"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 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.

I have restricted the above list to trees that are used as search data structures. If we include trees arising in specific applications (e.g., Huffman tree, DFS/BFS tree, alpha-beta tree), we

obtain an even more diverse collection of tree words. The list could be enlarged to include variants of these trees: thus there are subspecies of B-trees called B^+- and B^*-trees, etc. The list will continue to grow as the field progresses.

If there is a most important entry in the above list, it has to be *binary search tree*. It is the first non-trivial data structure that students encounter, after linear structures such as arrays, lists, stacks and queues. Trees are useful for implementing a variety of **abstract data types**. We shall see that all the common operations for search structures are easily implemented using binary search trees. Most basic algorithms on binary search trees have a worst-case behavior that is proportional to the height of the tree. Therefore it is vital to bound the height of trees with any given size (i.e., number of nodes). By an elementary argument, the height of a binary tree of size n is at least $\lg(1+n)$. A family of binary trees is said to be **balanced** if every tree in the family on n nodes has height $O(\log n)$. The implicit constant in the big-Oh notation here depends on the particular family. A balanced family may not be worth much unless it is equipped with efficient algorithms for inserting and deleting items from trees, while preserving membership in the family.

balance-ness is a family property

Among the balanced families invented in computer science, we can classify them into three basic schemes: **height-balanced**, **weight-balanced** and **degree-balanced**. For height- and weight-balanced schemes, we ensure that the relevant parameter (height or weight) of any pair of siblings is "approximately the same". For height-balancing, the difference in the height of two sibling are bounded; for weight-balancing, it is the ratio of their weights that are bounded. The "weight" of a node is typically the size of its subtree (but there is possibility for a more general way of assigning weight). Height- and weight-balancing are applied to binary trees, but our third scheme is aimed at non-binary trees. In degree-balancing, we ensure that every node has degree that "approximately the same" in the sense of bounded ratio. This requirement still results a vast number of possibilities, but nice families emerge if we impose another restriction, that every leaf in such schemes has the same depth.

This chapter will focus on one representative from each of these types. Height balancing is represented by the **AVL trees**. It was also the first balanced family to be invented (1962), and named for its Russian inventors Adel'son-Vel'skii and Landis. Weight balancing is represented by the family of bounded-balance binary trees introduced by Nievergelt and Reingold (1973). Finally degree balancing is represented by the family of (a,b)**-trees**, which generalizes the B-trees introduced by Bayer and McCreight (1972). Height-balancing requires less information to maintain than the weight-balancing: each node needs $\lg\lg n$ (or even constant) bits, as opposed to $\lg n$ bits. Weight-balancing has some extra flexibility that are needed for some applications. Although the algorithms for weight-balancing are simpler, their analysis is considerably more subtle. Degree-balanced trees turns out to be extremely important in practice due to their applications in data bases. Tarjan [18] gives a brief history of some balancing schemes. Brass [5] is an excellent resource 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 to implement them in their favorite programming language.

Students: expect to do hand-simulations of algorithms!

§1. Search Structures with Keys

Search structures store a set of objects subject to searching and modification of these objects. Here we will standardize some terminology for search structures. Search structures can be viewed as a collection of **nodes** that are interconnected by pointers. Abstractly, they are just directed graphs with labels on edges and vertices. Each node stores an object which we call an **item**. We will be informal about how we manipulate nodes — they will variously look like ordinary variables and other times like pointers[1] as in the programming language `C/C++`, or like references in `Java`. Let us look at some intuitive examples, relying on your prior knowledge about programming and variables.

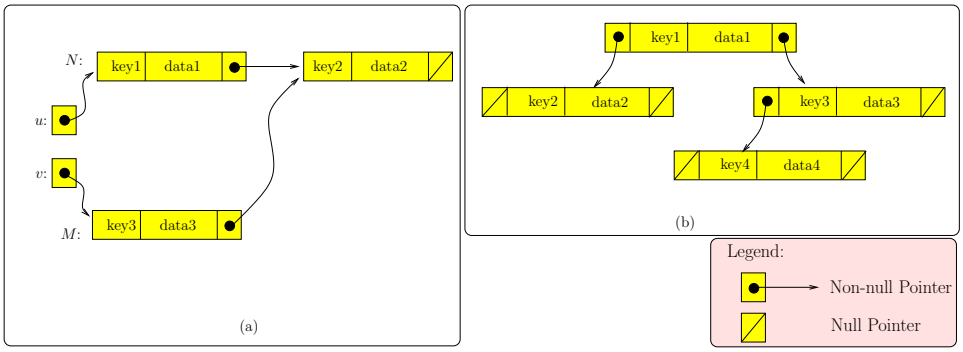
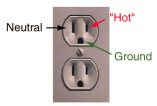


Figure 1: Two Kinds of Nodes: (a) linked lists, (b) binary trees

¶2. Keys and Items. We said each node stores an item. Each item is associated with a **key**. The rest of the information in an item is simply called **data**, so that we may regard an **item** as a pair $(Key, Data)$.

item = (key,data)

Besides an item, each node also stores one or more pointers to other nodes. Two simple types of nodes are illustrated in Figure 1: nodes with only one pointer (Figure 1(a)) are used to forming linked lists; nodes with two pointers can be used to form a binary trees (Figure 1(b)), or doubly-linked lists. Nodes with three pointers can be used in binary trees that require parent pointers. First, suppose N is a node variable of the type in Figure 1(a). Thus N has three **fields**, and we may name these fields as `key`, `data`, `next`. Each field has some data type. E.g. `key` is typically integer, `data` can be string, but `next` has to be a pointer to nodes. This field information constitutes the "type" of the node. To access the fields in node N, we write N.`key`, N.`data` or N.`next`. The type of N.`next` is not a node, but pointer to a node. In our figures, we indicate the values of pointers by a directed arrow. Node pointer variables act rather like node variables: if variable u is a pointer to a node, we also[2] write u.`key`, u.`data` and u.`next` to access the fields in the node. There is a special pointer value called[3] the **null pointer** whose value is denoted by `nil`. It points to nothing, but as a figure of speech, we say it points to the **nil node**. Of course the nil node is not a true node since it cannot store any information. In figures (cf. Figure 1) null pointers are indicated by a square box with a slash line.



Another convention is for null pointers to point to the "ground symbol" (from electrical engineering)

The following defines a node in the programming languages `C` and `Java`, respectively:

[1] The concept of **locatives** introduced by Lewis and Denenberg [11] may also be used: a locative u is like a pointer variable in programming languages, but it has properties like an ordinary variable. Informally, u will act like an ordinary variable in situations where this is appropriate, and it will act like a pointer variable if the situation demands it. This is achieved by suitable automatic referencing and de-referencing semantics for such variables.

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

[3] The null or nil terminology is often interchangeable in the literature.

```c
/* C program:  */
    struct Node {
       int key;
       struct Node* next;
    };
```

```java
// Java program:
    class Node {
       int key;
       Node next;
    }
```

Note the definition of a `Node` type is recursive because it refers to itself. In `C`, the node and node-pointers are different types, as illustrated by this snippet:

```
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.
```

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

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

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 (`key`, `data`, `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.

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

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'	$\&x$	$\&N$

The clue from the story of Alice and the White Knight

Examples of search structures:

(i) An *employee database* where each item is an employee record. The key of an employee record is the social security number, with associated data such as address, name, salary history, etc.

(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.

(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.

It is natural to refer 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. This is somewhat paradoxical since, for the users of the search structure, it is the data that is more important. With this caveat, we will normally ignore the data part of an item in our illustrations, thus *identifying the item with the key only.*

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

Binary search trees is an example of a keyed search structure. Usually, each node of the binary search trees stores an item. In this case, our terminology of "nodes" for the location of items happily coincides with the concept of "tree nodes". However, there are versions of binary

search trees whose items resides only in the leaves – the internal nodes only store keys for the purpose of searching.

usually, keys ≡ integers!

¶3. 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.

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 $u = \texttt{nil}$. 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

¶4. 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 $\texttt{insert}(I, S)$ which returns a new structure S' in which I is added. We may denote this as

E.g., mathematical variable vs. programming variables

$$S' \leftarrow \texttt{insert}(I, S) \quad \text{or} \quad \texttt{insert}(I, S) \rightarrow S'. \tag{1}$$

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:

$$S.\texttt{insert}(I) \tag{2}$$

which modifies the structure S to have an additional item I (all previous items are still in S). In OOPL terminology, (2) is calling a method of the class S. We call (1) the **functional form** of the insert operation, and (2) 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.*

¶5. 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	`make`()→*Structure*	creates a structure
		`kill`()	destroys a structure
(II)	Enumeration and Order	`list`() →*Node*	returns an iterator
		`succ`(*Node*)→*Node*	returns the next node
		`pred`(*Node*)→*Node*	returns the previous node
		`min`()→*Node*	returns a minimum node
		`max`()→*Node*	returns a maximum node
(III)	Dictionary-like Operations	`lookUp`(*Key*)→*Node*	returns a node with *Key*
o		`insert`(*Item*)→*Node*	returns the inserted node
		`delete`(*Node*)	deletes a node
		`deleteMin`()→*Item*	deletes a minimum node
(IV)	Set Operations	`Split`(*Key*)→ *Structure*	split a structure into two
		`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.`make`() will initialize the structure S, and S.`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.`merge`(S') can be simply written as "`merge`(S')".

Group (I): We need to initialize and dispose of search structures. Thus `make` (with no arguments) returns a brand new empty instance of the structure. The inverse of `make` is `kill`, to remove a structure. We view `make` as a constant time operation, but `kill` might not be in practice. Nevertheless we can view `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 `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 `min`() and `max`() operations are obvious. The successor `succ`(u) (resp., predecessor `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 `list`() can be implemented using `min`() and `succ`(u) or `max`() and `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,

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

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

The node u returned by `lookUp`(K) has the property that u.`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 `C`, nodes are usually represented by pointers. In this case, a standard convention is for the `lookUp` function to return a `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.

¶6. 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 [18]. 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

```
ListTraversal(L):
    u ← L.head
    While (u ≠ nil)
        VISIT(u)
        u ← u.next
    CLEANUP()
```

Here, VISIT(u) and CLEANUP() are **macros**, meaning that they stand for pieces of code that will be textually substituted before compiling and executing the program. We will indicate a macro ABC by framing it in a box like ABC. Macros should be contrasted to **subroutines**, which are independent procedures. In most situations, there is no semantic difference between macros and subroutines (except that macros are cheaper to implement). But see the implementation of `lookUp`(K) next. Note that macros, like subroutines, can take arguments. As a default, the macros do nothing ("no-op") unless we specify otherwise. We call ListTraversal a **shell program** — this theme will be taken up more fully when we discuss tree traversal below (§4). Since the while-loop (by hypothesis) visits every node in L, there is a unique node u (assume L is non-empty) with u.`next` = `nil`. This node is L.`tail`.

macros are not subroutines

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

- `lookUp`(K): We can use the above ListTraversal routine but expand "VISIT(u)" into the following code fragment:

  ```
  VISIT(u): if (u.key = K) Return(u)
  ```

  Since VISIT is a macro and not a subroutine, the Return in VISIT is *not* a return from VISIT, but a return from the `lookUp` routine! The CLEANUP macro is similarly replaced by

  ```
  CLEANUP(): Return(nil)
  ```

  The correctness of this implementation should be obvious.

- `insert`(K, D): We use the ListTraversal shell, but define VISIT(u) as the following macro:

  ```
  VISIT(u):    if (u.key = K) Return(nil)
  ```

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

```
CLEANUP():
    u ← new(Node)
    u.key ← K;  u.data ← D
    u.next ← L.head;  u.prev ← nil
    L.head.prev ← u    ◁ No-op if L.head = nil
    L.head ← u
    If (L.tail = nil) then L.tail ← u
    Return(u)
```

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:

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

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

¶7. 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 `lookUp`(K) is $\Theta(n)$ in the worst case because we have to traverse the entire list in the worst case. `insert`(K, D) is preceded by a `lookUp` and is $\Theta(n)$ in the worst case. The `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.

¶8. 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.

These data types are "abstract" because we make no assumption about the actual implementation.

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

- **Dictionary ADT**: `lookUp`[, `insert`[, `delete`]].
- **Ordered Dictionary ADT**: `lookUp`, `insert`, `delete`, `succ`, `pred`.
- **Priority queue ADT**: `deleteMin`, `insert`[, `delete`[, `decreaseKey`]].
- **Fully mergeable dictionary ADT**: `lookUp`, `insert`, `delete`, `merge`[, `Split`].

For instance, an ADT that supports only the three operations of `lookUp`, `insert`, `delete` is called a **dictionary ADT**. In these ADT's, there may be stripped-down versions where we omit some operations. These omitted operations are enclosed in square brackets: $[\cdots]$. Thus, a dictionary ADT without the `delete` operation is called a **semi-dynamic dictionary**, and if it further omits `insert`, it is called a **static dictionary**. Thus static dictionaries are down to a bare minimum of the `lookUp` operation. If we omit the `Split` 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)!*

Alternatively, some ADT's can be enhanced by additional operations. For instance, a priority 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

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

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

Recall the deletion operation is normally given the node u to be deleted, but an alternative is to be given the key K to be deleted. If the deletion are based on keys, we may think of the dictionary as a kind of **associative memory**. The operations `make` and `kill` (from group (I)) are assumed to be present in every ADT.

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

¶9. 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 ← make()
2.      for i = 1 to n do
            Q.insert(A[i])
3.      for i = 1 to n do
            A[i] ← Q.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 ¶6, 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 has the three operations of `lookUp`, `insert`, and `delete`.
(a) Suppose we omit `lookUp` from our ADT but still allow the other two operations. Call this ADT a WOM ("write only memory"). Discuss some possible uses for a WOM.
(b) Suppose you further omit `delete`. Call this 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".

Recall the definition and basic properties of binary trees in the Appendix of Chapter I. 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.`key` called its key. The keys in Figure 2 are integers, used simply as identifiers for the nodes.

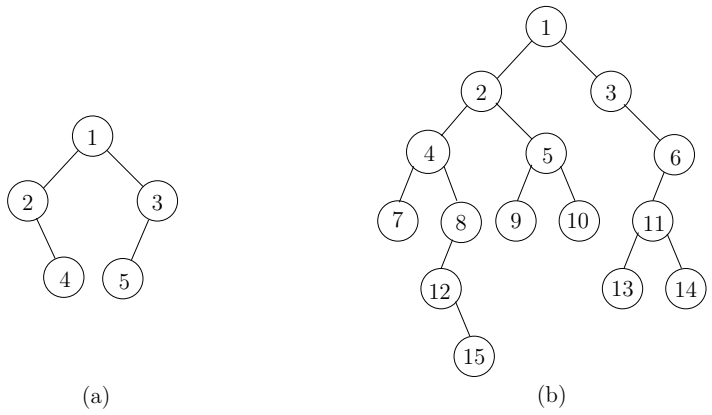


Figure 2: Two binary (not search) trees: (a) small, (b) big

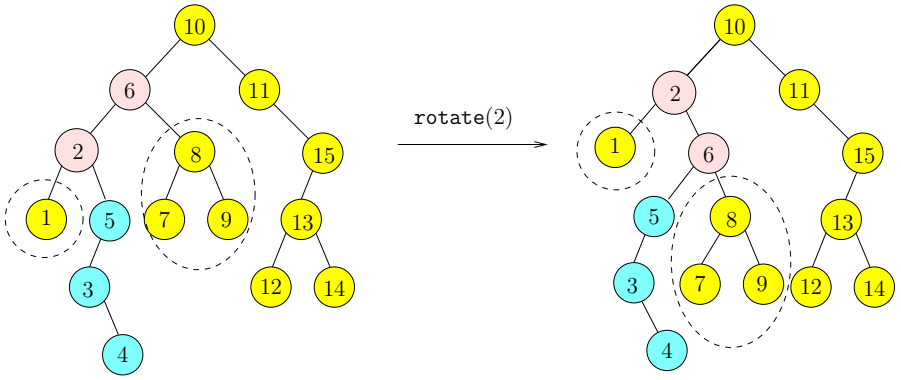
Briefly, a binary tree T is a set N of **nodes** where each node u_0 has two pointers, u_0.`left` and u_0.`right`.

- 1. The set N is either the empty set, or N has a special node u_0 called the **root**.

- 2. The remaining nodes $N \setminus \{u_0\}$ are partitioned into two sets of nodes that, recursively, form binary trees, T_L and T_R.
- 3. Moreover u_0.`left` and u_0.`right` point to the roots of T_L and T_R (resp.). The trees T_L, T_R are called the **left** and **right subtrees** of T. If these subtrees are empty, then the corresponding pointers (u_0.`left` and u_0.`right`) are `nil`.

A few other useful definitions. 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 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 $k \geq 1$ leaves iff it has $k - 1 \geq 0$ 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. 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 the other hand, for every $n \geq 0$, there is a unique complete binary tree of size n. A binary tree is said to be **linear** if each node u has at most one child. So it is essentially a linear list.



(a) (b)

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

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

BST are binary trees that satisfy the BST property!

A binary tree T is called a **binary search tree** (BST) if each node $u \in T$ has a field u.`key` that satisfies the **BST property**:

$$u_L\texttt{.key} < u\texttt{.key} \leq u_R\texttt{.key}. \tag{3}$$

for any left descendant u_L and any right descendant u_R of u. Please verify that the binary search trees in Figure 3 obey (3) at each node u.

The "standard mistake" is to replace (3) by $u.\texttt{left.key} < u.\texttt{key} \leq u.\texttt{right.key}$. By definition, a left (right) descendant of u is a node in the subtree rooted at the left (right) child of u. The left and right children of u are denoted by $u.\texttt{left}$ and $u.\texttt{right}$. This mistake focuses on a necessary, but not sufficient, condition in the concept of a BST. Self-check: construct a counter example to the standard mistake using a binary tree of size 3.

good quiz question...

> **Structural Induction for binary trees**: our definition of binary trees provide an in-built inductive structure on binary trees. Proofs using this structure is called "proof by structural induction", analogous to natural induction. *Most properties of binary trees are best proved by structural induction.* Likewise, algorithms for binary trees are often best described using structural induction.

