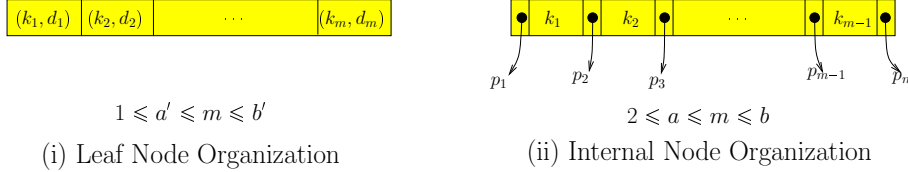
Figure 37: Organization of external and internal nodes in (a, b)-search trees

- EXTERNAL NODE: Each leaf stores a sequence of items, sorted by their keys. Hence we represent a leaf u with m items as the sequence,

$$u = (k_1, d_1; k_2, d_2; \cdots; k_m, d_m) \tag{42}$$

  where $k_1 < k_2 < \cdots < k_m$. Thus there are m items of the form (k_i, m_i). See Figure 37(i). In practice, d_i might only be a pointer to the actual location of the data. We must consider two cases.
  NON-ROOT LEAF: suppose leaf u is not the root. In this case, we require

$$a' \leqslant m \leqslant b'. \tag{43}$$

  ROOT-LEAF: suppose u is both a root and a leaf. There are no other nodes in this (a, b)-search tree. We now relax (43) to

$$0 \leqslant m \leqslant 2b' - 1. \tag{44}$$

  The reason for this condition will become clear when we discuss the insertion/deletion algorithms.

- INTERNAL NODE: Each internal node with m children stores an alternating sequence of keys and pointers (node references), in the form:

$$u = (p_1, k_1, p_2, k_2, p_3, \ldots, p_{m-1}, k_{m-1}, p_m) \tag{45}$$

  where p_i is a pointer to the i-th child of the current node. This is illustrated in Figure 37(ii). Note that the number of keys in this sequence is one less than the number m of children. The reader should compare (45) with (42). The keys are sorted so that

$$k_1 < k_2 < \cdots < k_{m-1}.$$

  For $i = 1, \ldots, m$, each key k in the i-th subtree of u satisfies

$$k_{i-1} \leqslant k < k_i, \tag{46}$$

  with the convention that $k_0 = -\infty$ and $k_m = +\infty$. Note that this is just a generalization of the binary search tree property in (5).

¶69. **Choice of the (a', b') parameters.** The (a, b) parameters are independent of the (a', b') parameters. Since our main focus is on the (a, b) parameters, it is convenient to choose default values for (a', b'). Here are two schemes for choosing (a', b'):

*So (a', b') is implicit. The **default** (i.e., canonical) choice is $a' = b' = 1$.*

- Simple Scheme: $a' = b' = 1$. This means each leaf stores exactly one item. This will be our **canonical choice** in examples (e.g., Figure 36) and in homework.

- Uniform Scheme: $(a', b') = (a, b)$. This is a more realistic choice in the context of B-tree applications where I/O from secondary memory is of major concern.

The simple scheme has a slight pedagogical advantage over the uniform scheme: (a, b)-search trees of any given shape need considerably fewer items under the simple scheme compared to the uniform scheme. E.g., the $(3, 4)$-search tree in Figure 36 has 14 items in the leaves. Under the uniform scheme, the same tree would need $\geq 14 \times 3 = 42$ items.
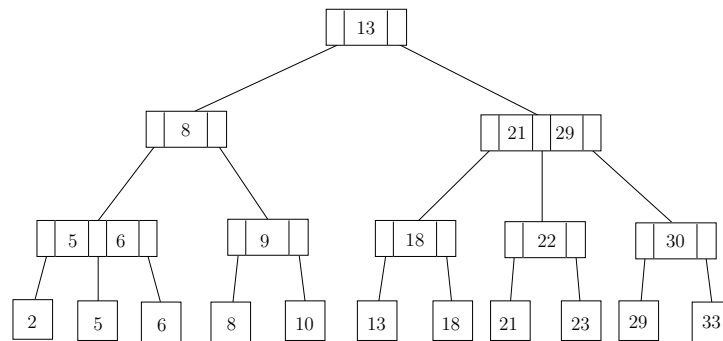

Figure 38: A $(2, 3)$-search tree.

Another (a, b)-search tree is shown in Figure 38, for the case $(a, b) = (2, 3)$. In contrast to Figure 36, here we draw it using a slightly more standard convention of representing the pointers as tree edges.

¶70. **How search keys are created/destroyed.** The keys in the (a, b)-search tree that are not attached to data items are called **search keys** in contrast to the **item keys** associated with items. Users only care about insertion and deletion of items, not in the management of search keys. So search keys must be automatically managed by our algorithms: *each insertion into a non-empty tree will create a new search key, as each deletion will destroy an existing search key.* Using our canonical scheme ($a' = b' = 1$), all search keys are in internal nodes of the tree. We may assume that each search key was originally a copy of an item key (but that item may have since been deleted). Here is the general situation. In Figure 39, we show (a) how the insertion of an item key K_0 creates a new search key (either K_0 or some K_2), and (b) the deletion of an item key K_2 destroys a search key (either K_1 or K_3).

This implies that if there are $n \geq 1$ items with keys $k_1 < k_2 < \cdots < k_n$ in the search tree, there are $n - 1 \geq 0$ search keys $k_1' < k_2' < \cdots < k_{n-1}'$ such that[22]

$$k_1 < k_1' \leq k_2 < k_2' \leq \cdots k_{n-1}' \leq k_n. \tag{47}$$

---

[21] Thus, if the root is the only node of the tree, it is both an internal and external note. We will discuss this special case below.

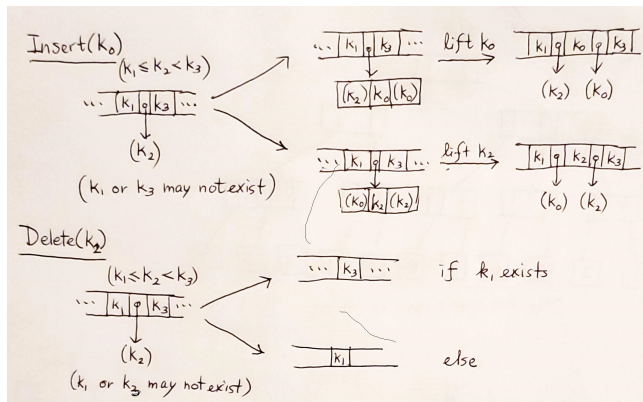[22] We have a similar property in (12) for external BST.

Figure 39: Creation and Destruction of search keys

It is possible to design our insertion/deletion algorithms so[23] that $k_i' = k_{i+1}$ for all $i \geqslant 1$

(Exercise). Thus k_1 (the key of the smallest item in the tree) is the only key that is not used as a search key.

¶71. Special Cases of (a, b) Parameters. The earliest and simplest (a, b)-search trees correspond to the case $(a, b) = (2, 3)$. These are called **2-3 trees** and were introduced by Hopcroft (1970). McCreight and Bayer [3] generalized them to **B-trees** which are (a, b)-search trees in which

$$b = 2a - 1 \tag{48}$$

and for any $a \geqslant 2$. Huddleston and Mehlhorn (1982) and Maier and Salveter (1981) observed that the slight variant of $b = 2a$ behaves better in an amortized sense (see [14, 9, Chap. III.5.3.1]. When $(a, b) = (2, 4)$, such trees have been studied by Bayer (1972) as **symmetric binary B-trees** and by Guibas and Sedgewick as **2-3-4 trees**. Another variant of 2-3-4 trees is **red-black trees**. The latter can be viewed as an efficient way to implement 2-3-4 trees, by embedding them in binary search trees. But this slight advantage comes at the price of complicated insertion and deletion algorithms. Thus it is clear that the concept of (a, b)-search trees serves to unify a variety of search trees. The *B*-tree relationship (48) is sharp in a certain sense, in the light of the "split-merge inequality" below.

¶72. Searching and its Complexity. The organization of an (a, b)-search tree supports an obvious lookup algorithm that is a generalization of binary search. Namely, to do `lookUp(key k)`, we begin with the root as the current node. In general, if u is the current node, we process it as follows, depending on whether it is a leaf or not:

- Base Case: suppose u is a leaf node given by (42). If k occurs in u as k_i (for some $i = 1, \ldots, m$), then we return the associated data d_i. Otherwise, we return the null value, signifying search failure.

---

[23]In Figure 39, when we delete the item key k_2, we removed the search key k_1 if it exists. We could also remove search key k_3. But if we want to ensure $k_i' = k_{i+1}$, then we must remove k_1 because $k_1 = k_2$ in this case.

- Inductive Case: suppose u is an internal node given by (45). Then we find the p_i such that $k_{i-1} \leqslant k < k_i$ (with $k_0 = -\infty, k_m = \infty$). Set p_i as the new current node, and continue by processing the new current node.

