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

– Rabindranath Tagore, *Fireflies*, 1928

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
BALANCED SEARCH TREES

Anthropologists inform us 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 list. The list can be enlarged to include variants of these trees: thus there are subspecies of B-trees called B^+- and B^*-trees, etc.

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. Algorithms on binary search trees have a worst-case behavior that is proportional to the height of the tree. The height of a binary tree on n nodes is at least $\lfloor \lg n \rfloor$. We say that a family of binary trees is **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

balance-ness is a family property

family. Such a family usually comes equipped with algorithms for inserting and deleting items from trees, while preserving membership in the family.

Many balanced families have been invented in computer science. They come in two basic forms: **height-balanced** and **weight-balanced schemes**. In the former, we ensure that the height of siblings are "approximately the same". In the latter, we ensure that the number of descendants of sibling nodes are "approximately the same". Height-balanced schemes require us to maintain less information than the weight-balanced schemes, but the latter has some extra flexibility that are needed for some applications. The first balanced family was invented by the Russians Adel'son-Vel'skii and Landis in 1962, and are called **AVL trees**. We will describe several balanced families, including AVL trees and red-black trees. The notion of balance can be applied to non-binary trees; we will study the family of (a, b)**-trees** and generalizations. Tarjan [11] gives a brief history of some balancing schemes.

STUDY GUIDE: all our algorithms for search trees are described in such a way that they can be internalized, and we expect students to carry out hand-simulations on concrete examples. We do not provide any computer code, but once these algorithms are understood, it should be possible to implementing them in your favorite programming language.

hand-simulations expected!

§1. Search Structures with Keys

Search structures store a set of objects subject to searching and modification of these objects. Search structures can be viewed as a collection of **nodes** that are interconnected by pointers. Abstractly, they are just directed graphs with edge and/or vertex labels. 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 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.

¶1. Keys and Items. 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 $(Key, Data)$. Besides an item, each node also stores one or more pointers to other nodes. Since the definition of a node includes pointers to nodes, this is a recursive definition. 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 it can almost anything, but `next` has to be a pointer to nodes. This field information constitutes the "type" of the node. To access these fields, we write N.`key`, N.`data` or N.`next`. The type of N.`next` is not that of 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`

[1] The concept of **locatives** introduced by Lewis and Denenberg [7] 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] For instance, `C++` would distinguish between nodes (N) and pointers (u) to nodes, and we would write $u \to$ `key`, $u \to$ `data`, etc.

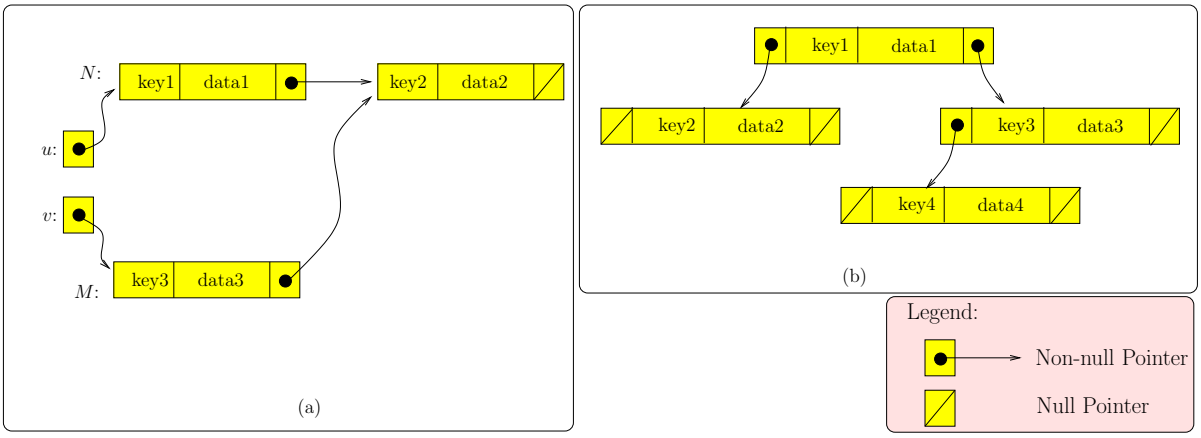


Figure 1: Two Kinds of Nodes: (a) linked lists, (b) binary trees

and u.`next` to access the fields in the node. There is a special pointer value called the **null pointer** or `nil` **value**. It points to nothing, but as a figure of speech, we may 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 box with a slash line.

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 un-specified order.

very informally speaking

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 admit all these four assignment possibilities, there is little distinction between manipulating nodes and their pointers. *This is what we meant earlier, when we said that our notion of nodes will variously look like ordinary variable N or pointers u.* Indeed the `Java` language eschews pointers, and introduces an intermediate concept called reference.

The four main players in our story are the two variables u and N, the pointer value of u, and the node that N refers to. This corresponds to the four references in the tale of Alice and the White Knight at the beginning of this chapter. We may use a 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.

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

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.* Our default is the **unique key assumption**, that the keys in a keyed search structure are unique. Equivalently, distinct items have the distinct keys. In the few places where we drop this assumption, it will be stated explicitly.

What is the point of 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!

¶2. 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. For simplicity, the default assumption is that items have unique keys. 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. 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 a given key K and S, returns a node u in S such that the item in u has key K. If no such node exists, it returns $u =$ `nil`. Next to `lookUp`, perhaps the next most important operation is `insert`.

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

§2. Abstract Data Types

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

Java fans: ADT = interface

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 languages 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.

¶3. 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*
		`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 the 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. 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 only one structure S, we may 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. These are constant time operations.

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 $\texttt{lookUp}(K)$ has the property that $u.\texttt{key} = K$. In conventional programming languages such as `C`, nodes are usually represented by pointers. In this case, the `nil` pointer is returned by the `lookUp` function when there is no item in S with key K.

In case no such item exists, or it is not unique, some convention should be established. At this level, we purposely leave this under-specified. Each application should further clarify this point. For instance, in case the keys are not unique, we may require that $\texttt{lookUp}(K)$ 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 '$\texttt{delete}(\texttt{lookUp}(K))$'. In case $\texttt{lookUp}(K)$ returns an iterator, we would expect the deletion to be performed over the iterator. Another useful extension is the **replacement operation** $\texttt{replace}(K, D)$: 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 $S.\texttt{deleteMin}()$ 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 $\texttt{delete}(\texttt{min}())$. But because of its importance, $\texttt{deleteMin}()$ 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 $S.\texttt{split}(K) \to S'$ 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 $S.\texttt{merge}(S')$ 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.

¶4. 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 [11]. For concreteness, we use the variety that Tarjan calls **endogeneous doubly-linked list**. Endogeneous means the item is stored in the node itself: thus from a node u, we can directly access $u.\texttt{key}$ and $u.\texttt{data}$. Doubly-linked means u has two pointers $u.\texttt{next}$ and $u.\texttt{prev}$. These two pointers satisfies the invariant $u.\texttt{next} = v$ iff $v.\texttt{prev} = u$. 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:

```
ListTraversal(L):
    u ← L.head
    while (u ≠ nil)
        [VISIT(u)]
        u ← u.next
    [CLEANUP()]
```

List traversal Shell

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 replace "VISIT(u)" by 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:

```
u.next.prev ← u.prev
u.prev.next ← u.next
del(u)
```

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

¶5. 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 balanced binary trees. 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. Both `insert`(K, D) and `delete`(u) are preceded by `lookUp`'s, which we know takes $\Theta(n)$ 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 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.

¶6. 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 further 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.

If the deletion in dictionaries are based on keys (see comment above) then we may think of a 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 all 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.

¶7. Application to 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`:

```
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.

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 ¶4, 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: 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 2.4: Repeat the previous question for the priority queue ADT. ♢

Exercise 2.5: 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".

the universal operation?

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 $u.\texttt{key}$ called its key. The keys in Figure 2 are integers, used simply as identifiers for the nodes.

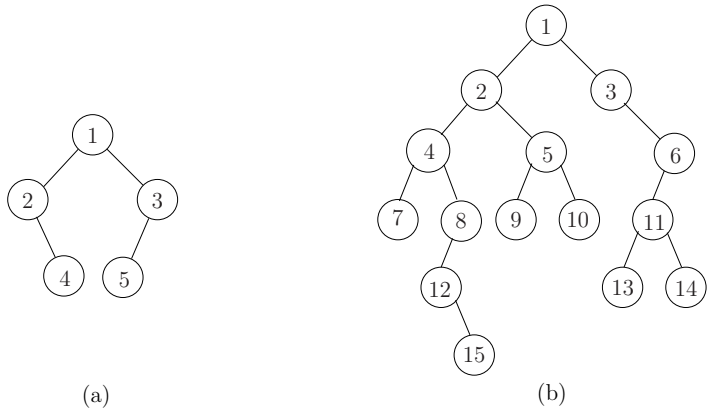


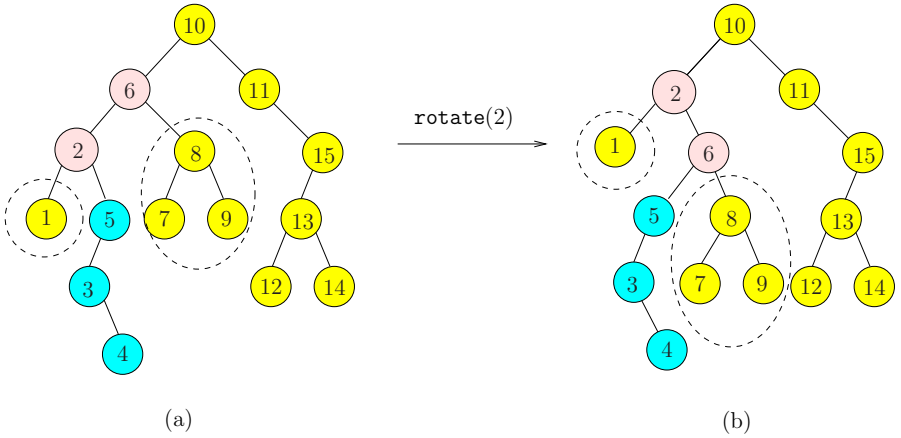
Figure 2: Two binary (not search) trees: (a) small, (b) big

Briefly, a binary tree T is a set N of **nodes** where each node u has two pointers, $u.\texttt{left}$ and $u.\texttt{right}$. The set N is either the empty set, or N has a special node u called the **root**. The remaining nodes $N \setminus \{u\}$ are partitioned into two sets of nodes that, recursively, form binary trees, T_L and T_R. If N is non-empty, then the root u has two fields, $u.\texttt{left}$ and $u.\texttt{right}$, that 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 $u.\texttt{left}$ ($u.\texttt{right}$) is $\texttt{nil}$.

A few other useful definitions: all the nodes of a binary tree of the same depth d forms the dth **level**. The dth level is **complete** if it has 2^d nodes. 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. Also recall the notion of a complete binary tree (Lecture I, Appendix A). A complete binary tree of whose size is one less than a power of 2 is said to be perfect.

Our definition of binary trees is based on **structural induction**. 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 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.



(a) (b)

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

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

BST are binary trees that satisfy the BST property!

$$u_L.\texttt{key} < u.\texttt{key} \leq u_R.\texttt{key}. \tag{1}$$

for all left descendent u_L and and all right descendent u_R of u. Please verify that the binary search trees in Figure 3 obey (1) at each node u.

The "standard mistake" is to replace (1) 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...

> Fundamental Rule about binary trees: *most properties about binary trees are best proved by induction on the structure of the tree. Likewise, algorithms for binary trees are often best described using structural induction.*

¶8. 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

$$\mu(h) = h + 1. \tag{2}$$

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

$$M(h) = 2^{h+1} - 1 \tag{3}$$

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 (3) 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

$$h \geq \lg(n + 1) - 1. \tag{4}$$

Informally, *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.

¶9. Lookup. The algorithm for key lookup 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 (remember the Fundamental Rule 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: (1) We can return the first node u we found to have the given key K. (2) We may insist on locating all nodes whose key is K.

In any case, requirement (2) can be regarded as an extension of (1), 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 (1) in the following.

¶10. 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.

¶11. Rotation. Roughly, to rotate a node u means to make the parent of u become its child. The set of nodes is unchanged. Rotation is not an operation in our list of ADT operation (§2), but it is critical for binary trees. On the face of it, rotation does not do anything essential: it is just redirecting some parent/child pointers. Two search structures that store exactly the same set of items are said to be **equivalent**. Rotation is an **equivalence transformation**, i.e., it transforms a binary search tree into an equivalent one. Remarkably, we shall show that rotation can[3] form the basis for all other binary tree operations.

The operation $\texttt{rotate}(u)$ 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 $\texttt{rotate}(u)$ amounts to the following transformation of T (see Figure 4).

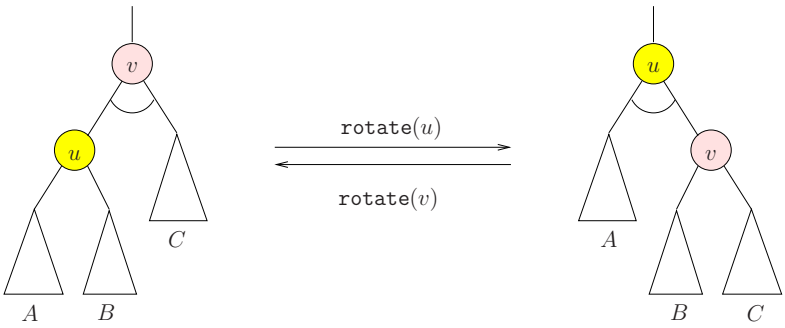


Figure 4: Rotation at u and its inverse.

[3] Augmented by the primitive operations of adding or removing a node.

In `rotate`(u), we basically want to invert the parent-child relation between u and its parent v. The other transformations are more or less automatic, given that the result is to remain a binary search tree. If the subtrees A, B, C (any of these can be empty) are as shown in Figure 4, then they must re-attach as shown. This is the only way to reattach as children of u and v, since 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. Actually, only the parent of the root of B has switched from u to v. Notice that 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; see Figure 4.

¶12. 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[4] 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.

¶* 13. 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.`parent` = `nil`, and u is type left iff u.`parent.left` $= u$. Clearly, `rotate`(u) is sensitive to the type of u. In particular, if u is a root then `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 $\overline{\texttt{right}} = \texttt{left}$.

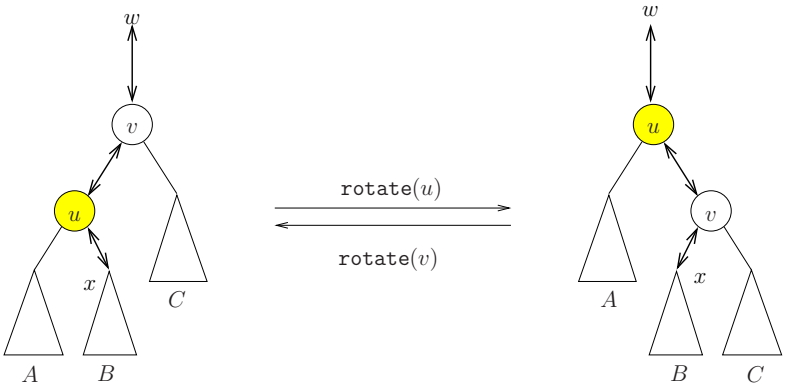


Figure 5: Links that must be fixed in `rotate`(u).

We are ready to discuss the `rotate`(u) subroutine. We assume that it will return the (same) node u. Assume u is not the root, and its type is $T \in \{\texttt{left}, \texttt{right}\}$. Let $v = u$.`parent`, $w = v$.`parent` and $x = u.\overline{T}$. Note that w and x might be `nil`. Thus we have potentially three

x is an inner grandchild of v

[4] If this were to happen, the subtrees A, B, C needs to be appropriately relabeled.

child-parent pairs:

$$(x, u), (u, v), (v, w). \tag{5}$$

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

$$(x, v), (v, u), (u, w). \tag{6}$$

These pairs are illustrated in Figures 5 and 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.

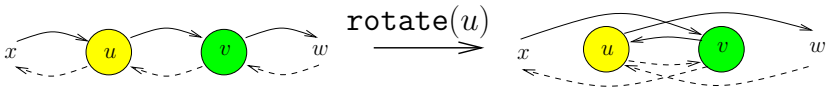


Figure 6: Simplified view of `rotate`(u) as fixing a doubly-linked list (x, u, v, w).

Such re-assignments must be done in the correct order. It is best to see what is needed by thinking of (5) as a doubly-linked list (x, u, v, w) which must be converted into the doubly-linked list (x, v, u, w) in (6). This is illustrated in Figure 6. For simplicity, we use the terminology of doubly-linked list so that u.`next` and u.`prev` are the forward and backward pointers of a doubly-linked list. Here is the code:

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

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

¶14. 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 with a pair of variants, called **left-rotation** and **right-rotation**. These can be defined 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 the usual 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 $\texttt{left-rotate}(u)$ or $\texttt{right-rotate}(u)$, we need to update one of the child pointers of p.

¶15. 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 $\texttt{rotate}(u)$ in this figure.]

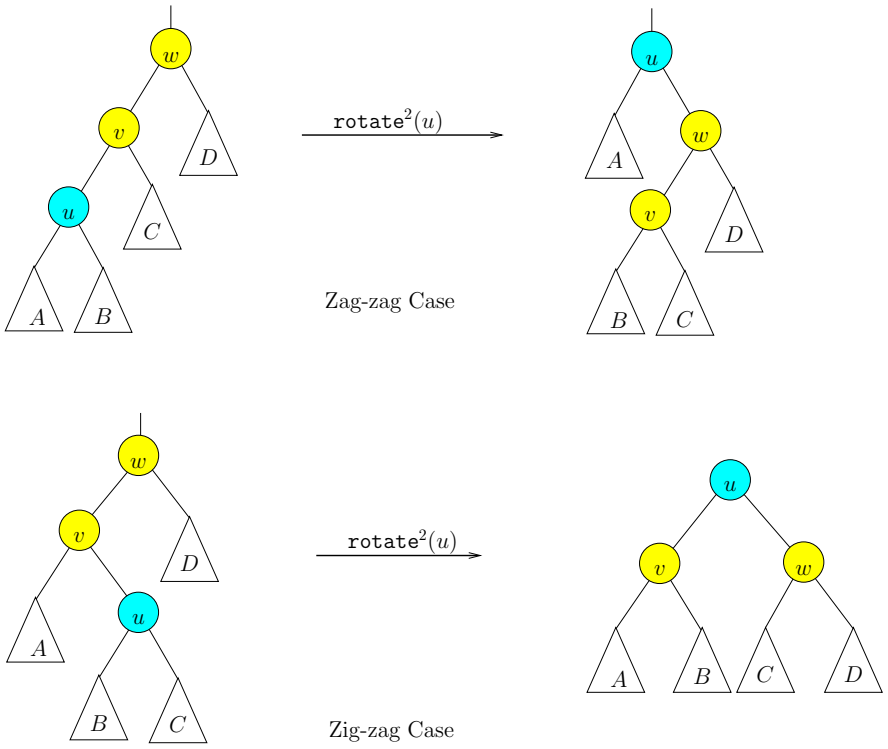
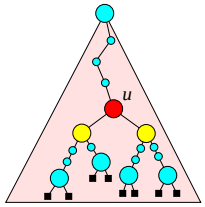


Figure 7: Two outcomes of $\texttt{rotate}^2(u)$

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**.

These two cases are also known as the zig-zig (or zag-zag) and zig-zag (or zag-zig) cases, respectively. 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.

¶16. Five Canonical Paths from a node. A **path** is a sequence of nodes $(u_0, u_1, \ldots, u_n)$ where each u_i is a child of u_{i-1}, or 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. 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.



5 *paths from a node*

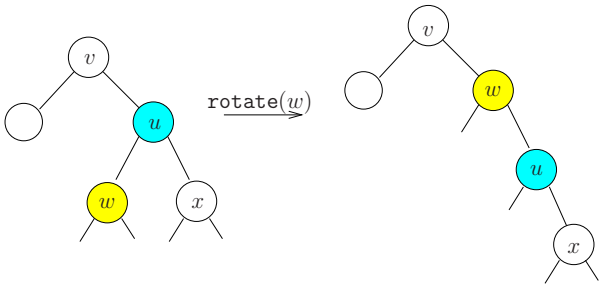


Figure 8: Reduction of the left-spine of u after $\texttt{rotate}(u.\texttt{left}) = \texttt{rotate}(w)$.

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

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

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

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

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

¶17. 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.

If we maintain information about the left and right spine heights of nodes (Exercise), and the right spine of u is shorter than the left spine, we can perform the while-loop using right-rotations to minimize the number of rotations. To avoid maintaining 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.

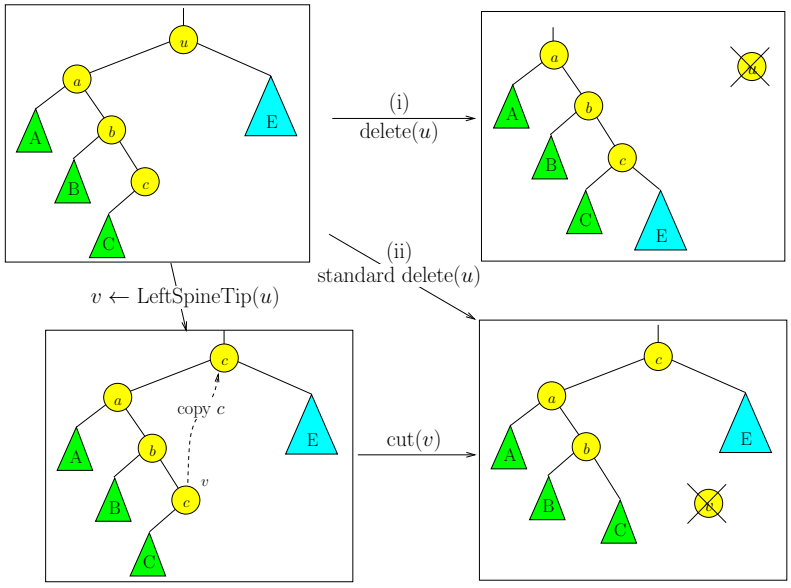


Figure 9: Deletion: (i) Rotation-based, (ii) Standard.

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

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

