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

balance-ness is a family property

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 [9] 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.

§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.

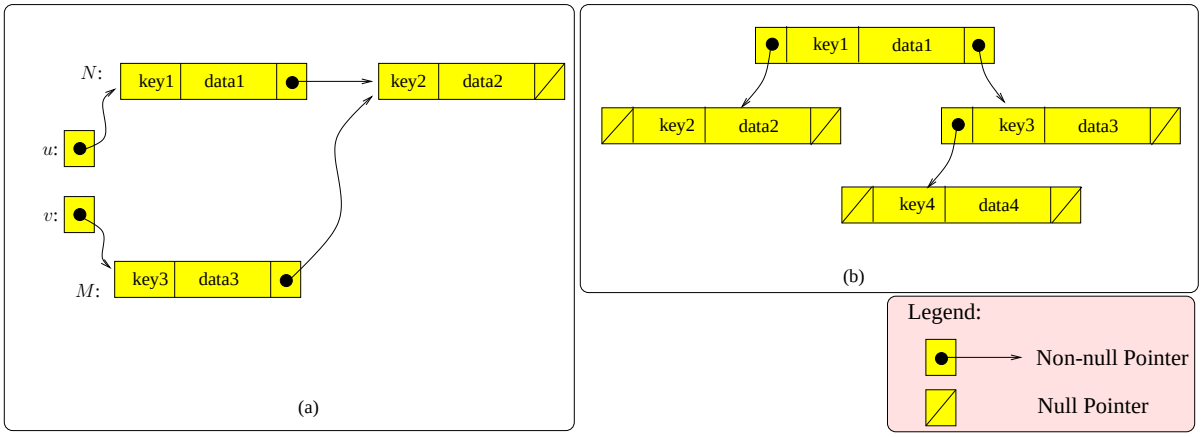


Figure 1: Two Kinds of Nodes: (a) linked lists, (b) binary trees

¶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

[1] The concept of **locatives** introduced by Lewis and Denenberg [6] 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.

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` 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 clue from the story of Alice and the White Knight

Recall the Lewis Carroll quotation at the beginning of this chapter: 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. Then u is 'Haddocks' Eyes', N is 'Ways and Means', the pointer value is 'The Aged, Aged Man', and the node is 'A-sitting On a Gate'.

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.

[2] For instance, `C++` would distinguish between nodes (N) and pointers (u) to nodes, and we would write $u \rightarrow$ `key`, $u \rightarrow$ `data`, 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 $u = \mathsf{nil}$. 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

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.

Java fans: ADT = interface

¶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, $S.\texttt{make}()$ will initialize the structure S, and $S.\texttt{max}()$ returns the maximum value in S.

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

Group (I): We need to initialize and dispose of search structures. Thus `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 $\texttt{list}()$ returns a node that is an iterator. This iterator is the beginning of a list that contains all the items in S in *some arbitrary* order. The ordering of keys is not used by the iterators. The remaining operations in this group depend on the ordering properties of keys. The $\texttt{min}()$ and $\texttt{max}()$ operations are obvious. The successor $\texttt{succ}(u)$ (resp., predecessor $\texttt{pred}(u)$) of a node u

refers to the node in S whose key has the next larger (resp., smaller) value. This is undefined if u has the largest (resp., smallest) value in S.

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

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

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

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

The node u returned by `lookUp`(K) has the property that $u.\texttt{key} = K$. 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 `lookUp`(K) returns an iterator that represents the entire set of items with key equal to K.

Both `insert` and `delete` have the obvious basic meaning. In some applications, we may prefer to have deletions that are based on key values. But such a deletion operation can be implemented as '`delete`(`lookUp`(K))'. In case `lookUp`(K) returns an iterator, we would expect the deletion to be performed over the iterator.

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

Group (IV): The final group of operations,

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

represent manipulation of entire search structures, S and S'. If $S.\texttt{split}(K) \to S'$ 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 [9]. 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.`key` and u.`data`. Doubly-linked means u has two pointers u.`next` and u.`prev`. These two pointers satisfies the invariant u.`next` $= v$ iff v.`prev` $= u$. We assume students understand linked lists, so the following discussion is a review of linked lists.

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

List traversal *Shell*

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

Here, VISIT(u) and CLEANUP() are **macros**, meaning that they stand for pieces of code that will be textually substituted before compiling and executing the program. We will indicate a macro ABC by framing it in a box like $\boxed{\text{ABC}}$. 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
    L.head:=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.

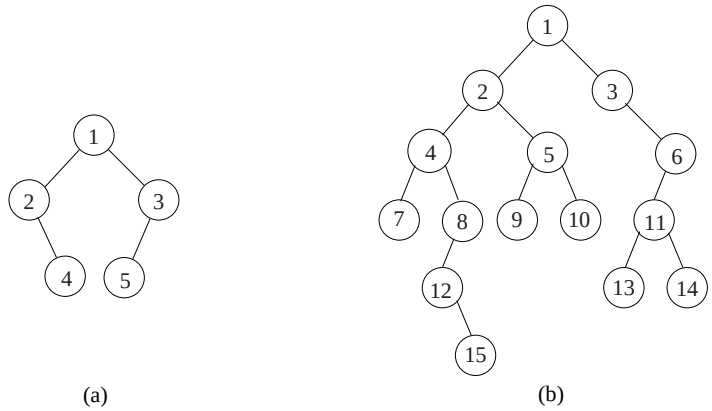


Figure 2: Two binary (not search) trees: (a) small, (b) big

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 nil.