The running time of the `lookUp` algorithm is $O(hb)$ where h is the height of the (a, b)-tree, and we spend $O(b)$ time at each node.

Analogous to AVL trees, we define $M(h)$ and $\mu(h)$ as the maximum and minimum (resp.) number of leaves in an (a, b)-tree of height h. Unlike AVL trees, the current definition focuses on the number of leaves rather than the size of the tree. That is because items are stored only in leaves of (a, b)-trees. Since $M(h)$ is attained if every internal node has b children, we obtain

$$M(h) = b^h. \tag{49}$$

Likewise, $\mu(h)$ is attained if every internal node (except the root) has a children. Thus

$$\mu(h) = \begin{cases} 1 & h = 0, \\ 2a^{h-1}. & h \geqslant 1. \end{cases} \tag{50}$$

It follows that if an (a, b)-tree with n leaves has height $h \geqslant 1$, then $2a^{h-1} \leqslant n \leqslant b^h$. Taking logs, we get

$$\log_b n \leqslant h \leqslant \log_a((an)/2). \tag{51}$$

See Exercise where we bound height of an (a, b)-tree based on the number of *items*; here one take into account the parameters (a', b') as well.

It is clear that b, b' determine the lower bound on h while a, a' determine the upper bound on h. Our design goal is to maximize a, b, a', b' for speed, and to minimize b/a for space efficiency (see below). Typically b/a is bounded by a small constant close to 2, as in B-trees.

¶73. The Split and Merge Inequalities. To support efficient insertion and deletion algorithms, the parameters a, b must satisfy an additional inequality in addition to (39). This inequality, which we now derive, comes from two low-level operations on (a, b)-search tree. They are the **split** and **merge** operations which are subroutines in the insertion and deletion algorithms. There is actually a family of such inequalities, but we first derive the simplest one ("the standard inequality").

The concept of **immediate siblings** is necessary for the following discussions. The children of any node have a natural total order, say $u_1, u_2, \ldots, u_m$ where m is the degree of u and the keys stored in the subtree rooted at u_i are less than the keys in the subtree rooted at u_{i+1} $(i = 1, \ldots, m - 1)$. Two siblings u_i and u_j are called **immediate siblings** of each other iff $|i - j| = 1$. So every non-root node u has at least one immediate sibling and at most two immediate siblings. The immediate siblings may be called **left sibling** or **right sibling**.

During insertion, a node with b children may acquire a new child. We say the resulting node is **overfull** because it now has $b + 1$ children. An obvious response is to **split** it into two nodes with $\lfloor (b + 1)/2 \rfloor$ and $\lceil (b + 1)/2 \rceil$ children, respectively. In order that the result is an (a, b)-tree, we require the following "split inequality":

$$a \leqslant \left\lfloor \frac{b + 1}{2} \right\rfloor. \tag{52}$$

Similarly, during deletion, we may remove node with only a children may lose a child. We say the resulting node with $a - 1$ children is **underfull**. We first consider borrowing a child from

an immediate sibling, provided the sibling has more than a children. If this proves impossible, we will **merge** the underfull node with its sibling of degree a. The merged node has degree $2a - 1$. To satisfy the degree bound of (a, b)-trees, we require $2a - 1 \leqslant b$. Thus we require the following "merge inequality":

$$a \leqslant \frac{b+1}{2}. \tag{53}$$

Clearly (52) implies (53). However, since a and b are integers, the reverse implication also holds! Thus (52) and (53) are equivalent. (53) is also equivalent to

$$2a \leqslant b + 1. \tag{54}$$

The smallest choices of parameters a, b subject to the split-merge inequality and also (39) is $(a, b) = (2, 3)$. The case of equality in (52) and (53) gives us $b = 2a - 1$; recall that $(a, 2a - 1)$-trees are known as B-**trees**.

On Mixed Integer Inequalities. The above argument uses the following little lemma: let x, y be real numbers such that

$$x \geqslant y. \tag{55}$$

Suppose you know that x is an integer. Then the (55) is equivalent to the apparently stronger inequality $x \geqslant \lceil y \rceil$. Similarly, if y is an integer, (55) is equivalent of $\lfloor x \rfloor \geqslant y$. Combining them:

Lemma 19 *Suppose at least one of x or y is an integer.*

(a)
$$x \geqslant y \quad \Longleftrightarrow \quad \lfloor x \rfloor \geqslant \lceil y \rceil. \tag{56}$$

(b)
$$x > y \quad \Longleftrightarrow \quad \lceil x \rceil \geqslant 1 + \lfloor y \rfloor. \tag{57}$$

Proof.

(a) Suppose x is an integer. Then

$$x \geqslant y \Longleftrightarrow x \geqslant \lceil y \rceil \Longleftrightarrow \lfloor x \rfloor \geqslant \lceil y. \rceil$$

A similar argument applies when y is an integer.

(b) If x is an integer, then

$$x > y \quad \Longleftrightarrow \quad x > \lfloor y \rfloor \quad \Longleftrightarrow \quad x \geqslant 1 + \lfloor y \rfloor. \quad \Longleftrightarrow \quad \lceil x \rceil \geqslant 1 + \lfloor y \rfloor.$$

A similar argument applies when y is an integer.

Q.E.D.

Neat trick!

¶74. **How to Split (or donate) and Merge (or borrow).** We have described conceptually what happens when we split (as a result of insertion) or merge (as a result of deletion). We now provide additional details of these two operations. Suppose we have a node N that has to be split, or two nodes N and M that have to be merged. We assume the *general case* where N and M are non-roots and non-leaves. The special case of the root or leaves is discussed below.

We need to **split** N because it is overfull, i.e., has $b+1$ children. This is illustrated in Figure 40. We split N into two new nodes N_1, N_2, one node with $\lfloor (b+1)/2 \rfloor$ pointers and the other with $\lceil (b+1)/2 \rceil$ pointers. In Figure 40, these two groups of pointers in N are separated by the search key 3. The parent P of N will replace its pointer to N with two pointers: one pointing to N_1 and the other pointing to N_2. Moreover, the search key 3 in P separates the pointers to N_1 and N_2. In general, when we split the $b+1$ pointers in N into $\lfloor (b+1)/2 \rfloor$ and $\lceil (b+1)/2 \rceil$, we will always have an extra search key (like 3) that can be sandwiched between the pointers to N_1 and N_2.
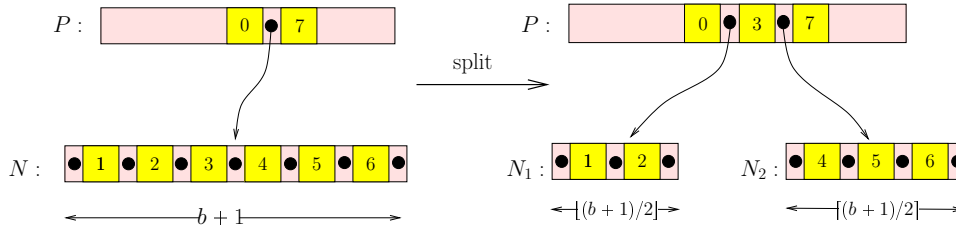
Figure 40: Splitting: N splits into N_1, N_2. Case $(a, b) = (3, 6)$ is illustrated.

do not borrow from cousin (near or distant ones). Why?

Next, suppose N is an underfull node with $a-1$ children. First we try to **borrow** from an (left or right) immediate sibling. This borrowing succeeds if any immediate sibling has degree $> a$ as shown in Figure 41. Thus, if borrowing is successful, the rebalancing process can stop. To borrow, we look to an immediate sibling (left or right), Suppose N borrows a child from its sibling M. After borrowing, N will have a children, but it will need a search key K to separate the newly borrowed child from its adjacent pointer. Key K can be taken from the parent of N (in Figure 41, we have $K = 2$). Since M lost a child, we now have an extra search key K' to spare — we send K' to the parent of N to replace K. In Figure 41, $K' = 3$.
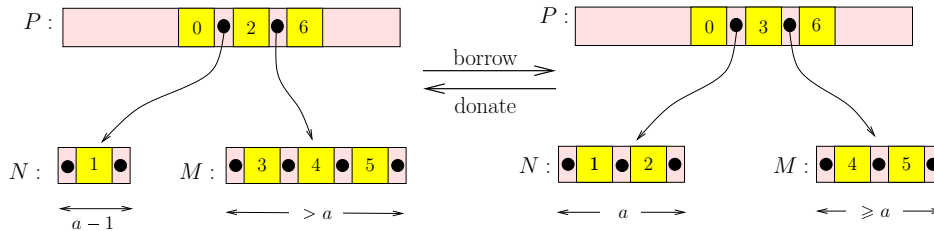
Figure 41: Borrowing: N borrows from M. Case of $(a, b) = (3, 6)$.