```
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).

¶19. Inorder listing of a binary tree.

Lemma 2. *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 2 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.

¶20. 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 3. *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.`right` $\neq$ `nil` then v is the tip of the right-spine of u.
(ii) If u.`right` $=$ `nil` then u is the tip of the left-spine of v.

It is easy to derive an algorithm for `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 `pred`(u) is similar. The Exercise develops a rotation-based version of successor or predecessor.

¶21. 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.

¶22. 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.`left` $=$ `nil`. At this point, we can perform $Cut(u)$ (see ¶17). If you like, you can perform right rotations instead of left rotations.

¶23. 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.`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

¶24. Complexity. Let us now discuss the worst case complexity of each of the above operations. They are all $\Theta(h)$ where h is the height of the tree. It is therefore desirable to be able to maintain $O(\log n)$ bounds on the height of binary search trees.

We stress that 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. If this cost is not an issue, then rotation-based algorithms are attractive because of 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: 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 3.3: Suppose the set of keys in a BST are no longer unique, and we want to modify the `lookUp`(u, K) function to return a linked list containing all the nodes containing key K in a subtree T_u rooted at u. Write the pseudo-code for $LookUpAll(u, K)$. ♢

Exercise 3.4: 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 3.5: TRUE or FALSE: Recall that a rotation can be implemented with 6 pointer assignments. Suppose a binary search tree maintains successor and predecessor links (denoted u.`succ` and u.`pred` in the text). Now rotation requires 12 pointer assignments. ♢

Exercise 3.6: (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 necessary modifications to your algorithms in (a) in case the binary trees do not have parent pointers. ♢

Exercise 3.7: 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. HINT: although we only require that you show the trees at the end of the operations, we recommend that you show selected intermediate stages. This way, we can give you partial credits in case you make mistakes!

(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 3.8: Give the code for rotation which uses temporary variables. ♢

Exercise 3.9: 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 $u.\texttt{next.prev.prev} \leftarrow u.\texttt{prev}$ 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 3.10: 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 3.11: Open-ended: The problem of implementing $\texttt{rotate}(u)$ 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., $\texttt{next}, \texttt{prev}, \texttt{left}, \texttt{right}$) 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 $\texttt{rotate}(u)$)? Under what conditions is the transformation achievable at all (using more intermediate variables? We also want to achieve minimum time. ♢

Exercise 3.12: The goal of this exercise is to show that *if T_0 and T_1 are two equivalent binary search trees, then there exists a sequence of rotations that transforms T_0 into T_1.* Assume the keys in each tree are distinct. We explore two strategies.

thus rotation is a "universal" equivalence transformation.

(a) One strategy is to first make sure that the roots of T_0 and T_1 have the same key. Then by induction, we can transform the left- and right-subtrees of T_0 so that they are identical to those of T_1. Describe an algorithm $A(T_1, T_2)$ that implements this strategy. The algorithm A does not modify T_2 at all, but transforms T_1 by rotations until T_1 has the same shape as T_2. Of course, we assume that T_1, T_2 are equivalent BST's.
(a') Let $R_A(n)$ be the worst case number of rotations of algorithm A on trees with n keys. Give a tight analysis of $R_A(n)$.
(b) Another strategy is to show that any tree can be reduced to a canonical form. For canonical form, we choose those binary search trees that form a left-list. A **left-list** is a binary tree in which every node has no right-child. If every BST can be rotated into a left-list, then we can rotate from any T_0 to any T_1 as follows: since T_0 and T_1 are equivalent, they can each be rotated into the same left-list L. To rotate from T_0 to T_1, we first transform T_0 to L, and then apply the *reverse* of the sequence of rotations that

transform T_1 to L. Give an explicit description of an algorithm $B(T)$ that transforms any BST T into an equivalent BST that is a left-list.
(b') Let $R_B(n)$ be worst case number of rotations for algorithm $B(T)$ on trees with n keys. Give a tight analysis of $R_B(n)$. ♢

Exercise 3.13: Prove Lemma 2, 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: remember the Fundamental Rule about binary trees. ♢

Exercise 3.14: 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 3.15: 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 3.16: 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 3.17: 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 3.18: 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 3.19: The text gave a conventional algorithm for successor of a node in a BST. Give the rotation-based version of the successor algorithm. ♢

Exercise 3.20: 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 3.21: Suppose we allow allow duplicate keys. Under (1), 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 3.22: Consider the priority queue ADT. Describe algorithms to implement this ADT when the concrete data structures are binary search trees.
(b) Analyze the complexity of your algorithms in (a). ♢

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 ¶4, when implemented ADT operations using linked lists.

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

¶25. 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.`left`, then you visit u, then recursive do in-order traversal of u.`right`. Here is the shell for this traversal:

Fundamental Rule of 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. For traversals, the BASE macro can be expanded into the following single line of code:

```
BASE(u):
    if (u=nil) Return.
```

The VISIT(u) macro is simply:

```
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 simply produce the sequence

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

This yields an interesting conclusion: *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.* This is because once we have such at BST, we can do an in-order traversal to list the keys in sorted order.

¶26. 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

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

¶27. 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

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

¶28. 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.

¶29. 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, modifying (simplifying!) it as follows:

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. *Hear! Hear!*

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. ♢

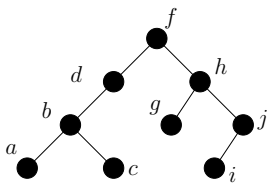


Figure 10:

Exercise 4.4: Here is the inorder and postorder 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 BST. ♢

Exercise 4.5: Tree reconstruction from key-listings in tree traversals. This is a slightly problem from the previous question. In the previous problem, we want to reconstruct a BST from the list of nodes from various traversals. Now, instead of nodes, we are given the

keys in a traversal. Instead of two lists, we only need one 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 4.6: 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: Give a recursive routine called $CheckBST(u)$ which checks whether the binary tree T_u rooted at a node u is a binary search tree (BST). Unfortunately, we cannot afford to simply return a Boolean value only, because in recursive calls, the parent need to receive some extra information from the children. So we design $CheckBST(u)$ to return a pair (ℓ, r) of keys. Normally, (ℓ, r) encodes the minimum and maximum keys in T_u. But we also use this pair to tell you whether T_u is BST or not: this happens when $\ell > r$. Assume that each non-nil node u has the three fields, $u.\texttt{key}, u.\texttt{left}, u.\texttt{right}$. ♢

Exercise 4.10: The previous exercise yields a recursive subroutine $CheckBST(u)$ to check if T_u is a BST. This exercise explores an alternative solution: the recursive subroutine $bCheckBST(u, min, max)$ that returns a Boolean value; the value is true iff T_u is a BST whose keys lies in the half-open range $[min, max)$. REMARK: we also define $bCheckBST(u)$ to be $bCheckBST(u, -\infty, +\infty)$. For simplicity, assume T_u has no duplicate keys. ♢

Exercise 4.11: In the previous homework, you designed a recursive subroutine $CheckBST(u)$ to check if the binary tree T_u rooted at u represents a BST. We now explore a different solution. Design a recursive subroutine $bCheckBST(u, min, max)$ that returns a Boolean value; the value is true iff T_u is a BST with keys lying in the range $[min, max]$. REMARK: We may define $bCheckBST(u)$ to be $bCheckBST(u, -\infty, +\infty)$. ♢

Exercise 4.12: 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 4.13: 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. If you like, your solution may consist of mutually recursive algorithms. Your overall algorithm must achieve this exponential complexity without any trivial redundancies. E.g., we should not be able to delete statements from your code and still achieve a correct program. Thus, we want to avoid a trivial solutions of this kind:

```
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

We discuss some important variations of our standard treatment of binary search trees (BST). For instance, an alternative way to use binary trees in search structures is to only store keys in the leaves. There are also notions of implicit BST: this means that the search keys are not explicitly stored as such in the tree. Another notion of implicitness is where the child/parent links of the BST is not directly stored, but computed. We can also store various auxiliary information in the BST such as height, depth or size information. We can also maintain additional pointers such as level links, or successor/predecessor links.

¶30. Extended binary trees. There is an alternative view of binary trees; following Knuth [5, p. 399], we call them **extended binary trees**. For emphasis, the original version will be called **standard binary trees**. In the extended trees, every node has 0 or 2 children; nodes with no children are called[5] **nil nodes** while the other nodes are called **non-nil nodes**. See Figure 11(a) for a standard binary tree and Figure 11(b) for the corresponding extended version. In this figure, we see a common convention (following Knuth) of representing nil nodes by black squares.

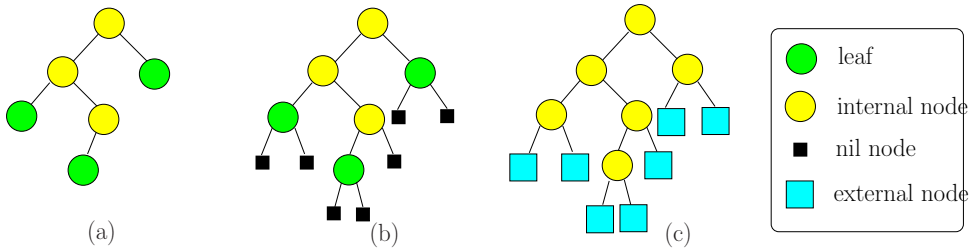


Figure 11: Binary Trees: (a) standard, (b) extended, (c) external

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

> *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.*

Who cares about nil nodes?

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 making the corresponding pointer take the `nil` value.

The concept of a "leaf" of an extended binary tree is apt to cause some confusion: we shall use the "leaf" terminology so as to be consistent with standard binary trees. A node of an extended binary tree is called a **leaf** if it is the leaf of the corresponding standard binary tree. Alternatively, a leaf in an extended binary tree is a node with two nil nodes as children. *Thus a nil node is never a leaf.*

¶31. Exogenous versus Endogenous Search Structures Tarjan [11], 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.* Figure 12 gives his interpretation of this definition for linked lists.

What is the relative advantage of either form? The exogenous case has an extra level of indirection (through pointers) which uses extra space. But on the other hand, it means that

[5] 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".

the actual data can be freely re-organized more easily, independently of the search structure. In databases, this freedom is important, and the exogenous search structure are called "indexes". Database users can freely create and destroy such indexes for the set of items. This allows a collection of items can be searched using different search criteria. Recall that each key is associated with some data, and such key-data pairs constitute the items for searching.

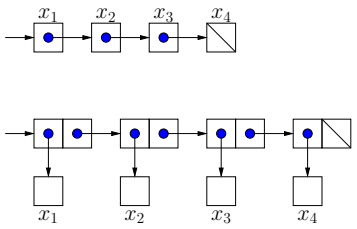


Figure 12: Endogenous/Exogenous

There is a simple way to understand the endogenous/exogenous distinction. In our framework of items as key-data pairs, we can interpret it to mean two ways to organize items. We can directly store the data with its associated key (endogenous), or we can store a pointer to the data with its associated key (exogenous). This seems consistent with Figure 12. However, this seems to be a weak and uninteresting distinction. For instance, should a standard BST be regarded as endo or exo? The weak distinction allows both views, since we can simply replace the data item in each BST node by a pointer to the data. We next provide a much stronger distinction between the endogenous and exogenous.