This 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).

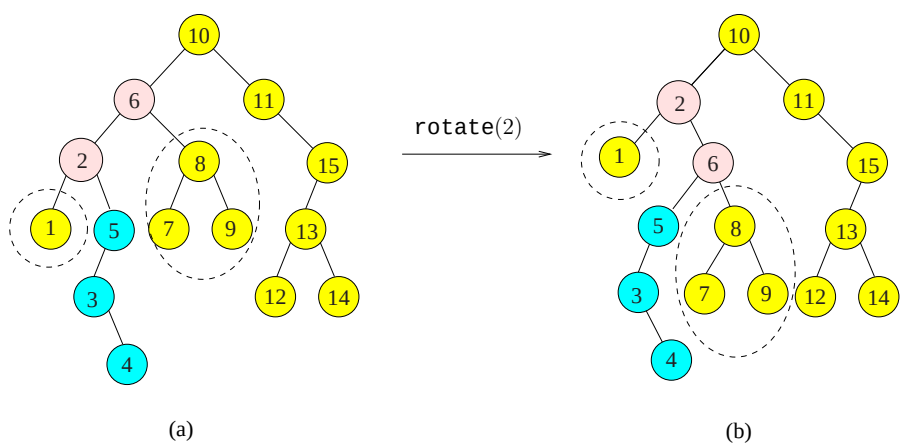


Figure 3: (a) Binary Search Tree on keys $\{1, 2, 3, 4, \ldots, 14, 15\}$. (b) After $\texttt{rotate}(2)$.

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

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).

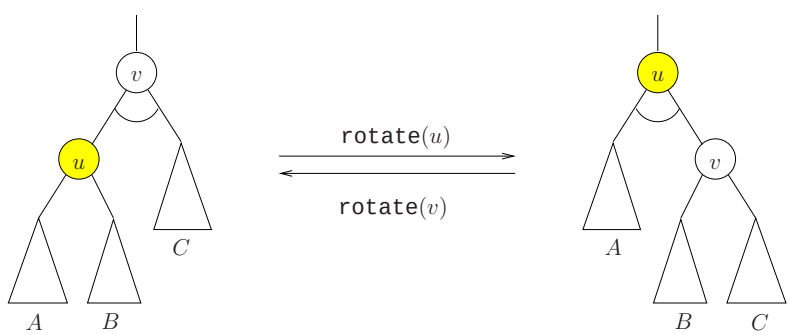


Figure 4: Rotation at u and its inverse.

In $\texttt{rotate}(u)$, 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 $\texttt{rotate}(u)$, 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 $\texttt{rotate}(u)$ followed by $\texttt{rotate}(v)$ 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.

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

¶* 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 $u.\texttt{parent} = \mathsf{nil}$, and u is type left iff $u.\texttt{parent}.\texttt{left} = u$. Clearly, $\texttt{rotate}(u)$ is sensitive to the type of u. In particular, if u is a root then $\texttt{rotate}(u)$ is the null operation. If $T \in \{\texttt{left}, \texttt{right}\}$ denote left or right type, its **complementary type** is denoted $\overline{T}$, where $\overline{\texttt{left}} = \texttt{right}$ and $\overline{\texttt{right}} = \texttt{left}$.

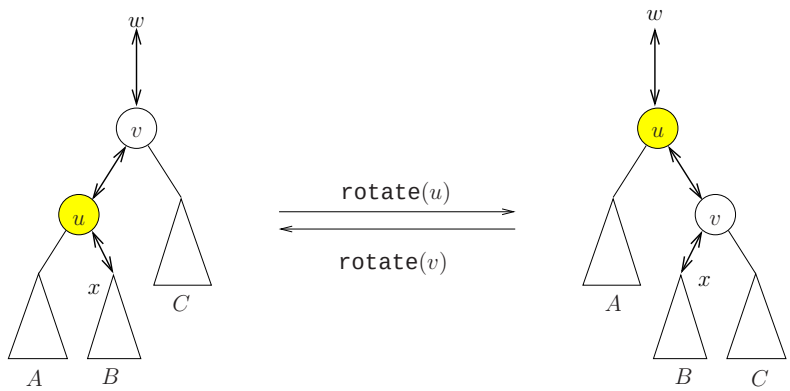


Figure 5: Links that must be fixed in $\texttt{rotate}(u)$.

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

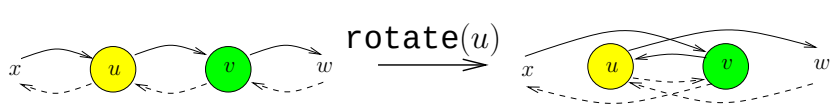


Figure 6: Simplified view of $\texttt{rotate}(u)$ as fixing a doubly-linked list (x, u, v, w).

Such re-assignments must be done in the correct order. It is best to see what is needed by thinking of (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 $u.\texttt{next}$ and $u.\texttt{prev}$ are the forward and backward pointers of a doubly-linked list. Here is the code:

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

We can now translate this sequence of 6 assignments into the corresponding assignments for binary trees: the u.`next` pointer may be identified with u.`parent` pointer. However, u.`prev` would be $u.T$ where $T \in \{\texttt{left}, \texttt{right}\}$ is the type of x. Moreover, v.`prev` is $v.\overline{T}$. Also w.`prev` is $w.T'$ for another type T'. A further complication is that x or/and w may not exist; so these conditions must be tested for, and appropriate modifications taken.