¶10. Height of binary trees. Let $M(h)$ and $\mu(h)$ (resp.) be the maximum and minimum number of nodes in a binary tree with height h. It is easy to see that

$$\mu(h) = h + 1. \tag{4}$$

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

$$M(h) = 2^{h+1} - 1 \tag{5}$$

and it is trivial to verify this for all h. Another way to see $M(h)$ is that it is equal to $\sum_{i=0}^{h} 2^i$ since there are at most 2^i nodes at level i, and this bound can be achieved at every level. The simple formula (5) tells us a basic fact about the minimum height of binary trees on n nodes: if its height is h, then clearly, $n \leq M(h)$ (by definition of $M(h)$. Thus $n \leq 2^{h+1} - 1$, leading to

$$h \geq \lg(n+1) - 1. \tag{6}$$

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

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

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

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

¶13. Rotation. Rotation is not an operation in our list of ADT operation (§2), but we view it as the critical operation for binary trees. Roughly, to rotate a node u means to make the parent of u become its child. The set of nodes is unchanged. On the face of it, rotation does nothing essential: it is just redirecting some parent/child pointers. Rotation is an **equivalence transformation**, i.e., it transforms a binary search tree into an equivalent one. Two search structures that store exactly the same set of items are said to be **equivalent**. Remarkably, we shall show that rotation can[4] form the basis for all other binary tree operations.

The operation `rotate`(u) is a null operation ("no-op" or identity transformation) when u is a root. So assume u is a non-root node in a binary search tree T. Then `rotate`(u) amounts to the following transformation of T (see Figure 4).

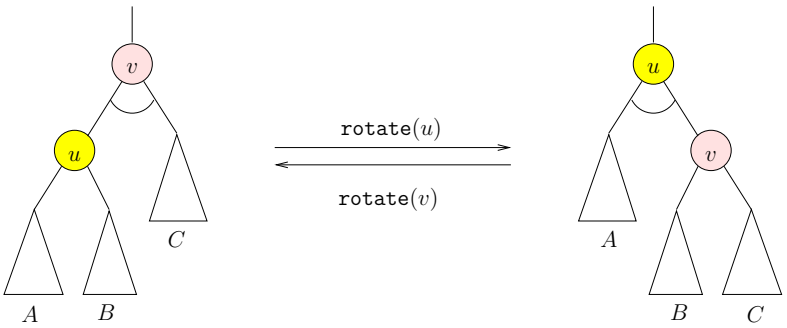


Figure 4: Rotation at u and its inverse.

As seen in Figure 4, `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

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

in the sense that each key in A is less than u which is less than any key in B, etc. So, the only way to maintain the BST property is to re-attach them as shown in Figure 4(right side). Only the parent of the root of B has switched from u to v; the parents of the root of A and C are unchanged. Furthermore, after `rotate`(u), the former parent of v (not shown) will now have u instead of v as a child. After a rotation at u, the depth of u is decreased by 1. Note that `rotate`(u) followed by `rotate`(v) is the identity or no-op operation; this is indicated by the left-arrow in Figure 4.

¶14. Graphical convention: Figure 4 encodes two conventions: consider the figure on the left side of the arrow. First, the edge connecting v to its parent is directed vertically upwards. This indicates that v could be the left- or right-child of its parent. Second, the two edges from v to its children are connected by a circular arc. This is to indicate that u and its sibling could[5] exchange places (i.e., u could be the right-child of v even though we choose to show u as the left-child). Thus Figure 4 is a compact way to represent four distinct situations.

[4] Augmented by the primitive operations of adding or removing a node.
[5] If this were to happen, the subtrees A, B, C needs to be appropriately relabeled.

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

Let us classify a node u into one of three **types**: *left*, *right* or *root*. This is defined in the 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 tested: u is type root iff $u.\texttt{parent} = \texttt{nil}$, and u is type left iff $u.\texttt{parent.left} = u$. Clearly, $\texttt{rotate}(u)$ is sensitive to the type of u. In particular, if u is a root then $\texttt{rotate}(u)$ is the null operation. If $T \in \{\texttt{left}, \texttt{right}\}$ denote left or right type, its **complementary type** is denoted $\overline{T}$, where $\overline{\texttt{left}} = \texttt{right}$ and $\overline{\texttt{right}} = \texttt{left}$.

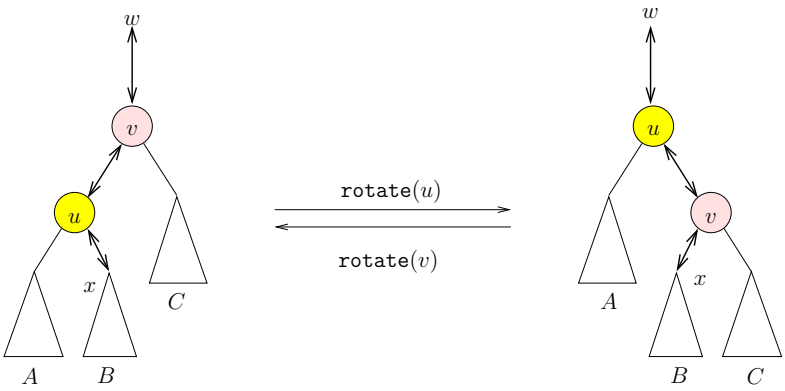


Figure 5: Links that must be fixed in $\texttt{rotate}(u)$.

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

x is an inner grandchild of v

$$(x, u), (u, v), (v, w). \tag{7}$$

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

$$(x, v), (v, u), (u, w). \tag{8}$$

These pairs are illustrated in Figures 5–6 where we explicitly show the parent pointers as well as child pointers. Thus, to implement rotation, we need to reassign 6 pointers (3 parent pointers and 3 child pointers). We show that it is possible to achieve this re-assignment using exactly 6 assignments.

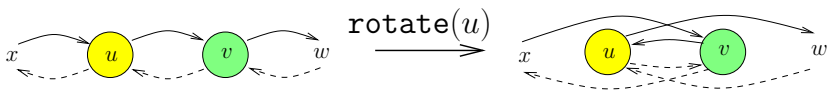


Figure 6: Simplified view of $\texttt{rotate}(u)$ as fixing a doubly-linked list (x, u, v, w).

Such re-assignments must be done in the correct order. It is best to see what is needed by thinking of (7) as a doubly-linked list (x, u, v, w) which must be converted into the doubly-linked list (x, v, u, w) in (8). This is illustrated in Figure 6. For simplicity, we use the terminology of doubly-linked list so that $u.\texttt{next}$ and $u.\texttt{prev}$ 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` is $w.T'$ for another type T'. A further complication is that x or/and w may not exist; so these conditions must be tested for, and appropriate modifications taken.

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

¶16. Variations on Rotation. The above rotation algorithm assumes that for any node u, we can access its parent u' and grandparent u''. This is true if each node has a parent pointer u.`parent`. *This is our default assumption for binary tree algorithms.* But even if we have no parent pointers, we could modify our algorithms to achieve the desired results because our search invariably starts from the root, and we can keep track of the triple (u, u', u'') which 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[6] as follows:

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

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

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

[6] Careful: "`left-rotate`(T, u)" in [?] corresponds our `right-rotate`(u) (and vice-versa).

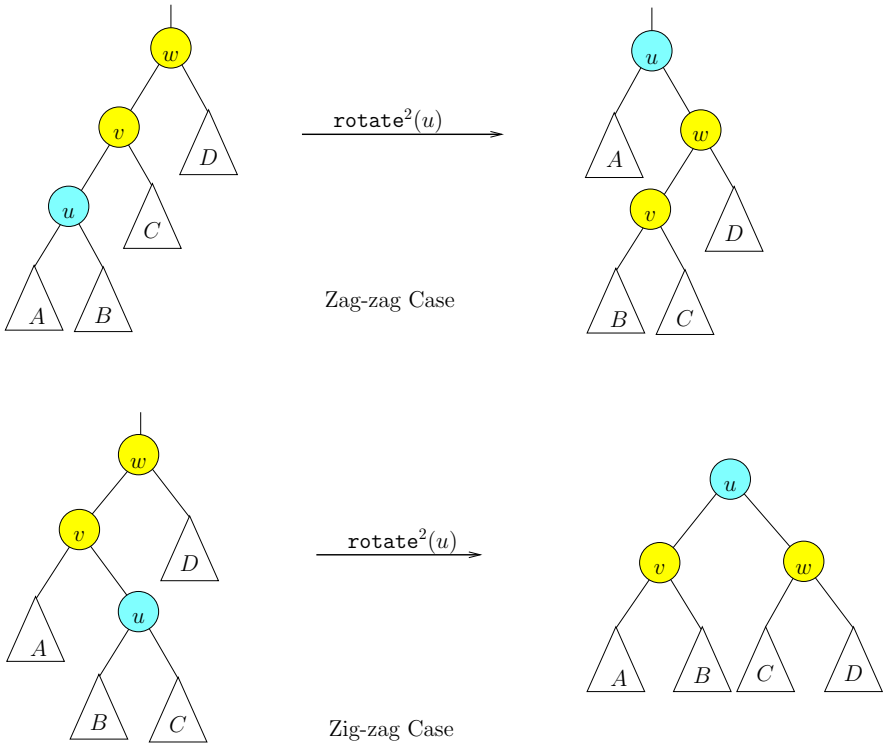


Figure 7: Two outcomes of $\texttt{rotate}^2(u)$

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

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

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

¶18. Rotation Distance Between Trees. Given two binary search trees, T and T', we define their **rotation distance** $R(T, T')$ to be the minimum number of rotations to convert T to T'. Of course, if T, T' are not equivalent, then $R(T, T') = \infty$. It is not hard to show that $R(T, T') = O(n^2)$ if T, T' are equivalent. Let $R(n)$ denote the rotation distance between two equivalent full BST's with n internal nodes (hence $n + 1$ external nodes). Sleator, Tarjan and Thurston (1988) has shown that $R(n) \leq 2n - 6$ for $n \geq 11$, and this bound is tight for n large enough. This is a deep result. There is no known polynomial time to compute $R(T, T')$. Culik and Wood (1982) has shown that $R(n) \leq 2n - 2$. 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).

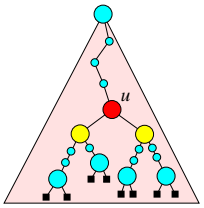
We describe a recursive procedure to convert T into L_n: Let u_0 be the internal node of rank 1. Clearly, the left child of u_0 is a leaf containing the minimum key in T.

```
Conversion to Left-List:
1.    Initialize u to be u_0, internal node of rank 1.
2.    While u ≠ root      ◁ The left subtree of u is a left-list
3.        While (u.right ≠ leaf)
4.            rotate(u.right)      ◁ Increment depth(u)
5.        u ← u.parent             ◁ Increment rank(u)
```

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

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

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



5 *paths from a node*

¶19. Five Canonical Paths from a node. A **path** in a binary tree is a sequence of nodes $(u_0, u_1, \ldots, u_n)$ where either (1) each u_i is a child of u_{i-1}, or (2) each u_i is a parent of u_{i-1}. The length of this path is n, and u_n is also called the **tip** of the path. Thus paths with positive length must fall under exactly one of two types: either (1) $depth(u_n) = depth(u_0) + n$ or (2) $depth(u_n) = depth(u_0) - n$. E.g., $(2, 4, 8, 12)$ is a path in Figure 2(b), with tip 12. Relative to a node u, there are 5 canonical paths that originate from u. The first of these is the path from u to the root, called the **root-path** of u. In figures, the root-path is displayed as an upward path, following parent pointers from the node u. E.g., if $u = 4$ in Figure 2(b), then the root-path is $(4, 2, 1)$. Next we introduce 4 downward paths from u. The **left-path** of u is simply the path that starts from u and keeps moving towards the left child until we cannot proceed further. The **right-path** of u is similarly defined. E.g., with $u = 4$ as before, the left-path is $(4, 7)$ and right-path is $(4, 8)$. Next, we define the **left-spine** of a node u is defined to be the path $(u, \text{rightpath}(u.\texttt{left}))$. In case $u.\texttt{left} = \texttt{nil}$, the left spine is just the trivial path (u) of length 0. The **right-spine** is similarly defined. E.g., with u as before, the left-spine is $(4, 7)$ and right-spine is $(4, 8, 12)$. The tips of the left- and right-paths at u correspond to the minimum and maximum keys in the subtree at u. The tips of the left- and right-spines, *provided they are different from u itself*, correspond to the predecessor and successor of u. Clearly, u is a leaf iff all these four tips are identical and equal to u.

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

How rotations affect the 5 paths

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

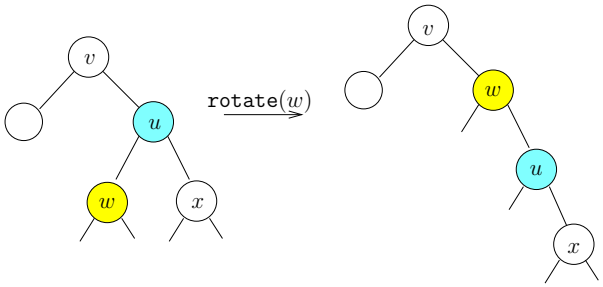


Figure 8: Reduction of the left-spine of u after $\texttt{rotate}(u.\texttt{left}) = \texttt{rotate}(w)$.

(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.

¶20. Deletion. Suppose we want to delete a node u. In case u has at most one child, this is easy to do – simply redirect the parent's pointer to u into the unique child of u (or `nil` if u is a leaf). Call this procedure $Cut(u)$. It is now easy to describe a general algorithm for deleting a node u:

the $Cut(u)$ operation

```
DELETE(T, u):
    Input: u is node to be deleted from T.
    Output: T, the tree with u deleted.
        While u.left ≠ nil do
            rotate(u.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 is shorter. To avoid maintaining spine height information, we can also do this: alternately perform left- and right-rotates at u until one of its 2 spines have length 0. This guarantees that the number of rotations is never more than twice the minimal needed.

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

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

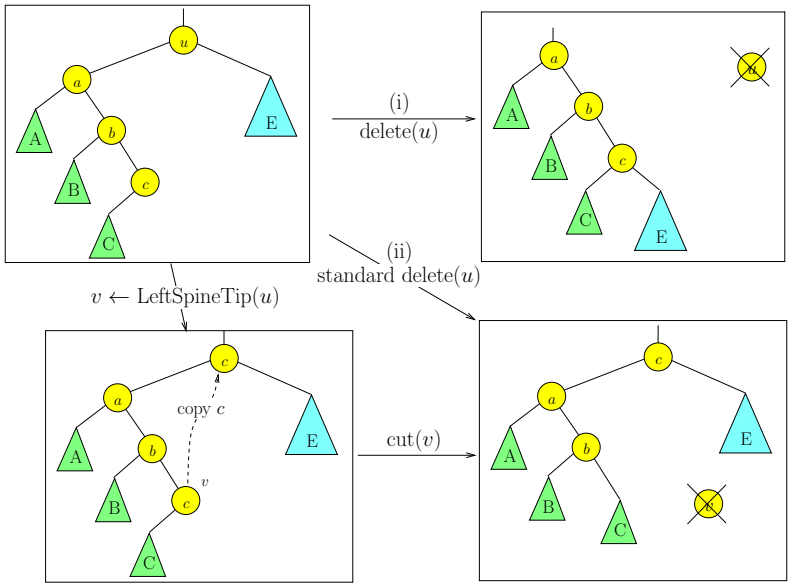


Figure 9: Deletion: (i) Rotation-based, (ii) Standard.

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 (removing the old item in u)
$Cut(v)$.

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

The rotation-based deletion is conceptually simpler, and will be useful for amortized algorithms later. However, the rotation-based algorithm seems to be slower as it requires an unbounded number of pointer assignments. To get a definite complexity benefit, we could perform this rotation in the style of splaying (Chapter VI, Amortization).

¶22. Inorder listing of a binary tree.

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

We leave the simple proof to an Exercise. For example, if T is the binary tree in Figure 2(b), then this lemma assigns the keys $\{1, \ldots, 15\}$ to the nodes of T as in Figure 3(a). In general,

the node that is assigned key i ($i = 1, \ldots, n$) by Lemma 3 may be known as the **ith node** of T. In particular, we can speak of the **first** ($i = 1$) and **last node** ($i = n$) of T. The unique enumeration of the nodes of T from first to last is called the **in-order listing** of T.

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

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

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

In some applications of binary trees, we want to maintain pointers to the successor and predecessor of each node. In this case, these pointers may be denoted $u.\texttt{succ}$ and $u.\texttt{pred}$. Note that the successor/predecessor pointers of nodes is unaffected by rotations. *Our default version of binary trees do not include such pointers.* Let us make some simple observations:

Lemma 4 *Let u be a node in a binary tree, but u is not the last node in the in-order traversal of the tree. Let $\texttt{succ}(u) = v$.*
(i) If $u.\texttt{right} \neq \texttt{nil}$ then v is the tip of the right-spine of u.
(ii) If $u.\texttt{right} = \texttt{nil}$ then u is the tip of the left-spine of v.

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

```
Succ(u):
Output: The successor node of u (if it exists) or nil.
 1.    if u.right ≠ nil    ◁ return the tip of the right-spine of u
 1.1       v ← u.right;
 1.2       While v.left ≠ nil, v ← v.left;
 1.3       Return(v).
 2.    else    ◁ return v where u is the tip of the left-spine of v
 2.1       v ← u.parent;
 2.2       While v ≠ nil and u = v.right,
 2.3            (u, v) ← (v, v.parent).
 2.4       Return(v).
```

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

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

¶25. Merge. To merge two trees T, T' where all the keys in T are less than all the keys in T', we proceed as follows. Introduce a new node u and form the tree rooted at u, with left subtree T and right subtree T'. Then we repeatedly perform left rotations at u until $u.\texttt{left} = \texttt{nil}$. At this point, we can perform $Cut(u)$ (see ¶20). If you like, you can perform right rotations instead of left rotations.

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

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

work out this little detail

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

Theorem 5 *We can do lookup, insert, delete, find successor or predecessor, min or max, merge and split of binary search trees with height at most h in worst-case time $O(h)$.*

Since $h = \Omega(\log n)$ for a tree of size n, the operations in this theorem are optimal if we can 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: Let us call a binary tree T at "zigzag-list" if it has exactly one leaf, and the unique path from the root u_0 to the leaf has the form

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

such that for all $i = 1, \ldots, n-1$, u_i is a left-child iff u_{i+1} is a right child. Describe the binary tree T after performing $\texttt{rotate}^n(u_n)$ (i.e., n repetitions of rotate u_n). What is the height of T? ♢

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 $\mathit{LookUpAll}(u, K)$. ♢

Exercise 3.5: The function Verify(u) is supposed to return `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))
```

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

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

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

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

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

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

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

Exercise 3.12: Open-ended: The problem of implementing `rotate`(u) without using extra storage or in minimum time (previous Exercise) can be generalized. Let G be a directed graph where each edge ("pointer") has a name (e.g., `next`, `prev`, `left`, `right`) taken from a fixed set. Moreover, there is at most one edge with a given name coming out of each node. Suppose we want to transform G to another graph G', just by reassignment of these pointers. Under what conditions can this transformation be achieved with only one variable u (as in `rotate`(u))? Under what conditions is the transformation achievable at 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 $k \geq 1$ repeated executions of the assignment $u \leftarrow \mathit{successor}(u)$ 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 (3), 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,

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

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

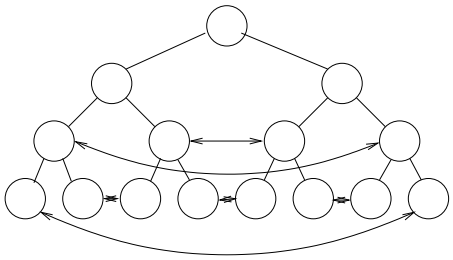


Figure 10: Partners

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 $u_L.\texttt{key} < u.\texttt{key} < u_R.\texttt{key}$. 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

§4. Tree Traversals and Applications

In this section, we describe systematic methods to visit all the nodes of a binary tree. Such methods are called **tree traversals**. Tree traversals provide "algorithmic skeletons" or **shells** for implementing many useful algorithms. We had already seen this concept in ¶6, when implemented ADT operations using linked lists.