¶32. External Binary Search Tree. To turn extended binary trees into search structures, we 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 (sentinels in algorithms). For a more substantial transformation, we will modify extended BST by replacing nil nodes with nodes 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 11(c). We can, of course, use external binary trees as a BST. But their special structure allows us to use them in a new way: we propose to store items only in external nodes, and to store only keys in internal nodes in order to allow Lookup of items in external nodes. For this purpose, the keys (whether in internal or external nodes) must satisfy the usual BST Property: at any internal node u,

So that is why 'internal nodes' are so-called...

$$u_L.\texttt{key} < u.\texttt{key} \leq u_R.\texttt{key} \tag{7}$$

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**.

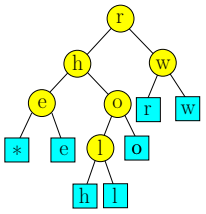
We interpret external BST as falling under Tarjan's notion of exogenous data structure, while the standard BST is considered an endogenous data structure. In Lecture VI, we will give an application of external BST in dynamic string compression, where a standard BST will not do. External BST illustrates the concept of **external search structures**. Informally, the idea is that items are kept separate[6] from from actual search structure itself. This important idea will be taken up again in §8 under (a, b)-search trees below.

The preceding description of external BST's is sufficient for Lookups, but in order to support insertion and deletion, we must introduce more properties. In particular, we must solve the question: *where do the keys in internal nodes come from?* Since the user only inserts and deletes items, or use keys to lookup items, it is incumbent on the the insertion and deletion algorithms to generate these keys automatically. We do not want to assume the ability to generate brand new keys; but it is reasonable to duplicate keys that are already in the external nodes. Using

[6] Though not in the weak sense of just redirecting the data by pointers.

this ability alone, we can now provide the 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)

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



An external BST

Each of these keys (with the exception of *) are duplicated in the internal nodes. In general, this is true of our external BST: with a single exception of the minimum item, there is a one-one correspondence between keys in the internal nodes and keys in the external nodes. This correspondence is possible because that the number I of internal nodes in a full binary tree and the number E of external nodes in a full binary tree satisfies the relation $E = I + 1$. This relation is easily proved by structural induction. This connection is made precise as follows:

External BST Property: For any internal node u and external node v,

$$u.\texttt{key} = v.\texttt{key} \iff v \text{ is the successor of } u. \tag{8}$$

We say u and v are **partners** when the either side of (8) holds. Thus, u is the left-most external node (storing the minimum key) iff it has no partner.

¶33. Implementation of External BST. Let us consider algorithms for insertion and deletions in External BSTs. We shall see that for the deletion algorithm, it will be useful for each external node u to point to its partner v. But it is not necessary for v to point back to u. In other words, we would like an external node u to have a field $u.\texttt{pred}$ to point to v. There is a natural implementation (the "standard implementation") of this. Let us assume that all nodes u, whether internal or external, are structurally the same and allocated the same fields. In particular, they will have the usual fields, $u.\texttt{left}, u.\texttt{right}$. If u is on internal node, $u.\texttt{left}, u.\texttt{right}$ are non-nil and point to the two children of u, as expected. For an external node u, these pointers have no use. So we shall re-purpose them, using $u.\texttt{left}$ to point to u's predecessor and setting $u.\texttt{right} = \texttt{nil}$. Thus, $u.\texttt{right} = \texttt{nil}$ becomes the criterion for an external node and $u.\texttt{left}$ is re-interpreted as $u.\texttt{pred}$. Finally, the leftmost external node u can be characterized by the property $u.\texttt{left} = u.\texttt{pred} = \texttt{nil}$.

We now make a simple observation about Lookup Operation: *suppose we reach an external node u during the Lookup of a key k. Then $u.\texttt{key} \le k$.* To see this, we observe that the root-path of u must contain the predecessor $v = u.\texttt{pred}$. Moreover, u must lie in the right subtree of v. Therefore, in the Lookup of k, we must have encountered v, and continued the search recursively at $v.\texttt{right}$. This amounts to the conclusion $v.\texttt{key} \le k$.

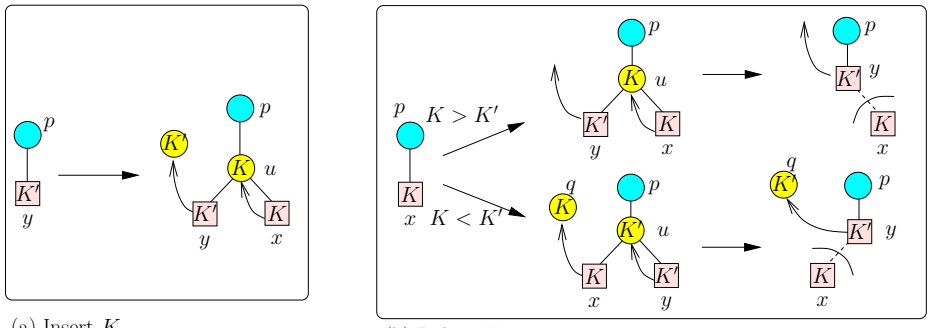
We now describe how insertion works, as illustrated in Figure 13(a). Say the item we want to insert is put into a node x, and we do a Lookup on $K = x.\texttt{key}$. This leads us to a external node y with key $K' = y.\texttt{key}$. If $K' = K$, then insertion fails. Otherwise, the above observation says $K' < K$. The remaining operations of the insertion is now simple. First, we create a new internal node u to take the place of y. We make y and x the left and right children of u, respectively. Moreover, the key in u will be (a duplicate of) K, and u is the predecessor of x. The predecessor of y is unchanged, but the parent p of y needs to replace y by u. The necessary assignments are given here:

u : $u.\texttt{parent} \leftarrow y.\texttt{parent}; u.\texttt{left} \leftarrow y;$ $u.\texttt{right} \leftarrow x;$ $u.\texttt{key} \leftarrow x.\texttt{key}$

x : $x.\texttt{parent} \leftarrow u; x.\texttt{left} \leftarrow u;$ $x.\texttt{right} \leftarrow \texttt{nil};$

y : $x.\texttt{parent} \leftarrow u;$

p : If $(p.\texttt{left} = y)$ then $p.\texttt{left} \leftarrow u$, else $p.\texttt{right} \leftarrow u$.



(a) Insert K (b) Delete K

Figure 13: Insertion and Deletion in external BST

In case of deletion of a key K, we first do a Lookup on K. This leads to an external node x. If $x.\texttt{key} \neq K$, then there is nothing to delete. Otherwise, we want to delete x. Let the sibling of x be y, and their common parent be u. In either case, we simply replace u by y, and discard x. But we need may have to update the key in an internal node. Let $K' = y.\texttt{key}$. If $K < K'$ and q is the predecessor of y, we must also update $q.\texttt{key}$ to K'. We leave the details to the reader. Deletion is illustrated in Figure 13(b). Observe that predecessor pointers are not needed for insertion, but are important for deletion.

¶34. 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).

¶35. Duplicate keys. We normally assume that the keys in a BST are distinct unless otherwise noted. But let us now briefly consider BST whose keys are not necessarily unique or distinct. When we do a lookup on a key K, *let us assume that we must visit every item with key K*. Now, this can be fairly expensive: in Figure 14(a), imagine having to search for the key 3.

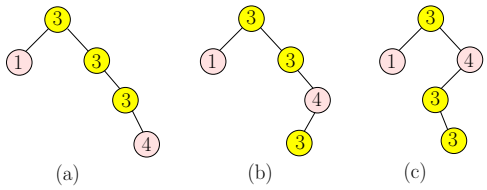


Figure 14: (a) arbitrary, (b) right-path rule

We consider this general question: suppose we have a node u whose key is K. How can we find all the nodes with key K in the subtree rooted at u? Let us define a procedure $FindAll(u)$ that returns a linked list L (i.e., iterator) containing all such nodes. The method is simple: the list L is initialized to u. Let v be the tip of the right-spine of u. If $v.Key = K$ then return L after appending with $FindAll(v)$; otherwise just return L. Note that if we have successor pointers, then v can be instantly located. This suggests that for BST's with duplicates, it may be worthwhile maintaining successor and predecessor pointers.

One way to handle simplify the algorithms for duplicate keys is to require the following **right-path rule**: *all items with the same key must lie on consecutive nodes of some right-path.* This is illustrated in Figure 14(b). We can view all the equal-key nodes on this right-path as a super-node for the purposes of maintaining height-balanced trees such as AVL trees. This amounts to keeping all the duplicate keys in a single linear list. So this rule should probably be restricted to situations where the number of duplicates is small.

Before discussing how to maintain this right-path rule, let us discuss how `lookUp` must be modified. When we look up on a key k, we can just return the first node that contains the key k. Alternatively, if there is a secondary key besides the (primary) key which might distinguish among the different items with primary key k, we can search the right-path for this secondary key. Now we must modify all our algorithms to preserve the right-path rule. In particular, insertion and rotation should be appropriately modified. What about deletion? If the argument of deletion is the node to be deleted, it is clearly easy to maintain this property. If the argument of deletion is a key k, we can either delete all items whose key is k or rely on secondary keys to distinguish among the items with key k.

Instead of the right-path rule, we could put all the equal-key items in an auxiliary linked list attached to a node. There are pros and cons in either approach. The "right path" organization of duplicate keys do not need any auxiliary structures. If the expected number of duplicated keys is small, it may be the best solution.

The right-path rule does not worry about balancing. Consider duplicate keys in the context of a balanced tree scheme like AVL trees. Imagine a BST with three keys, all duplicated. Then the BST Property, that $L < Root \leq R$ for all keys L (R) in the left (right) subtree, ensures that this tree is a right path. On the other hand, this right path violates the AVL Balance Property. To restore the AVL Balance Property, we must use the **modified BST Property**, namely, $L \leq Root \leq R$. The $FindAll(u)$ procedure above can be easily modified to recursively search both the left-spine tip as well as the right-spine tip.

¶36. Implicit Search Trees. 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 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 $i = 1, \ldots, n$,

$$HO(i): \quad A[i] \leq \min\{A[2i], A[2i+1]\}. \tag{9}$$

In (9), 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.

¶37. Parametric Binary Search Trees. Perhaps the most interesting variation of binary search trees is when the keys used for comparisons are only implicit. The information stored at nodes allows us to make a "comparison" and decide to go left or to go right at a node but this comparison may depend on some external data beyond any explicitly stored information. We illustrate this concept in the lecture on convex hulls in Lecture V.

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 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. REMARK: it suffices to show the external BST after each insertion.
(b) Show the tree after deleting `w` from the final tree in part(a). ♢

Exercise 5.3: We 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.
(a) Spell out the modifications to the Lookup, Insert, Delete Algorithms.
(b) What are the pros and cons of this approach? ♢

Exercise 5.4: 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 5.5: 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 5.6: Describe the various algorithms on binary search trees that store the size of subtree at each node. ♢

Exercise 5.7: 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 5.8: 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. 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 also have relatively simple insertion/deletion algorithms.

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 (2) and(3)), 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.

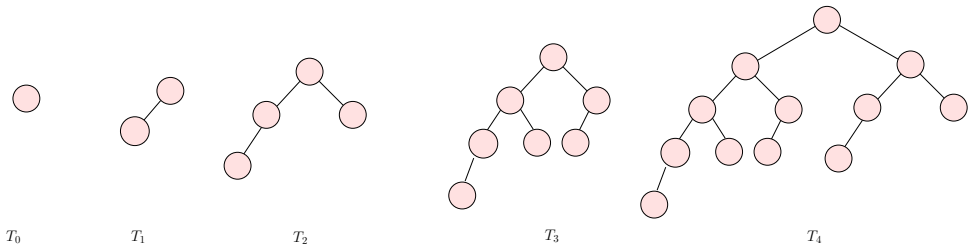


Figure 15: Canonical min-size AVL trees of heights 0, 1, 2, 3 and 4.

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

$$\mu(h) = 1 + \mu(h-1) + \mu(h-2), \qquad (h \geq 1). \tag{10}$$

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 (10) holds for $h = 1$. Here are some initial values of μ:

$$\mu(1) = 2, \mu(2) = 4, \mu(3) = 7, \mu(4) = 12, \mu(5) = 20, \mu(6) = 33, \mu(7) = 54, \mu(8) = 88$$

The function $\mu(h)$ helps us answer the following question: *what is the maximum height of an AVL tree of a given size n*? To see this, not that any AVL tree of size $\mu(h) - 1$ has height at most $h - 1$. So if $\mu(h) - 1 \geq n$ then a tree of size n has height at most $h - 1$. For instance, what is the maximum size of an AVL tree of size $n = 64$? We know that any AVL tree of size at most $87 = \mu(8) - 1$ has height at most $8 - 1 = 7$. So the answer is 7.

Let us now estimate the growth rate of μ. From (10), 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 $\phi = \frac{1+\sqrt{5}}{2}$, and this is be the positive root of the quadratic equation $x^2 - x - 1 = 0$. Hence, $\phi^2 = \phi + 1$.

Hey, to square ϕ, you just add 1

Lemma 4. *For $h \geq 0$, we have*

$$\phi^h \leq \mu(h) < 2\phi^h. \tag{11}$$

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

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

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.

We can further improve the lower bound on $\mu(h)$ in (11) by taking into account the "+1" term that was ignored in the above proof — See Exercises. It is the lower bound on $\mu(h)$ that is more important for us. For, if an AVL tree has n nodes and height h then

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

by definition of $\mu(h)$. The lower bound in (11) 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 4. Thus the height of AVL trees are at most 44% more than the absolute minimum. We have proved:

Corollary 5. *The family of AVL trees is balanced.*

¶38. 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.

REBALANCE PHASE: Let x be the parent of 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*

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.

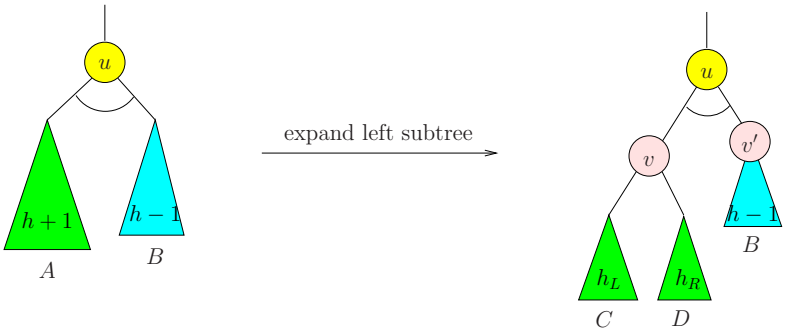


Figure 16: Node u is unbalanced after insertion or deletion.

¶39. 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)

$$h+1, \quad h, \quad h-1. \tag{12}$$

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.

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.

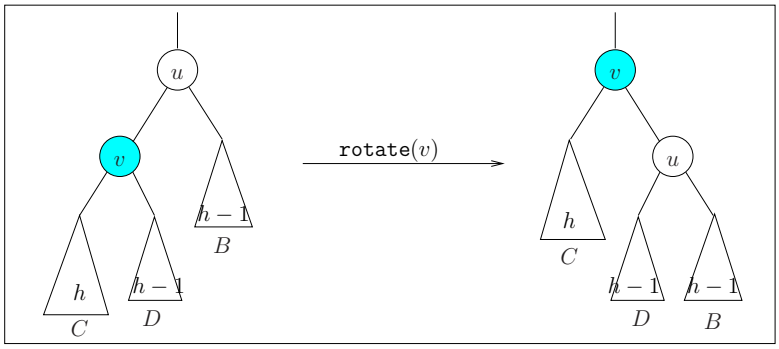
In both cases (I.a) and (I.b), the resulting subtree has height $h+1$. Since this was height before the insertion (see (12)), there are no unbalanced nodes further up the path to the root. Thus the insertion algorithm terminates with at most two rotations.

For example, suppose we begin with the AVL tree in Figure 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.