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

¶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 they do not require parent pointers.

¶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.]

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

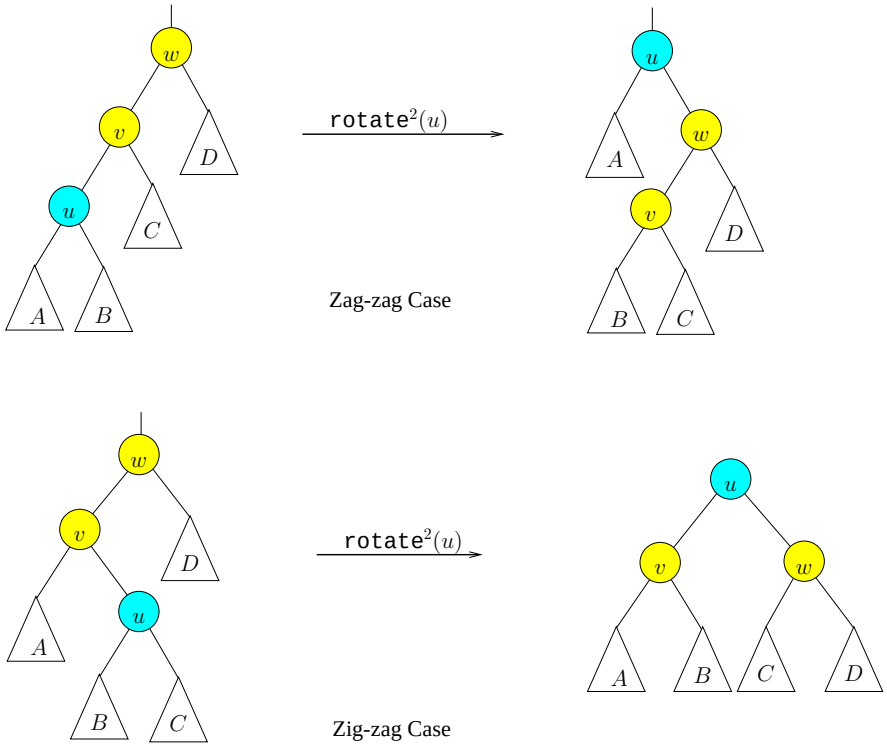
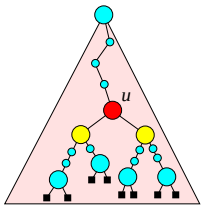


Figure 7: Two outcomes of $\texttt{rotate}^2(u)$

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 or right 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 $u.\texttt{left} = \text{nil}$, 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

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.

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*

How rotations affect the 5 paths

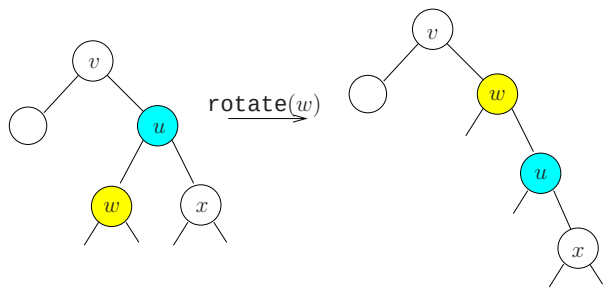


Figure 8: Reduction of the left-spine of u after $\texttt{rotate}(u.\texttt{left}) = \texttt{rotate}(w)$.

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.

We ask the reader to simulate the operations of $Delete(T, 10)$ 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**:

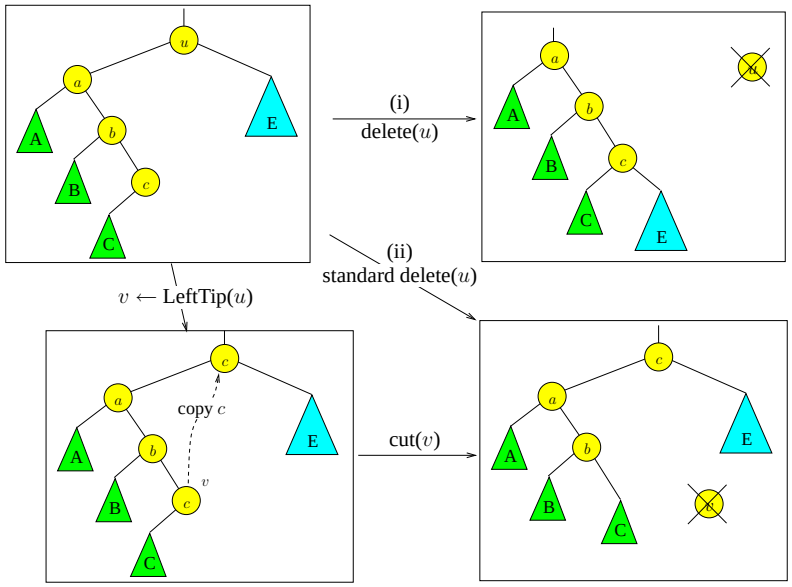


Figure 9: Deletion: (i) Rotation-based, (ii) Standard.