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

¶28. 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`).

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

BASE(u):
If $u =$ `nil`, Return.

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

VISIT(u):
Print u.`key`.

In illustration, consider the two binary trees in Figure 2. The numbers on the nodes are keys, 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:

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

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

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

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

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

SORT(A, B):
Input: A an array of n numbers to be sorted
Output: B, sorted output array
    Initialize a BST T with no items.
    For $i = 1, \ldots, n$
        Insert $A[i]$ into T
    $i \leftarrow 1$
    IN-ORDER(T) with suitable $\boxed{\text{VISIT}(u)}$ macro.

The $\boxed{\text{VISIT}(u)}$ amounts to the assignment $B[i\texttt{++}] \leftarrow u.\texttt{key}$.

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

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

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:

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

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

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

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

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

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

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

¶31. 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 BASE and 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

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

computing height in post-order

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

```
VISIT(u)
  if (u.left = nil) then L ← −1.
      else L ← u.left.H.
  if (u.right = nil) then R ← −1.
      else R ← u.right.H.
  u.H ← 1 + max{L, R}.
```

computing depth in pre-order

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

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

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

¶32. Return Shells. For some applications, we want a version of the above traversal routines that return some value. Call them "return shells" here. We illustrate this by modifying the previous postorder shell POST(u) into a new version rPOST(u) which returns a value of type T. For instance, T might be the type integer or the type node. The returned value from recursive calls are then passed to the VISIT macro:

```
rPOST(u)
  rBASE(u).
  L ← rPOST(u.left).
  R ← rPOST(u.right).
  rVISIT(u, L, R).
```

Note that both $rBASE(u)$ and $rVISIT(u, L, R)$ returns some value of type T.

As an application of this rPOST routine, consider our previous solution for computing the height of binary trees. There we assume that every node u has an extra field called $u.H$ that we used to store the height of u. Suppose we do not want to introduce this extra field for every node. Instead of POST(u), we can use rPOST(u) to return the height of u. How can we do this? First, BASE(u) should be modified to return the height of `nil` nodes:

```
rBASE(u):
  if (u=nil) Return(−1).
```

Second, we must re-visit the VISIT routine: *no pun intended*

```
rVISIT(u, L, R)
  Return(1 + max{L, R}).
```

The reader can readily check that rPOST solves the height problem elegantly. As another application of such "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 Lecture. 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 18. ♢

Exercise 4.3: BST reconstruction from node-listings in tree traversals.

(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. ♢

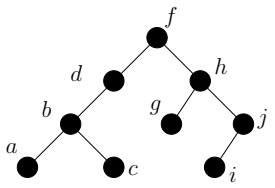


Figure 11:

Exercise 4.4: Tree reconstruction from key-listings in tree traversals. *Be 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 give an algorithm to compute the size of a node u (i.e., the number of nodes in the subtree rooted at u). Give two versions: (a) using a return shell, 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

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

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 +∞ if not BST
▷ Assume u is not nil
  If (u.left ≠ nil)
      L ← maxBST(u.left)
      If (L > u.key or L = ∞) return(∞)
  If (u.right ≠ nil)
      R ← minBST(u.right)
      If (R < u.key or R = ∞) return(∞)
  Return (CheckBST(u.left) ∧ (CheckBST(u.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. Variations on Binary Search Trees

¶33. Non-Standard BSTs This section will consider variations on the binary search trees which we have seen so far. Hence we will call the BST of the previous sections the **standard BST**. Here are some examples of non-standard BST's:

External BST Note that in standard BST's we store items[7] (i.e., key-data pairs) in every node of the tree. In contrast, **external BST** stores items only in their leaves; the

[7] 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.

internal nodes only have keys (without associated data) that serve as "guides" to lead to the items at the leaves.

Implicit Key BST This means that the search keys are not explicitly stored such in the tree. The information stored at each node, however, gives us a basis for deciding whether we should go to its left or right child. This comparison may also depend on some external data not explicitly stored in the node. This will be treated in Chapter VI (on convex hulls) In this application, it is called **parametric search**.

Implicit Pointer BST The child/parent links is a standard BST is explicitly stored at a node. If these links can somehow be computed based on some data about the node, we call them "implicit pointers". The main example is the **binary heap** where the BST on n nodes is represented by an array $A[1..n]$ where each $A[i]$ stores the key of a node u. Using the index i, we can compute the indices of the left and right children of u.

Auxiliary Information We may store useful information in the BST such as height, depth or size or range of keys at the subtree. Of course, we must modify our insertion/deletion algorithms to maintain these information.

Auxiliary Links We can maintain additional pointers such as level links, or successor/predecessor links. One interesting class of such search trees is the **finger tree**.

Before delving into non-standard BST's, we first take two detours to discuss *extended binary trees* and *exogenous search structures*. These discussions provide some context and vocabulary for our treatment of non-standard BSTs.

¶34. Extended binary trees. Underlying a BST is a purely structural object, the binary tree. Before looking into non-standard BST's, we first give an alternative account of these binary trees. It will give us some vocabulary for discussing some non-standard BST's. Following Knuth [9, p. 399], the alternative view of binary trees is called **extended binary trees**. In an extended binary tree, every node has 0 or 2 children; nodes with no children are called[8] **nil nodes** while the other nodes are called **non-nil nodes**. For emphasis, our original version of binary trees will be called **standard binary trees**.

See Figure 12(a) for a standard binary tree and Figure 12(b) for the corresponding extended version. In this figure, we see a common convention (following Knuth) of representing nil nodes by black squares.

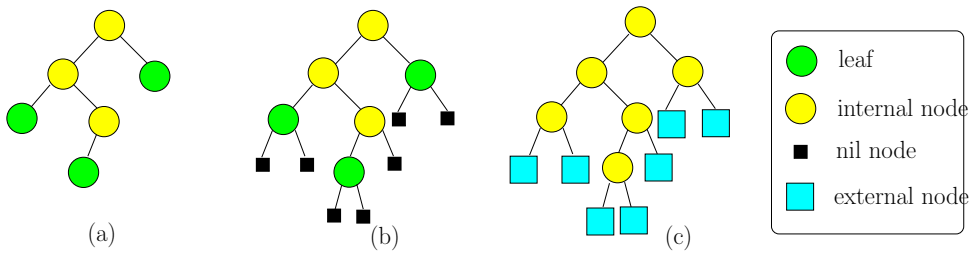


Figure 12: Three versions of binary trees: (a) standard, (b) extended, (c) external

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

[8] 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.

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

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

why care about nil nodes?

The concept of a "leaf" of an extended binary tree is apt to cause some confusion. We shall use the "leaf" terminology as follows: a node of an extended binary tree is called a **leaf** if it is the leaf of the corresponding standard binary tree. Thus a node in an extended binary tree is a leaf iff the node has two nil nodes as children. *Thus a nil node is never a leaf.* In Figure 12(a) and (b), the leaves are colored green.

¶35. External Binary Search Tree. To turn extended binary trees into search structures, we could just store keys in nodes satisfying the BST Property. The resulting "extended BST" is not very different from the standard BST. All we get from the nil nodes in extended BST is some book-keeping convenience (e.g., as sentinels in algorithms). For a more substantial transformation, we now modify extended BST by replacing nil nodes with a new type of node that can store items, calling them **external nodes**. Also the original nodes are now called **internal nodes**. The result is known as an **external binary tree**, and illustrated in Figure 12(c).

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

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

$$u_L.\texttt{key} < u.\texttt{key} \leq u_R.\texttt{key} \tag{9}$$

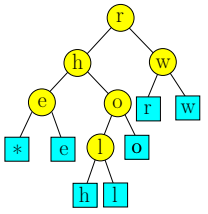
where u_L (resp., u_R) is any node in the left (resp., right) subtree at u. Note that u_L and u_R may be internal or external nodes. The new search structure is called, naturally, an **external BST**. Note that (9) allows $u.\texttt{key}$ may be equal to $u_R.\texttt{key}$ (but to $u_L.\texttt{key}$).

In summary, each node in an external BST is one of two kinds: an internal node that holds a key value and has two non-null children, or an external node that holds an item and has no children. Moreover, the keys in these nodes satisfy (9).

In Chapters V and VI, 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[9] do. Later in §8, we will see another substantial example of external search structure under the topic of (a, b)-search trees.

[9] The needed property is that the set of paths from the root to the items stored in the tree must form a prefix-free set.

The preceding description of external BST's is sufficient for Lookups, but in order to support insertion and deletion, we must answer this question: *where do the keys in internal nodes come from?* When the user inserts an item, a new internal node is created. What is the key in that internal node? We do not assume the insertion algorithm can generate brand new keys automatically; but we allow it to duplicate keys that are already in the BST. Using this duplication ability, we can now provide a solution of external BST's. Before going into details, let us look at the sample external BST shown in the right margin: the leaves store the seven items (represented by their keys only)



An External BST

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

Each of these keys are duplicated in the internal nodes. The sole exception is the minimum key `*` in the tree. This example illustrates the general property we will require in our External BST's:

Partner Property: If e is an external node and $e.\texttt{pred}$ is defined, then

$$e.\texttt{key} = e.\texttt{pred}.\texttt{key}. \tag{10}$$

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

This Partner Property is possible because, in a full binary tree, the number I of internal nodes and the number E of external nodes satisfy the relation $E = I + 1$. Although we assume the Partner Property in this book, such a property is not essential for external BST's: in general, we only have the relation

$$e.\texttt{key} \geq e.\texttt{pred}.\texttt{key}.$$

Without the Partner Property, the key in an internal node may not be associated with any item in the current BST (it probably came from an item that has been deleted). *With the partner property, it is sufficient to only show keys in internal nodes and the minimum key.* In other words, the remaining keys in external nodes can be inferred.

¶36. Exogenous versus Endogenous Search Structures Tarjan [18], 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 13 illustrates his interpretation of this definition for linked lists.

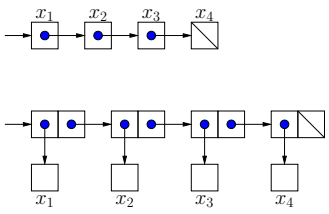


Figure 13: Endogenous/Exogenous

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.

¶37. Implementation of External BST with Partner Property. Let us consider algorithms for insertion and deletions in External BSTs. To support the Partner Property under deletion, it is convenient if each non-minimum external node e stores a pointer $e.\texttt{pred}$ that points to its predecessor (hence partner). For the minimum external node, $e.\texttt{pred} = \texttt{nil}$.

We are now ready to describe the 3 basic algorithms of an external BST: lookup, insert, delete.

```
Lookup(k)
Output: Returns an external node e.
    u ← root
    While (u is an internal node)
        if (u.Key ≤ k), u ← u.right
        else u ← u.left
    Return u
```

Note that when $u.Key = k$, we move to the right child of u.

Lemma 6 *If Lookup(k) returns the node e, then*

$$e.Key \leq k.$$

Proof. Observe that the root-path of e must contain the predecessor $i = e.\texttt{pred}$. Therefore, $i.Key \leq k$ since e lies in the right subtree of i. By the partner property, $i.Key = e.Key$, and thus

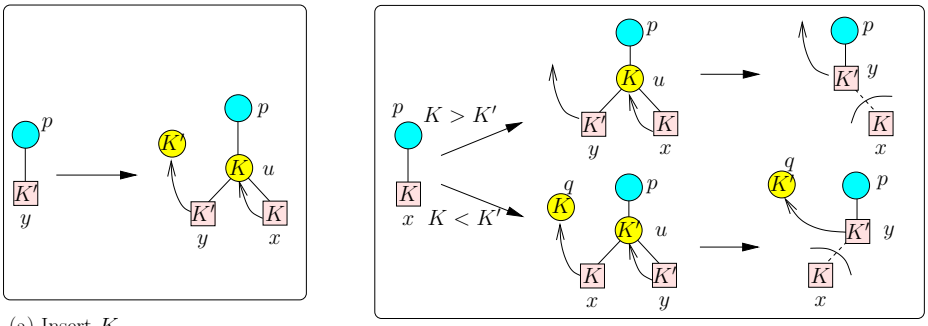
$$e.Key = i.Key \leq k.$$

Q.E.D.

Let us see how insertion works, as illustrated in Figure 14(a). Say the item we want to insert is placed into a new node x, and we begin by a Lookup on $K = x.\texttt{key}$. This leads us to an external node y with key $K' = y.\texttt{key}$. If $K' = K$, then insertion fails. Otherwise, the above observation implies that $K' < K$. The remaining operations of the insertion are now simple. First, we create a new internal node u to take the place of y. If p was the parent of y, we let

u.`parent` point to p. Make y and x the left and right children of u, respectively. Moreover, the key in u will be (a duplicate of) K, and we assign x.`left` $\leftarrow u$ (since u is the predecessor of x). Observe that the predecessor of y is unchanged. The necessary assignments are given here:

$$\begin{aligned}
u: &\quad u.\texttt{parent} \leftarrow y.\texttt{parent}; \quad u.\texttt{left} \leftarrow y; \quad u.\texttt{right} \leftarrow x; \quad u.\texttt{key} \leftarrow x.\texttt{key}\\
x: &\quad x.\texttt{parent} \leftarrow u; \quad x.\texttt{left} \leftarrow u; \quad x.\texttt{right} \leftarrow \texttt{nil};\\
y: &\quad y.\texttt{parent} \leftarrow u;\\
p: &\quad \textsf{If } (p.\texttt{left} = y) \textsf{ then } p.\texttt{left} \leftarrow u, \textsf{ else } p.\texttt{right} \leftarrow u.
\end{aligned}$$



(a) Insert K (b) Delete K

Figure 14: Insertion and Deletion in external BST

In case of the deletion of a key K, we first do a Lookup on K. This leads to an external node x. If x.`key` $\neq K$, then there is nothing to delete. Otherwise, let the parent of x be u. If u is the root of the BST, we may leave this easy case to the reader. Otherwise, let p be the parent of u and y the sibling of x. Essentially, we must delete x and u using these actions:

- The child pointer of p that pointed to u is updated to point at y. This effectively eliminates x and u.
- We update the Partner Property for y. If y is the minimum item in the tree, there is nothing to update. Otherwise, since y takes the place of u, we do:

$$y.\texttt{pred} \leftarrow u.\texttt{pred}, \quad p.\texttt{pred}.\texttt{key} \leftarrow y.\texttt{key}.$$

Deletion is illustrated in Figure 14(b). Assume that $K = x$.`key` and $K' = y$.`key`.

Here are two useful variations of the above algorithms:
(a) The Partner Property introduces redundancy among the keys. To remove the redundancy, we simply avoid storing keys in external nodes. In storing an item, i.e., a (key,data) pair, we store only the data in the external node u, but the key is stored in the predecessor of u.
(b) We can ignore the Partner Property – so the predecessor of an external node u need not have the same key as u. In the above algorithms we no longer need to update keys in predecessors (in fact, we do not need predecessor pointers). But the above insertion algorithm has an extra case to handle. However, when we insert a new item, its parent (which is a new node) will initially have the key of the new item. Over time, because of deletions, the internal keys might not be a duplicate of any key in an external node.

¶38. Auxiliary Information. In many 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).

¶39. Duplicate keys. We normally assume that the keys in a BST are distinct. Let us now relax this requirement. In other words, let us allow duplicate keys. We continue to assume that items are stored in internal nodes as well as in leaves of the BST. Let `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.

Next, we design a recursive procedure called $FindAll(u)$ that will return a list *containing all the nodes in the subtree at u that has the same key as $u.Key$.* To find all items that has a given key K, we can simply compose $FindAll$ with `lookUp`:

$$FindAll(\texttt{lookUp}(K)).$$

We next describe the operations of $FindAll(u)$ where node $u.Key = K$. Here is the easy recursive procedure. If u has no right-child then $FindAll(u) = (u)$, i.e., the list has only one node u. Otherwise, let v be the successor of u, i.e., v is the tip of the right-spine of u. Since v is the successor of u, it follows that $v.Key \geq K$, i.e., either $v.Key = K$ or $v.Key > K$. Then

$$FindAll(u) = \begin{cases} (u) & \text{if } v.Key > K \\ (u; FindAll(v)) & \text{else.} \end{cases}$$

We leave it as an exercise to prove the correctness of this procedure.

If we have successor pointers, then v can be located in $O(1)$ time. Then the complexity of $FindAll(u)$ is $O(L)$ where L is the length of the list returned by $FindAll(u)$. This suggests that for BST's with duplicates, we ought to maintain successor pointers.

¶40. Implicit Pointer BST. Pointers take up space, and we would like to reduce them. In discussing rotations, we noted that parent pointers could be omitted in all our algorithms for binary search trees. See the Exercise for these algorithms. But the ultimate step in this direction is to eliminate all pointers! Such trees are sometimes called **implicit trees**. Without explicit pointers to indicate parent/child relationships, we must use index arithmetic. So implicit trees are normally stored in an array. The most famous example is the **heap structure**: this is defined to be binary tree whose nodes are indexed by integers following this rule: the root is indexed 1, and if a node has index i, then its left and right children are indexed by $2i$ and $2i + 1$, respectively. Moreover, if the binary tree has n nodes, then the set of its indices is the set $\{1, 2, \ldots, n\}$. A heap structure can therefore be represented naturally by an array $A[1..n]$, where $A[i]$ represents the node of index i. If, at the ith node of the heap structure, we store a key $A[i]$ and these keys satisfy the **heap order (HO) property** for each $i = 1, \ldots, n$,

$$HO(i): \quad A[i] \leq \min\{A[2i], A[2i + 1]\}. \tag{11}$$

In (11), it is understood that if $2i > n$ (resp., $2i + 1 > n$) then $A[2i]$ ($A[2i + 1]$) is taken to be ∞. Then we call the binary tree a **heap**. Here is an array that represents a heap:

$$A[1..9] = [1, 4, 2, 5, 6, 3, 8, 7, 9].$$

In the exercises we consider algorithms for insertion and deletion from a heap. This leads to a highly efficient method for sorting elements in an array, in place.