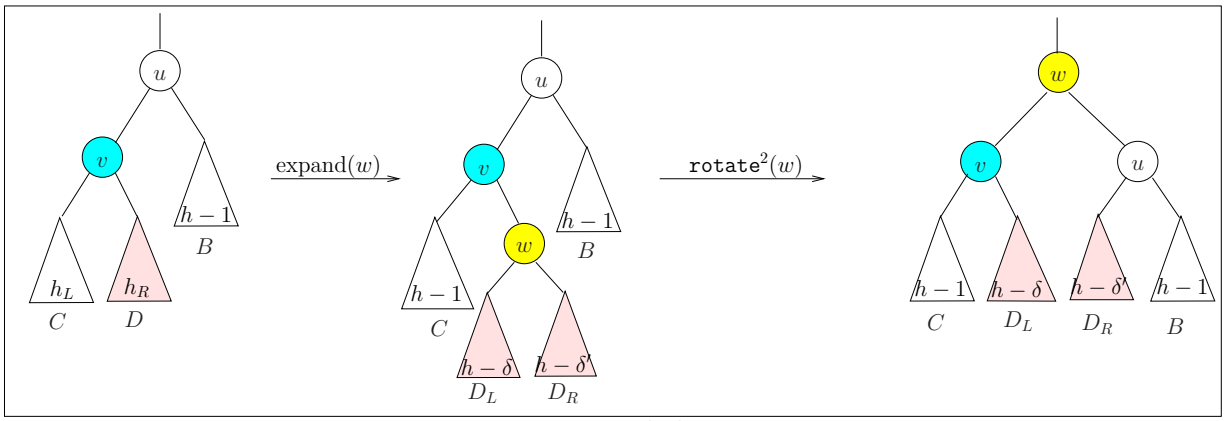
¶40. Deletion Rebalancing. Suppose the imbalance in Figure 16 comes from a deletion. The previous heights of u, v, v' must have been

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

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



CASE (I.a)



CASE (I.b)

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

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.

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

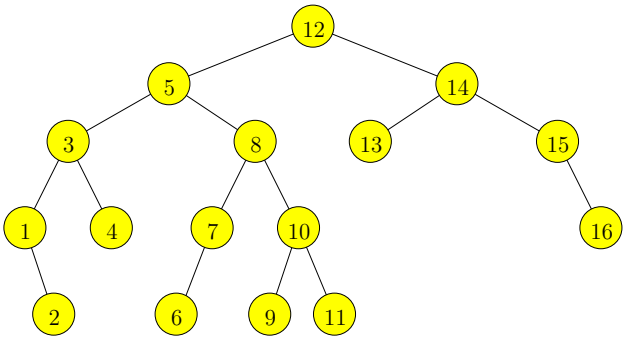


Figure 18: An AVL tree

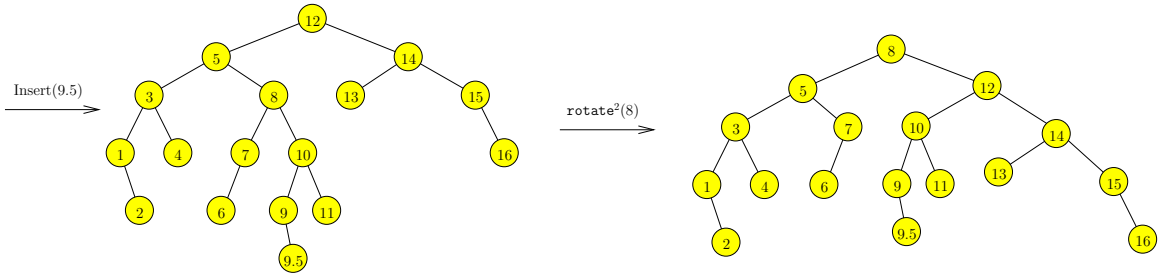


Figure 19: Inserting 9.5 into an AVL tree

¶41. 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 states $(00, 01, 10)$ for each node u. Let 00 mean that $u.\texttt{left}$ and $u.\texttt{right}$ have the same height, and 01 mean that $u.\texttt{left}$ has height one less than $u.\texttt{right}$, and similarly for 10. In simple implementations, we could just use an integer to represent this information. We leave it as an exercise to determine how to use these bits during rebalancing.

Hey, I thought it is $\Theta(\lg n)$

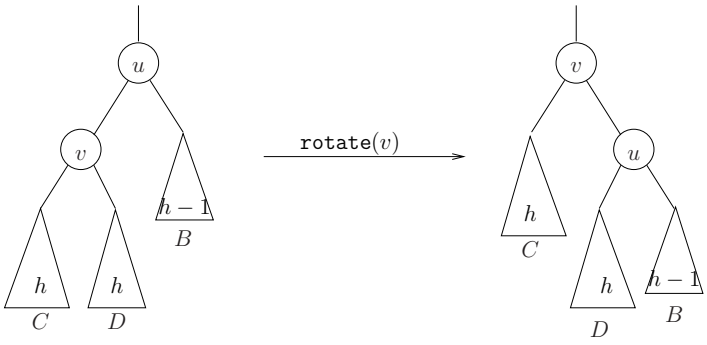


Figure 20: CASE (D.c): `rotate`(v)

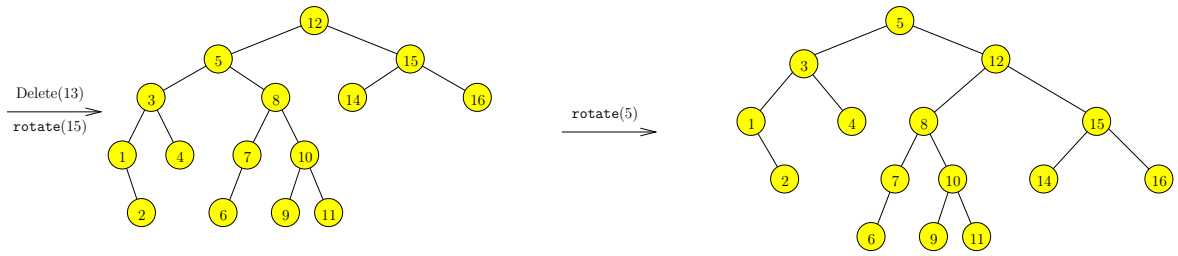


Figure 21: Deleting 13 from the AVL tree in Figure 18

¶42. Relaxed Balancing. Larsen [6] 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).

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 $O(|T|)$. HINT: Use shell programming. ♢

Exercise 6.3: 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 6.4: What is the minimum number of nodes in an AVL tree of height 10? ♢

Exercise 6.5: Prove that $\mu(h) = a\phi^h + b\widehat{\phi}^h - 1$ where $\phi, \widetilde{\phi} = \frac{1\pm\sqrt{5}}{2} = 1.6180\ldots, -0.6180\ldots$, and a, b are suitable constants. Determine a, b. ♢

Exercise 6.6: My pocket calculator tells me that $\log_\phi 100 = 9.5699\cdots$. What does this tell you about the height of an AVL tree with 100 nodes? ♢

Exercise 6.7: Show an AVL T with minimum number of nodes such that the following is true: there is a node x in T such that if you delete this node, the AVL rebalancing will require

two rebalancing acts. Note that a double-rotation counts as one, not two, rebalancing act. Can you make the 2 rebalancing acts be two single rotations? Two double-rotations? One single and one double rotation? ♢

Exercise 6.8: Consider the AVL tree in Figure 22.

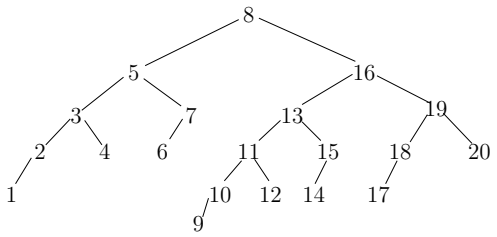


Figure 22: An AVL Tree for deletion

(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 22 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 6.9: (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 6.10: 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 22? ♢

Exercise 6.11: 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 6.12: (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 6.13: 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 6.14: 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 6.15: 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 (3) and $\mu(h)$ in (10).
(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 6.16: 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 6.17: Draw two AVL trees, with n keys each: the two trees must have different heights. Make n as small as you can. ♢

Exercise 6.18: 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 6.19: 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 upper and lower bounds on $\overline{\mu}(h)$. ♢

Exercise 6.20: 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 6.21: 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 6.22: 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 6.23: 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 6.24: 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 6.25: 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 6.26: Relaxed AVL Trees

Let us define **AVL(2) balance condition** to mean that at each node u in the binary tree, $|balance(u)| \leq 2$.

(a) Derive an upper bound on the height of a AVL(2) tree on n nodes.

(b) Give an insertion algorithm that preserves AVL(2) trees. Try to follow the original AVL insertion as much as possible; but point out differences from the original insertion.

(c) Give the deletion algorithm for AVL(2) trees. ♢

Exercise 6.27: To implement we reserve 2 bits of storage per node to represent the balance information. This is a slight waste because we only use 3 of the four possible values that the 2 bits can represent. 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 6.28: 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 null, $h'(u) := -2$ (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 the 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 conceptual difficulty of trying to use the family of 'AVL' trees as a general search structure? ♢

Exercise 6.29: 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) Prove 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 6.30: The AVL insertion algorithm makes two passes over its search path: the first pass is from the root down to a leaf, the second pass goes in the reverse direction. Consider the following idea for a "one-pass algorithm" for AVL insertion: during the first pass, before we visit a node u, we would like to ensure that (1) its height is less than or equal to the height of its sibling. Moreover, (2) if the height of u is equal to the height of its sibling, then we want to make sure that if the height of u is increased by 1, the tree remains AVL.

♢

End Exercises

§7. Size Balanced Trees

We specify that the ratio $s_L : s_R$ of the sizes of the two subtrees at any node should lie between $1/\beta$ and β where $\beta > 1$ is fixed for the family. This defines a balanced family of trees. This is more flexible than height balanced trees, and this is important for some applications. The price we pay is that we need up to $\lg n$ bits of balance information at each node.

We introduce another form of balance in binary trees. The **size** of a node u is the number of nodes in the subtree rooted at u. If Size$u$ is the size of u, **size balance** at u is

$$B(u) := \text{Size}u.\texttt{left}/\text{Size}u.\texttt{right}.$$

Let $0 < \alpha < 1/2$. A binary tree T has **bounded balance** α if $\alpha < B(u) \geq 1 - \alpha$ for each internal node u. For short, we say T is "$BB(\alpha)$". The family of $BB(\alpha)$ trees is a balanced family, *i.e.*, has logarithmic height. Bounded balance trees were introduced by Nievergelt and Reingold [10]. For more information, see Anderson [1].

In bounded balance trees, each node must store its own size. Thus $O(\log n)$ space is required at each node. This is inferior to the $O(\log \log n)$ space needed for height balanced schemes. Compensating for this, weight balanced trees are more flexible than height balanced trees for some applications. Give an example of multidimensional...

Let us first bound the height of a binary tree that is $BB(\alpha)$. Let $H(n)$ be the maximum height of a $BB(\alpha)$ binary tree of size n. Clearly, $H(1) = 0$ and $H(n) \leq 1 + H(n - \lceil \alpha n \rceil)$. This gives $H(n) \leq 1 + H(n - \alpha n - 1)$.

Exercises

Exercise 7.1: Show that weight balanced families are balanced in the usual sense. ♢

Exercise 7.2: Since in practice we need to reserve 2 bits of information per node in an AVL tree, let us try to take full advantage of this. Consider AVL trees in which the balance information at each node is $b = -1, 0, 1, 2$. Here b is the height of the left subtree minus the height of the right subtree. What are the advantages of this new flexibility? ♢

Exercise 7.3: Work out the details about how to use only one balance bit per AVL node. ♢

Exercise 7.4: Design a one-pass algorithm for AVL insertion and AVL deletion. ♢

Exercise 7.5:

(a) Show how to maintain the min-heap property in a binary tree under `insert` and `deleteMin`.

(b) Modify your solution in part (a) to ensure that the depth of the binary tree is always $O(\log n)$ if there are n items in the tree. ♢

Exercise 7.6: 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 23. Now define the

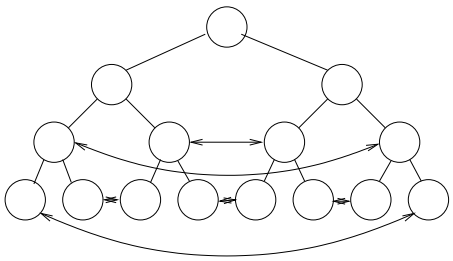


Figure 23: Partners

following parallel operations. The first operation is *sort*1: 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 *sort*2, and it is analogous to *sort*1 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

§8. (a,b)-Search 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 a choice of two integers,

$$2 \leq a < b. \tag{13}$$

An (a,b)**-tree** is a rooted, ordered[7] tree with the following structural constraints:

- DEPTH PROPERTY: All leaves are at the same depth.
- DEGREE BOUND: Let m be the number of children of an internal node u. This is also known as the **degree** of u. In general, we have the bounds

$$a \leq m \leq b. \tag{14}$$

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

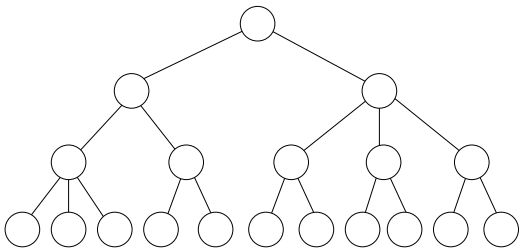


Figure 24: A $(2,3)$-tree.

Figure 24 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 where leaves have the same depth, we allow the number of children of an internal node to vary over a larger 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 iff c,d satisfy

$$2 \leq c \leq a < b \leq d. \tag{15}$$

E.g., Figure 24 could have represented an $(2,10)$-tree but not a $(3,4)$-tree.

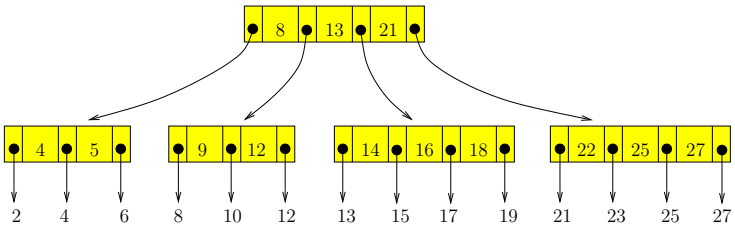


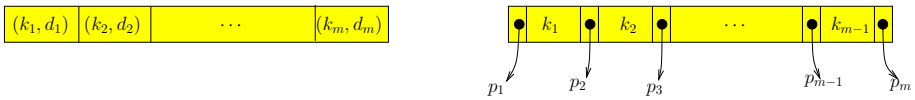
Figure 25: A (3,4)-search tree on 14 items

¶43. From Structure to Search. 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

[7] "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).

provide some intuition by looking an example of such a search tree in Figure 25. This tree is structurally a $(3,4)$-tree, at a minimum; but it could be a (a,b)-tree for any $2 \le a \le 3$ and $b \ge 4$. 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 25 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 follows. First, the organization in leaves are different than in internal nodes, as illustrated in Figure 26. The leaf organization is controlled by another pair of parameters a', b' that satisfy the inequalities $1 \le a' \le b'$. They are independent of a, b, but like a, b, they control the minimum and maximum number of items in leaves. Specifically:



$1 \le a' \le m \le b'$ (i) Leaf Node Organization

$2 \le a \le m \le b$ (ii) Internal Node Organization

Figure 26: Organization of nodes in (a,b)-search trees

- LEAF: Each leaf stores a sequence of items, sorted by their keys. Hence we represent a leaf u with m items as the sequence,

$$u = (k_1, d_1, k_2, d_2, \ldots, k_m, d_m) \tag{16}$$

  where $k_1 < k_2 < \cdots < k_m$. See Figure 26(i). In practice, d_i might only be a pointer to the actual location of the data. We must consider two cases. **NON-ROOT CASE:** suppose leaf u is not the root. In this case, we require

$$a' \le m \le b'. \tag{17}$$

  ROOT CASE: suppose u is the root. Since it is also a leaf, there are no other nodes in this (a,b)-search tree. We now require $0 \le m \le 2b' - 1$. This is relaxed compared to non-root leaves above. 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:

$$u = (p_1, k_1, p_2, k_2, p_3, \ldots, p_{m-1}, k_{m-1}, p_m) \tag{18}$$

  where p_i is a pointer to the i-th child of the current node. Note that the number of keys in this sequence is one less than the number m of children. Contrast with the organization (16) for a leaf-node. See Figure 26(ii). 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

$$k_{i-1} \le k < k_i, \tag{19}$$

  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 (1).

¶44. 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 are not significant (and they play roles analogous to a, b). There are two canonical choices: the simplest is $a' = b' = 1$. This means each leaf stores exactly one item. All our examples (e.g., Figure 25) use this default choice. Another canonical choice is $a' = a, \quad b' = b$. These considerations highlights the different roles that keys play in the leaves and in internal nodes.

So (a',b') is implicit!

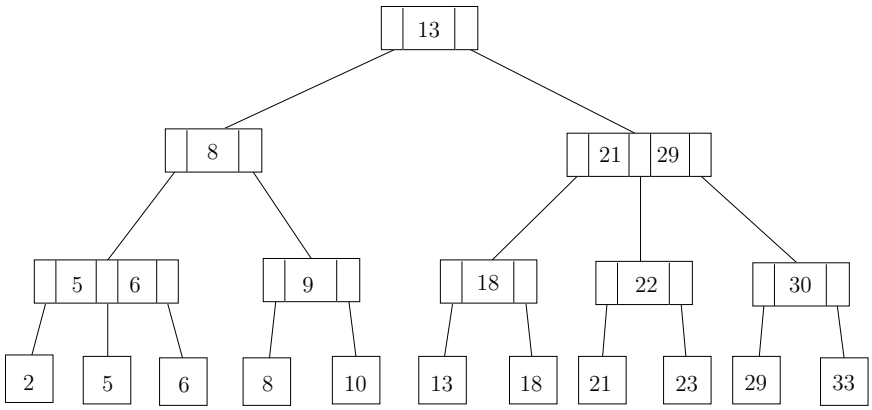


Figure 27: A $(2,3)$-search tree.