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

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

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

¶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 i**th 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 $u.\texttt{right} \neq$ nil then v is the tip of the right-spine of u.
(ii) If $u.\texttt{right} =$ nil then u is the tip of the left-spine of v.

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

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

The algorithm for $\texttt{pred}(u)$ is similar.

¶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 $u.\texttt{left} =$ 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, $u.\texttt{key}$ is either the smallest key in T that is greater or equal to K or the largest key in T that is less than or equal to K. Now we can repeatedly rotate at u until u becomes the root of T. At this point, we can split off either the left-subtree or right-subtree of T, renaming them as T and T' appropriately. This pair (T, T') of trees is the desired result.

work out this little detail.

¶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 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 $u.\texttt{succ}$ and $u.\texttt{pred}$ 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 $u \leftarrow successor(u)$ 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: 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**. Here is the recursive procedure to perform an in-order traversal of a tree rooted at u:

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. $\boxed{\text{BASE}(u)}$.
1. `In-order`(u.`left`).
2. $\boxed{\text{VISIT}(u)}$.
3. `In-order`(u.`right`).

This recursive program uses two macros called BASE and VISIT. For traversals, the BASE macro can be expanded into the following single line of code:

$\boxed{\text{BASE}(u)}$:
if (u=nil) Return.

The VISIT(u) macro is simply:

$\boxed{\text{VISIT}(u)}$:
    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.*