If N is unable to borrow, we must **merge**: if M is an immediate sibling of N, and since N cannot borrow from M, we know that M has degree a. Thus we can merge M and N into a new node N' with degree $2a - 1$. Note that N' needs a search key K'' to separate the pointers of N from those of M. Key K'' can be taken from the parent of N because now, the parent has one less child. In Figure 42, $K'' = 2$.

Degenerate case of merging: when $a = 2$, an underfull node has degree one. In Figure 42, it means that the node N has no keys before the merge. But the process described is still valid.

¶75. Policies for Split/Donate and Merge/Borrow. So far, we have basically described the "standard" split/merge procedures, and they immediately lead to the "standard" insert/delete algorithms for (a, b)-search trees. But these merge/split procedures remain non-deterministic. Our goal is to enunciate some policies and conventions to ensure canonical
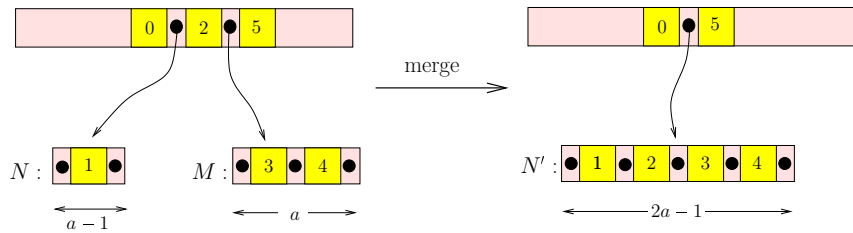
Figure 42: Merging: N and M merges into N'. Case of $(a, b) = (3, 6)$.

2344   answers.

2345   The careful reader will notice an asymmetry between splitting and merging in the: splitting
2346   is *unconditional* (we can always split) but merging is *conditional* (we may be unable to merge
2347   and have to borrow instead). Towards a more symmetrical view, we introduce the concept of
2348   **donation**, viewed as the inverse of borrowing. Donation appears to be new in the literature,
2349   but we will see its benefits below. Indeed, if we simply reverse the direction of transformation in
2350   Figure 41, we have the **donation** operation: node N donates to node M iff M borrows from N.
2351   Instead of an unconditional split of an *overfull* node, it can be pre-empted by making donation
2352   to an immediate sibling. Thus we can view these 4 operations as two super operations: the
2353   **split/donate operation** and **merge/borrow operation**. These super operations have can
2354   be greatly generalized to (a, b, c)-trees later.

2355   However our immediate concern is to specify a canonical behavior for insertion and deletion
2356   for (a, b)-search trees. We may characterize the different behaviors as "policies":

2357   • **Eager split policy** for overfull node: *when a node is overfull, we immediately split it.*
2358   By definition, the **Standard Insertion** algorithm is an implementation of this policy.
2359   We like this policy as it is unconditional.

2360   • **Marriage proposal policy** for an underfull node: the underfull node u will make a
2361   (marriage) proposal to an immediate sibling v. The proposal is for marriage (i.e., merger
2362   of u and v), but if marriage proves impossible (i.e., degree of v is too large), then u
2363   will borrow from v. So the proposal is always satisfied, even though it is not known in
2364   advance whether it will be a merge or a borrow event. To make the choice of v unique,
2365   we assume that v is the **right** immediate sibling iff u has a right sibling. By definition,   Do the right thing!
2366   the **Standard Deletion** implements this right proposal policy. This policy is attractive
2367   because as we will see in the I/O aware algorithm below, this only requires bringing one
2368   I/O block into the main memory.

2369   Let us briefly note alternatives to the above two policies. For an overfull node, we can use
2370   **Reluctant split policy**: we try to donate to an immediate sibling if possible; failing this,
2371   we split. For an underfull node, we can use the **reluctant merge policy**: we try to borrow
2372   from an immediate sibling if possible, but failing this, we will merge with one of them. A
2373   possible motivation for these "reluctant policies" is that splitting and merging changes the tree
2374   structure. Unfortunately, both policies may need to bring into two I/O blocks (i.e., the left and
2375   right immediate siblings) into main memory. A compromise is the the **half-reluctant** versions
2376   of these policies where we test at most one sibling into main memory.

¶76. **Special Treatment of Root.** Now we must take care of these split, borrow, merge, and donate operations for the special case of roots and leaves. Consider splitting a root, and merges of children of the root:

(i) Normally, when we split a node u, its parent gets one extra child. But when u is the root, we create a new root with two children. This explains the exception we allow for roots to have between 2 and b children. (Alternatively, instead of letting the degree of the root drop to 2, we could keep the lower bound at a but let the upper bound increase to $ab - 1$.)

(ii) Normally, when we merge two immediate siblings u and v, the parent loses a child. But when the parent is the root, the root may now have only one child. In this case, we delete the root and its sole child is now the root.

So this is how (a, b)-trees change height!

Notice that case (i) is the only means for increasing the height of the (a, b)-tree; likewise case (ii) is the only means for decreasing height.

¶77. **Special Treatment of Leaves.** Now consider leaves that are constrained to have between a' to b' items. In order for the splits and merges of leaves to proceed as above, we need the analogue of the split-merge inequality,

$$a' \leqslant \frac{b' + 1}{2}. \tag{58}$$

Recall that the two canonical choices for (a', b') are $(a', b') = (a, b)$ or $(a', b') = (1, 1)$. The inequality (58) holds in both choices.

Suppose u splits into two consecutive leaves, say u and u'. In this case, the parent needs a key to separate the pointers to u and u'. This key can be taken to be the minimum key in u'. Conversely, if two consecutive leaves u, u' are merged, the key in the parent that separates them is simply discarded.

So this is where internal keys are generated or eliminated!

Note that internal keys are generated or eliminated only in the preceding scenarios. Otherwise, when we split or merge internal nodes, existing internal keys are just shuffled around and no key is generated or eliminated.

However, a rather unique case arise when the leaf is also the root! Call this the **root-leaf case**. We cannot treat it like an ordinary leaf that has between a' to b' items. We allow a root-leaf is have between 0 and $2b' - 1$ items. The lower bound of 0 is clearly necessary to allow (a, b)-trees with no items. The upper bound is predicated on the fact that when the root-leaf is overfull with $2b'$ items, it can create a new root with two leaves, each with a' items!

¶78. **I/O Aware Description of Insertion/Deletion.** We use an "I/O aware" description of the insertion/deletion process. First, we must recall that the nodes of (a, b)-search trees normally reside in secondary storage (say, a disk), and they must be explicitly swapped in or out of main memory. These are called **I/O operations**. Once the nodes is in main memory, we can modify and modify these nodes, and eventually write them back into secondary memory. We assume that I/O operations are much more expensive than any operation in main memory. Typically, I/O operations are two to three orders of magnitude slower. Therefore in our complexity analysis, we only count I/O operations. For that matter, the earlier LookUp algorithm should also be viewed in this I/O aware manner: as we descend the search tree, we are really bringing into main memory each new node to examine. In the case of Lookup, there is no need

be I/O aware!

to write the node back into disk. This raises another point – we should distinguish between pure-reading or reading-cum-writing operations when discussing I/O. We simply count pure reading as one I/O, and reading-cum-writing as two I/O's. In the following, we will describe `lookUp`, `insert` and `delete` as **secondary memory algorithms** in which data movement between disk and main memory are explicitly invoked.

The following is a unified algorithm that encompasses both insertion and deletion. The algorithm is relatively simple, comprising a single while-loop:

INSERT/DELETE Algorithm
▷ *UPDATE PHASE*
To insert an item (k, d) or delete a key k, first do a lookUp on k.
LookUp will eventually bring a leaf node u into the main memory; u is where the the insertion or deletion takes place.
Call u the **current node**.
▷ *REBALANCE PHASE*
While u is overfull or underfull, do:
    1.   If u is root, handle as a special case and terminate.
    2.   Bring the parent p of u into main memory.
    3.   Bring needed sibling(s) u_j's $(j = 1, 2, \ldots)$ of u into main memory.
    4.   Do the desired transformations (split, merge, borrow, donate)
          on u, u_j's and p.
          ◁ *In main memory, nodes may temporarily have $> b$*
          ◁ *or $< a$ children. Nodes may be created or deleted*
    5.   Write back into disk any modified node other than p.
    6.   Make p the new current node (renamed as u) before repeating this loop.
Write the current node u to secondary memory and terminate.

REMARKS:

- In Step 5, we do not write a node u back into disk unless it has been modified. In particular, when we split or merge, then modified children must be written back to disk (the parent will be written out too, but in the next iteration).

- We were deliberately vague in Step 3 of the above algorithm for two reasons: first, the vague description can cover generalized split/merge operations when we introduce (a, b, c)-trees shortly. But even in standard (a, b)-trees, this allows for different policies to be implemented.

- For (a, b)-trees, we could ensure that the main memory holds at most 3 nodes at any moment: the current node u, its parent p and at most one sibling.