Another (a,b)-search tree is shown in Figure 27, for the case $(a,b) = (2,3)$. In contrast to Figure 25, here we draw it using a slightly more standard convention of representing the pointers as tree edges.

¶45. 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

$$b = 2a - 1 \tag{20}$$

(for any $a \geq 2$), we obtain the generalization of $2-3$ trees called **B-trees**. These were introduced by McCreight and Bayer [3]. When $(a,b) = (2,4)$, the 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 [8].

The B-tree relationship (20) is optimal in a certain[8] sense. Nevertheless, there are other benefits in allowing more general relationships between a and b. E.g., if we replace (20) by $b = 2a$, the amortized complexity of such (a,b)-search trees algorithms can improve [4].

¶46. 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:

[8] I.e., assuming a certain type of split-merge inequality, which we will discuss below.

- Base Case: suppose u is a leaf node given by (16). 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 (18). 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.

It is best to define $M(h)$ and $\mu(h)$ as the maximum and minimum (resp.) number of leaves in a (a,b)-tree of height h. Note that this definition differs from that in AVL trees in that we focus on the number of leaves rather than the size of the tree. That is because items is stored only in leaves of (a,b)-trees. Since $M(h)$ is attained if every internal node has b children, we obtain

$$M(h) = b^h. \tag{21}$$

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

$$\mu(h) = \begin{cases} 1 & h = 0, \\ 2a^{h-1}. & h \geq 1. \end{cases} \tag{22}$$

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

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

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 in general, b, b' determines the lower bound on h and 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.

¶47. Organization within a node. The keys in a node of an (a,b)-search tree must be ordered for searching, and manipulation such as merging or splitting two list of keys. Conceptually, we display them as in (18) and (16). Since the number of keys is not necessarily a small constant, the organization of these keys is an issue. In practice, b is a medium size constant (say, $b < 1000$) and a is a constant fraction of b. These ordered list of keys can be stored as an array, a singly- or doubly-linked list, or even as a balanced search tree. These have their usual trade-offs. With an array or balanced search tree at each node, the time spent at a node improves from $O(b)$ to $O(\log b)$. But a balanced search tree takes up more space than using a plain array organization; this will reduce the value of b. Hence, a practical compromise is to simply store the list as an array in each node. This achieves $O(\lg b)$ search time but each insertion and deletion in that node requires $O(b)$ time. This tradeoff is reasonable under the assumption that searches are much more frequent than insertion/deletions. In Exercises, you may assume this default organization. *When we take into account the effects of secondary memory (see below), the time for searching within a node is negligible compared to the time accessing each node.* This argues that the overriding goal in the design of (a,b)-search trees should be to maximize b and a.

the central tenet of (a,b)-trees!

¶48. The Standard Split and Merge Inequalities for (a,b)-Search trees. To support efficient insertion and deletion algorithms, the parameters a, b must satisfy an additional inequality in addition to (13). This inequality, which we now derive, comes from two low-level operations on (a,b)-search tree. These **split** and **merge** operations are called as subroutines by the insertion and deletion algorithms (respectively). 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} $(i = 1, \ldots, m-1)$. Then 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:

$$a \leq \left\lfloor \frac{b+1}{2} \right\rfloor. \tag{23}$$

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 are forced to **merge** a node with $a-1$ children with a node with a children. The resulting node has $2a-1$ children, and to satisfy the branching factor bound of (a,b)-trees, we have $2a - 1 \leq b$. Thus we require the following merge inequality:

$$a \leq \frac{b+1}{2}. \tag{24}$$

Clearly (23) implies (24). However, since a and b are integers, the reverse implication also holds! Thus (23) and (24) 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 (13) is $(a,b) = (2,3)$; this case has been mentioned above. The case of equality in (23) and (24) gives us $b = 2a - 1$; this is another special case mentioned earlier, and in the literature, the $(a, 2a-1)$-search trees are known as B**-trees**. Sometimes, the condition $b = 2a$ is used to define B-trees; this behaves better in an amortized sense (see [8, Chap. III.5.3.1]).

Neat Trick. The above argument uses the following little lemma: suppose x, y are real numbers such that

$$x \geq y. \tag{25}$$

If you also know that x is an integer, then the (25) is equivalent to the apparently stronger inequality

$$x \geq \lceil y \rceil . \tag{26}$$

Similarly, if y is an integer, then (25) is equivalent to the apparently stronger inequality

$$\lfloor x \rfloor \geq y. \tag{27}$$

Combining both remarks, if either x or y is an integer, then (25) is equivalent to the apparently stronger inequality

$$\lfloor x \rfloor \geq \lceil y \rceil . \tag{28}$$

The following lemma captures the preceding argument that says that (24) implies (23). It amounts to strengthening an inequality if one side is known to be integer. We will have occasion to re-use this argument several times below.

Lemma 6. *Let x, y be real numbers satisfying $x \leq y$.*
(a) If x is an integer, the inequality is equivalent to $x \leq \lfloor y \rfloor$.
(b) If y is an integer, the inequality is equivalent to $\lceil x \rceil \leq y$.

¶49. 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 28. 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 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.

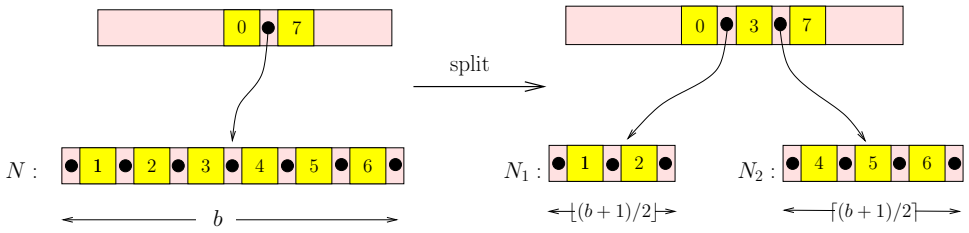


Figure 28: Splitting: N splits into N_1, N_2. Case $(a, b) = (3, 6)$ is illustrated.

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

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

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 29. Suppose N borrows a new from its sibling 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.

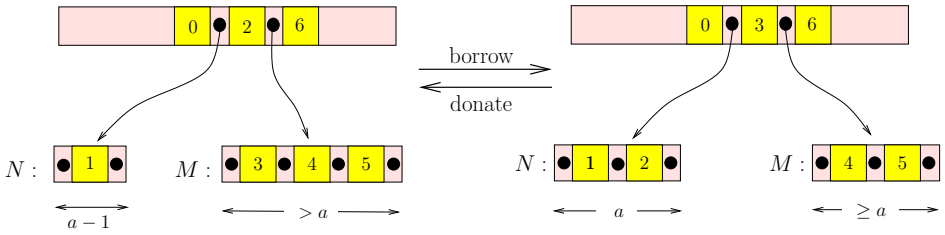


Figure 29: Borrowing: N borrows from M. Case of $(a, b) = (3, 6)$.

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 30.

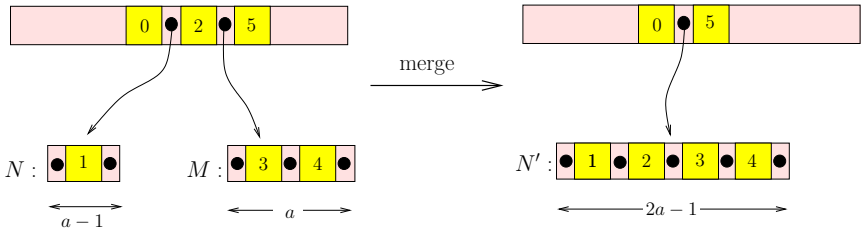


Figure 30: Merging: N and M merges into N'. Case of $(a, b) = (3, 6)$.

The careful reader will notice an asymmetry in the above three processes. We have the concept of borrowing, but it as much sense to talk about its inverse operation, **donation**. Indeed, if we simply reverse the direction of transformation in Figure 29, 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 not usually discussed in the literature. But we will see its benefits below.

¶50. 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.

(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.

Notice: cases (i) is the *only* means for increasing the height of the (a,b)-tree; likewise case (ii) is the only means for decreasing height.

¶51. Special Treatment of Leaves. Now consider leaves: in order for the splits and merges of leaves to proceed as above, we need the analogue of the split-merge inequality,

$$a' \leq \frac{b'+1}{2}. \tag{29}$$

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 above scenarios. In particular, when we split or merge internal nodes, the internal keys are just moved around.

However, a rather unique case arise when the leaf is also a root! We cannot treat it like an ordinary leaf having between a' to b' items. So let us introduce the parameters a'_0, b'_0 to control the minimum and maximum number of items in a root-leaf. Let us determine constraints on (a'_0, b'_0) relative to (a', b'). Initially, there may be no items in the root, so we must allow $a'_0 = 0$. Also, when the number of items exceed b'_0, we must split into two or more children with at least a' items. The standard literature allows the root to have 2 children and this requires $2a' \leq b'_0 + 1$ (like the standard split-merge inequality). Hence we require

$$b'_0 \geq 2a' - 1. \tag{30}$$

not $b'_0 \geq 2a'_0 - 1$

In practice, it seems better to allow the root to have a larger degree than a smaller degree. Thus, we might even want distinguish between leaves that are non-roots and the very special case of a root that is simultaneously a leaf. Such alternative designs are explored in Exercises.

¶52. 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.

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 less than $< a$ 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 (rename it as u) to prepare for 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).

¶53. 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 if it were possible to donate/borrow. We will now make the Standard Algorithms explicit.

(STANDARD INSERTION) For standard insertion, if node u is overfull, the standard algorithm immediately splits u into two nodes (and recurse). *That is all.* What are some possible enhancements? It seems worthwhile to try donation first. To donate, we must bring into main memory an immediate sibling. If the attempted donation fails, and we have another immediate sibling, it seems worthwhile to attempt to donation again. Of course, when both attempts fail, we do the usual split.

(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. If we succeed in borrowing from u', the algorithm terminates; otherwise, we can merge u with u' (and recurse). *That is all.* A possible enhancement in case attempt to borrow fails is to make a second attempt to borrow. Of course this is only possible if we have another immediate sibling u''.

Observe that standard INSERT/DELETE only need to hold in main memory at most three nodes at any moment: current node, its parent and one sibling. These enhancements seems to favor better space utilization and encourage earlier termination. In the worst case, the enhanced algorithm is slower than standard algorithm because failure to donate/borrow has an associated cost. This is quantified in our analysis next. It may be possible to justify the enhancements using amortized or probabilistic analysis.

¶54. I/O Analysis of Standard and Enhanced Algorithms. Let us analyze the Standard Insertion (no enhancement): 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.

Suppose we enhance the insertion algorithm: each failed donation costs one I/O (to read in a sibling), but a successful donation costs[9] us two I/O's (reading in the sibling and writing it out again). Of course, a successful donation happens only once, and costs one more I/O than the standard algorithm. So the total number of I/O's is $(2 + 3I) + F + S$ where F is the number of failures and $S = 1$ or 0 (depending on whether there is a successful donation). Since $S + F \leq 2I$, we can bound the number of I/O's by $2 + 5I$.

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.

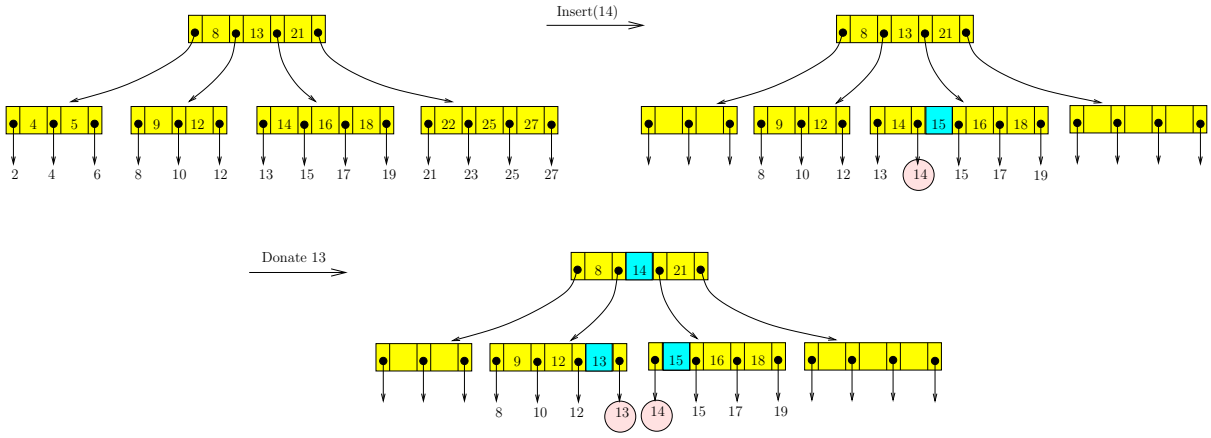


Figure 31: Inserting 14 into a $(3, 4)$-search tree.

Insertion Example: Consider inserting the item (represented by its key) 14 into the tree in Figure 25. This is illustrated by Figure 31. Note that $a' = b' = 1$. After inserting 14, we get an overfull node with 5 children. In standard Insertion, we would immediately split. But with the enhancement, we try to donates 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.)

[9] The cost of reading and writing u is separately accounted for, as part of the standard algorithm accounting.

¶55. 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 (24) on (a, b)-trees implies that the space utilization in such trees can never[10] 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 only slightly larger than $1 : 2$. We now 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

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

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

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 32.

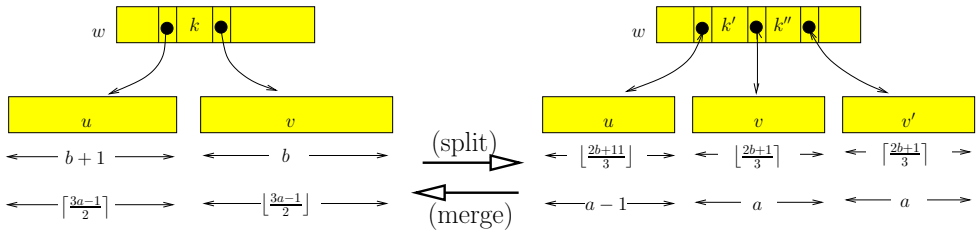


Figure 32: Generalized (2-to-3) split and (3-to-2) merge

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

$$a \leq \left\lfloor \frac{2b+1}{3} \right\rfloor. \tag{31}$$

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[11] 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 $\lfloor (3a-1)/2 \rfloor$

[10] 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.

[11] Normally, we expect v, v' to be immediate siblings of u (to the left and right of u). But if u is the eldest or youngest sibling, then we may have to look slightly farther for the second sibling.

and $\lceil (3a-1)/2 \rceil$ children, respectively. Assuming

$$a \geq 3, \tag{32}$$

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

$$\left\lceil \frac{3a-1}{2} \right\rceil \leq b \tag{33}$$

Because of integrality constraints, the floor and ceiling symbols could be removed in both (31) and (33). Thus both inequality are seen to be equivalent to

$$a \leq \frac{2b+1}{3}. \tag{34}$$

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.

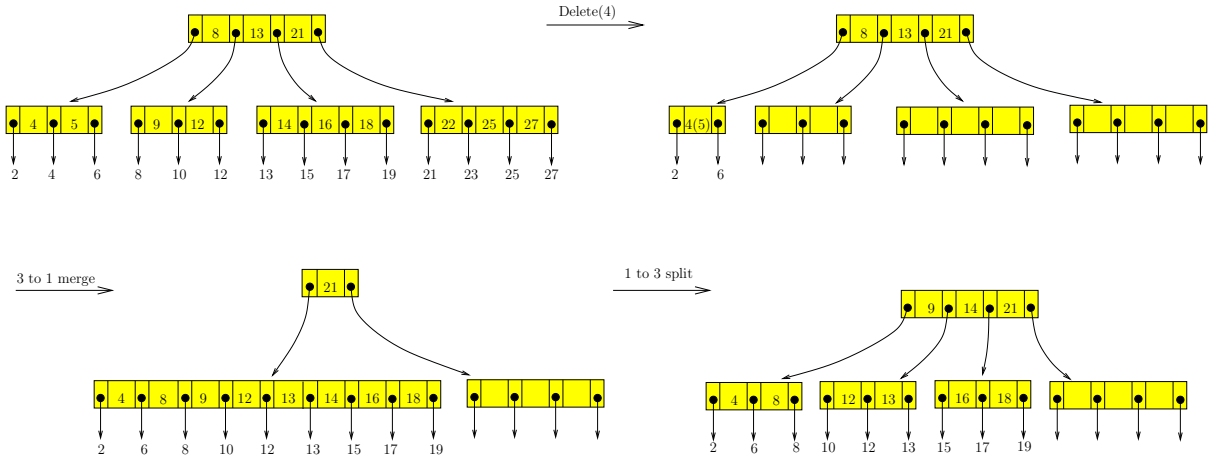


Figure 33: Deleting 4 from $(3, 4, 2)$-search tree.