In general, implicit data structures are represented by an array with some rules for computing the parent/child relations. By avoiding explicit pointers, such structures can be very efficient to navigate.

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 with Partner Property, but avoid key redundancy (i.e., do not store keys in the external nodes). ♢

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: The text noted that External BST's need not maintain the Partner Property. Provide the Lookup/Insert/Delete Algorithms. What is the extra case needed in Insertion? ♢

Exercise 5.7: 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 5.8: 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 5.9: Prove that the $FindAll(u)$ procedure in the text is correct. ♢

Exercise 5.10: 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 5.11: Implement the algorithms for insert, delete, etc, on binary search trees that maintains the size of subtree at each node. ♢

Exercise 5.12: Repeat the previous question, but where we maintain the average value of the keys in each subtree. ♢

Exercise 5.13: 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 5.14: 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.

♢

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:

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

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 (4) and(5)), 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 this idea.

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 15 shows the first few min-size AVL trees. Of course, we can exchange 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 15.

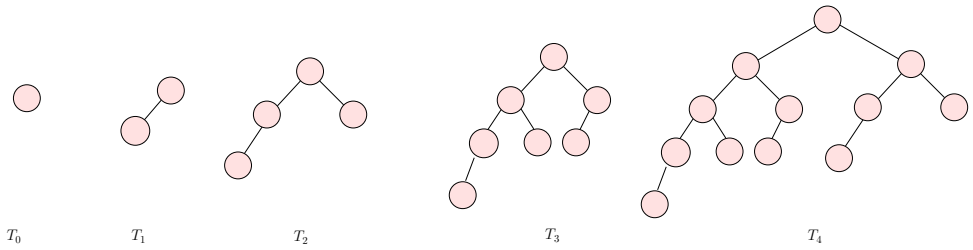


Figure 15: Canonical min-size AVL trees of heights 0, 1, 2, 3 and 4.

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

$$\mu(h) = 1 + \mu(h-1) + \mu(h-2), \qquad (h \geq 1). \tag{12}$$

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 (12) 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 (12), we have $\mu(h) > 2\mu(h-2)$ for $h \geq 1$. It is then easy to see by induction that $\mu(h) > 2^{h/2}$ for all $h \geq 1$. 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

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

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 positive 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 $h \geq 0$, we have*

$$\phi^{h+1} + \widehat{\phi} \leq \mu(h) < 2\phi^h. \tag{13}$$

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

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

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

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

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

Q.E.D.

This lemma tells us that $\mu(h) + 1 = A\phi^h$ where $\phi \leq A < 2$. We can obtain the correct value of A in an exercise. 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

$$\mu(h) \leq n$$

by definition of $\mu(h)$. The lower bound in (13) then implies $\phi^h \leq n$. Taking logs, we obtain

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

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

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

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

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

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

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

Must use standard deletion!

REBALANCE PHASE: Let x be parent of the node that was just inserted, or just *cut* during deletion, in the UPDATE PHASE. The path from x to the root will be called the **rebalance path**. We now move up this path, rebalancing nodes along this path as necessary.

It remains to give details for the REBALANCE PHASE. If every node along the rebalance path is AVL-balanced, then there is nothing to do in the REBALANCE PHASE. Otherwise, let u be the first unbalanced node we encounter. It is clear that u has a balance of ± 2. In general, we fix the balance at the "current" unbalanced node and continue searching upwards along the rebalance path for the next unbalanced node. By symmetry, we may suppose that u 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 16.

u = unbalanced

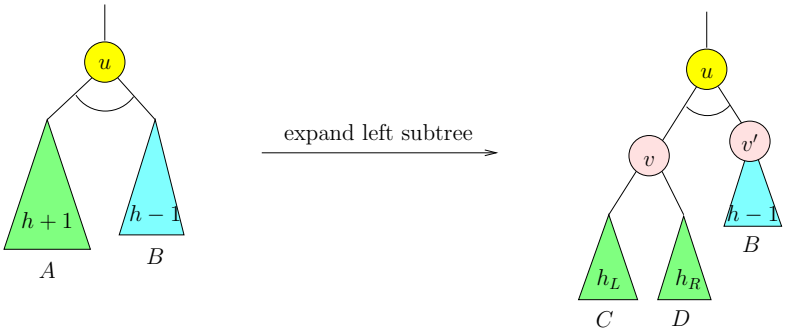


Figure 16: Node u is unbalanced after insertion or deletion.

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.

¶42. 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)

$$h+1, \quad h, \quad h-1. \tag{14}$$

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 17.

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 u is now $h+1$. We call this the "single-rotation" case, in contrast to the next case.

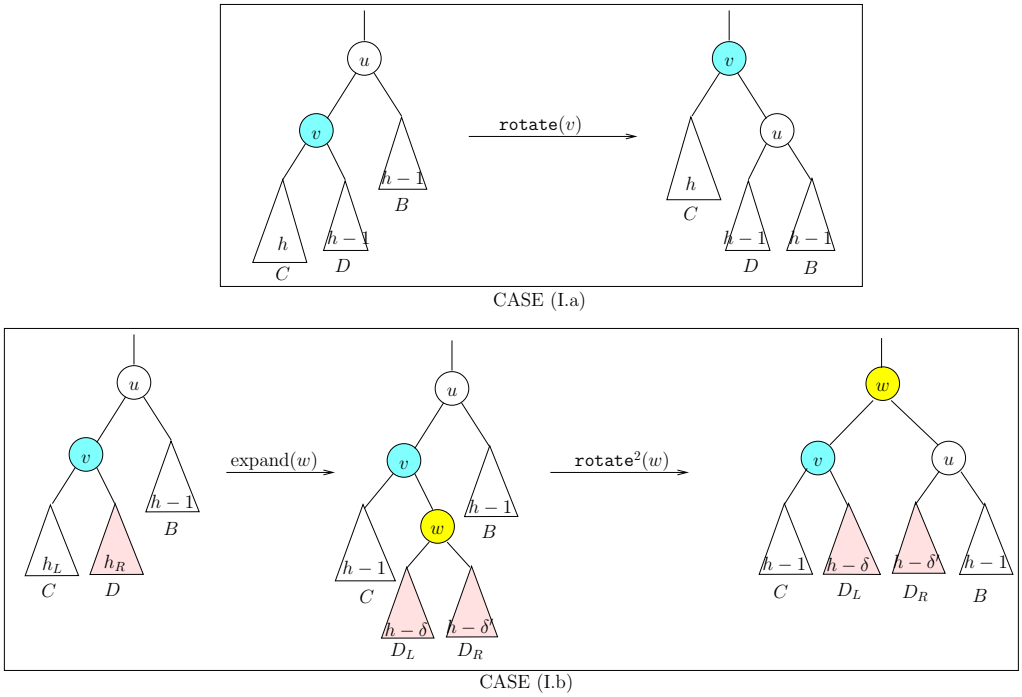


Figure 17: AVL Insertion: CASE (I.a) and CASE (I.b)

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. The two children of w will have heights $h - \delta$ and $h - \delta'$ where $\delta, \delta' \in \{1, 2\}$. 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 (14)), there are no unbalanced nodes further up the path to the root. Thus the insertion algorithm terminates with at most two rotations.

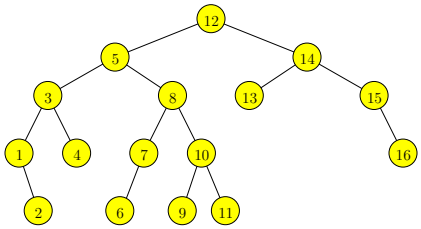


Figure 18: An AVL tree

For example, suppose we begin with the AVL tree in Figure 18, and we insert the key 9.5. This yields the unbalanced tree on the left-hand side of Figure 19. 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 19, 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 19.

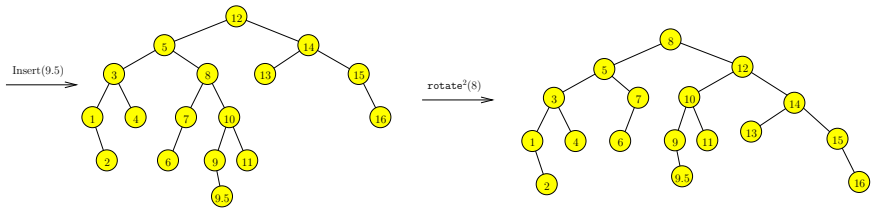


Figure 19: Inserting 9.5 into an AVL tree

¶43. Deletion Rebalancing. Suppose the imbalance in Figure 16 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.

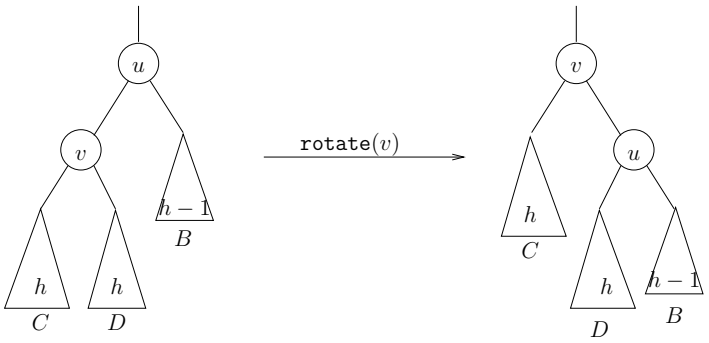


Figure 20: CASE (D.c): `rotate`(v)

CASE (D.c): $h_L = h_R = h$. This case is new, and is illustrated in Figure 20. 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 18. 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 21. 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 21.

Both insertion and deletion take $O(\log n)$ time. In case of deletion, we may have to do

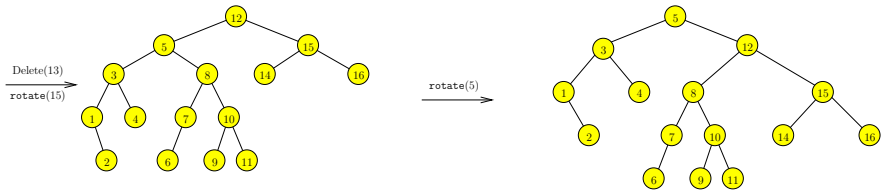


Figure 21: Deleting 13 from the AVL tree in Figure 18

$O(\log n)$ rotations but a single or double rotation suffices for insertion.

¶44. Almost-AVL Trees. Let us summarize the global correctness argument for the preceding algorithms. It is useful to introduce a terminology. A binary tree T is **almost-AVL** if every node in T is AVL-balanced with one single exception: the exceptional node, known as the **defective node**, has children whose heights differ by exactly 2. The above arguments show that *given an almost-AVL tree, by a single or double rotation, we can make it AVL or almost-AVL. In case it becomes almost-AVL, the depth of the defective node is reduced by* 1. Since the depth of the defective node is reduced, this process must eventually halt.

¶45. 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 to indicate three possible balance states $(00, 01, 10)$ for each node u. Let 00 mean that u.`left` and u.`right` have the same height, and 01 mean that u.`left` has height one less than u.`right`, and similarly for 10. In simple implementations, we could just use an integer to represent this information. Of course, the state 11 is unused. So theoretically, we only need $\lg 3 = 1.58\ldots$ bits per node. See Exercises.

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

¶46. Merging AVL Trees. 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 less than the keys in T_R. We want to merge them into a new AVL tree in time $O(\log n)$ where the size of T_L and T_R are $O(n)$. The steps are illustrated in Figure 22.

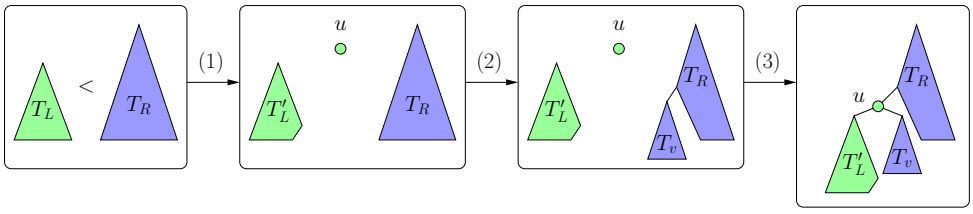


Figure 22: Steps in merging T_L and T_R.

(1) Assume the height of T_L is at most the height of T_R. We first delete the tip of the right-path of T_L. Let the resulting tree be T'_L and the deleted node be u. We may assume that the height h of T'_L is at least 1. (If $h = 0$, it means T_L has only two nodes in the first place, and we could simply insert the 2 nodes of T_L into T_R.)

(2) Now, walk down the left-path of T_R until you find the first node v whose height is $\leq h$ or v is tip of the left-path. Note that the heights of consecutive nodes along the left-path differs by 1 or 2. Thus there are logically three possibilities:

- $\texttt{Ht}(v) = h$.
- $\texttt{Ht}(v) = h - 1$.
- $\texttt{Ht}(v) > h$ and v is the tip of the left-path.

We claim that the last case cannot happen. Otherwise, the tip v has height ≥ 2 and has no left-child. This implies v is not AVL-balanced, contradiction.

(3) Now construct a tree T_u rooted at u with left-subtree equal to T'_L and right subtree equal to T_v. Note that T_u is AVL of height $1 + h$. Moreover, we will "graft" T_u into T_R by replacing the subtree at v by T_u. Now T_R is AVL balanced except that the nodes along the path from u to the root might have balance of $+2$. We rebalance along the root-path of u in the manner of AVL insertion.

To summarize: Let $T_R.\texttt{merge}(T_L)$ denote the preceding sequence of operations. Then step (1) amounts to $u \leftarrow T_L.\texttt{lookUp}(k)$ followed by $u \leftarrow T_L.\texttt{delete}(u)$. Step (2) searches the right path of T_R for the first node v of height at least the (updated) height of T_L. Let T_u be the tree rooted at u with left subtree equal to the (updated) T_L, and right subtree equal to T_v (the subtree rooted at v). Step (3) replaces T_v by T_u, followed by rebalance along the root-path of u. Now, T_R has the merged result, but T_L is set to the empty tree.

We have proved:

Lemma 9 *Given two AVL trees T_L and T_R such that $T_L < u < T_R$, we can merge them into a single AVL tree in time $O(|ht(T_L) - ht(T_R)|)$.*

¶47. 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 23 for an illustration with $t = 5$.

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.* In particular, the trees in L are sorted in the form $S_{i_1} < x_{i_1} < S_{i_2} < x_{i_2} < \cdots$ 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

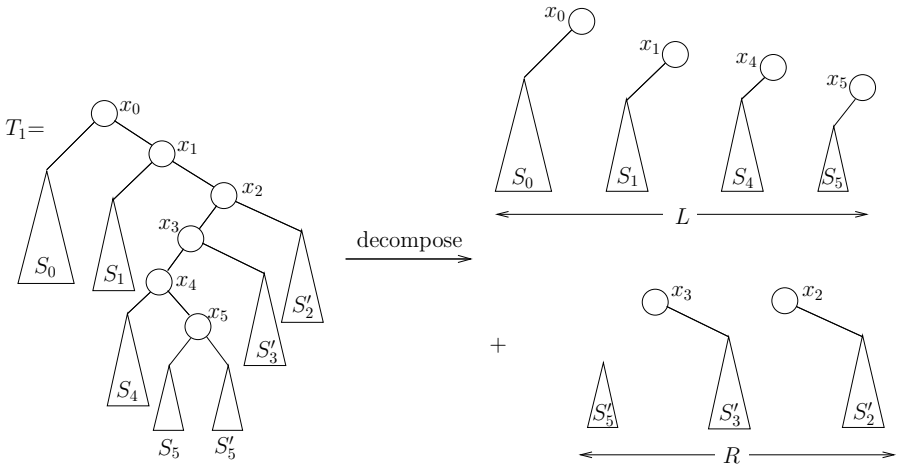


Figure 23: Split: decomposition of T.

Figure 23, if x_i and S_i are put in L, we display them together as one tree with x_i as root; a similar remark holds if x_i, S'_i are put in R.

Composition Phase: We must now combine the trees in L into T_L, and similarly combine the trees in R into T_R. Let us focus on the set of trees in L (R is similarly treated). It is not hard to do see that 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 let us now show that $O(\log n)$ suffices. Let us note that the trees in L can in fact be relabeled and ordered as follows:

$$L_1 > y_1 > L_2 > y_2 > \cdots > L_\ell > y_\ell > L_{\ell+1} \tag{15}$$

where the y_j's are singleton trees (coming from the x_i's), and

$$\texttt{Ht}[L_1] \geq \texttt{Ht}[L_2] \geq \cdots \geq \texttt{Ht}[L_{\ell+1}] \geq 0. \tag{16}$$

Note that $L_{\ell+1}$ could be empty. For instance, the set L in Figure 23 is relabeled as in Figure 24 with L_5 as an empty tree.

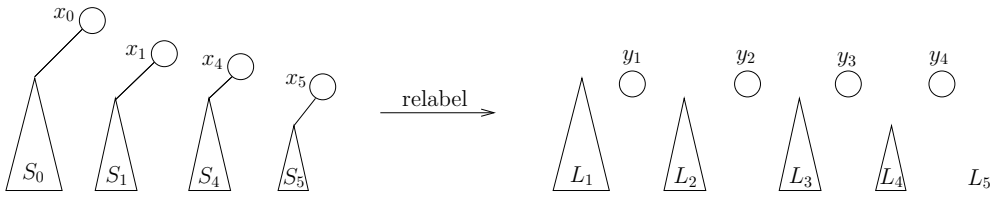


Figure 24: Relabeling the trees in the set L.

The basic idea is to merge the trees from right to left. More precisely: initially, we merge $L_\ell, y_\ell, L_{\ell+1}$ and let the result be denoted $L'_{\ell-1}$. Inductively, assume that

$$L_{i+1}, y_{i+1}, \ldots, y_\ell, L_{\ell+1}$$

have been merged into a tree denoted by L'_i. If $i > 0$, we continue inductively by merging

$$L_i, y_i, L'_i \tag{17}$$

to form L'_{i-1}. Otherwise, $i = 0$ and we have merged all the trees in L into one AVL tree L'_0. This completes the merge algorithm.