¶79. I/O Analysis of Standard Algorithms. Let us now count the number of I/O operations for our standard algorithms. Clearly, a LookUp uses exactly as many reads as the height of the tree.

Next, consider Standard Insertion: there is the initial reading and final writing of the current node. In each iteration of the while loop, the current node u is overfull, and we need to bring

---

2442 in a parent, split u into u and u', and write out u and u'. Thus we have 3 I/O operations per
2443 iteration. Thus the overall I/O cost is $2 + 3I$ where I is the number of iterations (of course I
2444 is bounded by the height). There is one other possibility: the last iteration might be the base
2445 case where u is a root. In this case, we split u and write them out as two nodes. So this case
2446 needs only 2 I/O's, and our bound of $2 + 3I$ is still valid.

2447   We do a similar analysis of the enhanced insertion algorithm: each failed donation costs one
2448 I/O (to read in a sibling), but a successful donation costs[24] two I/O's (reading in the sibling
2449 and writing it out again). Of course, a successful donation happens only once, and costs one
2450 more I/O than the standard algorithm. So the total number of I/O's is $(2+3I)+F+S$ where F
2451 is the number of failures and $S = 1$ or 0 (depending on whether there is a successful donation).
2452 Since $S + F \leqslant 2I$, we can bound the number of I/O's by $2 + 5I$.

2453   Consider Standard Deletion: there is an initial reading and final writing of the current node.
2454 In each iteration of the while loop, the current node u is underfull, and we need to bring in a
2455 parent and a sibling u', and then writing out the merger of u and u'. Again, this is 3 I/O's per
2456 iteration. A possibly exception is in the last iteration where we achieve a borrowing instead of
2457 merging. In this case, we must write out both u and u', thus requiring four I/O's. Thus the
2458 overall I/O cost is bounded by $3 + 3D$ where D is the number of iterations.

2459   **¶80. Achieving $2/3$ Space Utility Ratio.**   A node with m children is said to be **full** when
2460 $m = b$; for in general, a node with m children is said to be (m/b)**-full**. Hence, nodes can be
2461 as small as (a/b)-full. Call the ratio $a : b$ the **space utilization ratio**. This ratio is < 1 and
2462 we like it to be as close to 1 as possible. The standard inequality (53) on (a, b)-trees implies
2463 that the space utilization in such trees can never[25] be better than $\lfloor (b + 1)/2 \rfloor /b$, and this can
2464 be achieved by B-trees. This ratio is as large as $2 : 3$ (achieved when $b = 3$), but as $b \to \infty$, it
2465 is asymptotically equal to $1 : 2$.

2466   Of course, in practice, the potential to waste about half of the available space is not good.
2467 Hence we next address the issue of achieving ratios that are arbitrarily close to 1, for any choice
2468 of a, b. First, we show how to achieve $2/3$ asymptotically.

     Consider the following modified insertion: to rebalance an node u with $b + 1$ children, we
first look at a sibling v to see if we can **donate** a child to the sibling. If v is not full, we may
donate to v. Otherwise, v is full and we can take the $2b+1$ children in u and v, and divide them
into 3 groups as evenly as possible. So each group has between $\lfloor (2b + 1)/3 \rfloor$ and $\lceil (2b + 1)/3 \rceil$
keys. More precisely, the size of the three groups are

$$\lfloor (2b + 1)/3 \rfloor , \quad \lfloor (2b + 1)/3 \rceil, \quad \lceil (2b + 1)/3 \rceil$$

for any integer n:

2469 where "$\lfloor (2b + 1)/3 \rceil$" denotes **rounding** to the nearest integer. For instance, $\lfloor 4/3 \rfloor + \lfloor 4/3 \rceil +$ $\lfloor n/3 \rfloor + \lfloor n/3 \rceil +$
2470 $\lceil 4/3 \rceil = 1 + 1 + 2 = 4$ and $\lfloor 5/3 \rfloor + \lfloor 5/3 \rceil + \lceil 5/3 \rceil = 1 + 2 + 2 = 5$. Nodes u and v will (respectively) $\lceil n/3 \rceil = n$
2471 have one of these groups as their children, but the third group will be children of a new node.
2472 See Figure 43.

2473   We want these groups to have between a and b children. The largest groups has size
2474 $\lceil (2b + 1)/3 \rceil$ and this $\leqslant b$, automatically. For the smallest group to have size at least a, we

---

[24]The cost of reading and writing u is separately accounted for, as part of the standard algorithm accounting.
[25]The ratio $a : b$ is only an approximate measure of space utility for various reasons. First of all, it is an
asymptotic limit as b grows. Furthermore, the relative sizes for keys and pointers also affect the space utilization.
The ratio $a : b$ is a reasonable estimate only in case the keys and pointers have about the same size.
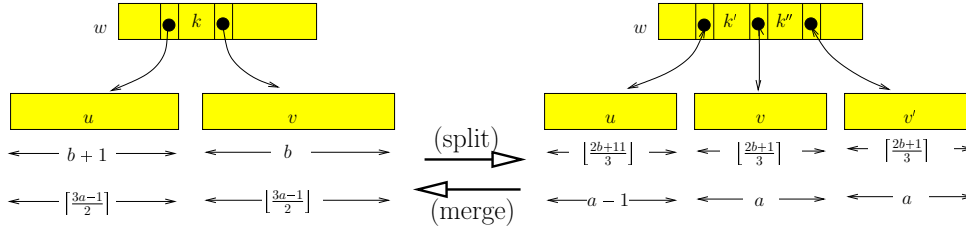
Figure 43: Generalized (2-to-3) split and (3-to-2) merge

require

$$a \leqslant \left\lfloor \frac{2b+1}{3} \right\rfloor. \tag{59}$$

This process of merging two nodes and splitting into three nodes is called **generalized split** because it involves merging as well as splitting. Let w be the parent of u and v. Thus, w will have an extra child v' after the generalized split. If w is now overfull, we have to repeat this process at w.

   Next consider a modified deletion: to rebalance an underfull node u with $a-1$ nodes, we again look at an adjacent sibling v to **borrow** a child. If v has a children, then we look at another sibling v' to borrow. If both attempts at borrowing fails, we merge the $3a-1$ children[26] in the nodes u, v, v' and then split the result into two groups, as evenly as possible. Again, this is a **generalized merge** that involves a split as well. The sizes of the two groups are $\lfloor (3a-1)/2 \rfloor$ and $\lceil (3a-1)/2 \rceil$ children, respectively. Assuming

$$a \geqslant 3, \tag{60}$$

v and v' exist (unless u is a child of the root, which is handled separately). For lower bound on degree, we require $\lfloor (3a-1)/2 \rfloor \geqslant a$, which is equivalent to $(3a-1)/2 \geqslant a$ (by integrality of a), which clearly holds. For upper bound on degree, we require

$$\left\lceil \frac{3a-1}{2} \right\rceil \leqslant b \tag{61}$$

Because of integrality constraints, the floor and ceiling symbols could be removed in both (59) and (61). Thus both inequalities are be equivalent to

$$a \leqslant \frac{2b+1}{3}. \tag{62}$$

As in the standard (a, b)-trees, we need to make exceptions for the root. Here, the number m of children of the root satisfies the bound $2 \leqslant m \leqslant b$. So during deletion, the second sibling v' may not exist if u is a child of the root. In this case, we can simply merge the level 1 nodes, u and v. This merger is now the root, and it has $2a-1$ children. This suggests that we allow the root to have between a and $\max\{2a-1, b\}$ children.

¶81. Example of Generalized Merge.   Consider deleting the item (represented by its key) 4 from the tree in Figure 36. The is illustrated in Figure 44. After deleting 4, the current node u is underfull. We try to borrow from the right sibling, but failed. But the right sibling of the right sibling could give up one child.

---

[26]We assume $a \geqslant 3$. Normally, we expect v, v' to be immediate siblings of u. But if u has only one sibling, then v will have two siblings, one of which is u. We pick v'' to be the other sibling of v''.
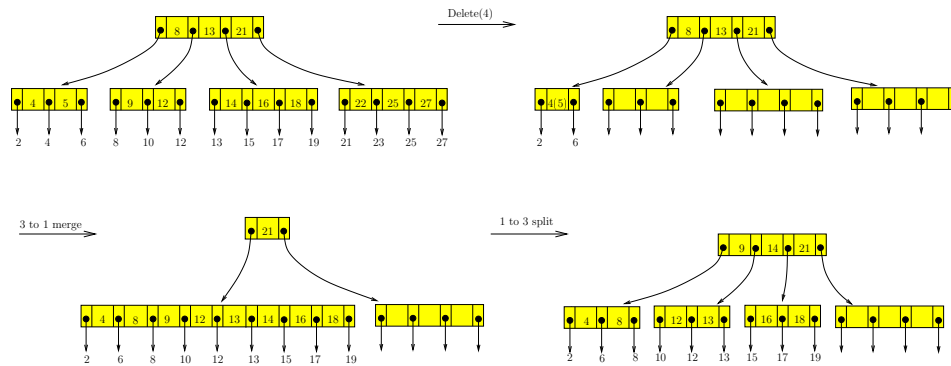
Figure 44: Deleting 4 from $(3, 4, 2)$-search tree.