¶56. Example of Generalized Merge. Consider deleting the item (represented by its key) 4 from the tree in Figure 25. The is illustrated in Figure 33. 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 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 should be able combine this merge-then-split steps into one 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 (24); 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 25. 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.

¶57. 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[12] do we know that these data structures are endogenous? We deduce it from the observation 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 before we can conclude that an item with key k is, or is not, stored in the search tree. In contrast, items in (a,b)-search trees are stored in the leaves only. Tarjan [11, p. 9] calls a search structure **endogenous** if items are directly stored in the nodes of the search structure; otherwise it is **exogenous**. Thus, standard BST are endogenous while (a,b)-search trees are exogenous. Recall that in §5 we also saw the exogenous form of BST, called an external BST.

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

There are two consequence 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*. In illustration, see Figure 27 where 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 items.

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

¶58. 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 (16) 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 most searches in such a data base is presumably based on these values. It seems to make less sense to build an index based on age or salary, although we could.

¶59. Disk I/O Considerations: How to choose 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

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

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 block size

If the number of items stored in the (a,b)-tree is too many to be stored in main memory, the same would be true of the internal nodes of the (a,b)-tree. Hence each of these internal nodes are also stored on disk, and they are read into main memory as needed. Thus `lookUp`, `insert`and `delete` are known as **secondary memory algorithms** because data movement between disk and main memory must be explicitly invoked. Typically, it amounts to bringing a specific disk block into memory, or writing such a block back to disk.

¶60. On (a,b,c)-trees: Generalized Split-Merge for (a,b)-trees. Thus insertion and deletion algorithms uses 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. The only global constraint on c is that it be a positive integer: $c \geq 1$. 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 we can donate a child to these siblings. If so, we are 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.

  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 possible 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. When $c=1$, we have the B-trees; when $c=2$, we achieve the 2/3-space utilization ratio above. 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**,

$$c+1 \leq a \leq \frac{cb+1}{c+1}. \tag{35}$$

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 (35) 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 (35) guarantees success.

We are interested in the maximum value of a in (35). Using the fact that a is integer, this amounts to

$$a = \left\lfloor \frac{cb+1}{c+1} \right\rfloor. \tag{36}$$

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!*

Example: What is the simplest generalized B-tree where $c=3$? Then $b > a \geq c+1 = 4$. So the smallest choices for these parameters are $(a,b,c) = (4,5,3)$.

¶61. 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 (35). We view (35) 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. So the more interesting case is the smallest possible value of c. To determine the smallest c, we can easily verify (Exercise) this fact:

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

For instance, we now see that $(4,5,c)$ has only one solution ($c=3$) because $(4,5,2)$ is invalid.

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 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.

¶62. 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 has 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.

¶63. 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. 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. It might well turn out that these extra I/O's are un

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.

¶64. 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 [8]). This was the beginning of a technique called "fringe analysis" which Yao [12] introduced in 1974. Nakamura and Mizoguchi [9] 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 (36), 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 (a,b)-search tree: (i) an array, (ii) a singly-linked list, (iii) a 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 treat this as 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. The lecture notes has some discussion of these issues.

(b) What is the worst case time to search for a key in an (a,b)-search tree with two million items? We need one number (not expression) for each of the four schemes as answer: the unit for each number is CPU cycles (or "CPUC").

MAKE THESE ASSUMPTIONS: Assume $(a',b') = (a,b)$, i.e., number of items in leaves is between a' and b'. 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 for 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 can use calculators, but I encourage you to make calculator-free simplified estimates whenever possible (e.g., log base 2 of one million is 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: 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" Modify our insert/delete algorithms so that we try to share with direct siblings or cousins before doing the generalized split/merge.

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 8.10: Suppose you have an (a, b, c)-tree of height h. What is the number of disk I/O's in the worst case when inserting an item? Assume the the root is always in main memory. Give an expression involving c and h. ♢

Exercise 8.11: 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 8.12: 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: What is the the best ratio achievable under (24)? Under (34)? ♢

Exercise 8.14: 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 (24). ♢

Exercise 8.15: Consider the tree shown in Figure 25. 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 8.16: 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 8.17: Is it possible to design (a, b, c) trees so that the root is not treated as an exception? ♢

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 reasonably, considering that the root should

probably be kept in memory all the time and so do not have to obey the b constraint. Here is the 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: We want to explore the weight balanced version of (a, b)-trees.
(a) Define such trees. Bound the heights of your weight-balanced $(a, b$-trees.
(b) Describe an insertion algorithm for your definition.
(c) Describe a deletion algorithm. ♢

Exercise 8.20: How can we choose the a parameter (see (36)) in generalized B-trees in a more relaxed manner so that the repeated splits/merges during insertion and deletions are minimized? ♢

End Exercises

§A. APPENDIX: Red-Black Trees

Red-black trees form a balanced family that, in some sense, is the most economical among height-balanced binary trees. Each node in such a tree is colored[13] red or black, where red nodes are considered deficient. Such trees have some nice properties (see Exercise on treaps below). The drawback is the relative complexity of their insertion and deletion algorithms. There is an alternative view of red-black Trees that connects then to $(2,4)$-trees. Since $(2,4)$-trees (as a special case of (a,b)-trees) have simpler algorithms, we can think of the red-black tree algorithms as the unraveled $(2,4)$-tree algorithms.

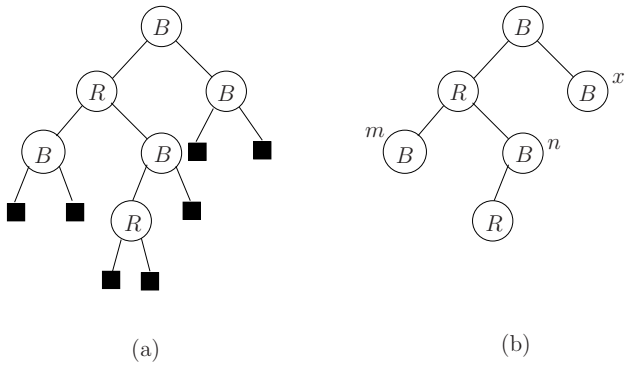
To make these algorithms easier to understand, it is convenient to introduce the "extended" version of red-black trees:

An **extended red-black tree** is an extended binary search tree in which each node is colored either red or black. The color scheme satisfies properties $B(x)$, $H(x)$ and $P(x)$ at each node x:

Basis property $B(x)$:	If x is a nil node, then it is black.
Height property $H(x)$:	The number of black nodes in a path from node x to any nil node is invariant. This invariant number, minus one, is called the **black height** of x. So a nil node has black height of 0.
Parent property $P(x)$:	If x is red then its *p*arent (if any) is black.

A **(standard) red-black tree tree** is a standard binary search tree T whose nodes are colored either red or black such that its corresponding extended version T' (with nil nodes colored black) is an extended red-black tree. The two versions of red-black tree are interchangeable. Sometimes it is easier to work with the extended version but most of the time we assume the standard version. Figure 34 illustrates a red-black tree in its two forms.

The theoretical advantage of red-black trees over AVL trees is that we use only 1 bit of balance information per node, as opposed to $\lg 3$ bits in AVL trees.



(a) (b)

Figure 34: Two equivalent red-black trees: (a) extended version, (b) standard version.

There are some easy consequences of the basis, height and parent properties:
(i) Each node in a red-black tree of height two or more must have two children. (Recall that a

[13] This color terminology is borrowed from accounting practice, as in balance sheets being in the red or in the black.

node has height 0 iff it is a leaf.)
(ii) A node with exactly one child must be black, and its unique child must be a leaf with color red.

With these observations, we can now give a direct definition of red-black trees (without going through the extended binary search trees). At each node x, the following three properties hold:

Basis property $B'(x)$:	If x has only one child, then that child is a leaf with color red.
Height property $H'(x)$:	The number of black nodes in a path from node x to any leaf is invariant. This invariant number is the **black height** of x.
Parent property $P'(x)$:	If x is red then its *p*arent (if any) is black.

Note that the new parent property $P'(x)$ is identical to $P(x)$. In the following descriptions, we shall continue to use $B(x)$ and $H(x)$ instead of $B'(x)$ and $H'(x)$.

Let T be an ordinary binary tree whose nodes are colored red or black and x is a node of T. The meaning should be clear when we say that T **violates the parent property** $P(x)$, or equivalently, there is a P**-violation at** x. Similarly for the height property, we speak of violating $H(x)$ or a H-violation at x. It may be helpful[14] for the reader to think of the red nodes as deficient in a certain sense. The parent property ensures that there are no two consecutive deficient nodes in a path.

Note that if the root of a red-black tree is red, we can just color it black, and the result is still a red-black tree. For this reason, some literature assumes that the root of a red-black tree is always black.

If x is a node in a red-black tree T, let $\texttt{bht}[x] = \texttt{bht}_T[x]$ denote the black height of x in T and $\texttt{bht}[T]$ denote the black height of the root of T. For instance, if x is a black leaf, $\texttt{bht}[x] = 1$. For any path p, we also let $\texttt{bht}[p]$ denote the number of black nodes in p. Red-black trees are automatically balanced:

Lemma 7. *Suppose a red-black tree T has n nodes and black height h. Then $n \geq 2^h - 1$ and hence $h \leq \lg(n+1)$. Hence T has height at most $2\lfloor \lg(n+1) \rfloor$.*

*Proof.*We prove that $n \geq 2^h - 1$ by induction on $h \geq 0$. This is true when $h = 0$, for T has 0 or 1 node. If $h \geq 1$, then the left and right subtree at T has black height at least $h - 1$ (they could have height h). By induction, they each have at least $2^{h-1} - 1$ nodes. Thus T has at least $2(2^{h-1} - 1) + 1 = 2^h - 1$ nodes, as claimed. The rest of the lemma is immediate: from $n \geq 2^h - 1$ we get $h \leq \lfloor \lg(n+1) \rfloor$. By the parent property, the height of the tree is at most $2h$. **Q.E.D.**

¶65. Operations on red-black trees. We next consider the basic operations of `lookUp`(*Key*), `insert`(*Item*) and `delete`(*Node*) on red-black trees. Since red-black trees are binary search trees, the lookup operation can be done as for binary search trees. The following terminology will be useful in our descriptions. The usual terminology for binary trees views them as a kind of family tree where each node has at most 2 children and reproduction is

[14] The colors in our trees comes from "red ink" and "black ink". That is, they are accounting analogies, with no racial connotations!

asexual. Extending this analogy, the children of a sibling are called **nephews**. But we may distinguish one of these nephews as a **near-nephew** and the other as a **far-nephew**. For instance, in Figure 34(b), the nodes n, m are respectively the near-nephew and far-nephews of x. Of course, the reciprocal relation is an **uncle**: x is the unique uncle of m and of n. Note that a node is either a near-nephew or a far-nephew of its uncle, but not both.

Exercises

Exercise A.1: For each of the following binary trees, say whether it could be the shape of a red-black tree or not. If yes, show a coloring scheme; if not, argue why not. i ♢

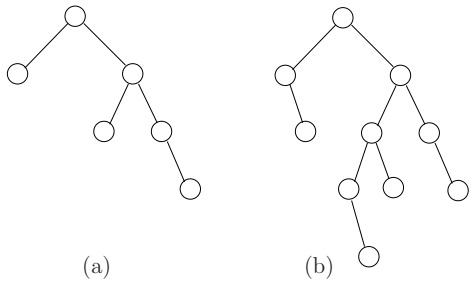


Figure 35: Some binary trees.

Exercise A.2:
(a) What is the maximum possible height (not black height) of a red-black tree with 10 keys? Argue rigorously why your bound is correct.
(b) Show a lower bound (draw a read-black tree with this bound). ♢

Exercise A.3: Give a direct definition of red-black trees for ordinary binary search trees. (In our text, we defined it indirectly, via extended binary search trees.) ♢

Exercise A.4: A "2-4 tree" is a search tree in which each interior node has degree 2, 3 or 4 children, and every path from the root to a leaf has the same length. It is a search tree in the sense that if a node has d ($d = 2, 3, 4$) children, then the node stores $d - 1$ items. The keys of these items $K_1 < K_2 < \cdots < K_{d-1}$ are ordered and we have a generalization of the binary search tree property. This facilitates searching for any key.
(a) Show that every red-black tree can be interpreted as a 2-4 tree. HINT: assume that the root is black.
(b) Do a detail comparison of the insertion algorithms in $(2, 4)$-trees and red-black trees.
(c) Repeat part (b) for deletion. ♢

Exercise A.5: (Cormen-Leiserson-Rivest) Alternative definition of black height: define the **black height** of node u to be the number of black nodes along any path from u to a nil node, but *not* counting u. That is, if u is red, then our black height is 1 less than the alternative definition. In particular, if a red-black tree has a red node, we can change it to black and its height is not changed according to this definition. One disadvantage of this definition is the need to refer to nil nodes in its definition. But does this alternative definition make any difference to our algorithms? ♢

Exercise A.6: An alternative definition of an extended red-black trees is in terms of a rank function $r(x)$ for each node x with these properties: (i) If the parent $p(x)$ of x exists, then $r(x) \leq r(p(x)) \leq r(x) + 1$. (ii) If the grandparent $p(p(x))$ exists, then $r(x) < r(p(p(x)))$. (iii) If x is a leaf, $r(x) = 1$ and if $x = \texttt{nil}$ then $r(x) = 0$. Show the "equivalence" of such trees and red-black trees; part of the problem is to define "equivalence" precisely. HINT: use the definition of black height in the previous exercise. ♢

Exercise A.7: (Olivié) A binary tree is **half-balanced** if for every node x, the length of the longest path from x to a leaf is at most twice as long as the length of the shortest path from x to a leaf. As in the previous exercise, show the "equivalence" of such trees and red-black trees. ♢

Exercise A.8: Let $M(h)$ denote the maximum number of key-bearing nodes in a red-black tree with black height of h. Write the recurrence equation for $M(h)$. Solve the recurrence. NOTE: what if we do the same for $m(h)$, denoting the minimum number of key-bearing nodes in a red-black tree with black height h? ♢

End Exercises

§A.1. Red-Black Insertion

The following description is designed so that the student can easily commit to memory the insertion algorithm. Consequently, the student should be able to perform hand-simulation of the algorithm on actual trees. For insertion, we prefer to treat the tree as a standard binary tree.

Suppose we want to insert a node x into a tree T. There are three steps in insertion:

1. Insert x into T as in a binary search tree.
2. Make x a red node.
3. Rebalance at x.

We must explain the third step of rebalancing. After steps 1 and 2, x is a red leaf. We verify that the tree automatically satisfies the height property. Next, property $P(u)$ holds at each node u except possibly when $u = x$. In fact, the only possible violation of red-black tree properties is when the parent of x is red. We are done if the parent of x is black. Hence assume otherwise.

This single violation is key to understanding the insertion operation, and deserves a definition: let x be any node of binary search tree T whose nodes are colored red or black. We call T an **almost red-black tree at node** x if T satisfies the basis and height properties. Moreover, it satisfies property $P(u)$ for all nodes $u \neq x$ but does not satisfy $P(x)$. In this case, we also say T has a P**-violation** at x or an **insertion violation** at x.

¶66. Rebalancing an almost red-black tree at x. Rebalancing an almost red-black tree means to convert the tree into a red-black tree. This is reduced to a sequence of (repeated) "rebalancing steps". Each rebalancing step either (I) transfers the violation at x to the grandparent of x, or (II) remove the violation completely (and we terminate). The color of the **uncle** of x decides which of these two cases applies. We consider the following scenario (see Figure 36):

Insertion scenario.
We are trying to rebalance an almost red-black tree T at x. Both x and its parent y are red. Let z be the parent of y and u the sibling of y (so u is the uncle of x). Then z must be black but the color of u is unknown and is the critical determinant of the action we take.