¶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}.
```

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

computing depth in pre-order

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

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

¶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: Give the in-order, pre-order and post-order listing of the tree in Figure 15. ♢

Exercise 4.2: 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. ♢

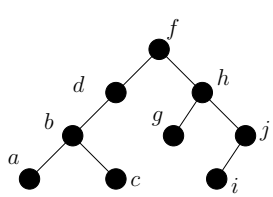


Figure 10:

Exercise 4.3: 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 4.4: 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 4.5: 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 4.6: 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 4.7: 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 4.8: 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). You must figure out the information to be returned by $CheckBST(u)$; this information should also tell you whether T_u is BST or not. Assume that each non-nil node u has the three fields, $u.\texttt{key}, u.\texttt{left}, u.\texttt{right}$. ♢

Exercise 4.9: 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 4.10: 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 [4, 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.

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

1. For any extended binary tree, if we delete all its nil nodes, we obtain a standard binary tree.
2. Conversely, for any standard binary tree, if we give every leaf two nil nodes as children and for every internal node with one child, we give it one nil node as child, then we obtain a corresponding extended binary tree.

[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".

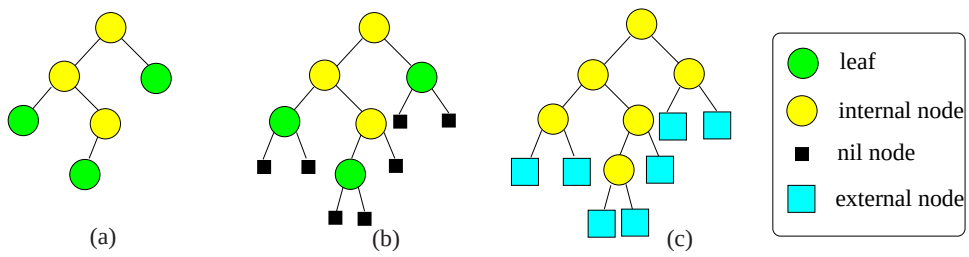


Figure 11: Binary Search Trees: (a) standard, (b) extended, (c) external

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.

Who cares about nil nodes?

The concept of a "leaf" of an extended binary tree is apt to cause some confusion: we shall use the "leaf" terminology 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 The above notion of extended binary tree is purely structural, as there is no discussion of its use in search structures. But clearly we can associate keys/items with nodes of the extended search to turn them into the corresponding notion of **extended binary search tree**. Here, the non-nil nodes store keys in the usual nodes but the nil nodes do not hold keys. So extended BST does not really add anything new: the nil nodes are just place holders or sentinels. In the following, we take this idea one step further: instead of nil nodes, we replace them by nodes that can store items, and moreover, we ban items from the internal nodes. This is the idea of "external search structures" which we next describe. This terminology for standard, extended and external binary search trees is illustrated in Figure 11.

Recall that each key is associated with some data, and such key-data pairs constitute the items for searching. There are two ways to organize such items. One way is to directly store the data with the key. The other way is for the key to be paired with a pointer to the data. Following[6] Tarjan [9], we call the latter organization an **exogenous search structure** In contrast, if the data is directly stored with the key, it is an **endogenous search structure**. What is the relative advantage of either form? The exogenous case has an extra level of indirection (the pointer) which uses extra space. But on the other hand, it means that 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. The concept of (a, b)-trees below illustrates such exogenous search structures.

More precisely, an **external BST** is a full binary tree in which every internal node u stores a key

[6] He used this classification for linked lists data structure.

u.*Key*, and every leaf u stores an item $(u.Key, u.Data)$. Moreover, the usual BST property holds:

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

where u_L (resp., u_R) is a node in the left (resp., right) subtree at u. Note that u_L and u_R may be internal nodes or leaves. The leaves in an external BST may be called **external nodes**.

Now I understand why 'internal' nodes are so-called...