One way to break down this process is to imagine that we merge u with the 2 siblings to its right (a 3-to-1 merge) to create supernode. This requires bringing some keys (8 and 13) from the parent of u into the supernode. The supernode has 9 children, which we can split evenly into 3 nodes (a 1-3 split). These nodes are inserted back into the parent. Note that keys 9 and 14 are pushed into the parent. An implementation may combine these merge-then-split steps into a single more efficient process.

If we view b as a hard constraint on the maximum number of children, then the only way to allow the root to have $\max\{2a - 1, b\}$ children is to insist that $2a - 1 \leqslant b$. Of course, this constraint is just the standard split-merge inequality (53); so we are back to square one. This says we must treat the root as an exception to the upper bound of b. Indeed, one can make a strong case for treating the root differently:

(1) It is desirable to keep the root resident in main memory at all times, unlike the other nodes.

(2) Allowing the root to be larger than b can speed up the general search.

The smallest example of a $(2/3)$-full tree is where $(a, b) = (3, 4)$. We have already seen a $(3, 4)$-tree in Figure 36. The nodes of such trees are actually $3/4$-full, not $2/3$-full. But for large b, the "$2/3$" estimate is more reasonable.

¶82. **Exogenous and Endogenous Search Structures.** The keys in the internal nodes of (a, b)-search trees are used purely for searching: they are not associated with any data. In our description of standard BSTs (or their balanced versions such as AVL trees), we never explicitly discuss the data that are associated with keys. So how[27] did we know that these data structures are endogenous? We observe that, in looking up a key k in a BST, if k is found in an internal node u, we stop the search and return u. This means we have found the item with key k. The item is not necessarily stored in u, because we could store a pointer to the real location of the item. For (a, b)-search tree, we cannot stop at any internal node, but must proceed until we reach a leaf.

There are two consequences of this dual role of keys in (a, b)-search trees. First, even if items have unique keys (our default assumption), the keys in the external search structure could have duplicate keys. In the case of (a, b)-search trees, we could have as many as one copy of the key per level. Second, the keys in the internal nodes *need not correspond to the keys of items in the leaves.* E.g., in Figure 38 the key 13 appears in an item as well as in an internal node, and the key 9 in an internal node does not correspond to any item.

Q: Is this school bus going left or right?
Adult: I don't know.
Child: Going Left.
Adult: Huhh???

Can't we require the keys in internal nodes to correspond to keys of stored items?

[27] The child knows that if the school bus were going right then you could see the entrance door.

¶**83. Database Application.** One reason for treating (a, b)-trees as external search structures comes from its applications in databases. In database terminology, (a, b)-search tree constitute an **index** over the set of items in its leaves. A given set of items can have more than one index built over it. If that is the case, at most one of the index can actually store the original data in the leaves. All the other indices must be contented to point to the original data, i.e., the d_i in (42) associated with key k_i is not the data itself, but a reference/pointer to the data stored elsewhere. Imagine a employee database where items are employee records. We may wish to create one index based on social security numbers, and another index based on last names, and yet another based on address. We chose these values (social security number, last name, address) for indexing because typical searches in the data base is presumably based on these values. It seems to make less sense to build an index based on age or salary, for instance.

¶**84. Disk I/O Considerations: maximizing the parameter b.** There is another reason for preferring external search structures. In databases, the number of items is very large and these are stored in disk memory. If there are n items, then we need at least n/b' internal nodes. This many internal nodes implies that the nodes of the (a, b)-trees is also stored in disk memory. Therefore, while searching through the (a, b)-tree, each node we visit must be brought into the main memory from disk. The I/O speed for transferring data between main memory and disk is relatively slow, compared to CPU speeds. As a rule of thumb, consider each I/O operation is three orders of magnitude slower than CPU operations. Moreover, disk transfer at the lowest level of a computer organization takes place in fixed size **blocks** (or pages). E.g., in UNIX, block sizes are traditionally 512 bytes but can be as large as 16 Kbytes. To minimize the number of disk accesses, we want to pack as many keys into each node as possible. So the ideal size for a node is the block size. Thus the parameter b of (a, b)-trees is chosen to be the largest value so that a node has this $b$lock size. Below, we discuss constraints on how the parameter a is chosen.

roughly: 1000 *CPU cycles per I/O*

parameter b is determined by <u>b</u>*lock size*

¶**85. Organization of an internal node.** The keys in an internal node of an (a, b)-search tree must be organized for searching, and manipulation such as merging or splitting two list of keys. Conceptually, we display them as in (45) but their actual organization may be quite different. Since the number of keys an internal node may not be small, their organization merit some discussion. In practice, b is a medium size constant (say, $b < 1000$) and a is a constant fraction of b. We consider four methods to organize these keys:

(1) an ordered array

(2) an ordered singly-linked list

(3) an ordered doubly-linked list

(4) a balanced BST

The time to search an internal node is $O(b)$ in methods (2) and (3) but $O(\log b)$ in methods (1) and (4). On the other hand, insertion and deletion is $O(b)$ in methods (1)-(3) but $O(\log b)$ in (4). That is as far as data-structure theory helps you. The optimal design calls for a more holistic look. We must account for two strong constraints:

(A) THE BLOCK CONSTRAINT: That is, the size of an internal node is fixed (being the size of a block of memory as determined by the operating system). *In Exercises, we ask the student to take a nominal value of $2^{12} = 4096$ bytes (or "4 Kbytes") for this size.* Since the

2574   space demand increases from methods (1) to (4), the value of b is expected to decrease as we
2575   move from methods (1) to (4). This will have an impact on the height of the search tree.

2576   (B) THE I/O CONSTRAINT: Our algorithms perform their actions only in main memory
2577   while the nodes reside in secondary (disk) memory. The operation to move a node in or out
2578   of the main memory is called an **I/O operation**. The I/O constraint refers to the fact that
2579   such operations are 2 or 3 orders of magnitude slower than a CPU operation. *In Exercises, the*
2580   *student is asked to assume a nominal value of* 1000 *CPU cycles per block swap.* In view of the
2581   I/O constraint, the overriding goal in the design of (a, b)-search trees should be to maximize b.
2582   When maximizing b, we remember there are some fixed overhead in each node: for instance,
2583   we assume that each internal node must maintain the current degree of the node (this is an
2584   integer between a and b). This space counts towards the overhead.

central tenet of
(a, b)-trees?

    Let us briefly consider how to choose b in method (1), i.e., storing keys in an ordered array.
As noted, we can search for a key in $O(\lg b)$ time but each insertion and deletion may need $O(b)$
time. This tradeoff is reasonable under the assumption that searches are much more frequent
than insertion/deletions in a node. If the overhead is H bytes, then we may choose b to be

$$\left\lfloor (2^{12} - H)/(K + P) \right\rfloor$$

2585   where each key uses K bytes and each data item takes P bytes. Note that data might have
2586   unpredictable size, but if we only store a pointer to the actual location of the data, then
2587   allocating a fixed amount of P bytes is reasonable (typically, $P <= 8$).

2588   For the remaining methods (2)-(4), it seems best to divide the internal node into fixed size
2589   "chunks" for use in the linked lists or as BST nodes. These "chunks" can be kept in a FREEL-
2590   IST, using the well-known FREELIST technique from Operating Systems. Our algorithm for
2591   inserting or deleting keys into the node can ask for a new chunk or return a chunk to the
2592   FREELIST very efficiently. The head of this FREELIST may be counted towards the fixed
2593   overhead of the node. This is explored in the Exercise.

2594   **¶86. On (a, b, c)-trees: Generalized Split-Merge.**   Our insertion and deletion algorithms
2595   use the strategy of "share a key if you can" in order to avoid splitting or merging. Here,
2596   "sharing" encompasses borrowing as well as donation. The 2/3-space utility method is now be
2597   generalized by the introduction of a new parameter $c \geqslant 1$. Call these (a, b, c)**-trees**. We use
2598   the parameter c as follows.

2599   • **Generalized Split** of u: When node u is overfull, i.e., with degree $b + 1$, we will examine
2600   up to $c - 1$ siblings to see if any of these siblings has degree $< b$. If so, we can donate a
2601   child of u to its sibling which in turn donates a child to the next sibling, etc, until the
2602   sibling of degree $< b$ receives a donated child. We are now done. Otherwise, we merge
2603 c nodes (node u plus $c - 1$ siblings), and split the merger into $c + 1$ nodes. We view c
2604   of these nodes as re-organizations of the original nodes, but one of them is regarded as
2605   new. We must insert this new node into the parent of u. The parent will be transformed
2606   appropriately, and the process repeated.