We have made $O(\log n)$ merges, each in time $O(\log n)$, for a total time of $O(\log^2 n)$. But we prove a better bound:

Lemma 10 *The time to merge all the trees in L is $O(\texttt{Ht}[L_1] + \ell) = O(\log n)$.*

The basic idea is as follows: the inductive merge step (17) takes time

$$\mathcal{O}(1 + |\texttt{Ht}[L_i] - \texttt{Ht}[L'_i]|) \tag{18}$$

But it is not hard to see that $\texttt{Ht}[L'_i] \leq 1 + \texttt{Ht}[L_{i+1}]$. Hence (18) implies that the ith merge takes time

$$\mathcal{O}(2 + \texttt{Ht}[L_i] - \texttt{Ht}[L_{i+1}]).$$

By telescopy, the overall cost of merging is $\mathcal{O}(\texttt{Ht}[L_1] + \ell) = \mathcal{O}(\log n)$, as claimed.

¶48. Relaxed Balancing. Larsen [10] 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).

It is evident that 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 [6] investigated empirically such trees for $k \leq 5$, showing that for modest increase in lookup time, the amount of restructuring is decreased by a factor of 10.

Exercises

Exercise 6.1: 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 21.
(b) Delete the key 4 from the final AVL tree in Figure 21. NOTE: part(b) is independent of part(a). ♢

Exercise 6.2: Give an algorithm to check if a binary search tree T is really an AVL tree. Your algorithm should take time $O(|T|)$. Use shell programming. ♢

Exercise 6.3: Figure 25 shows two rebalance paths (see ¶III.39) of the form

$$(u_0(-), u_1(0), u_2(+), u_3(+), u_4(+)u_5(-))$$

where u_0 is the root, and the balance information at each node is indicated as $0/-/+$. The 2 paths differ in their left-or-right child relation of successive nodes, as seen in Figure 25. The paths are pink in Figure 25, and the sibling of each u_i is the blue node v_i.
(i) AVL Insertion for Figure 25(a): suppose there is an unbalance at u_3 in the Rebalance Phase. Explain what will happen in the rest of the Phase.
(ii) Repeat part (i) but using Figure 25(b).
(iii) AVL Deletion for Figure 25(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 25(b). ♢

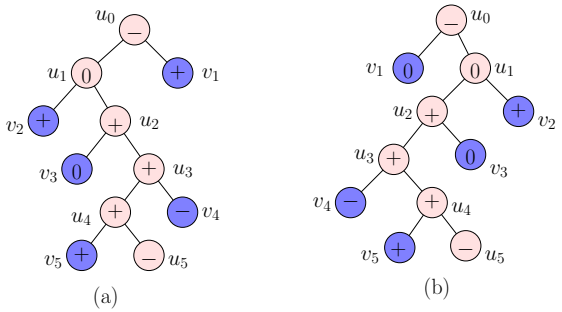


Figure 25: Two rebalance paths

Exercise 6.4: Draw an AVL tree with 12 nodes such that, by deleting one node, you achieve these effects (respectively):
(a) Rebalancing requires two double-rotations.
(b) Rebalancing requires one single rotation and one double-rotation.
(c) Rebalancing requires two single rotations.

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

Exercise 6.5: What is the minimum number of nodes in an AVL tree of height 10? ♢

Exercise 6.6: 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 $h \geq 2$, with $\mu(0) = 1, \mu(1) = 2$.
(a) First consider a general linear recurrence of the form

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

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

$$t(n) = \sum_{i=1}^{k} a_i t(n-i). \tag{19}$$

by[10] 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

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

where $\phi, \widehat{\phi}$ are the roots of $x^2 - x - 1$ and A, B are $\frac{2}{\sqrt{5}} \pm 1 = 1.8944, -0.1056$.
(c) (Yuejie Wang, f2020) The transformation of part(a) has limitations. Show that the homogeneous linear recurrence (19) can always be achieved by $t(n) = T(n) + cn^s$ for some constant c and some $0 \leq s < k$. ♢

[10] 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 $t(n)$.

Exercise 6.7: (Lixuan Zhu, f'2013) Show that the exact solution for the non-homogeneous Fibonacci recurrence

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

is

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

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 6.8: 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 $\phi^{h+1} < \mu(h) + 1 \leq 2\phi^h$. ♢

Exercise 6.9: 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 6.10: Consider the AVL tree in Figure 26.

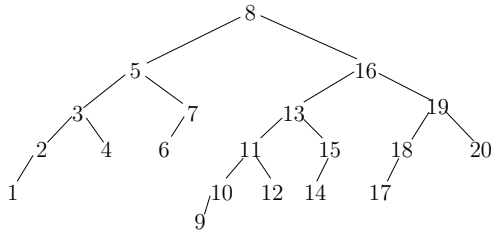


Figure 26: An AVL Tree for deletion

(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 26

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 6.11: (a) Draw two AVL trees, both of height 4. One has maximum size and the other has minimum size.
(b) Starting with an empty AVL tree, insert the following set of keys, in this order:

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

Now delete key 9. Please show the tree at the end of each operation. ♢

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 26? ♢

Exercise 6.13: 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? ♢

Exercise 6.14: 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 $n \to \infty$. ♢

Exercise 6.15: (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. ♢

Exercise 6.16: 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 $19, 18, 17, 16, 15, 14, 13, 12, 11$. ♢

Exercise 6.17: 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 $n \geq 1$. ♢

Exercise 6.18: 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 (5) and $\mu(h)$ in (12).
(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? ♢

Exercise 6.19: Show that for each n in the range $[\mu(h), M(h)]$ we can find an AVL tree of height h of size n. ♢

Exercise 6.20: 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 6.21: Draw two AVL trees, with n keys each: the two trees must have different heights. Make n as small as you can. ♢

Exercise 6.22: 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 6.23: Let $\overline{\mu}(h)$ be the number of *n*on-isomorphic *m*in-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 6.24: Improve the lower bound $\mu(h) \geq \phi^h$ by taking into consideration the effects of "+1" in the recurrence $\mu(h) = 1 + \mu(h-1) + \mu(h-2)$.
(a) Show that $\mu(h) \geq F(h-1) + \phi^h$ 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 $h \geq 2$.
(b) Further improve (a). ♢

Exercise 6.25: Prove the following connection between ϕ (golden ratio) and F_n (the Fibonacci numbers):

$$\phi^n = \phi F_n + F_{n-1}, \qquad (n \geq 1)$$

Note that we ignore the case $n = 0$. ♢

Exercise 6.26: 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 6.27: 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 6.28: 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 6.29: 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 6.30: Let $AVL[2]$ trees be simply 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 6.31: Prove that the family $AVL[k]$ (for any $k \geq 1$) is a balanced family. ♢

Exercise 6.32: Give tight bounds on the height of $AVL[k]$ ($k \geq 1$) 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 6.33: 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 6.34: 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 $h \leq 5$.
(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 6.35: 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. ♢

Exercise 6.36: (Open) Let a node u of height $h+1$ have four grandchildren with heights $\{h_0, h_1, h_2, h_3\}$. In an AVL tree, we may assume $h_0 = h-1$. What about the other 3 heights? We can have $\{h_1, h_2, h_3\} = \{h-1-\delta, h-2, h-2-\delta'\}$ where δ, δ' are $0, 1$ values. Suppose we now enforce the possible values for $\{h_1, h_2, h_3\}$. Explore such trees and algorithms. ♢

End Exercises

§7. Ratio Balanced Trees

¶49. We consider another form of balance in binary trees. Instead of balancing the heights of the left and right subtrees, we now balance their sizes. Typically, we want the ratio of the left and right sizes to be within a fixed range $(\rho, 1/\rho)$ for some $0 < \rho < 1/2$. These are called **weight balanced trees** and were first introduced by Nievergelt and Reingold (1973). We will see that their insertion/deletion algorithms are simpler than in AVL trees, but the analysis is more involved. It turns out that weight balancing is more "robust" than height balancing in a certain sense that is crucial for applications such as multidimensional range search trees [12]. We will also look at an interesting combination of weight and height balancing called **logarithmic BST** from Roura [17].

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 `nil`nodes in the extended binary tree corresponding to T (see ¶34). Figure 27(a) shows a binary tree and Figure 27(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

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

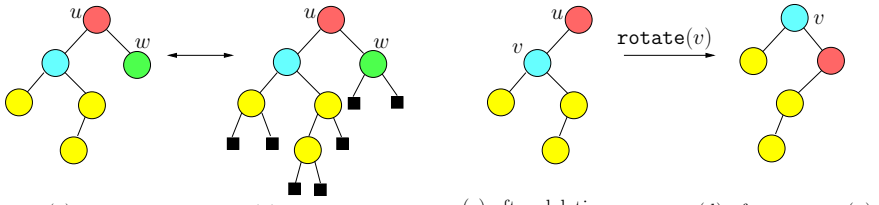
Let us see why we prefer `esize` to `size` in the definition of ratio(u). The recursive equations for `esize` and `size` are thus:

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

The "$1 + \cdots$" term in the definition of $\texttt{size}(u)$ is inconvenient when computing ratios of sizes. Moreover, when taking ratio of two sizes, we might divide by a zero size. On the other hand, `esize` is at least 1, and so such ratios are always well-defined.

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

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



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

Figure 27: (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.

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.

Let $\beta \in (0, \frac{1}{2}]$. Nievergelt and Reingold defined the **root balance** of u to be $B(u) := \texttt{esize}(u.\texttt{left})/\texttt{esize}(u)$. For instance, the root u in Figure 27(a) has ratio 5/2 but root balance 5/7. A binary tree T is said to have **β-bounded balance** if[11]

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

connection between bounded balance and ratio balance: T is β-bounded balanced iff it is $\beta/(1-\beta)$-ratio balanced. Conversely, T is ρ-ratio balanced iff if is $\rho/(1+\rho)$-bounded balanced. In view of this, we can freely go back and forth between the two concepts. We prefer to use the ratio balance concept because it offers some[12] pedagogical advantage.

Bounded balanced trees are also known as **weight balanced trees**. Following Nievergelt and Reingold, a β-bounded balanced trees is said to be in "$BB[\beta]$". By analogy, we say a ρ-ratio balanced is in "$RB[\rho]$". It will be convenient to define

$$\overline{\beta} := 1 - \beta, \qquad \widetilde{\rho} := 1 + \rho. \tag{20}$$

Using these notations, the above connection between bounded balance and ratio balance can be written in the symmetrical form

$$\beta = \rho/\widetilde{\rho}, \qquad \rho = \beta/\overline{\beta}.$$

E.g., if $\beta = 1/4$ then $\widetilde{\beta} = 3/4$ and therefore $\rho = (1/4)/(3/4) = 1/3$. Thus, $BB[1/4] = RB[1/3]$.

Let us look at 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 $u.\texttt{left} = \texttt{nil}$, and ratio($u$) = 2 if $u.\texttt{right} = \texttt{nil}$. This shows that if T is ρ-ratio balanced and $\rho \geq 1/2$, 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) $\in \{1/3, 1, 3\}$.

In our development, we will focus on the class $RB[1/3]$, or equivalently, on $BB[1/4]$. It can be verified that for trees of size up to 5, the AVL trees and the $RB[1/3]$ trees are co-extensive: a tree is AVL iff it is $RB[1/3]$. But Figure 27(a) shows a binary tree of size 6 that is $BB[1/3]$ but not AVL. This tree is not AVL because we know that the smallest AVL tree of height 3 has size $\mu(3) = 7$.

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

[12] We can work with ratios that are integers, e.g., ratio(u) = 3. Also the critical ratio β_0 is not an algebraic integer but critical ratio ρ_0 is an algebraic integer.

Weight balanced trees were introduced by Nievergelt and Reingold [16], with corrected analysis by Blum and Mehlhorn [4]. Nievergelt and Reingold realized that β must not get too close to $\frac{1}{2}$: in fact, $\beta < 1 - \frac{1}{\sqrt{2}}$ is necessary. Blum and Mehlhorn observed that the rebalance rule also run into problems when the tree size is too small, and their solution is to assume β is bounded away from 0. It is intuitively clear why this is necessary: any simple rebalance rule applied to a small tree could introduce a large discrete change that violates $BB[\beta]$ for small β. Thus, the rebalancing theorem of Blum and Mehlhorn requires $2/11 < \beta < 1 - \frac{1}{\sqrt{2}}$. Our notes provide a simplified and complete analysis for $BB[1/4]$ trees.

¶50. Basic Properties To show that $RB[\rho]$ trees is a balanced family, let $H(n)$ denote the maximum height of a $RB[\rho]$ tree of size $\leq n$.

this definition works even for non-integer n: $H(n) = H(\lfloor n \rfloor)$

To bound the height of $RB[\rho]$ trees, as usual, let $\mu(h)$ denote the minimal `esize` of a $RB[\rho]$ tree of height h. For example, if $\rho = 1/3$, then we see that

$$\mu(0) = 2, \quad \mu(1) = 3, \quad \mu(2) = 4, \quad \mu(3) = \cdots$$

(FIGURE) To get a recurrence for $\mu(h)$, wlog, assume that the left subtree has `esize` $\mu(h)$, and the right subtree has esize R where $R < \mu(h)$. We want the minimal integer R satisfying $R/\mu(h) > \rho$, or

$$R > \mu(h)\rho. \tag{21}$$

There are two cases: if ρ irrational, then we can ensure (21) by choosing

$$R := \lceil \mu(h)\rho \rceil . \tag{22}$$

This choice is minimal for any h. In case ρ is irrational, then $R = \lceil \mu(h)\rho \rceil$ is sufficient. Unfortunately, this will not do in case ρ is rational: in case $\mu(h)\rho$ is an integer, (22) does not ensure the strict inequality (21). Note that in most applications, we are interested in rational ρ.

Lemma 11 *Let $\rho = p/q$ a rational number in reduced form.*

$$\mu(1+h) = 1 + \mu(h) + \lceil (1 + p\mu(h))/q \rceil .$$

Equivalently,

$$\mu(1+h) = \left\lceil \frac{1 + q + (p+q)\mu(h)}{q} \right\rceil .$$

Proof. The lemma amounts to the claim that the integer expression

$$R = \lceil (q^{-1} + \mu(h))\rho \rceil$$

is the minimal one that satisfies (21). We have

$$\begin{aligned} R > \mu(h)p/q \quad &\iff \quad R \geq q^{-1} + \mu(h)p/q \quad (\text{since } \mu(h)p \text{ is an integer}) \\ &\iff \quad R \geq \left\lceil \tfrac{1+p\mu(h)}{q} \right\rceil \quad (\text{since } R \text{ is an integer}). \end{aligned}$$

This leads to the recurrence

$$\mu(h+1) = \left\lceil 1 + \mu(h) + \frac{1 + p\mu(h)}{q} \right\rceil = \left\lceil \frac{q + q\mu(h) + (1 + p\mu(h))}{q} \right\rceil = \left\lceil \frac{1 + q + (p+q)\mu(h)}{q} \right\rceil .$$

Q.E.D.

Lemma 12 (Height Bound)

$$H(n) = \log_{1+\rho}(n+1) - O(1).$$

Proof. Let u be the root of $RB[\rho]$ tree with esize $n+1$. Then $n+1 = a+b$ where $\texttt{esize}(u.\texttt{left}) = a$ and $\texttt{esize}(u.\texttt{right}) = b$. WLOG, let $a \leq b$ and hence the nontrivial inequality is $\rho < a/b$. Thus $b\rho < a = n+1-b$, or $b(1+\rho) < 1+n$ or $b < \frac{1+n}{1+\rho}$. Since b is an integer, this is equivalent to

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

Let $\beta(n) := \left\lceil \frac{1+n}{1+\rho} \right\rceil - 1$. It follows that $H(n)$ satisfies the exact recurrence

$$H(n) = 1 + H(\beta(n)) \tag{23}$$

Expanding, $H(n) = 2 + H(\beta^2(n)) = \cdots = i + H(\beta^i(n))$. Note that $\beta^2(n) = \left\lceil \left\lceil \frac{1+n}{1+\rho} \right\rceil /(1+\rho) \right\rceil - 1 \geq \left\lceil \frac{1+n}{(1+\rho)^2} \right\rceil - 1$. By induction on i, we get $\beta^i(n) \geq \left\lceil \frac{1+n}{(1+\rho)^i} \right\rceil - 1$. This proves that

$$H(n) \geq \log_{\widetilde{\rho}}(n+1) - O(1)$$

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

It remains to prove an upper bound on $H(n)$. Note that $\beta(n) < \frac{1+n}{1+\rho}$ implies $H(n) \leq 1 + H(\left\lceil \frac{1+n}{1+\rho} \right\rceil)$. By induction on i, we get $H(n) \leq i + H(\left\lceil \frac{1+n}{(1+\rho)^i} \right\rceil)$, and thus

$$H(n) \leq \log_{\widetilde{\rho}}(n+1).$$

Q.E.D.

This lemma shows that $RB[\rho]$ is a balanced family. We now claim a new kind of height balance condition, to be contrasted with the AVL-type height balance:

Lemma 13 (Ratio of Heights) *If u is a node in a $RB[\rho]$ tree, then the ratio of the heights of $u.\texttt{left}$ and $u.\texttt{right}$ can be arbitrarily close to $\lg(1+\rho)$. Moreover, it is at most $\lg(1+\rho)$.*

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

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

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

To see this, for each $h \geq 0$, there is an AVL tree T_h of height h and size $\mu(h) \leq 2\phi^h - 1$ (see (13)). Let S_h denote the perfect binary tree of height h and size $2^{h+1} - 1$. Consider the

AVL tree whose left subtree is T_h and right subtree is S_h. The ratio balance of the root is $\leq 2\phi^h/2^{h+1}$, which is asymptotically $(\phi/2)^h$. Since $(\phi/2)^h \to 0$ as $h \to \infty$, the ratio balance of AVL trees can be arbitrarily close to 0. This proves our claim.

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

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

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

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

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

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 $S = \{x_1, \ldots, n\} \subseteq \mathbb{R}^2$ of points 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** $u.\texttt{aux}$ 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 $v = u.\texttt{left}$, as in Figure 30(i). After rotation, the auxiliary structure $v.\texttt{aux}$ can be taken from the previous value of $u.\texttt{aux}$. However, the new value of $\texttt{esize}(u)$ is now $b+c$, and $u.\texttt{aux}$ must be a new auxiliary BST tree obtained as a merge of the original $v.\texttt{right.aux}$ with the original $u.\texttt{right.aux}$. 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.

height rebalancing is fragile; ratio rebalancing is robust!

With an AVL tree or 2-3 trees (say), this update complexity cannot be[13] 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.