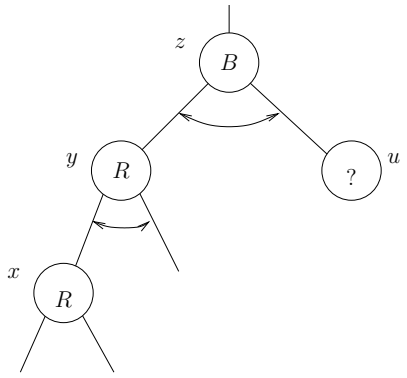
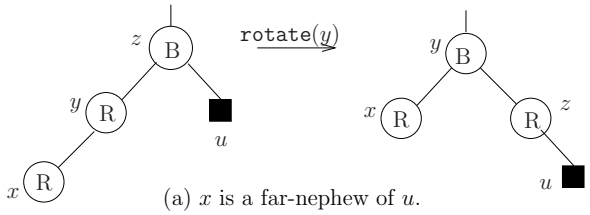


Figure 36: Insertion scenario: violation at x.

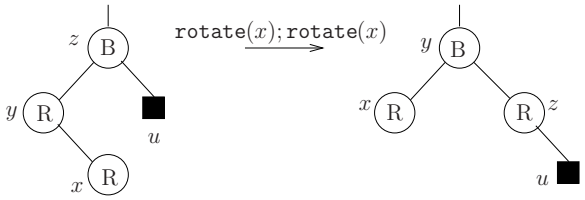
¶67. Justification for Insertion Scenario. Why does this scenario hold? Note that if y did not exist, we would not have a violation at x. If the grandparent z of x did not exist, then we can simply color y black and the result would be a red-black tree. *This recoloring is the only operation in the insertion algorithm that increases the black height of a tree.* What about u? In general, its existence is guaranteed by the basis property $B(z)$. However, if x has *just* been inserted, u may not exist (that is, u may be a nil node). In this case, it is easy to see that x has no sibling or children. This situation can easily be fixed as illustrated in Figure 37:

FixSuperNode(x):
(a) If x is a far-nephew of u, we rotate at y,
blacken y and redden its children x, z.
(b) If x is a near-nephew of u, we rotate twice at x.
We blacken x and redden its children (y and z).

In either case, the reader verifies that the result is a red-black tree.



(a) x is a far-nephew of u.



(b) x is a near-nephew of u.

Figure 37: When u does not exist: two cases.

This transformation of the triple x, y, z will be encountered again. It is helpful to think of x, y, z as a *supernode* (hence the name of this little routine). Hence (a) and (b) are just two cases in the rebalancing of a supernode. The operation in (b) is quite common and is called a **double rotation**. In general, a double rotation at a node x is defined if x is the near-nephew of its uncle and the operation amounts to two consecutive rotations of x.

To conclude, this "justification" of the insertion scenario amounts to a procedure (let us call it Convert) to transform an insertion violation into the insertion scenario, or else to remove the insertion violation.

¶68. Rebalance Step. We assume the insertion scenario and consider two cases.

¶69. CASE I: The uncle u is red. This case is easy: we simply redden z and blacken both y and u (see Figure 38).

It is not hard to check that the result is either a red-black tree or an almost red-black tree at z. In the latter case, we recursively do a rebalance at z. In short, the rebalance step has either removed the insertion violation or moved it closer to the root. Eventually the recursive process must stop.

¶70. CASE II: The uncle u is black. In this case, we transform the tree as indicated in Figure 39: This amounts to rebalancing the supernode x, y, z. That is, we just call the routine FixSuperNode(x). Since the result is a red-black tree, the black uncle case is a terminal one.

We expand on Figure 39 a little bit. The left-hand side is a combination of 4 cases, depending on whether x is a left or right child of y and whether y is a left or right child of z. Note that the roots of the four subtrees T_1, T_2, T_3 and T_4 are all black (u, the root of T_4, is

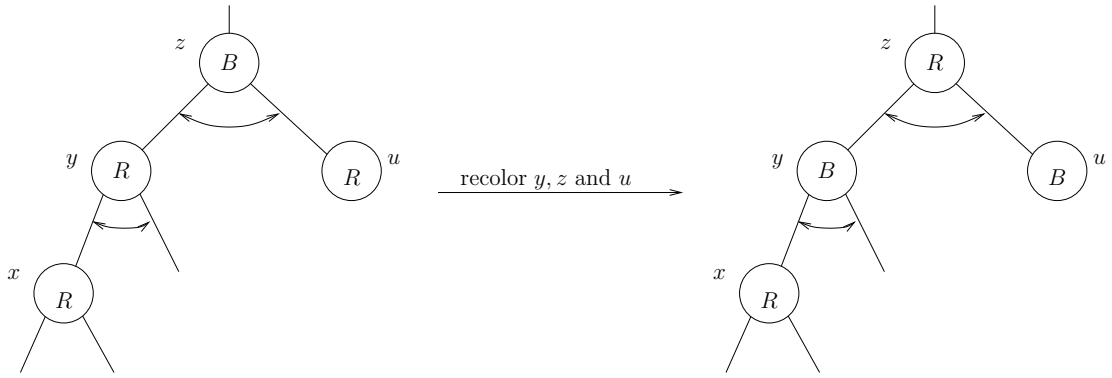


Figure 38: Red uncle: possible new violation at z.

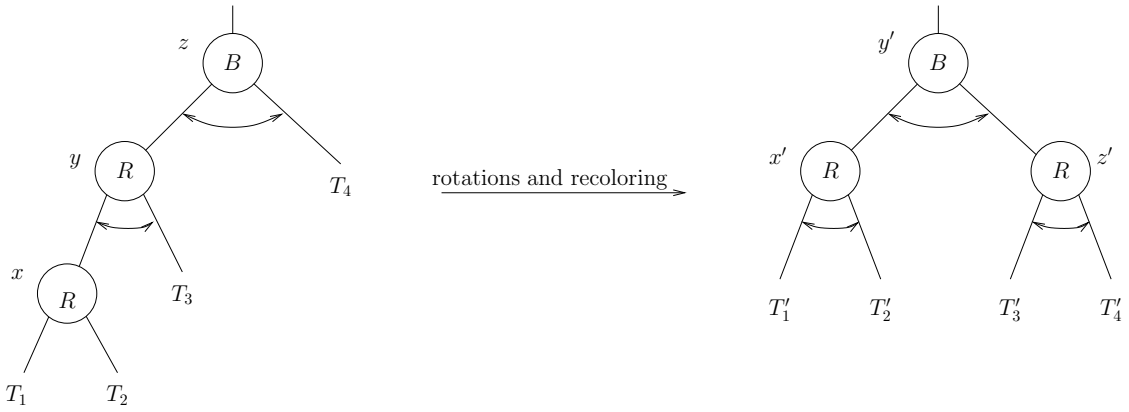


Figure 39: Black uncle: root of T_4, the uncle of x, is black.

black by assumption and the others are black by the parent property). This transformation is completely specified by the following requirements:

- x', y', z' is simply x, y, z in non-decreasing order of their keys.
- T'_1, T'_2, T'_3 and T'_4 are the the subtrees T_1, T_2, T_3 and T_4 in non-decreasing order.

The reason this is enough to determine the tree on the right-hand side of Figure 39 is because the result must be a binary search tree. Take the possibility where x is the right child of y, and y the left child of z (see Figure 40). Here, if we perform a double rotations at x, redden z and blacken x, we obtain the desired red-black tree. We can similarly work out the other three possibilities.

This completes our discussion of the black uncle case and hence of the rebalance step. To summarize the insertion algorithm:

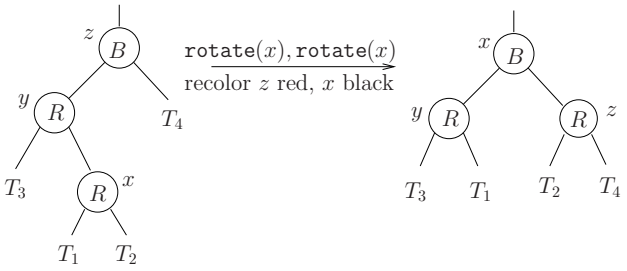


Figure 40: Black Uncle: x is right child of y, y is left child of z.

Insert(**1** I):
1. Insert I as in a binary search tree; color the inserted node red.
2. while there is a violation, do
2.1 Convert to the insertion scenario.
2.2 Perform a rebalance step.

Exercises

Exercise A.9: Insert the following sequence of keys (in succession) into an initially empty tree: $10, 8, 3, 1, 4, 2, 5, 9, 6, 7$. ♢

Exercise A.10: Verify carefully that after the rotations and re-colorings in the rebalance steps always result in a red-black tree or an almost red-black tree. ♢

Exercise A.11: Draw a red-black tree T_1 with black height 2 and specify a key k such that inserting k into T_1 will increase the black height of T_1. Draw the red-black tree after inserting k. **HINT:** when does the black height increase in the insertion procedure? ♢

Exercise A.12: Is the recoloring in our rebalancing steps ad hoc? Try to give a more systematic account of how to assign colors. (This is certainly another useful aid to memory.) ♢

Exercise A.13: (One pass version) Our insertion algorithm requires two passes, one pass down and the other pass up the tree. Design an alternative insertion algorithm which has only one pass. HINT: the idea is "preemptive rebalance". Try to make sure that a node is black before you visit it. ♢

End Exercises

§A.2. Red-Black Deletion.

Deletion is slightly more involved than insertion. Again, we design the algorithm to be remembered, so that the reader to perform hand-simulation of the algorithm. For this description, we prefer to use the terminology of extended binary trees.

Suppose we want to delete the item in a node u in a red-black tree T. We delete item in u as we would in an ordinary binary tree, and then we rebalance to ensure that the result is still a red-black tree. It is important that we use the "standard deletion algorithm" (see §3) which does not use rotations. Note that the standard deletion algorithm may not actually remove the node u. Instead, some node u' with only one child will be removed.

¶71. Deletion violation. Again, the deletion of node u' causes a "violation", which we now explain. Imagine the nodes of our search tree to carry "tokens" – a red node has no token and a black node has one token. The nil nodes (recall we view T as an extended binary tree now) are automatically black and so carries a token. Thus, the black height of a node u is one less than the number of tokens along any path from u to a nil node. The height property simply says that the number of black tokens along a path from any node to a nil node is invariant. Thus if the deleted node u' is red, we do not change the number of tokens along any path, so the height property is preserved. So assume that the deleted node u' is black. Recall that u' has at most one child. If u' has no parent, then the result is again a red-black tree. Hence assume that u' has a parent y. After deleting u', one of the children of y becomes a node x in place of u'. Note that x is a nil node if u' was a leaf, but otherwise x was the only child of u'. *Let us give the token of u' to x after deletion.* Now the height property is restored in the sense that every path to a nil node still contains the same number of tokens!

What can still go wrong? Well, if x already has a token, then giving x another token means that x now has two tokens. We say a node is **doubly-black** if it has 2 tokens. First observe that if u' was not a leaf, then we know that x must be a red leaf. This means that x is black, not doubly-black. But if u' is a leaf, then x is a nil node and hence it is now doubly-black. In figures, we use R, B, D to denote red, black and doubly-black nodes, respectively.

An extended binary search tree T in which an extended node x is colored doubly-black (D), with the remaining nodes are colored black (B) or red (R), is said to be an **almost red-black tree at** x if T satisfies the basis and parent properties, and also the modified height property, interpreted in terms of counting tokens as above. We also say[15] that T has a **deletion violation at** x.

We summarize the deletion algorithm:

1. Use the standard deletion algorithm to delete (the item in) u.
   Let u' be the node that is physically removed in this deletion.
2. If u' is red or has no parent, we terminate with a red-black tree.
   {`Henceforth assume` u' `is black and has parent` y`.`}
3. Let x be the extended node that is the child of y in place of u'. If x is non-nil, we again terminate.
4. Otherwise, we doubly blacken the nil node x.
5. Call the rebalancing procedure at node x.

[15] We used the same terminology "almost red-black tree" when the tree has an insertion violation; this ambiguity is not a problem because we normally are in either insertion or deletion mode, but not both simultaneously. The context will make the type of implied violation clear.

In the remainder of this section, we describe the rebalancing procedure for an almost red-black tree at a node x. The rebalancing procedure is recursive and consists of repeated application of a "rebalancing step". Each step either removes the violation at x, or move it to some node nearer the root. Each rebalancing step assumes the following basic scenario (see Figure 41):

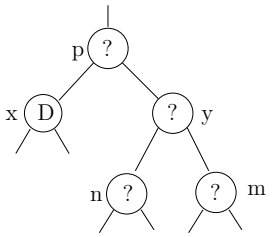


Figure 41: Deletion scenario: violation at x.

Deletion scenario.
We have an almost red-black tree which is doubly-black at x. The node x may be nil. The parent and sibling of x is p and y, respectively. The children of y are n and m, which are respectively the near-nephew and far-nephew of x.

¶72. Conversion to the Deletion Scenario. The illustration in Figure 41 is completely general, up to a mirror symmetry (that is, x may be the right child of p, etc). How can we "justify" this scenario? Suppose we have a deletion violation at x. If p did not exist, we can simply color x black and the result is a red-black tree. If p exists then the height property implies the existence of y. Now the black-height of y is equal to the black height of x, which is at least 2. Hence y is not nil. Note that if x is non-nil, then the two children m, n of y must also be non-nil. This justification amounts to a tiny procedure Convert for bringing an deletion violation into the deletion scenario (or, failing that, to terminate in a red-black tree).

¶73. Rebalancing Step. We now describe the rebalancing step under the hypothesis of the deletion scenario. There are 3 cases to consider, depending on the colors of y, m, n. The simplest is the following.

¶74. I. All-black case. The sibling y and the two nephews m, n are all black (see Figure 42). Then by coloring y red and by giving a black token to the parent p, we get either a red-black tree (if p was originally red) or an almost red-black tree at p (if p was originally black). Thus the deletion violation is either removed or moved closer to the root.

¶75. II. Red-nephew case. Suppose some nephew of x is red. So y is black. There are two possibilities:
(a) The far-nephew m is red: See Figure 43. We can rotate at y, give the color of p to y, and recolor m, p and x to be black. The reader can verify that the result is a red-black tree. So this is a terminal case.

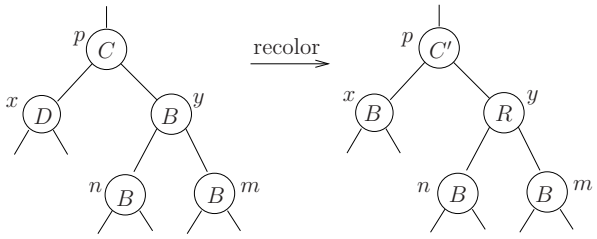


Figure 42: x is doubly black, with black sibling and nephews.

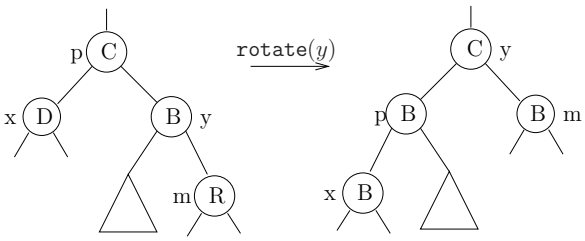


Figure 43: Far-nephew m is red.

(b) The near-nephew n is red: See Figure 44. We may further assume that the far-nephew m is black. By rotating at n, blackening n and reddening y, we have reduced this to case (a) where the far-nephew is red. (But note that case (a) will immediately cause a rotation at n, so in effect, we have a double rotation at n and this case may be regarded as terminal also.)

HINT: We can combine both cases and view this as the rebalancing of a supernode (cf. the FixSuperNode routine above). More precisely, in case (a), we rebalance the supernode m, y, p. Then y is given the old color of p, and m, p are blackened. Case (b) is similar: we rebalance the supernode m, y, p. (That is, the role of m taken over by n.) Then y is given the old color of p; and n, p are blackened. In both cases, we make x black and terminate.

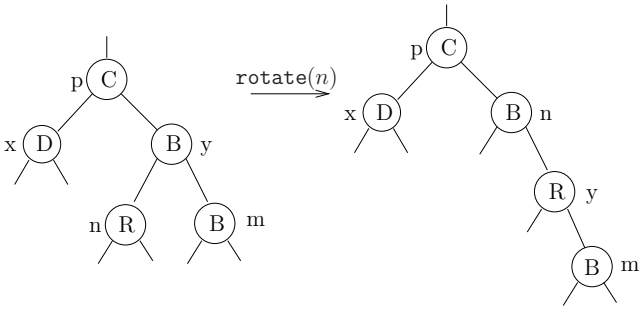


Figure 44: Near-nephew n is red.

¶76. III. Red sibling case. The sibling y of x is red (Figure 45). So the common parent p of x and y is black. In this case, we rotate at y, redden p and blacken y. The result is still an almost red-black tree at x, except that the sibling of x is black. This means we have reduced our situation to cases I or II.