2607   We stress that there is no sharp distinction between donation and splitting: we view of
2608   them as different possibilities for a single **generalized split subroutine**: starting from
2609   an overfull node u, we successively bring into main memory a sequence of contiguous
2610   siblings of u (they may be right or left siblings) until we either (i) find one that has less
2611   than b children, or (ii) brought in the maximum number of $c - 1$ siblings. In case (i), we
2612   do donation, and in case (ii) we split these c siblings into $c + 1$ siblings.

- **Generalized Merge** of u: When node u is underfull, i.e., with degree $a - 1$, we will examine up to c siblings to see if we can borrow a child of these siblings. If so, we are done. Otherwise, we merge $c + 1$ nodes (node u plus c siblings), and split the merger into c nodes. We view c of the original nodes as being re-organized, but one of them being deleted. We must thus delete a node from the parent of u. The parent will be transformed appropriately.

  Again, we view borrowing and merging as two possibilities for a single **generalized merge** subroutine: starting from an underfull node u, we successively bring into main memory a sequence of contiguous siblings of u until we either (i) find one that has more than a children, or (ii) brought in the maximum number of c siblings. In case (i), we do borrowing, and in case (ii) we merge $c + 1$ siblings into c siblings.

In summary, the generalized merge-split of (a, b, c)-trees transforms c nodes into $c+1$ nodes, or vice-versa. The case $c = 1$ corresponds to the B-trees. The case $c = 2$ was discussed earlier in the algorithms for achieving a 2/3-space utilization ratio. In general, they achieve a space utilization ratio of $c : c + 1$ which can be arbitrarily close to 1 (we also need $b \to \infty$). Our (a, b, c)-trees must satisfy the following **generalized split-merge inequality**,

$$c + 1 \leqslant a \leqslant \frac{cb + 1}{c + 1}. \tag{63}$$

The lower bound on a ensures that generalized merge or split of a node will always have enough siblings. In case of merging, the current node has $a - 1$ keys. When we fail to borrow, it means that c siblings have a keys each. We can combine all these $a(c+1) - 1$ keys and split them into c new nodes. This merging is valid because of the upper bound (63) on a. In case of splitting, the current node has $b + 1$ keys. If we fail to donate, it means that $c - 1$ siblings have b keys each. We combine all these $cb + 1$ keys, and split them into $c + 1$ new nodes. Again, the upper bound on a (63) guarantees success.

We are interested in the maximum value of a in (63). Using the fact that a is integer, this amounts to

$$a = \left\lfloor \frac{cb + 1}{c + 1} \right\rfloor. \tag{64}$$

The corresponding (a, b, c)-tree will be called a **generalized B-tree**. Thus generalized B-trees are specified by two parameters, b and c.   *why, (b,c)-trees!*

What is the simplest generalized B-tree for specific values of c? When $c = 1$, it is $(a, b, c) = (2, 3, 1)$ (corresponding to the classic $(2, 3)$-tree). When $c = 2$, we noted that it is $(a, b, c) = (3, 4, 2)$. For $c = 3$, it is $(a, b, c) = (4, 5, 3)$. You might see a pattern here:

Lemma 20 *For any c, the smallest set of parameters (a, b, c) satisfying the generalized split-merge inequalities (63) is $(a, b, c) = (c + 1, c + 2, c)$.*

Proof. The first inequality in

$$c + 1 \leqslant a \leqslant \frac{cb + 1}{c + 1}$$

is trivially satisfied, since $a = c + 1$. The second inequality is $c + 1 \leqslant (cb + 1)/(c + 1)$ reduces to $(c + 1)^2 \leqslant c(c + 2) + 1$ when $b = c + 2$. This inequality is in fact an equality.   **Q.E.D.**

¶**87. Using the c parameter.** An (a, b, c)-trees is structurally indistinguishable from an (a, b)-tree. In other words, the set of all (a, b, c) trees and the set of all (a, b) trees are the same ("co-extensive").

Call a pair (a, b) **valid** if $2 \leqslant a < b$. Likewise, a triple (a, b, c) is **valid** if it satisfies (63). We view (63) as specifying a lower bound $a \geqslant c + 1$ and an upper bound $a \leqslant (cb + 1)/(c + 1)$ on the a parameter. *For any valid (a, b), can we find a c such that (a, b, c) is valid?* The answer is yes: choose $c = a - 1$. Then clearly the lower bound on a holds. The upper bound becomes to $a \leqslant ((a - 1)b + 1)/a$ or $a^2 \leqslant (a - 1)b + 1$. But this is clearly satisfied since $b \geqslant a + 1$.

The choice $c = a - 1$ is the largest possible choice. In analogy to B-Trees which satisfy (48), We might define "C-trees" as (a, b, c) trees whose parameters satisfy

$$b = 2a - 1, \quad c = a - 1. \tag{65}$$

The (a, b, c) parameters C-trees are given

$$(2, 3, 1), (3, 5, 2), (4, 7, 3), (5, 9, 4), \dots.$$

So the more interesting case is the smallest possible value of c. To determine the smallest c, we can easily verify (Exercise) this fact:

$$\textit{Assume } c + 2 \leqslant a. \textit{ If } (a, b, c) \textit{ is valid, so is } (a, b, c + 1).$$

In general, we like to assume the parameters (a, b) is hard-coded (it is optimally determined from the block size, etc). However, the c parameter need not be hard coded — the c parameter is only used during insertion/deletion algorithms, and we can freely change c (within the range of validity) in a dynamic fashion. Thus, we might store c in a global variable. E.g., if we implement a `C++` class for (a, b, c)-search structures, we can store c as a static member of the class. Why would we want to modify c? Increasing c improves space utilization but slows down the insertion/deletion process. Therefore, we can begin with $c = 1$, and as space becomes tight, we slowly increase c. And conversely we can decrease c as space becomes more available. This flexibility a great advantage of the c parameter.

¶**88. A Numerical Example.** Let us see how to choose the (a, b, c) parameters in a concrete setting. The nodes of the search tree are stored on the disk. The root is assumed to be always in main memory. To transfer data between disk and main memory, we assume a UNIX-like environment where memory blocks have size of 512 bytes. So that is the maximum size of each node. The reading or writing of one memory block constitute one disk access. Assume that each pointer is 4 bytes and each key 6 bytes. So each (key,pointer) pair uses 10 bytes. The value of b must satisfy

$$10b \leqslant 512.$$

You could say that the number of keys is one less than the number of pointers, and so the correct inequality is $10b \leqslant (512 + 4) = 516$. Although this makes no difference for the current calculation, in general it can make a difference of one. But for simplicity, we dispense with this refinement because we are already ignoring other details: for instance, each node will need to store a parent pointer, and probably its degree (i.e., number of children).

Hence we choose $b = \lfloor 512/10 \rfloor = 51$. Suppose we want $c = 2$. In this case, the optimum choice of a is $a = \left\lfloor \frac{cb + 1}{c + 1} \right\rfloor = 34$.

To understand the speed of using such $(34, 51, 2)$-trees, assume that we store a billion items in such a tree. How many disk accesses in the worst is needed to lookup an item? The worst case is when the root has 2 children, and other internal nodes have 34 children (if possible). A calculation shows that the height is 6. Assuming the root is in memory, we need only 6 block I/Os in the worst case How many block accesses for insertion? We need to read c nodes and write out $c + 1$ nodes. For deletion, we need to read $c + 1$ nodes and write c nodes. In either case, we have $2c + 1$ nodes per level. With $c = 2$ and $h = 6$, we have a bound of 30 block accesses.

...since a path from root to leaf has 7 nodes!

For storage requirement, let us bound the number of blocks needed to store the internal nodes of this tree. Let us assume each data item is 8 bytes (it is probably only a pointer). This allows us to compute the optimum value of a', b'. Thus $b' = \lfloor 512/8 \rfloor = 64$. Also, $a' = \left\lfloor \frac{cb' + 1}{c + 1} \right\rfloor = 43$. Using this, we can now calculate the maximum and number of blocks needed by our data structure.

¶89. Preemptive or 1-Pass Algorithms. The above algorithm uses 2-passes through nodes from the root to the leaf: one pass to go down the tree and another pass to go up the tree. There is a 1-pass versions of these algorithms. Such algorithms could potentially be twice as fast as the corresponding 2-pass algorithms since they could reduce the bottleneck disk I/O. The basic idea is to preemptively split (in case of insertion) or preemptively merge (in case of deletion).

First consider the standard insertion algorithm (where $c = 1$). During the Lookup phase, as we descend the search path from root to leaf, if the current node u is already full (i.e., has b children) then we will pre-emptively split u. Splitting u will introduce a new child to its parent, v. We may assume that v is in core, and by induction hypothesis, v is not full. So v can accept a new child without splitting. this involves no extra disk I/O. But this preemptive splitting of u is not without I/O cost – since v is modified, it must be written back into disk. This may turn out to be an unnecessary I/O in our regular algorithm. So, in the worst case, we could double the number of disk I/O's compared to the normal insertion algorithm.