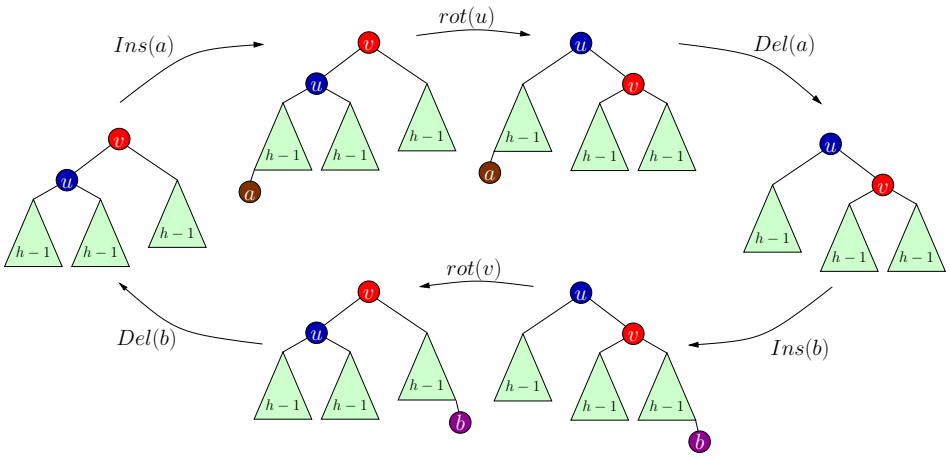


Figure 28: Cycle: $(Ins(a) \to Del(a) \to Ins(b) \to Del(b))$

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.

¶52. 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 the treatment of AVL trees, where we focus on height balance 1 instead of some arbitrary (but fixed) height balance of 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 $(1/3, 3)$.

In practice, it is seldom the case that we need to use more than one choice of ρ. Nevertheless see [7] in the following box.

[13] See Theorem 3.1, [19, p.608].

Implementation and Correctness. Yoichi Hirai and Kazuhiko Yamamoto [7] 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 leaf x of T. 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. We therefore move up this path, rebalancing any such node. The following Rebalance procedure accomplishes this:

```
    Rebalance(u)
1.  While (u ≠ u.parent)
2.        u ← Rebalance-Step(u)
3.        u ← u.parent
```

At the end of this loop, T would be balanced. Lines 1 and 3 use the convention that u is the root iff $u = u$.`parent`. In Line 2, we call Rebalance-Step(u) to ensure that the subtree rooted at u is RB balanced:

```
    Rebalance-Step(u)
1.  If (ratio(u) ≥ 3)
2.        v ← u.left
3.        If (ratio(v) ≥ 3/5),
4.              Return(rotate(v))            ◁ See Figure 30(i)
5.        else   ◁ ratio(v) < 3/5
6.              Return(rotate²(v.right))     ◁ See Figure 30(ii)
7.  elif (ratio(u) ≤ 1/3)
8.        v ← u.right
9.        ... omitted (by symmetry) ...
10. else
11.       Return(u)    ◁ u is already RB balanced
```

Note that `rotate`(u) performs the rotation at u and then returns the node u. A simple illustration of this rebalancing is given in Figure 29.

Note that we compare ratio(v) to 3/5 in Line 3 to decide whether to do single or double rotation. The constant 3/5 is judiciously chosen to be a "simple" rational number: in the literature, the value 3/5 is typically[14] an irrational number, requiring some care in implementation (cf. [17]). Both Lines 4 and 6 return the node that is being rotated (i.e., we assume `rotate`(v) returns v). The omitted code in Line 9 are just a copy of Lines 3-6, except that we interchange

[14] E.g., Nievergelt and Reingold need to use the value $\sqrt{2}$ in comparison.

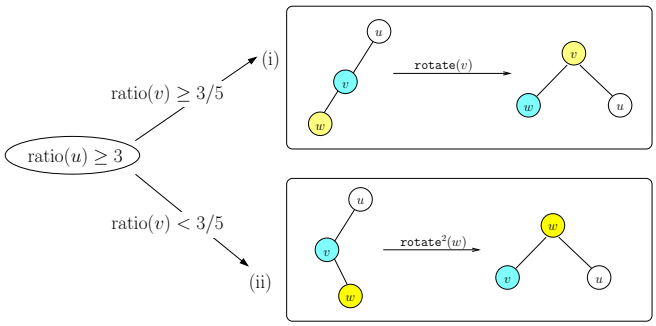


Figure 29: Simple example of Rebalance-Step(u)

`left` and `right`, and replace the condition "ratio(v) $\geq 3/5$" by "ratio(v) $\leq 5/3$". The decisions of Lines 2-6 are illustrated in Figure 30.

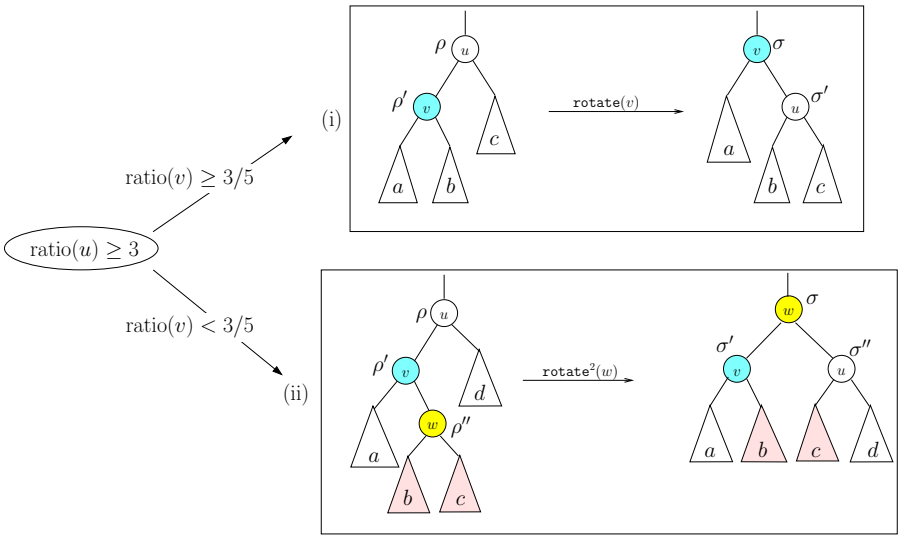


Figure 30: Rebalance-Step: (i) Single Rotate (ii) Double Rotate

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).

¶53. Analysis of Ratio-Balanced Trees. Although the algorithms for RBT are simple, their analysis is somewhat[15] intricate. The follow is a new analysis following an approach of Willard and Leuker [19].

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 $\text{ratio}(w) = \rho''$. 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. Of course, this follows generally from Task(b), but there are some details for small size trees that must be resolved.

We now proceed with these 3 tasks.

¶54. 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):

$$\rho = \frac{a+b}{c}, \qquad \rho' = \frac{a}{b}. \tag{24}$$

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

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

The transformed ratios σ, σ' can likewise be given in terms of a, b, c. Next, plugging in the ρ's we get:

$$\left.\begin{array}{lll} \sigma' & = \frac{b}{c} & = \frac{\rho}{\widetilde{\rho'}} \\ \sigma & = \frac{a}{b+c} & = \frac{\rho\rho'}{\widetilde{\rho+\rho'}}. \end{array}\right\} \tag{25}$$

The expressions (25) are what we seek. An Exercise shows that no exact relation like (25) are possible if we use $\texttt{size}$ instead of $\texttt{esize}$.

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

[15] The book of Mehlhorn [14, p. 193] declines a proof, because it would be "very tedious and inelegant".

$$\rho = \frac{a+b+c}{d}, \qquad \rho' = \frac{a}{b+c}, \qquad \rho'' = \frac{b}{c}. \tag{26}$$

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

$$\left.\begin{array}{rcll} \sigma' & = & \frac{a}{b} & = \frac{\rho'\widetilde{\rho''}}{\rho''} \\ \sigma'' & = & \frac{c}{d} & = \frac{\rho}{\rho'\widetilde{\rho''}+\rho''+1} \\ \sigma & = & \frac{a+b}{c+d} & = \frac{\rho(\rho''+\rho'\widetilde{\rho''})}{1+\rho+\rho''+\rho'\widetilde{\rho''}}. \end{array}\right\} \tag{27}$$

The expressions in (27) are what we seek.

¶55. Task (b): We now want to show that the Rebalance-Step results in a RB tree. The key observation is that the expression for σ is monotonic increasing or monotonic decreasing as a function of each of the variables ρ, ρ'. This observation implies that the minimum $\sigma_{\min}$ and maximum $\sigma_{\max}$ achieved by σ as ρ, ρ' vary within prescribed ranges of values is easily calculated: just plug in one of the two extremal values for ρ, and also for ρ'. Similarly, we can determine $\sigma'_{\min}, \sigma'_{\max}$, and $\sigma''_{\min}, \sigma''_{\max}$.

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

	New Ratio	(ρ, ρ', ρ'')	$S_{\min}$	$S_{\min} - \frac{1}{3}$	$S_{\max}$	$3 - S_{\max}$
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	$\sigma = \frac{\rho(\rho''+\rho'\widetilde{\rho''})}{1+\rho+\rho''+\rho'\widetilde{\rho''}}$	$(+,+,+)$	$\frac{3(\frac{1}{3}+\frac{1}{3}(1+1/3))}{1+3+(1/3)+\frac{1}{3}(1+1/3)} = \frac{21}{43}$	$\frac{22}{129}$	$\frac{4(3+(3/5)(1+3))}{1+4+3+(3/5)(1+3)} = \frac{27}{13}$	$\frac{12}{13}$
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	$\sigma'' = \frac{\rho}{\rho'\widetilde{\rho''}+\rho''+1}$	$(+,-,-)$	$\frac{3}{(3/5)(1+3)+3+1} = 15/32$	$\frac{17}{32}$	$\frac{4}{(1/3)(1+1/3)+1/3+1} = 9/4$	$\frac{3}{4}$

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

For instance, row R1 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':

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

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

With these monotonicity properties, it is now easy to plug in the extremal values of ρ, ρ', ρ'' that will maximize or minimize the expression $S \in \{\sigma, \sigma', \sigma''\}$. We will work with the assumption that $\rho \in [3, 4]$. Let $S_{\min}$ and $S_{\max}$ denote these maximas. The computed entries for $S_{\min}$ and $S_{\max}$ are shown in their respective columns in Table 1. For instance, row R1 shows that, in the case of single rotation, $\sigma_{\min} = 9/23$ and $\sigma_{\max} = 3/2$.

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

In conclusion, the entries in Table 1 show that

$$1/3 < S_{\min} \leq \text{ratio}(x) \leq S_{\max} < 3$$

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

Lemma 14 *Assume every node x in T_u satisfies*

$$\text{ratio}(x) \in \begin{cases} (\frac{1}{3}, 3) & \text{if } x \neq u, \\ [\frac{1}{4}, \frac{1}{3}] \cup [3, 4] & \text{if } x = u. \end{cases}$$

Then after performing Rebalance-Step*(u), T_u is an RB tree.*

¶56. Task (c): We wish to reduce the correctness of the Rebalance procedure to the correctness of Rebalance-Step procedure (Lemma 14). This amounts to ensuring that after insertion or deletion in an RB tree, each node u still satisfies

$$\frac{1}{4} \leq \text{ratio}(u) \leq 4. \tag{28}$$

Let ρ (resp., σ) be the ratio of u after (resp., before) the operation. If $\frac{1}{3} < \rho < 3$, then there is nothing to show. Otherwise, we may assume that $\rho \geq 3$ by symmetry. Hence it remains to show $\rho \leq 4$.

For the proof, let n be the size of T_u before the insertion or deletion, and we may write $\sigma = a/b$ in "non-reduced form", i.e., $a+b = n+1$ where $\texttt{esize}(u.\texttt{left}) = a$ and $\texttt{esize}(u.\texttt{right}) = b$. Since $\sigma < 3$, we immediately have $a < 3b$.

The correctness of insertion is very simple: After insertion, we have $\rho = (a+1)/b \geq 3$ or $a + 1 \geq 3b$. Thus $a < 3b \leq a + 1$. We conclude $a + 1 = 3b$ or $\rho = 3b/b = 3$. This confirms (28) for insertion.

ANALYSIS of SMALL TREES. Before proving the correctness of deletion, we look at maximum value of ρ attainable with deletion. Let $\max \rho(n)$ be the maximum when n is the size of u before deletion. Likewise define $\max \sigma(n)$ to be the maximum value of σ for size n. Table 2 shows these values for small n.

$n = \texttt{size}(u)$	$\max \sigma(n)$	$\max \rho(n)$
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

The values of $\max \sigma(n)$ in Column 2 are given in "non-reduced form" a/b as above. From this non-reduced form, the value of $\max \rho(n)$ in Column 3 can be simply read-off as $a/(b-1)$ (the exception is $n = 2$, for the obvious reason). Successive entries in Column 2 are easily generated using this rule: *if* $\max \sigma(n) = a/b$, *then* $\max \sigma(n+1)$ *is* $(a+1)/b$ *if* $(a+1)/b < 3$; *otherwise it is* $a/(b+1)$. 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:

$$\text{For } n = \texttt{size}(u) \geq 7, \text{ the non-reduced denominator is at least 3.} \tag{29}$$

The most interesting row in Table 2 is for $n = 6$: Just before deletion, the ratio is equal to $\max \sigma(6) = 5/2$; the corresponding binary tree is illustrated in Figure 27(a) After deletion, the maximum ratio is $\max \rho(6) = 5$, a clear violation of (28). This bad case is shown in Figure 27(c). For all other values of n, the maximum ratio after deletion will be less than 5.

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 $\rho \geq 3$ we have $a \geq 3b-3$. Thus $3b-3 \leq a < 3b$. 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 $b \geq 3$, we will have $\rho \leq 4$, proving (28). According to (29), $b \geq 3$ for all $n \geq 7$.

It remains to consider the case $n \leq 6$. By Table 2, the only situation where ρ exceed 4 is when the $n = 6$. It turns Rebalancing-Step actually works correctly for this case.

To see this, refer to Figure 27 as guide. Start with the RB tree T_u in Figure 27(a). Suppose we delete w. Note that $\text{ratio}(v) = 2/3$ where $v = u.\texttt{left}$. But another possibility is $\text{ratio}(v) = 3/2$. In either case, $\text{ratio}(v) \geq 3/5$ is satisfied, and so our rebalance rule will call for a single rotation. In both cases the result is an RB tree.

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

¶57. A Hybrid Scheme – Logarithmic BST. Roura [17] 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 $2^{h-1} < size(u) \leq 2^h$. 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 $\geq 1 + 2^{h-2}$. Therefore size of w is at least $2 + size(u) + 2(1 + 2^{h-2}) \geq 2 + (2^{h-1} + 1) + (2 + 2^{h-1}) = 5 + 2^h$. 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 31.
- $\overline{\mu}(h)$ is the minimum size of a Log BST tree of log-height h. For instance, the first few values are given by $\overline{\mu}(-1, 0, 1, 2, 3, 4) = (0, 1, 2, 4, 8, 16)$.

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

$$\overline{\mu}(h) = 2^h. \tag{30}$$

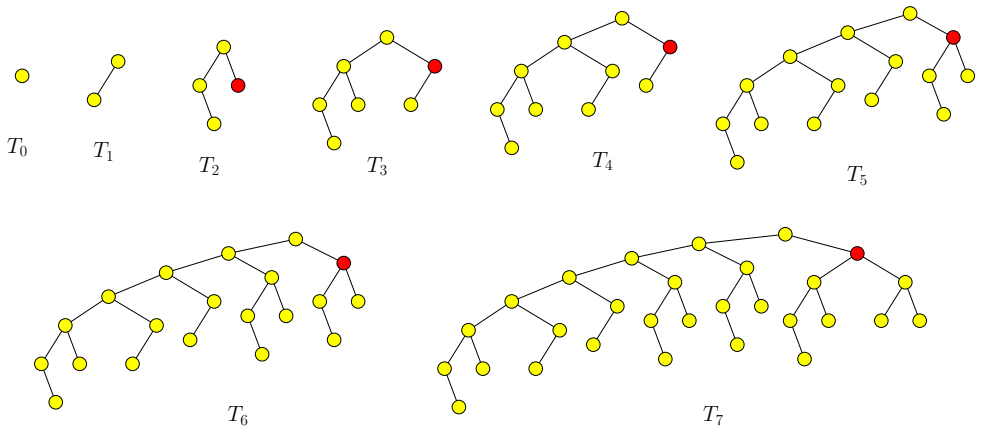


Figure 31: Min-size Logarithmic BST's

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

Next we derive a recurrence for $\mu(h)$. We must not be misled by the AVL recurrence, and simply claim that $\mu(h + 1) = 1 + \mu(h) + \mu(h - 1)$. The correct recurrence is

$$\mu(h + 1) = 1 + \mu(h) + 2^{\lfloor \lg \mu(h) \rfloor - 1}. \tag{31}$$

To see this, let T_{h+1} be a min-size tree of height $h+1$. Wlog, assume that its left subtree has height h and therefore size $\mu(h)$. But what is the size of its right subtree? Well, the log-height of the left subtree is $\ell(\mu(h)) = 1 + \lfloor \lg \mu(h) \rfloor$. Therefore the log-height of the left subtree must be $\lfloor \lg \mu(h) \rfloor$. By (30), the minimum size tree with this log-height $2^{\lfloor \lg \mu(h) \rfloor - 1}$. This is precisely the the quantity we incorporate into the recurrence (31). Using (31), 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

We make two observations about the entries in this table: the case $h = 4$ marks the first 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 31 is non-AVL for all $h \geq 4$. 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 31, 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 (31). 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 32.

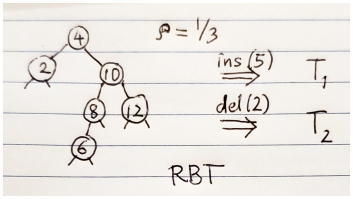


Figure 32: RB tree of size 6: Insert 5 and Delete 2

(i) Insert key 5 and draw the result T_1.

(ii) Delete key 2 and draw the result T_2.