¶32. 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. One way to handle 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.* 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. 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.

¶33. 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).

¶34. 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.

¶35. Implicit Binary Trees. By an implicit tree, we mean one that does not have explicit pointers which determine the parent/child relationships of nodes. An 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 property for each $i = 1, \ldots, n$,

$$HO(i): \quad A[i] \leq \min\{A[2i], A[2i+1]\}. \tag{7}$$

In (7), it is understood that if $2i > n$ (resp., $2i+1 > n$) then $A[2i]$ ($A[2i+1]$) is taken to be ∞. Then we call the binary tree a **heap**. Here is an array that represents a heap:

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

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

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

EXERCISES

Exercise 5.1: Describe what changes is needed in our binary search tree algorithms for the exogenous case. ♢

Exercise 5.2: Suppose we insist that for exogenous binary search trees, each of the keys in the internal nodes really correspond to keys in stored items. Describe the necessary changes to the deletion algorithm that will ensure this property. ♢

Exercise 5.3: 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 5.4: Describe the various algorithms on binary search trees that store the size of subtree at each node. ♢

Exercise 5.5: 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 5.6: 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.

$\diamondsuit$

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 either 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 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[7] 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.

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 call T_h a **min-size AVL tree** (for height h). Figure 12 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, we could compute the number of non-isomorphic min-sized AVL trees of a given height. But we can 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 draw such canonical trees in Figure 12.

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

$$\mu(h) = 1 + \mu(h-1) + \mu(h-2), \qquad (h \geq 1). \tag{8}$$

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 (8) holds for $h = 1$.

[7] For instance, if we say the height of the empty tree is $-\infty$, then $\mu(1) = 3$. This definition of AVL trees could certainly be supported. See Exercise for an exploration of this idea.

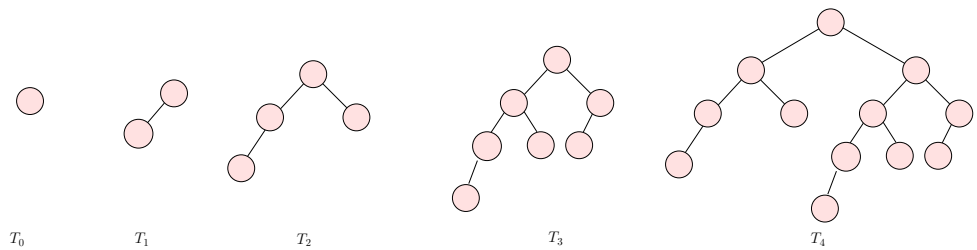


Figure 12: Canonical min-size AVL trees of heights 0, 1, 2, 3 and 4.

From (8), we have $\mu(h) \geq 2\mu(h-2)$ for $h \geq 1$. It is then easy to see by induction that $\mu(h) \geq 2^{h/2}$ for all $h \geq 1$. Writing $C = \sqrt{2} = 1.4142\ldots$, we have thus shown

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

The next lemma improves this simple lower bound on $\mu(h)$ and also provide a matching upper bound: let $\phi = \frac{1+\sqrt{5}}{2} > 1.6180$. This is the golden ratio and it is easily seen to be the positive root of the quadratic equation $x^2 - x - 1 = 0$. Hence, $\phi^2 = \phi + 1$ (in words: to square ϕ, you add 1).

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

$$\phi^h \leq \mu(h) < 2\phi^h. \tag{9}$$

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.

The bounds of this lemma are asymptotically tight; an exercise below derive an exact bound for $\mu(h)$. Actually, it is the lower bound on $\mu(h)$ that is more important. Let us derive a consequence of the lower bound on $\mu(h)$. If an AVL tree has n nodes and height h then

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

by definition of $\mu(h)$. The lower bound in (9) 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.*

¶36. 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. REMARK: We assume here the *standard* deletion algorithm, not its rotational variant. Furthermore, the node containing the deleted key and the node we *physically* removed may be different.

REBALANCE PHASE: Let x be the parent of node that was just inserted, or just *physically* deleted, in the UPDATE PHASE. We now retrace the path from x towards the root, rebalancing nodes along this path as necessary. For reference, call this the **rebalance path**.