Suppose the height is h. At the minimum, we need $h + O(1)$ disk I/O operations, just to do the lookup. (Note: The "$O(1)$" is to fudge some details about what happens at a leaf or a root, and is not important.) It may turn out that the regular insertion algorithm uses $h + O(1)$ disk I/O's, but the pre-emptive algorithm uses $3h + O(1)$ disk I/O's (because of the need to read each node u and then write out the two nodes resulting from splitting u). So the preemptive insertion algorithm is slower by a factor of 3. Conversely, it may turn out that the regular insertion algorithm has to split every node along the path, using 3 I/O's per iteration as it moves up the path to the root. Combined with the h I/O operations in Lookup, the total is $4h + O(1)$ I/O operations. In this case, the pre-emptive algorithm uses only $3h + O(1)$ disk I/O's, and so is faster by a factor of 4/3. Similar worst/best case analysis can be estimated for generalized insertion with $c \geqslant 2$.

For deletion, we can again do a preemptive merge when the current node u has a children. Even for standard deletion algorithm ($c = 1$), this may require 4 extra disk I/O's per node: we have to bring in a sibling w to borrow a key from, and to then write out u, w and their parent.

But there is another intermediate solution: instead of preemptive merge/split, we simply **cache** the set of nodes from the root to the leaf. In this way, the second pass does not involve any disk I/O, unless absolutely necessary (when we need to split and/or merge). In modern computers, main memory is large and storing the entire path of nodes in the 2-pass algorithm

seems to impose no burden. In this situation, the preemptive algorithms may actually be slower than a 2-pass algorithm with caching.

¶90. Background on Space Utilization.   Using the $a : b$ measure, we see that standard B-trees have about 50% space utilization. Yao showed that in a random insertion model, the utilization is about $\lg 2 \sim 0.69\%$. (see [14]). This was the beginning of a technique called "fringe analysis" which Yao [21] introduced in 1974. Nakamura and Mizoguchi [16] independently discovered the analysis, and Knuth used similar ideas in 1973 (see the survey of [2]).

Now consider the space utilization ratio of generalized B-trees. Under (64), we see that the ratio $a : b$ is $\frac{cb+1}{(c+1)} : b$, and is greater than $c : c + 1$. In case $c = 2$, our space utilization that is close to $\lg 2$. Unlike fringe analysis, we guarantee this utilization in the worst case. It seems that most of the benefits of (a, b, c)-trees are achieved with $c = 2$ or $c = 3$.

———Exercises

Exercise 8.1: Begin with the $(2, 3)$-search tree in Figure 38. Perform the following sequence of operations:

$$(a) \quad Ins(20), \qquad (b) \quad Ins(19), \qquad (c) \quad Del(10)$$

Assume the standard insert/delete algorithms. A single operation may cause multiple rebalancing acts (split, merge, borrow) which you must indicate.

$\diamondsuit$

Exercise 8.2: Consider the $(2,3)$-tree in Figure 45. Draw the following trees $T_1, \ldots, T_5$. *Assume the eager split policy in this question.*
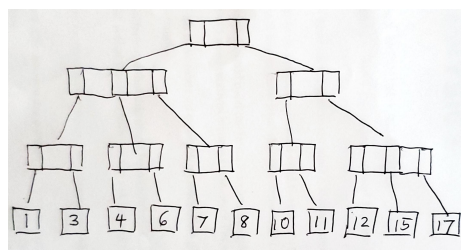

Figure 45: (2,3)-tree on 11 items

(i) T_1 is the $(2,3)$-search tree that results after you fill in the keys in the internal nodes of Figure 45. You must choose the minimum possible integer keys.

(ii) T_2 is like T_1, but where you choose the maximum possible *integer* keys.

(iii) T_3 is the result of inserting key 13 into T_2.

(iv) T_4 is the result of inserting key 9 into T_3.

(v) T_5 results from the minimum number of insertions into T_4 to ensure that the height increases. Describe all the possible *integers* that you can insert to achieve this.

$\diamondsuit$

Exercise 8.3: This is a continuation of the previous question where we consider deletions. The **eager merge policy** for deletion states: if a node u is underfull, we find a sibling v to borrow from v, or failing that, we merge with v. For definiteness, assume v is a right sibling (if one exists); otherwise v is the left sibling. Now if there are two siblings, and borrowing from the right sibling fails, the **reluctant merge policy** would try to borrow from the left sibling (if it exists) and failing that, merge with the left-sibling. *Assume the eager merge-right policy in this question.*

     1. Consider the tree T_2 from the previous question. Draw T_6, resulting from deleting key 4 from the tree T_2 in the previous question.

     2. What is the minimum number of keys you must delete from T_6 in order to reduce its height?

     3. What are the pros and cons of the eager merge policy for (a, b)-trees in the I/O model?

$\diamondsuit$

Exercise 8.4: Suppose we have a $(5, 6)$-search tree. What is the smallest possible value of c so that we have a valid $(5, 6, c)$-search tree? $\diamondsuit$

Exercise 8.5: Suppose we define $M(h)$ and $\mu(h)$ to be the maximum and minimum *size* of an (a, b)-tree of height h. Give the formula for $M(h)$ and $\mu(h)$. $\diamondsuit$

in text, $M(h), \mu(h)$ were defined for number of leaves, not size

Exercise 8.6: Prove the claim that if $c + 1 \leqslant a$ and (a, b, c) is valid then so is $(a, b, c + 1)$.

$\diamondsuit$

Exercise 8.7: We consider B-trees in this question. Since a B-tree node contains many keys, it must support searching within its nodes. We consider four different **schemes** for this internal organization:

     (i) an ordered array,

     (ii) an ordered singly-linked list,

     (iii) an ordered doubly-linked list,

     (iv) an AVL tree.

Consider a specific numerical example: each B-tree node is stored in a **block** whose size is $4096 = 2^{12}$ bytes. An address within the block is called a **local pointer**, and it needs only 12 bits. But for simplicity, we assume local pointers use two bytes. The address of each block in the secondary storage (i.e., disc) is specified by a **block pointer** that uses 4 bytes. Assume that each key take 6 bytes. Now answer these two questions for the schemes (i)-(iv):

     (a) What is the maximum value of the parameter b under the scheme? Be sure to show your calculations.

(b) What is the worst case time to search for a key in a B-tree with two million items? Give the answer in terms of the number of CPU cycles ("CPUC") for each of the four schemes. State explicit numbers, not expressions. REMARK: in the text, we only count the number of I/O operations. But for part(b), you must also count the cost of searching within each block.

Use the following additional ASSUMPTIONS:

(A1) Each internal node must store a parent pointer as well as its degree (i.e., number of children). Use 2 bytes for the degree. For schemes (ii)-(iv), we also reserve two bytes for book-keeping purposes.

(A2) Since this is a B-tree, once you know b, you can set $a \leftarrow \lfloor (b+1)/2 \rfloor$.

(A3) Each disk I/O takes 1000 CPU cycles.

(A4) Assume that $(a', b') = (a, b)$. This determines the number of items in a leaf. Is this reasonable? Well, assume that an item represented by the pair $(key, data)$ uses 6 bytes for key and 4 bytes for $data$. Intuitively, the $data$ is only a pointer to the actual data in disc that may use much more space! Therefore, the space requirement for an external node is comparable that of an internal node. We assume the search structure in an external node uses the same scheme (i)-(iv), but you may assume it has the same performance as the internal nodes.

(A5) If searching for a key takes $O(\lg n)$ or $O(n)$ CPU time, always assume that "4" is the constant in big-Oh notation. E.g., searching for a key in an AVL tree with $n = 100$ keys takes $4 \lg n = 4 \times 7 = 28$ CPUC (taking $\lg 100 = 7$). E.g., Searching in a linked list of length $n = 100$ takes $4n = 400$ CPUC.

(A6) The root of the search tree is always in main memory, so you never need to read or write the root.

(A7) You may use calculators, but we encourage simplified estimates whenever possible (e.g., you can assume $\lg 10^6 \simeq 20$).

$\Diamond$

Exercise 8.8: Consider an (a, b)-tree with n items and height h.

(a) Give upper and lower bounds on the height in terms of n. Your bounds will depend on the parameters (a, b) and (a', b'). HINT: see (51).

(b) Assume $100 = a' = a$ and $200 = b' = b$. If there are billion 10^9 items, what is the range of the corresponding height?