These 2 parts are independent of each other! ♢

Exercise 7.2: 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? ♢

Exercise 7.3: Our code for Rebalance-Step(u) omitted the logic when ratio(u) $\leq 1/3$. Please write this out explicitly. Implement it in you favorite programming language. ♢

Exercise 7.4: The threshold constant 3/5 in Rebalance Step requires an exception when $\texttt{size}(u) = 5$. Find another simple rational constant that works without exception. ♢

Exercise 7.5: Prove that $\max \sigma(n) \leq 5$. ♢

Exercise 7.6: 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 7.7: In (23), 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., $\texttt{esize} = n$). Prove the recurrence:

$$H'(n) = 1 + H'(\lceil n/\widetilde{\rho} \rceil + 1).$$

♢

Exercise 7.8: (Michael Mathieu, 2012) Instead of $\texttt{esize}(u)$, suppose we want to use $\texttt{size}(u)$ to formulate the ratio inequalities. We seek the analogue of the relations (25) 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 7.9: 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 7.10: 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 7.11: 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 7.12: 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 7.13: 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 7.14: 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 $D(n)$? ♢

Exercise 7.15: Generalize the data structure in the previous question to range queries in $\mathbb{R}^d$ ($d \geq 3$). ♢

Exercise 7.16: 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. ♢

Exercise 7.17: 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**. ♢

End Exercises

§8. Generalized B-Trees

We consider another class of trees that is very important in practice, especially in database applications. These are no longer binary trees, but are parametrized by two integers,

$$2 \leq a < b. \tag{32}$$

An (a, b)**-tree** is a rooted, ordered[16] tree with the following structural constraints:

[16] "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).

- 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

$$a \leq m \leq b. \tag{33}$$

The root is an exception, with the bound $2 \leq m \leq b$.

Perhaps the most important case is when $b = 2a - 1$: these will be called **B-trees**, and is from McCreight and Bayer [3]. Hence we regard (a, b)-trees as **generalized B-trees**.

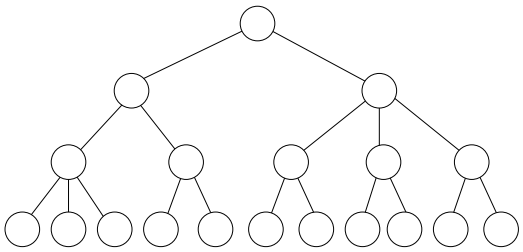


Figure 33: A $(2, 3)$-tree.

Figure 33 illustrates an (a, b)-tree for $(a, b) = (2, 3)$. To see the intuition behind these conditions, compare with binary trees. In binary trees, the leaves do not have to be at the same depth. To re-introduce some flexibility into trees when all leaves have the same depth, we allow the degree of an internal node to vary over a range $[a, b]$. Moreover, in order to ensure logarithmic height, we require $a \geq 2$. This means that if there are n leaves, the height is at most $\log_a(n) + \mathcal{O}(1)$. *Therefore, (a, b)-trees constitute a balanced family of trees.* Notice that an (a, b)-tree is also (c, d)-tree if c, d satisfy

$$2 \leq c \leq a < b \leq d. \tag{34}$$

E.g., Figure 33 could have represented an $(2, 10)$-tree but not a $(3, 4)$-tree.

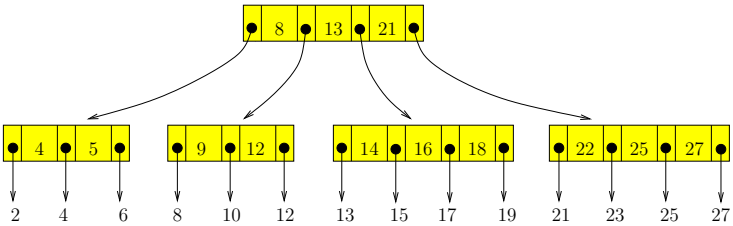
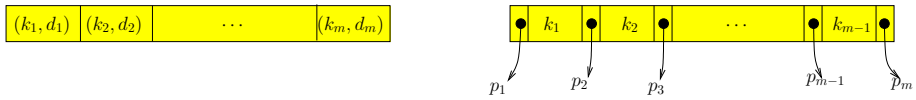


Figure 34: A (3,4)-search tree on 14 items

¶58. (a, b)-Tree as External Search Tree. The definition of (a, b)-trees imposes purely structural requirements. To use such trees as search structures, we need to store keys and items in the tree nodes. These keys and items must be suitably organized. Before giving these details, we provide some intuition for such search trees by looking at the example in Figure 34. This tree is structurally a $(3, 4)$-tree. Of course, it could an (a, b)-tree for any $2 \leq a \leq 3$ and $b \geq 4$. It has 14 leaves, each storing a single item. The keys of these items are $2, 4, 6, 8, \ldots, 23, 25, 27$. Recall that an item is a $(\texttt{key}, \texttt{data})$ pair, but as usual, we do not display the associated data in items. *We continue our default assumption that items have unique keys.* But we see in

Figure 34 that the key 13 appears in the root as well as in a leaf. In other words, although keys are unique in the leaves, they might be duplicated (once) in the internal nodes (e.g., key 27). Conversely, the keys in the internal nodes (e.g. key 5) *need not* correspond to keys of items.

We define an (a, b)**-search tree** to be an (a, b)-tree whose nodes are organized as an **external search tree**: as in external BST (see §5), it means that items are only stored in the leaves. Leaves are also called **external nodes**. The different organization of the internal and external nodes are illustrated in Figure 35. The external node organization is controlled by an independent pair of parameters a', b' that satisfy the inequalities $1 \leq a' \leq b'$. 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:



$1 \leq a' \leq m \leq b'$ $\qquad$ $2 \leq a \leq m \leq b$

(i) Leaf Node Organization $\qquad$ (ii) Internal Node Organization

Figure 35: 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,

$$u = (k_1, d_1; k_2, d_2; \cdots; k_m, d_m) \tag{35}$$

where $k_1 < k_2 < \cdots < k_m$. Thus there are m items of the form (k_i, m_i). See Figure 35(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

$$a' \leq m \leq b'. \tag{36}$$

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 (36) to

$$0 \leq m \leq 2b' - 1. \tag{37}$$

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:

$$u = (p_1, k_1, p_2, k_2, p_3, \ldots, p_{m-1}, k_{m-1}, p_m) \tag{38}$$

where p_i is a pointer to the i-th child of the current node. This is illustrated in Figure 35(ii). Note that the number of keys in this sequence is one less than the number m of children. The reader should compare (38) with (35). 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

$$k_{i-1} \leq k < k_i, \tag{39}$$

with the convention that $k_0 = -\infty < k_i < k_m = +\infty$. Note that this is just a generalization of the binary search tree property in (3).

¶59. Choice of the (a', b') parameters. Since the a', b' parameters are independent of a, b, it is convenient to choose some default value for our discussion of (a, b) trees. This decision is justified because the dependence of our algorithms on the a', b' parameters is not significant. We use two canonical choices: the simplest is $a' = b' = 1$. This means each leaf stores exactly one item. All our examples (e.g., Figure 34) use this default choice. Another canonical choice is $a' = a, \quad b' = b$.

So (a', b') is implicit!

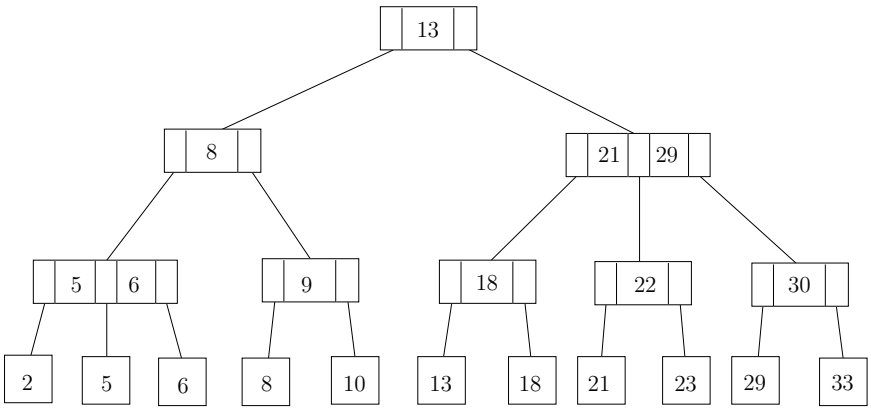


Figure 36: A $(2, 3)$-search tree.

Another (a, b)-search tree is shown in Figure 36, for the case $(a, b) = (2, 3)$. In contrast to Figure 34, here we draw it using a slightly more standard convention of representing the pointers as tree edges.

¶60. Special Cases of (a, b)-Search Trees. 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). By choosing

$$b = 2a - 1 \tag{40}$$

(for any $a \geq 2$), we obtain the generalization of $2 - 3$ trees called **B-trees**. These were introduced by McCreight and Bayer [3]. Huddleston and Mehlhorn (1982) and Maier and Salveter (1981) observed that the slight variant of $b = 2a$ behaves better in an amortized sense (see [13, 8, 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 the price of this efficiency is complicated algorithms for insertion and deletion. Thus it is clear that the concept of (a, b)-search trees serves to unify a variety of search trees. The terminology of (a, b)-trees was used by Mehlhorn [13].

The B-tree relationship (40) is sharp in a certain sense: this will be shown via the "split-merge inequality" below.

¶61. 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 (35). 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.
- Inductive Case: suppose u is an internal node given by (38). Then we find the p_i such that $k_{i-1} \leq k < k_i$ (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 a (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

$$M(h) = b^h. \tag{41}$$

Likewise, $\mu(h)$ is attained if every internal node (except the root) has a children. Thus

$$\mu(h) = \begin{cases} 1 & h = 0, \\ 2a^{h-1}. & h \geq 1. \end{cases} \tag{42}$$

It follows that if an (a, b)-tree with n leaves has height $h \geq 1$, then $2a^{h-1} \leq n \leq b^h$. Taking logs, we get

$$1 + \log_a((n-1)/2) \leq h \leq \log_b n.$$

We leave to an Exercise to bound the number of *items* stored in an (a, b)-tree, but here we must 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.

¶62. Standard Split and Merge Inequalities. To support efficient insertion and deletion algorithms, the parameters a, b must satisfy an additional inequality in addition to (32). 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:

$$a \leq \left\lfloor \frac{b+1}{2} \right\rfloor. \tag{43}$$

Similarly, during deletion, we may remove a child from a node that has only a children. 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 $2a-1 \leq b$. Thus we require the following merge inequality:

$$a \leq \frac{b+1}{2}. \tag{44}$$

Clearly (43) implies (44). However, since a and b are integers, the reverse implication also holds! Thus (43) and (44) are equivalent, and they will be known as the **split-merge inequality**. The smallest choices of parameters a, b subject to the split-merge inequality and also (32) is $(a,b) = (2,3)$ because by definition of (a,b) trees, $a \geq 2$. The case of equality in (43) and (44) gives us $b = 2a-1$; recall that these are known as the B-trees.

On Mixed Integer Inequalities. The above argument uses the following little lemma: let x, y be real numbers such that

$$x \geq y. \tag{45}$$

Suppose you also know that at least one of x or y is an integer. Then the (45) is equivalent to the apparently stronger inequality $x \geq \lceil y \rceil$ or $\lfloor x \rfloor \geq y$. Combining them:

Neat trick!

Lemma 15 *If at least one of x or y is an integer, then*

$$x \geq y \iff \lfloor x \rfloor \geq \lceil y \rceil . \tag{46}$$

We will re-use this argument several times below.
Similar ideas apply for strict inequalities: If x is an integer, then

$$x > y \iff x > \lfloor y \rfloor \iff x \geq 1 + \lfloor y \rfloor . \tag{47}$$

If y is an integer, then

$$x > y \iff \lceil x \rceil > y \iff \lceil x \rceil \geq 1 + y. \tag{48}$$

¶63. How to Split, Borrow, and Merge. Once (a,b) is known to satisfy the split-merge inequality, we can design algorithms for insertion and deletion. However, we will first describe the subroutines of split, borrow and merge first. We begin with the *general case* of internal nodes that are non-root. The special case of leaves and root will be discussed later.

Suppose we need to **split** an overfull node N with $b+1$ children. This is illustrated in Figure 37. 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. The parent P of N will replace its pointer to N with two pointers to N_1 and N_2. But what is the key to separate the pointers to N_1 and N_2? The solution is to use a key from N: there are b keys in the original node, but only $b-1$ keys are needed by the two new nodes. The extra key can be moved in the parent node, sandwiched between the pointers to N_1 and N_2, as indicated.

Next, suppose N is an underfull node with $a-1$ children. First we try to **borrow** from an immediate sibling if possible. This is because after borrowing, the rebalancing process can stop. To borrow, we look to an immediate sibling (left or right), provided the sibling has more than a children. This is illustrated in Figure 38. Suppose N borrows a new from its sibling

do not borrow from cousin (near or distant ones). Why?

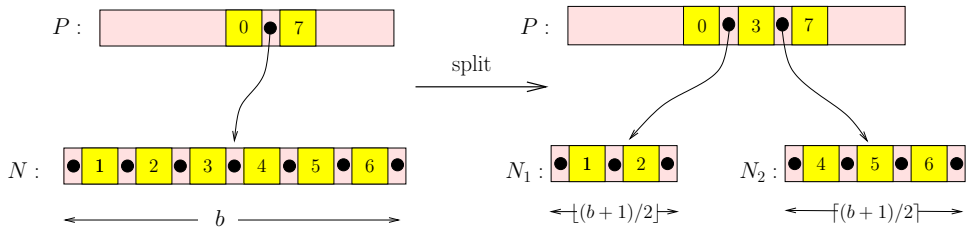


Figure 37: Splitting: N splits into N_1, N_2. Case $(a, b) = (3, 6)$ is illustrated.

M. After borrowing, N will have a children, but it will need a key to separate the new pointer from its adjacent pointer. This key is taken from its parent node. Since M lost a child, it will have an extra key to spare — this can be sent to its parent node.

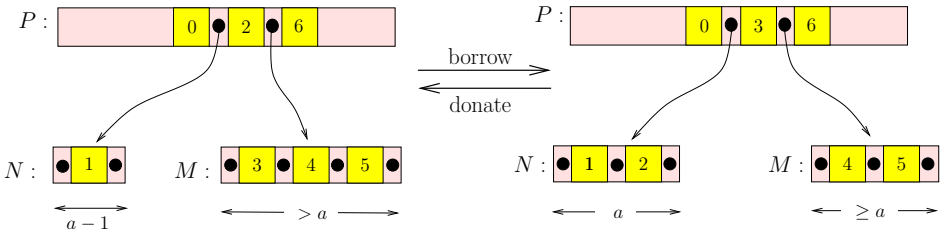


Figure 38: Borrowing: N borrows from M. Case of $(a, b) = (3, 6)$.

If N is unable to borrow, we resort to **merging**: let M be an immediate sibling of N. Clearly M has a children, and so we can merge M and N into a new node N' with $2a - 1$ children. Note that N' needs an extra key to separate the pointers of N from those of M. This key can be taken from the parent node; the parent node will not miss the loss because it has lost one child pointer in the merge. This is illustrated in Figure 39.

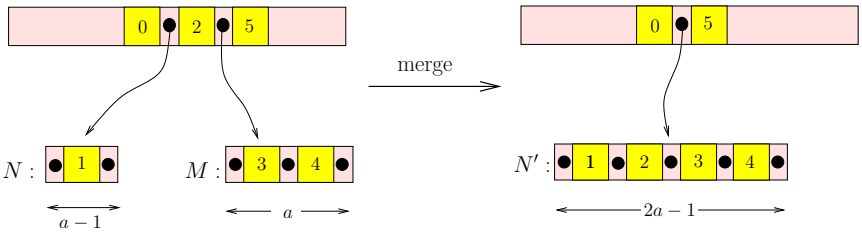


Figure 39: Merging: N and M merges into N'. Case of $(a, b) = (3, 6)$.

The careful reader will notice an asymmetry in the above three processes. We have the concept of borrowing, but it makes as much sense to talk about its inverse operation, **donation**. Indeed, if we simply reverse the direction of transformation in Figure 38, we have the **donation** operation (node N donates a key to node M). Just as the operation of merging can be preempted by borrowing, the operation of splitting can be preempted by donation! Donation is rarely discussed in the literature. But we will see its benefits below.

¶64. 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.

¶65. 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,

$$a' \leq \frac{b'+1}{2}. \tag{49}$$

Recall that the two canonical choices for (a',b') are $(a',b') = (a,b)$ or $(a',b') = (1,1)$. The inequality (49) 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!

¶66. Mechanics of Insertion and Deletion. We are ready to present the algorithm for insertion and deletion. It is important that we describe these algorithms in an "I/O aware" manner, meaning 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. Furthermore, I/O operations are much more expensive than CPU operations (two to three orders of magnitude slower). Therefore in complexity analysis below, we will 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 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.

be I/O aware!

We now present 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.
Let u be the leaf node where the insertion or deletion takes place.
At this point, u is in main memory.
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.

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).

¶67. Standard Insert/Delete and Enhancements. We were deliberately vague in Step 3 of the above algorithm for two reasons: first, the vague description can cover generalized split/merge operations to be described shortly. Second, even in the "standard Insert/Delete" algorithms, there are some possible enhancements and/or variations. These enhancements are associated with attempts to avoid split/merge by doing donate/borrow. Why? After a split/merge, we must continue going along the root-path to see if any parent is now unbalanced. On the other hand, if we can do a donate/borrow, then we can terminate right away. Such enhancements result in better space utilization and encourage earlier termination. Hence we now make the Standard Algorithms explicit, in order to point out the enhancements.

(STANDARD INSERTION) For standard insertion, if node u is overfull, the standard algorithm immediately splits u into two nodes (and recurse). *That is all.*

As noted, we should try to donate instead of splitting. To donate, we find an immediate sibling of degree less than b. There can be up to two such siblings. When all immediate siblings are full, we fail to donate and we must split as in the standard insertion. Refer[17] to this non-standard algorithm as **enhanced insertion**.

Example: Consider inserting the item (represented by its key) 14 into the tree in Figure 34. This is illustrated by Figure 40. Note that $a' = b' = 1$. After inserting 14, we get an over-

[17] Technically, we are getting ahead of ourselves. The $(3, 4)$-search tree is actually what we will call a $(3, 4, 2)$-search tree later. Enhancement refers to the third parameter $c > 1$ in (a, b, c)-search trees.