It remains to give details for the REBALANCE PHASE. If every node along the rebalance path is balanced, then there is nothing to do in the REBALANCE PHASE. Otherwise, let u be the first unbalanced node we encounter as we move upwards from x to the root. 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. Let u be the current unbalanced node. By symmetry, we may suppose that u has balance 2. Suppose its left child is node v and has height $h+1$. Then its right child v' has height $h-1$. This situation is illustrated in Figure 13.

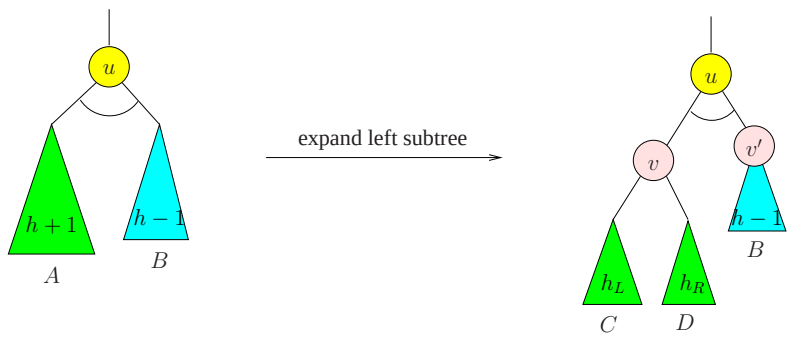


Figure 13: Node u is unbalanced after insertion or deletion.

By definition, 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.

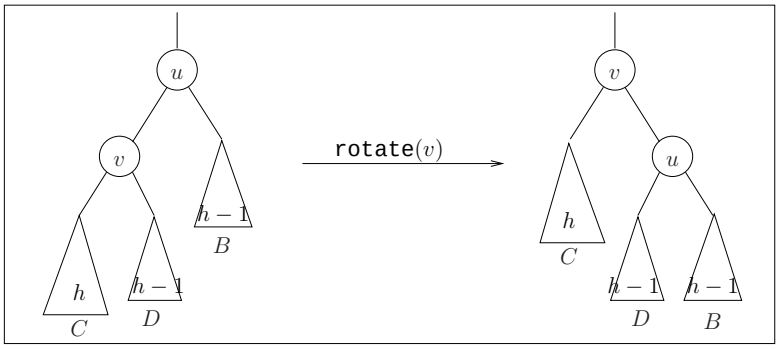
¶37. Insertion Rebalancing. Suppose that this imbalance came about because of an insertion. What was the heights of u, v and v' before the insertion? It is easy to see that the previous heights are (respectively)

$$h+1, \quad h, \quad h-1. \tag{10}$$

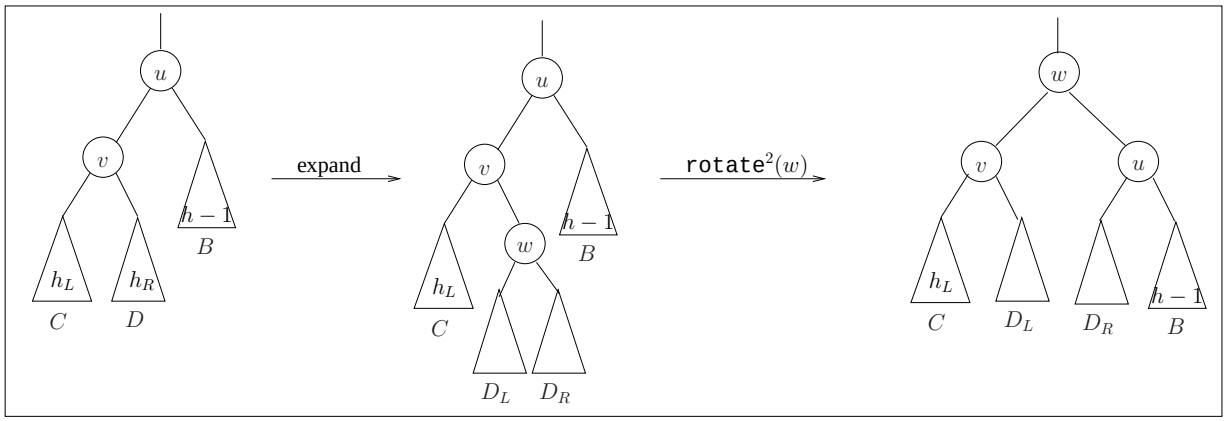
The inserted node x must be in the subtree rooted at v. Clearly, the heights h_L, h_R of the children of v satisfy $\max(h_L, h_R) = h$. Since v is currently balanced, we know that $\min(h_L, h_R) = h$ or $h-1$. But in fact, we claim that $\min(h_L, h_R) = h-1$. To see this, note that if $\min(h_L, h_R) = h$ 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 14.

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 $h+1$.

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 of $h-1$ and $h-1-\delta$ ($\delta = 0, 1$). It turns out that it does not matter which of these is the left child (despite the apparent asymmetry of the situation). If we double rotate w (*i.e.*, $\texttt{rotate}(w), \texttt{rotate}(w)$), the result is a balanced tree rooted at w of height $h+1$.



CASE (I.a)



CASE (I.b)

Figure 14: AVL Insertion: CASE (I.a) and CASE (I.b)

In both cases (I.a) and (I.b), the resulting subtree has height $h+1$. Since this was height before the insertion (see (10)), 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 15, and we insert the key 9.5. The resulting transformations is shown in Figure 16.

¶38. Deletion Rebalancing. Suppose the imbalance in Figure 13 comes from a deletion. The previous heights of u, v, v' must have been

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

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

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

CASE (D.b): $h_L = h-1$ and $h_R = h$. This is like case (I.b) and treated the same way, by performing a double rotation at w. Again, this is a non-terminal case.

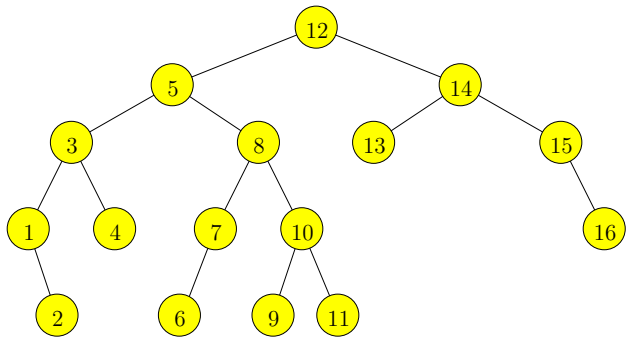


Figure 15: An AVL tree

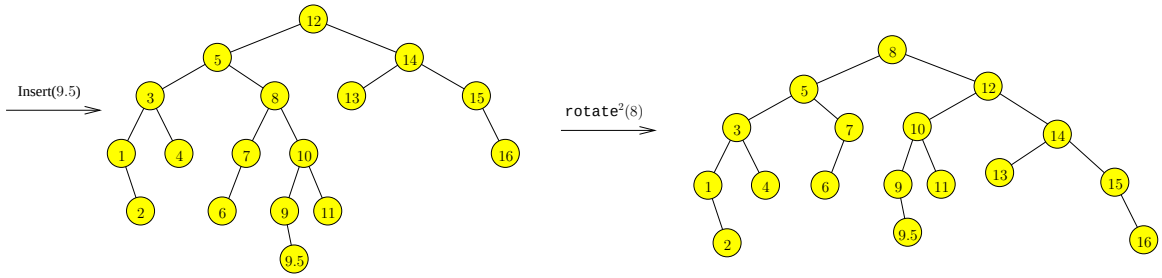


Figure 16: Inserting 9.5 into an AVL tree

CASE (D.c): $h_L = h_R = h$. This case is new, and is illustrated in Figure 17. 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 15. After deleting 13, the node 14 is unbalanced. This is restored by a single rotation at 15. Now, the root containing 12 is unbalanced. Another single rotation at 5 will restore balance. The result is shown in Figure 18.

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.

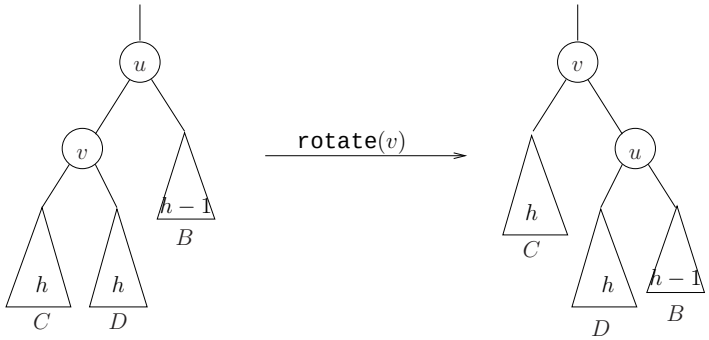


Figure 17: CASE (D.c): `rotate`(v)

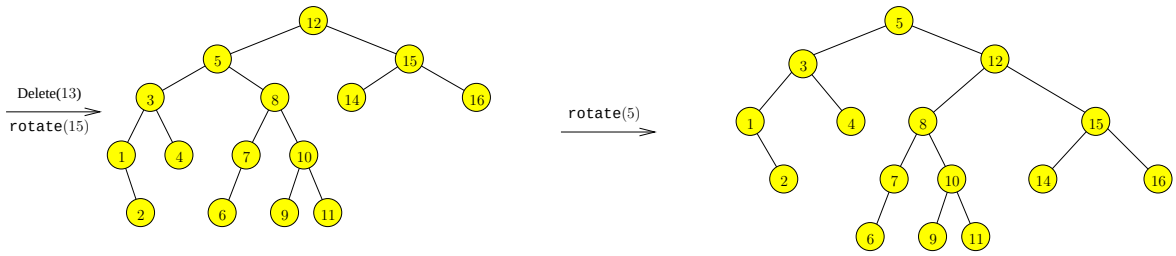


Figure 18: Deleting 13 from the AVL tree in Figure 15

¶39. 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 $\Theta(\lg n)$!

¶40. Relaxed Balancing. Larsen [5] 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: Let T be the AVL tree in 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) Delete the key 10 from T.
(b) Insert the key 2.5 into T. This question is independent of part (a).
Re-do parts (a) and (b), but using the AVL tree in Figure 3(b) instead. ♢

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

Exercise 6.4: Prove that $\mu(h) = a\phi^h + b\widetilde{\phi}^h - 1$ 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 6.5: 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 6.6: Draw 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. Draw T and the node x. ♢

Exercise 6.7: Consider the AVL tree in Figure 19.

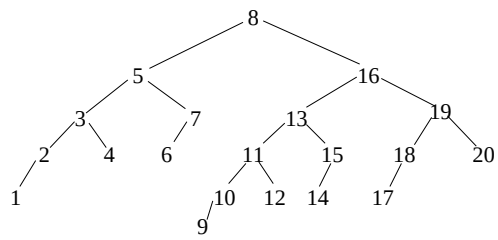


Figure 19: An AVL Tree for deletion

(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 19 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 6.8: 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 19? ♢