$\Diamond$

Exercise 8.9: Justify the following statements about (a, b)-search trees:
(a) If we only have insertions into an (a, b)-tree, then the keys in an internal node are just copies of keys of items found in the leaves.
(b) It is possible to maintain the property in part (a) even if there are both insertions and deletions. $\Diamond$

Exercise 8.10: Suppose you have an (a, b, c)-tree of height h. What is the number of disk I/O's in the worst case when inserting an item? Assume the the root is always in main memory. Give an expression involving c and h. $\Diamond$

Exercise 8.11: In the text, we did a worst/best case comparison between standard insertion and preemptive insertion algorithms. Please do the same for the standard deletion and the preemptive deletion algorithms. More precisely, answer these questions:
(a) What is the maximum number of I/O operations when doing a standard insertion into an (a, b)-search tree of height h?
(b) Repeat part (a), but now assume the pre-emptive insertion algorithm.
(c) In the best case scenario, how much faster is preemptive insertion?
(d) In the worst case scenario, how much slower is preemptive insertion?
(e) Based on the considerations above, should we do preemptive or regular insertion? $\diamond$

Exercise 8.12: We consider the worst case behavior of insertion and deletion in (a, b)-trees. Assume the standard insertion and deletion algorithms (not the enhanced ones). We are interested in counting the number of blocks that must be *written* into memory. For instance, in a LookUp, we must *read* h blocks of memory if the height of the tree is h, but we do not need to write any blocks. Each node is assumed to be stored in one block in the disk. Also assume the root is always in main memory. Assume $(a', b') = (a, b)$.
(a) Suppose $b = 2a - 1$. For each h and n, construct a B-tree of height h and a sequence of $2n$ requests

$$(D_1, I_1, D_2, I_2, \ldots, D_n, I_n)$$

such that these requests results in $3nh + O(1)$ block writes. Here D_i is a deletion and I_i is an insertion $(i = 1, \ldots, n)$.
(b) Suppose $b = 2a$. The **potential** $\phi(u)$ of node u in the (a, b)-search tree is defined to be

$$\phi(u) = \begin{cases} 4 & \text{if } degree(u) = a - 1 \\ 1 & \text{if } degree(u) = a \\ 0 & \text{if } a < degree(a) < b \\ 3 & \text{if } degree(u) = b \\ 6 & \text{if } degree(u) = b + 1 \end{cases}.$$

Note that during insertion (deletion), we allow nodes to be temporarily have degree $b + 1$ $(a - 1)$. Moreover, the potential of the (a, b)-tree is just the sum of the potential of all the nodes in the tree.
(b.1) Show that each split, borrow, or merge operation results in the reduction of potential in the tree.
(b.2) Conclude that if we begin with an empty (a, b)-tree, and start to perform a sequence of n requests

$$(R_1, R_2, R_3, \ldots, R_n)$$

where each R_i is either an insert or a delete, the total number of block writes is only $O(n)$. $\diamond$

Exercise 8.13: We noted that if an (a, b)-search tree has $n \geq 1$ items, then it has $n - 1$ search keys, as indicated in (47). Describe how you must modify insertion and/or deletion in order to ensure that $k_i' = k_{i+1}$ for all i. $\diamond$

Exercise 8.14: Our "Enhanced Standard Insert/Delete" algorithms try to share (i.e., donate or borrow) children from siblings only. Suppose we now relax this condition to allow sharing among "immediate first cousins"
(a) Modify our insert/delete algorithms so that we try to share with direct siblings or cousins before doing the generalized split/merge.
(b) Carry out the worst case I/O analysis.

REMARKS: "Immediate cousin" means an adjacent node in the same level that is not a sibling. "First cousin" means nodes in the same level that shares a grandparent. Note that we stick to $c = 1$ in this question; do not consider generalized merge/split.   ◇

Exercise 8.15: Suppose you have an (a, b, c)-tree of height h. What is the number of disk I/O's in the worst case when inserting an item? Assume the the root is always in main memory. Give an expression involving c and h.   ◇

Exercise 8.16: Do the same worst/best analysis as the previous question, but assuming an arbitrary $c \geqslant 2$:
(I) Compare insertion algorithms (regular and pre-emptive)
(D) Compare deletion algorithms (regular and pre-emptive)   ◇

Exercise 8.17: Let us suppose that each node stores, not just pointers to its children, but also track the degree of its children.
(a) Discuss how to maintain this additional information during insert and deletes the (a, b, c)-search trees.
(b) How can this additional information speed up your algorithm?   ◇

Exercise 8.18: What is the the best ratio achievable under (53)? Under (62)?   ◇

Exercise 8.19: Give a detailed analysis of space utilization based on parameters for (A) a key value, (B) a pointer to a node, (C) either a pointer to an item (in the exogenous case) or the data itself (in the endogenous case). Suppose we need k bytes to store a key value, p bytes for a pointer to a node, and d bytes for a pointer to an item or for the data itself. Express the space utilization ratio in terms of the parameters

$$a, b, k, p, d$$

assuming the inequality (53).   ◇

Exercise 8.20: Consider the tree shown in Figure 36. Although we originally viewed it as a $(3, 4)$-tree, we now want to view it as a $(2, 4)$-tree (yes we can). For insertion/deletion we further treat it as a $(2, 4, 1)$-tree.
(a) Insert an item (whose key is) 14 into this tree. Draw intermediate results.
(b) Delete the item (whose key is) 4 from this tree. Draw intermediate results.   ◇

Exercise 8.21: To understand the details of insertion and deletion algorithms in (a, b, c)-trees, we ask you to implement in your favorite language (we like Java) the following two $(2, 3, 1)$-trees and $(3, 4, 2)$-trees.   ◇

Exercise 8.22: Suppose we want the root, if non-leaf, to have at least a children. But we now allow it to have more than b children. This is reasonable, considering that the root should probably be kept in memory all the time and so do not have to obey the b constraint. Consider this idea: we allow the root, when it is a leaf, to have up to $a'a - 1$ items. Here, (a', b') is the usual bound on the number of items in non-root leaves. Similarly, when it is a non-leaf, it has between a and $\max\{a^a - 1, b\}$ children. Show how to consistently carry out this policy.   ◇

Exercise 8.23: Explore the bounded balanced version of (a, b)-trees. Bound the height of bounded balanced (a, b)-trees, and provide insertion and deletion algorithms.   ◇

_____ End Exercises

References

[1] A. Anderson. General balanced trees. *J. Algorithms*, 30:1–18, 1989.

[2] R. A. Baeza-Yates. Fringe analysis revisited. *ACM Computing Surveys*, 27(1):109–119, 1995.

[3] R. Bayer and McCreight. Organization of large ordered indexes. *Acta Inform.*, 1:173–189, 1972.

[4] N. Blum and K. Mehlhorn. On the average number of rebalancing operations in weight-balanced trees. *Theor. Computer Science*, 11:303–320, 1980.

[5] P. Brass. *Advanced Data Structures*. Cambridge University Press, Cambridge, UK, 2008.

[6] T. H. Corman, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press and McGraw-Hill Book Company, Cambridge, Massachusetts and New York, second edition, 2001.

[7] C. C. Foster. A generalization of AVL trees. *Comm. of the ACM*, 16(8):513–517, 1973.

[8] Y. Hirai and K. Yamamoto. Balancing weight-balanced trees. *J. of Functional Programming*, 21(3):287–307, 2011.

[9] S. Huddleston and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Inform.*, 17:157–184, 1982.

[10] D. E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume 1. Addison-Wesley, Boston, 2nd edition edition, 1975.

[11] K. S. Larsen. AVL Trees with relaxed balance. *J. Computer and System Sciences*, 61:508–522, 2000.

[12] H. R. Lewis and L. Denenberg. *Data Structures and their Algorithms*. Harper Collins Publishers, New York, 1991.

[13] G. Lueker and D. Willard. A data structure for dynamic range queries. *Inf. Proc. Letters*, 15:209–213, 1982.

[14] K. Mehlhorn. *Datastructures and Algorithms 1: Sorting and Sorting*. Springer-Verlag, Berlin, 1984.

[15] K. Mehlhorn. *Datastructures and Algorithms 3: Multi-Dimensional Searching and Computational Geometry*. Springer-Verlag, Berlin, 1984.

[16] T. Nakamura and T. Mizoguchi. An analysis of storage utilization factor in block split data structuring scheme. *VLDB*, 4:489–195, 1978. Berlin, September.

[17] J. Nievergelt and E. M. Reingold. Binary trees of bounded balance. *SIAM J. Computing*, 2(2):33–43, 1973.

[18] S. Roura. A new method for balancing binary search trees. In F. Orejas, P. Spirakis, and J. van Leeuwen, editors, *LNCS*, volume LNCS 2076, pages 469–480. Springer, 2001. Proc. Automata, Languages and Programming.

[19] R. E. Tarjan. *Data Structures and Network Algorithms.* SIAM, Philadelphia, PA, 1974.

[20] D. Willard and G. Lueker. Adding range restriction capability to dynamic data structures. *J. ACM*, 32:597–617, 1985.

[21] A. C.-C. Yao. On random 2-3 trees. *Acta Inf.*, 9(2):159–170, 1978.