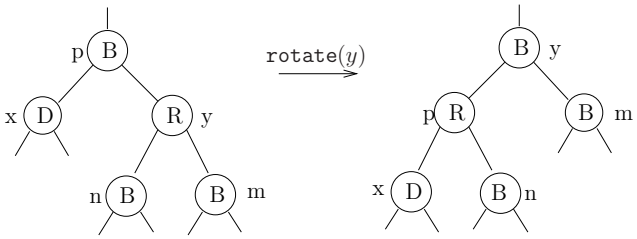


Figure 45: Red sibling case.

But there is a subtle point here – the depth of x is increased. To be sure that we are not in a non-terminating loop, we must analyze deeper: since case II is terminal, we only have to worry about a reduction to case I (all-black case) which may or may not terminate. But if we were reduced to the all-black case, it is easy to check that we would terminate after the necessary recoloring. Remark: this check is just for our analysis – the algorithm need not do anything special.

HINT: we can view case III as rebalancing the supernode m, y, p, somewhat like the red far-nephew case.

The above analysis shows that the only recursive case is the all-black case. This recursion can repeat at most $2 \lg n$ times before we reach the root. But, regardless of how rebalancing steps are performed, only a constant number of rotations are performed.

Exercises

Exercise A.14:
(a) Execute the following the red-black tree insert and delete operations (the meaning is self-explanatory): $Ins(5, 7, 3, 9, 10, 8), Del(10)$.
(b) Instead of $Del(10)$, do $Del(3)$. ♢

Exercise A.15: Verify that the deletion operation makes only a constant number of rotations, not $\Theta(\lg n)$ rotations. (You should flesh out some of the claims in the text about termination.) ♢

Exercise A.16: Write the deletion algorithm in a reasonable pseudo-code, making explicit any assumptions about the data structure and basic operations. Notes: there should be a procedure called Convert. Presumably, the argument to Convert is a node x where we have a deletion violation. The technical problem is that x may be nil. So a solution is that we call convert with the pair of nodes, Convert(x, y) where y is the parent of x. Here, either x or y may be `nil`. ♢

Exercise A.17: (a) When does the black height of a tree decrease in a deletion?
(b) Is it possible to have a red-black tree so that its black height increases when you insert a certain key, and its black height decreases when you delete a certain node? ♢

Exercise A.18: (One pass version) Our deletion algorithm requires two passes, one pass down and the other pass up the tree. Design an alternative deletion algorithm which has only

one pass. HINT: do "preemptive rebalance" by making sure that a node is red before you visit it. ♢

Exercise A.19: Give an alternative description (not code) for the deletion algorithm without reference to nil nodes. That is, view them as standard binary search trees. ♢

Exercise A.20:
(a) Show that we can modify the colors of a red-black tree so that each node of height 1 is black. Note that a leaf has height 0.
(b) Modify the insertion and deletion algorithms for such red-black trees. ♢

Exercise A.21: Let S be a set of n points in the plane. The **treap** data structure of E. McCreight stores a set S of points using their x-coordinate as key. I also stores at each node u the largest y-coordinate among all the points in the subtree at u. The underlying data structure is a binary search tree.
(a) Assume that binary search tree is a red-black tree. Show how to insert and delete points from treaps.
(b) Analyze the complexity of the algorithms in (a).
(c) Can you achieve the same complexity if you use AVL trees?
(d) Can you achieve the same complexity if you use 2-3 trees? ♢

Exercise A.22: (open-ended) Suppose we have *tricolored binary search trees* in which each node is colored red, black or doubly-black, satisfying some suitable modified Basis, Height and Parent properties. Work out the insert and delete algorithms for such search trees. Discuss advantages or disadvantages of these trees. ♢

End Exercises

§A.3. Merge and Split.

Suppose we want to implement the additional operations of merging and splitting, *i.e.*, the operations of a fully mergeable dictionary (§2). We shall write

$$T_1 < T_2$$

to indicate that all the keys in T_1 are less than the keys in T_2.

¶77. Merge. Consider how to merge T_1 and T_2 under the assumption $T_1 < T_2$. First we delete the maximum item u in T_1, and still call the resulting tree T_1. This takes $O(\log n)$ time. So we have to solve the related problem of merging the following three trees,

$$T_1 < u < T_2.$$

If $\texttt{bht}[T_1] = \texttt{bht}[T_2]$, merging is trivial: we just make u the root with T_1, T_2 as the left and right subtrees. So now assume $\texttt{bht}[T_1] > \texttt{bht}[T_2]$ (the other case is similarly treated).

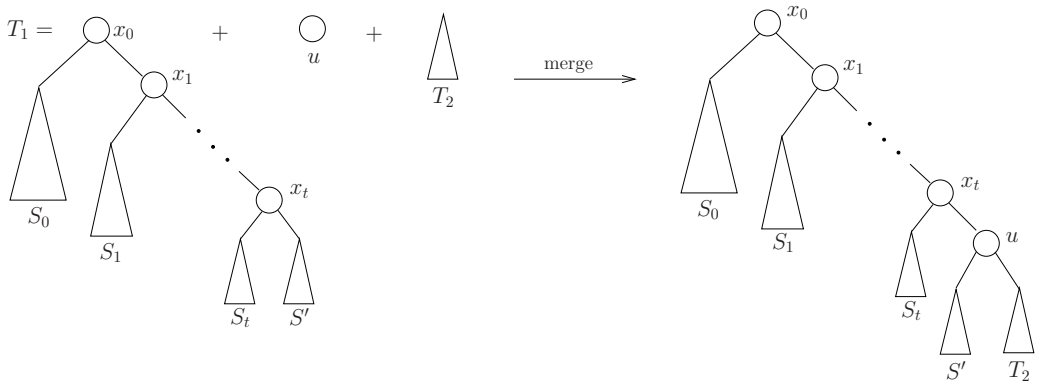


Figure 46: Decomposition of T_1 and merging of T_1, u, T_2.

Let us now walk down the right subpath $(x_0, x_1, \ldots)$ of T_1, terminating at a node x_t whose right subtree $S'_t = S'$ has the same black height as T_2. In general, the left and right subtrees of x_i are denoted S_i and S'_i, respectively (see Figure 46). Note that we may assume that S'_t has a black root (by choosing x_t appropriately). We install u as the right child of x_t, make S' and T_2 the left and right subtrees of u. We also color u red. The result is an almost red-black tree with possibly an insertion violation at u (if x_t is red). As in the insertion algorithm, we can now perform the rebalancing algorithm to convert an almost red-black tree into a red-black tree. The time to carry out the rebalancing is

$$O(1 + \texttt{bht}[T_1] - \texttt{bht}[T_2]) \tag{37}$$

which is $O(\log n)$. This concludes our description of merging.

¶78. Splitting. Next consider the problem of splitting a red-black tree T_1 at a key k. This is slightly more complicated, and will use the merging algorithm just described as a subroutine.

We 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 it is equal to the predecessor or successor of k in the tree. See Figure 47 for a particular case where $t = 5$.

Again, let S_i, S'_i denote the left and right subtrees of x_i. Let us form two collections L and R of RBT's: for each i, if the key in x_i is greater than k, we put x_i (viewed as a RBT with one key) and S'_i into R. Otherwise, we put x_i and S'_i into L. This takes care of every key in T_1, with the possible exception of a subtree of x_t: if x_t is put into R, then we put S_t into L. Otherwise, x_t is put into L and we put S'_t into R. In Figure 47, 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.

Clearly, our task is completed if we now combine the trees in L into a new T_1, and similarly combine the trees in R into a tree which is returned as the value of this procedure call.

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 in fact, 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:

$$L_1 > y_1 > L_2 > y_2 > \cdots > L_\ell > y_\ell > L_{\ell+1} \tag{38}$$

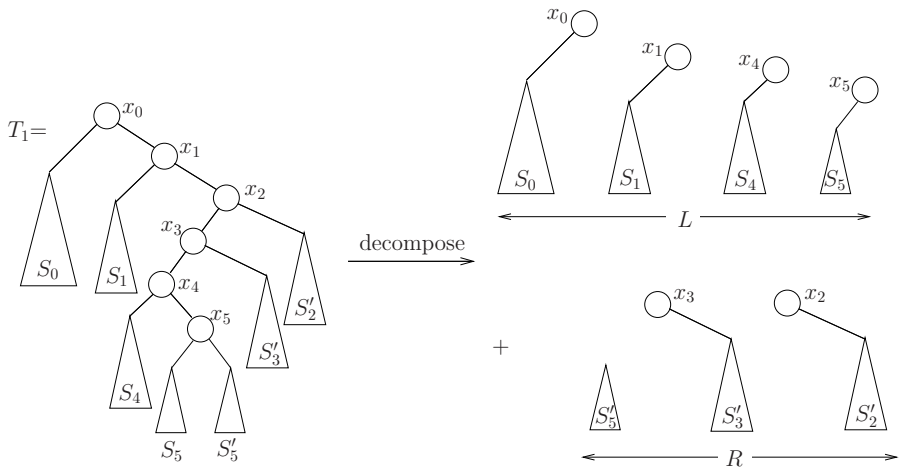


Figure 47: Split: decomposition of T_1.

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

$$\texttt{bht}[L_1] \geq \texttt{bht}[L_2] \geq \cdots \geq \texttt{bht}[L_{\ell+1}] \geq 0. \tag{39}$$

Note that $L_{\ell+1}$ could be empty. For instance, the set L in Figure 47 is relabeled as in Figure 48 with L_5 as an empty tree. The basic idea is to merge the trees from right to left. More precisely:

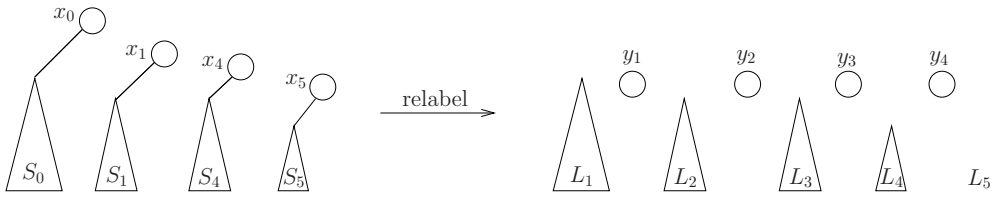


Figure 48: Relabelling the trees in the set L.

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 merge

$$L_i, y_i, L'_i \tag{40}$$

to form L'_{i-1}. Otherwise, L'_0 is the final result and we stop. This completes the merge algorithm.

The main result is the following:

LEMMA 8. *The time to merge all the trees in L is $O(\texttt{bht}[L_1] + \ell) = O(\log n)$.*

Verifying this requires some careful analysis but the idea is as follows. The inductive merge step (40) takes time

$$O(1 + |\texttt{bht}[L_i] - \texttt{bht}[L'_i]|). \tag{41}$$

Now, if we could assume that $\texttt{bht}[L'_i] \leq 1 + \texttt{bht}[L_{i+1}]$ then we could replace (41) by

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

and the overall cost (after telescoping) would be $O(\texttt{bht}[L_1] + \ell)$,

¶79. An $O(\log n)$ bound for splitting. We refer to the collection L (see equations (38) and (39)). Let us call L_i **red** or **black** according as its root is red or black. Notice that if L_i is red, then $\texttt{bht}[L_{i+1}] > \texttt{bht}[L_i]$. We claim: for $i = 2, \ldots, \ell$,

$$\texttt{bht}[L_{i-1}] = \texttt{bht}[L_i] \quad \Longrightarrow \quad \texttt{bht}[L_i] > \texttt{bht}[L_{i+1}].$$

For, if $\texttt{bht}[L_{i-1}] = \texttt{bht}[L_i]$ then the root of L_i is a near nephew of the root L_{i-1} and the parent $p = x_i$ of L_i is red. Similarly, if $\texttt{bht}[L_i] = \texttt{bht}[L_{i+1}]$, we conclude that the parent $q = x_{i+1}$ of L_{i+1} is red. But p is the parent of q, by the height property. This is a contradiction because now the parent property is violated, proving our claim.

Suppose inductively, for some $h = 1, \ldots, \ell$, we have already merged

$$L_{h+1}, y_{h+1}, L_{h+2}, \ldots, y_\ell, L_{\ell+1}$$

into a RBT denoted L'_h. In case

$$\texttt{bht}[L'_h] > \texttt{bht}[L_h]$$

we will say that an **inversion** has occurred at L'_h.

LEMMA 9 (Inversion Lemma). *In case of an inversion at L'_h, the following holds.*

(i) $\texttt{bht}[L'_h] = 1 + \texttt{bht}[L_h]$.

(ii) *Either L'_h is black or* $\texttt{bht}[L_{h-1}] > \texttt{bht}[L_h]$.

*Proof.*To see this lemma, consider the previous step which combined $L_{h+1}, y_{h+1}, L'_{h+1}$ into L'_h. We use three easy remarks. First, it is clear from the merge algorithm that

$$\texttt{bht}[L'_h] \leq 1 + \max\{\texttt{bht}[L_{h+1}], \texttt{bht}[L'_{h+1}]\}. \tag{42}$$

Second, if equality is achieved in (42) then L'_h is black (using a basic property of our rebalancing procedure for removing insertion violations). Third, if $\texttt{bht}[L'_{h+1}] > \texttt{bht}[L_{h+1}]$ and L'_{h+1} is black, then (42) is a strict inequality. (Similarly if $\texttt{bht}[L'_{h+1}] < \texttt{bht}[L_{h+1}]$ and L_{h+1} is black then we also have a strict inequality.)

There are two cases.
CASE I: there is no inversion at L'_{h+1}, *i.e.*, $\texttt{bht}[L_{h+1}] \geq \texttt{bht}[L'_{h+1}]$. Then clearly property (i) holds and L'_h is black (thus satisfying (ii)).
CASE II: there is an inversion at L'_{h+1}. We assume that this inversion satisfies the hypothesis in our lemma. So, either (a) L'_{h+1} is black or (b) $\texttt{bht}[L_h] > \texttt{bht}[L_{h+1}]$. In subcase (a), in order to have an inversion at L'_h, our first remark implies that $\texttt{bht}[L_h] = \texttt{bht}[L_{h+1}]$. But this means $\texttt{bht}[L_{h-1}] > \texttt{bht}[L_h]$, satisfying property (ii). To see property (i), note that remark 3 implies $\texttt{bht}[L'_h] \leq \texttt{bht}[L'_{h+1}]$ and hence $\texttt{bht}[L'_h] = \texttt{bht}[L'_{h+1}]$. By induction, property (i) says that $\texttt{bht}[L'_{h+1}] = 1 + \texttt{bht}[L_{h+1}] = 1 + \texttt{bht}[L_h]$.

In subcase (b), property (i) follows because

$$\begin{aligned}
\texttt{bht}[L'_h] &\leq 1 + \texttt{bht}[L'_{h+1}] \\
&\leq 2 + \texttt{bht}[L_{h+1}] \quad \text{(by induction)} \\
&\leq 1 + \texttt{bht}[L_h] \quad \text{(subcase b)} \\
&\leq \texttt{bht}[L'_h] \quad \text{(inversion assumption).}
\end{aligned}$$

Property (ii) holds since L'_h is black by the third remark. **Q.E.D.**

Since $1 + \texttt{bht}[L_h] \geq \texttt{bht}[L'_h] \geq \texttt{bht}[L_{h+1}]$, the cost of combining L_h, y_h, L'_h into L'_{h-1} is

$$O(1 + \texttt{bht}[L_h] - \texttt{bht}[L_{h+1}]).$$

Summing up for $h = 1, \ldots, \ell$, we obtain the bound $O(\ell + \texttt{bht}[L_1] - \texttt{bht}[L_{\ell+1}]) = O(\log n)$. This concludes our proof.

Exercises

Exercise A.23: Let T_1 be the tree obtained in Exercise 6.1.
(a) Merge this with the tree T_2 with two keys: 12 and 11. The root of T_2 is 12, assumed to be black.
(b) Now split the tree obtained in (a) at the key 6. ♢

Exercise A.24: Our $O(\log n)$ bound for merging the $O(\log n)$ red-black trees in the split algorithm has fairly tight "constants" because of the inversion lemma. Give a simpler proof of an $O(\log n)$ bound by using only the assumptions of equations (38) and (39). That is, do not assume that the trees came from any particular process so that they have certain properties. ♢

End Exercises

References

[1] A. Anderson. Improving partial rebuilding by using simple balance criteria. In *Lect. Notes in C.S.*, volume 382, pages 393–402, 1989. Proc. **Workshop on Algorithms and Data Structures**, Aug. 17-19, 1989, Carleton University, Ottawa, Canada.

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

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

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

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

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

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

[10] J. Nievergelt and E. M. Reingold. Binary trees of bounded balance. *SIAM J. Computing*, 2(1), 1973.

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

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