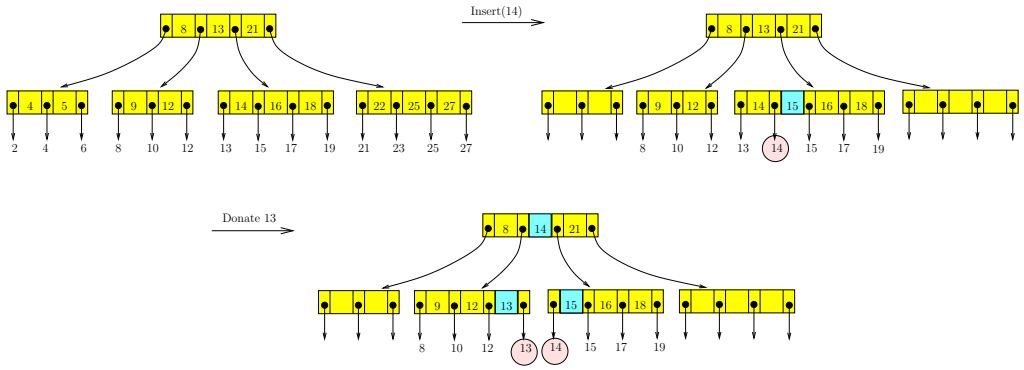


Figure 40: Enhanced Insertion of 14 into a $(3, 4)$-search tree.

full node with 5 children. In standard Insertion, we would immediately split. But with the enhancement, we try to donate to our left sibling. In this case, this is possible since the left sibling has less than 4 children. (We shall see later that this donation action is also consistent with treating this as a $(3, 4, 2)$-tree.)

(STANDARD DELETION) For standard deletion, if a node u is underfull, the standard algorithm tries to borrow from an immediate sibling u'. Notice that u' could be either a left or a right sibling and it must have degree greater than a. If we fail to borrow from u', then we merge u with u' as before. *That is all.*

In **enhanced deletion**, we try to borrow to avoid merging. Although we have failed to borrow from u', if we have another immediate sibling u'' of degree $> a$, then we could borrow from u''. Otherwise, we merge as in the standard insertion.

Observe that these INSERT/DELETE algorithms only need to hold in main memory at most three nodes at any moment: current node, its parent and at most one sibling. So far, we have implicitly assumed that the degree of each internal node is stored in the node itself. The standard insertion/deletion algorithms use this information. For the enhanced algorithms, this is a problem: we may needlessly bring a sibling into memory in a failed attempt to donate/borrow. To avoid this, the degree of each child of node u will now be stored in u itself. Intuitively, we modify (38) to

$$u = (p_1, d_1, k_1; p_2, d_2, k_2; p_3, d_3, \ldots, p_{m-1}, d_{m-1} k_{m-1}; p_m, d_m). \tag{50}$$

This way, we can immediately tell whether donation/borrowing is possible with an immediate sibling, without bringing that sibling into main memory. So the enhanced algorithms incur no extra I/O operations. Note that maintaining (50) is quite easy because just before we write a node u into main memory, we have its parent p also in main memory. So the degree of u can be recorded.

It would interesting to quantify the gains from these enhancements using amortized or probabilistic analysis.

¶68. 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 in a parent, split u into u and u', and write out u and u'. Thus we have 3 I/O operations per iteration. Thus the overall I/O cost is $2+3I$ where I is the number of iterations (of course I is bounded by the height). There is one other possibility: the last iteration might be the base case where u is a root. In this case, we split u and write them out as two nodes. So this case needs only 2 I/O's, and our bound of $2+3I$ is still valid.

Consider Standard Deletion: there is an initial reading and final writing of the current node. In each iteration of the while loop, the current node u is underfull, and we need to bring in a parent and a sibling u', and then writing out the merger of u and u'. Again, this is 3 I/O's per iteration. A possibly exception is in the last iteration where we achieve a borrowing instead of merging. In this case, we must write out both u and u', thus requiring four I/O's. Thus the overall I/O cost is bounded by $3+3D$ where D is the number of iterations. Next consider the cost of enhancements: failed borrowing costs only one I/O (to read the sibling). But any successful borrowing is already a part of the main accounting. Since the number F of failures is at most D, we obtain an upper bound of $3+4D$ in the enhanced algorithm. If we use the representation (50), then the enhanced deletion incurs no extra cost.

¶69. Achieving 2/3 Space Utility Ratio. A node with m children is said to be **full** when $m=b$; for in general, a node with m children is said to be (m/b)**-full**. Hence, nodes can be as small as (a/b)-full. Call the ratio $a:b$ the **space utilization ratio**. This ratio is <1 and we like it to be as close to 1 as possible. The standard inequality (44) on (a,b)-trees implies that the space utilization in such trees can never[18] be better than $\lfloor (b+1)/2 \rfloor / b$, and this can be achieved by B-trees. This ratio is as large as $2:3$ (achieved when $b=3$), but as $b \to \infty$, it is asymptotically equal to $1:2$.

Of course, in practice, the potential to waste about half of the available space is not good. Hence we next address the issue of achieving ratios that are arbitrarily close to 1, for any choice of a, b. First, we show how to achieve $2/3$ asymptotically.

Consider the following modified insertion: to remove 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

$$\lfloor (2b+1)/3 \rfloor, \quad \lfloor (2b+1)/3 \rceil, \quad \lceil (2b+1)/3 \rceil$$

where "$\lfloor (2b+1)/3 \rceil$" denotes **rounding** to the nearest integer. For instance, $\lfloor 4/3 \rfloor + \lfloor 4/3 \rceil + \lceil 4/3 \rceil = 1+1+2 = 4$ and $\lfloor 5/3 \rfloor + \lfloor 5/3 \rceil + \lceil 5/3 \rceil = 1+2+2 = 5$. Nodes u and v will (respectively) have one of these groups as their children, but the third group will be children of a new node. See Figure 41.

for any integer n: $\lfloor n/3 \rfloor + \lfloor n/3 \rceil + \lceil n/3 \rceil = n$

We want these groups to have between a and b children. The largest groups has size $\lceil (2b+1)/3 \rceil$ and this $\leq b$, automatically. For the smallest group to have size at least a, we require

$$a \leq \left\lfloor \frac{2b+1}{3} \right\rfloor . \tag{51}$$

[18] 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.

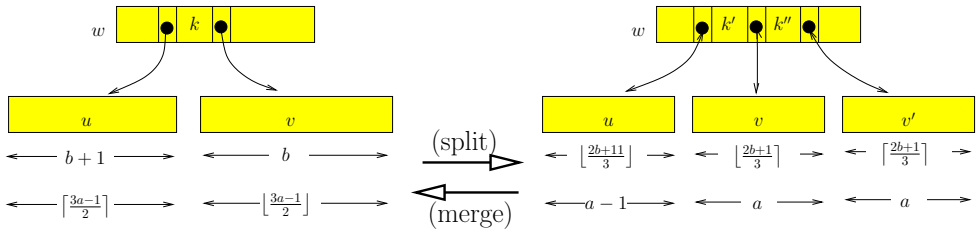


Figure 41: Generalized (2-to-3) split and (3-to-2) merge

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 remove 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[19] 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

$$a \geq 3, \tag{52}$$

v and v' exist (unless u is a child of the root, which is handled separately). For lower bound on degree, we require $\lfloor (3a-1)/2 \rfloor \geq a$, which is equivalent to $(3a-1)/2 \geq a$ (by integrality of a), which clearly holds. For upper bound on degree, we require

$$\left\lceil \frac{3a-1}{2} \right\rceil \leq b \tag{53}$$

Because of integrality constraints, the floor and ceiling symbols could be removed in both (51) and (53). Thus both inequalities are be equivalent to

$$a \leq \frac{2b+1}{3}. \tag{54}$$

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 $2 \leq m \leq b$. 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.

¶70. Example of Generalized Merge. Consider deleting the item (represented by its key) 4 from the tree in Figure 34. The is illustrated in Figure 42. 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.

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 (6 and 12) from

[19] We assume $a \geq 3$. 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''.

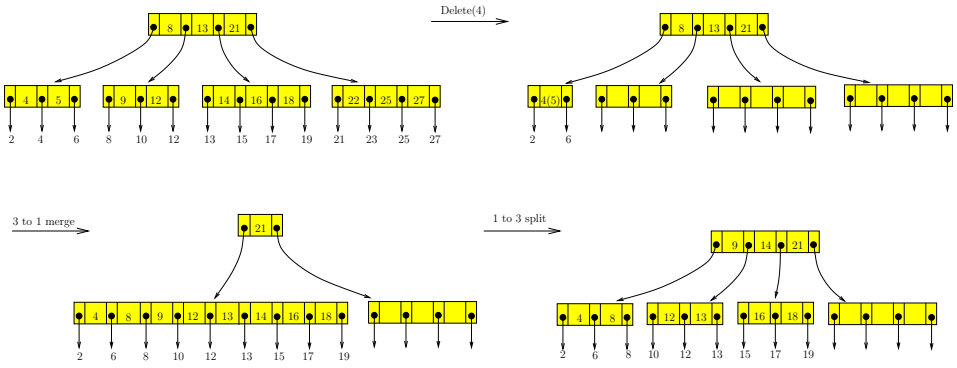


Figure 42: Deleting 4 from $(3, 4, 2)$-search tree.

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 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 $2a - 1 \leq b$. Of course, this constraint is just the standard split-merge inequality (44); 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 34. 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.

¶71. 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[20] 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.

***Q**: Is this school bus going left or right?*
***Adult**: I don't know.*
***Child**: Going Left.*
***Adult**: Huhh???*

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 36 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.

Can't we require the keys in internal nodes to correspond to keys of stored items?

[20] The child knows that if the school bus were going right then you could see the entrance door.

¶72. 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 (35) 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.

¶73. 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 $\underline{b}$lock size

¶74. 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 (38) 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

space demand increases from methods (1) to (4), the value of b is expected to decrease as we move from methods (1) to (4). This will have an impact on the height of the search tree.

(B) THE I/O CONSTRAINT: Our algorithms perform their actions only in main memory while the nodes reside in secondary (disk) memory. The operation to move a node in or out of the main memory is called an **I/O operation**. The I/O constraint refers to the fact that such operations are 2 or 3 orders of magnitude slower than a CPU operation. *In Exercises, the student is asked to assume a nominal value of* 1000 *CPU cycles per block swap.* In view of the I/O constraint, the overriding goal in the design of (a, b)-search trees should be to maximize b. When maximizing b, we remember there are some fixed overhead in each node: for instance, we assume that each internal node must maintain the current degree of the node (this is an 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

$$\lfloor (2^{12} - H)/(K + P) \rfloor$$

where each key uses K bytes and each data item takes P bytes. Note that data might have unpredictable size, but if we only store a pointer to the actual location of the data, then allocating a fixed amount of P bytes is reasonable (typically, $P <= 8$).

For the remaining methods (2)-(4), it seems best to divide the internal node into fixed size "chunks" for use in the linked lists or as BST nodes. These "chunks" can be kept in a FREELIST, using the well-known FREELIST technique from Operating Systems. Our algorithm for inserting or deleting keys into the node can ask for a new chunk or return a chunk to the FREELIST very efficiently. The head of this FREELIST may be counted towards the fixed overhead of the node. This is explored in the Exercise.

¶75. On (a, b, c)-trees: Generalized Split-Merge. Our insertion and deletion algorithms use the strategy of "share a key if you can" in order to avoid splitting or merging. Here, "sharing" encompasses borrowing as well as donation. The 2/3-space utility method will now be generalized by the introduction of a new parameter $c \geq 1$. Call these (a, b, c)**-trees**. We use the parameter c as follows.

- **Generalized Split** of u: When node u is overfull, we will examine up to $c - 1$ siblings to see if any of these siblings has degree $< b$. If so, we can donate a child of u to its sibling which in turn donates a child to the next sibling, etc, until the sibling of degree $< b$ receives a donated child. We are now done. Otherwise, we merge c nodes (node u plus $c - 1$ siblings), and split the merger into $c + 1$ nodes. We view c of these nodes as re-organizations of the original nodes, but one of them is regarded as new. We must insert this new node into the parent of u. The parent will be transformed appropriately, and the process repeated.

  We stress that there is no sharp distinction between donation and splitting: we view of them as different possibilities for a single **generalized split subroutine**: starting from an overfull node u, we successively bring into main memory a sequence of contiguous siblings of u (they may be right or left siblings) until we either (i) find one that has less than b children, or (ii) brought in the maximum number of $c - 1$ siblings. In case (i), we do donation, and in case (ii) we split these c siblings into $c + 1$ siblings.

- **Generalized Merge** of u: When node u is underfull, 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**,

$$c+1 \leq a \leq \frac{cb+1}{c+1}. \tag{55}$$

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 (55) 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 (55) guarantees success.

We are interested in the maximum value of a in (55). Using the fact that a is integer, this amounts to

$$a = \left\lfloor \frac{cb+1}{c+1} \right\rfloor. \tag{56}$$

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 16 *For any c, the smallest set of parameters (a,b,c) satisfying the generalized split-merge inequalities (55) is $(a,b,c)=(c+1,c+2,c)$.*

Proof. The first inequality in

$$c+1 \leq a \leq \frac{cb+1}{c+1}$$

is trivially satisfied, since $a=c+1$. The second inequality is $c+1 \leq (cb+1)/(c+1)$ reduces to $(c+1)^2 \leq c(c+2)+1$ when $b=c+2$. This inequality is in fact an equality. **Q.E.D.**

¶76. 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 $2 \leq a < b$. Likewise, a triple (a,b,c) is **valid** if it satisfies (55). We view (55) as specifying a lower bound $a \geq c+1$ and an upper bound $a \leq (cb+1)/(c+1)$ 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 $a \leq ((a-1)b+1)/a$ or $a^2 \leq (a-1)b+1$. But this is clearly satisfied since $b \geq a+1$.

The choice $c = a-1$ is the largest possible choice. In analogy to B-Trees which satisfy (40), We might define "C-trees" as (a,b,c) trees whose parameters satisfy

$$b = 2a-1, \quad c = a-1. \tag{57}$$

The (a,b,c) parameters C-trees are given

$$(2,3,1),(3,5,2),(4,7,3),(5,9,4),\ldots.$$

So the more interesting case is the smallest possible value of c. To determine the smallest c, we can easily verify (Exercise) this fact:

$$\textit{Assume } c+2 \leq a. \textit{ If } (a,b,c) \textit{ is valid, so is } (a,b,c+1).$$

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.

¶77. 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

$$10b \leq 512.$$

You could say that the number of keys is one less than the number of pointers, and so the correct inequality is $10b \leq (512+4) = 516$. 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.

¶78. 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 $c \geq 2$.

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.

¶79. 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 [13]). This was the beginning of a technique called "fringe analysis" which Yao [20] introduced in 1974. Nakamura and Mizoguchi [15] 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 (56), 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: 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? ♢

Exercise 8.2: 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 $\mu(h)$. ♢

in text, $M(h), \mu(h)$ were defined for number of leaves, not size

Exercise 8.3: Prove the claim that if $c + 1 \leq a$ and (a, b, c) is valid then so is $(a, b, c + 1)$. ♢

Exercise 8.4: Consider tradeoffs in using one of the following schemes to organize the nodes of an B-tree where $b = 2a - 1$: (i) an ordered array, (ii) an ordered singly-linked list, (iii) an ordered doubly-linked list, (iv) a balanced binary search tree.
Consider a specific numerical example: block size is $4096 = 2^{12}$ bytes, and each **block pointer** is 4 bytes, and each key 6 bytes. A **local pointer** within the block uses 12 bits, but for simplicity we use two bytes. Please be sure to note other information you need in a node, such as a parent pointer, the degree, etc.
(a) What is the maximum value of the parameter b under each of the schemes (i)-(iv)? 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? We need an explicit number (not an numerical expression) for each of the four schemes as answer: the unit for each number is CPU cycles (or "CPUC").

ASSUMPTIONS: Assume $(a', b') = (a, b)$ to control the number of items in leaves. Each disk I/O takes 1000 CPU cycles. If searching for a key takes $O(\log n)$ or $O(n)$ CPU time, always assume that "4" is the constant in big-Oh notation. E.g., searching for a key in a balanced BST with $n = 100$ keys takes $4 \log n = 4 \times 7 = 28$ CPUC. Searching in a list of length $n = 100$ takes $4n = 400$ CPUC. The root of the search tree is always in main memory, so you never need to read or write the root. Assume $a = \lfloor (b + 1)/2 \rfloor$. You may use calculators, but feel free to use simplified estimates whenever possible (e.g., $\lg 10^6 = 20$). ♢

Exercise 8.5: Consider an (a, b)-tree with n items and height h. Give upper and lower bounds on the height in terms of n. Your bounds will depend on the parameters (a, b) and (a', b'). ♢

Exercise 8.6: 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. ♢

Exercise 8.7: 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 8.8: 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? ♢

Exercise 8.9: 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 $O(n)$. ♢

Exercise 8.10: 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 8.11: 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 8.12: Do the same worst/best analysis as the previous question, but assuming an arbitrary $c \geq 2$:
(I) Compare insertion algorithms (regular and pre-emptive)
(D) Compare deletion algorithms (regular and pre-emptive) ♢

Exercise 8.13: 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 8.14: What is the the best ratio achievable under (44)? Under (54)? ♢

Exercise 8.15: 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 (44). ♢

Exercise 8.16: Consider the tree shown in Figure 34. Although we previously viewed it as a $(3, 4)$-tree, we now want to view it as a $(2, 4)$-tree. 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 8.17: 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 8.18: 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 8.19: 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] C. C. Foster. A generalization of AVL trees. *Comm. of the ACM*, 16(8):513–517, 1973.

[7] Y. Hirai and K. Yamamoto. Balancing weight-balanced trees. *J. of Functional Programming*, 21(3):287–307, 2011.

[8] S. Huddleston and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Inform.*, 17:157–184, 1982.

[9] D. E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume 1. Addison-Wesley, Boston, 2nd edition edition, 1975.

[10] K. S. Larsen. AVL Trees with relaxed balance. *J. Computer and System Sciences*, 61:508–522, 2000.

[11] H. R. Lewis and L. Denenberg. *Data Structures and their Algorithms*. Harper Collins Publishers, New York, 1991.

[12] G. Lueker and D. Willard. A data structure for dynamic range queries. *Inf. Proc. Letters*, 15:209–213, 1982.

[13] K. Mehlhorn. *Datastructures and Algorithms 1: Sorting and Sorting*. Springer-Verlag, Berlin, 1984.

[14] K. Mehlhorn. *Datastructures and Algorithms 3: Multi-Dimensional Searching and Computational Geometry*. Springer-Verlag, Berlin, 1984.

[15] T. Nakamura and T. Mizoguchi. An analysis of storage utilization factor in block split data structuring scheme. *VLDB*, 4:489–195, 1978. Berlin, September.

[16] J. Nievergelt and E. M. Reingold. Binary trees of bounded balance. *SIAM J. Computing*, 2(2):33–43, 1973.

[17] 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.

[18] R. E. Tarjan. *Data Structures and Network Algorithms*. SIAM, Philadelphia, PA, 1974.

[19] D. Willard and G. Lueker. Adding range restriction capability to dynamic data structures. *J. ACM*, 32:597–617, 1985.

[20] A. C.-C. Yao. On random 2-3 trees. *Acta Inf.*, 9(2):159–170, 1978.