Exercise 6.9: (8+6 Points)
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 6.10: 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 6.11: 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 complete binary tree at the end of inserting key $2^n - 1$ for all $n \geq 1$. ♢

Exercise 6.12: 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 (8).
(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 6.13: 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 6.14: Draw two AVL trees, with n keys each: the two trees must have different heights. Make n as small as you can. ♢

Exercise 6.15: 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 6.16: 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 6.17: 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 6.18: 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 6.19: 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 6.20: 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 6.21: 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 6.22: 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 6.23: 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 6.24: 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 6.25: 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 6.26: 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.

The following example illustrates the difficulty of designing such an algorithm:

Imagine an AVL tree with a path $(u_0, u_1, \ldots, u_k)$ where u_0 is the root and u_i is a child of u_{i-1}. We have 3 conditions:

(a) Let $i \geq 1$. Then u_i is a left child iff i is odd, and otherwise u_i is a right child. Thus, the path is a pure zigzag path.

(b) The height of u_i is $k - i$ (for $i = 0, \ldots, k$). Thus u_k is a leaf.

(c) Finally, the height of the sibling of u_i is $h - i - 1$.

Suppose we are trying to insert a key whose search path in the AVL tree is precisely $(u_0, \ldots, u_k)$. Can we preemptively balance the AVL tree in this case? ♢

END EXERCISES

§7. (a, b)-Search Trees

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

$$2 \leq a < b. \tag{11}$$

An (a, b)-**tree** is a rooted, ordered tree with the following requirements:

- DEPTH BOUND: All leaves are at the same depth.
- BRANCHING BOUND: Let m be the number of children of an internal node u. In general, we have the bounds
$$a \leq m \leq b. \tag{12}$$
The root is an exception, with the bound $2 \leq m \leq b$.

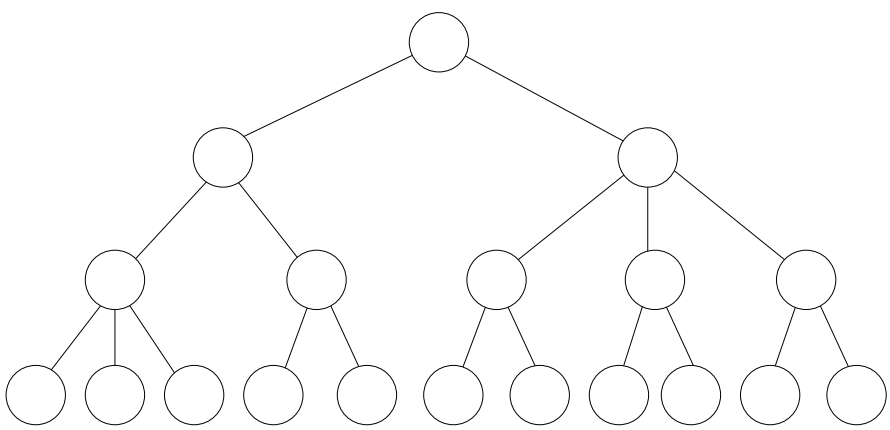


Figure 20: A $(2, 3)$-tree.

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 n leaves, the height is at most $\log_a(n) + \mathcal{O}(1)$. Therefore, (a, b)-trees forms a balanced family of trees.

The definition of (a, b)-trees imposes purely structural requirements. Figure 20 illustrates an (a, b)-tree for $(a, b) = (2, 3)$. But to use (a, b)-trees as a search structure, we need to store keys and items in the nodes of the trees. These keys and items must be suitably organized. Before giving these details, we can build some intuition by studying an example of such a search tree in Figure 21. The 14 items stored in this tree are all at the leaves, with the keys $2, 4, 6, \ldots, 23, 25, 27$. As usual, we do not display the associated data in items. The keys in the internal nodes do not correspond to items.

Recall that an item is a $(\texttt{key}, \texttt{data})$ pair. We define an (a, b)-**search tree** to be an (a, b)-tree whose nodes are organized as follows. The organization in leaves are different than in internal nodes,

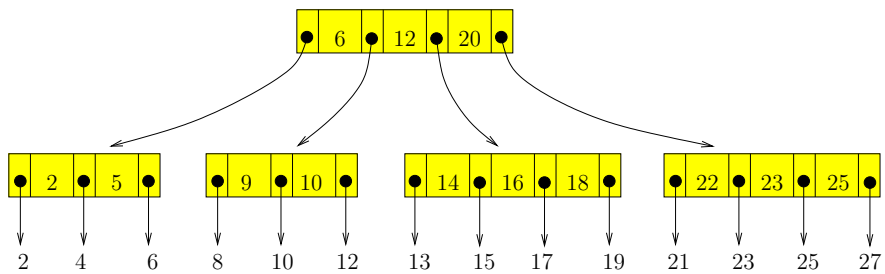


Figure 21: A (3,4)-search tree on 14 items

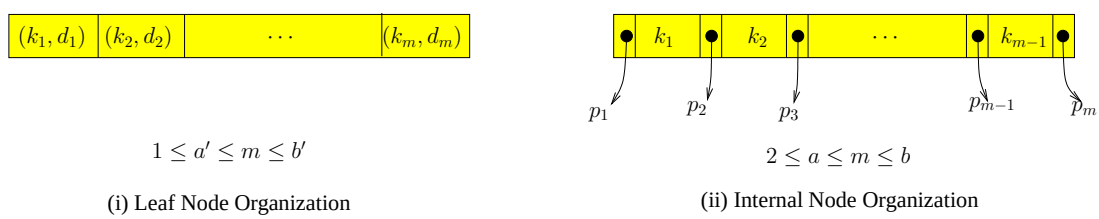


Figure 22: Organization of nodes in (a, b)-search trees

as illustrated in Figure 22. 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:

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

  $$u = (k_1, d_1, k_2, d_2, \ldots, k_m, d_m) \tag{13}$$

  where $k_1 < k_2 < \cdots < k_m$. See Figure 22(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

  $$a' \le m \le b'. \tag{14}$$

  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:

  $$u = (p_1, k_1, p_2, k_2, p_3, \ldots, p_{m-1}, k_{m-1}, p_m) \tag{15}$$

  where p_i is a pointer (or reference) 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 (13) for a leaf-node. See Figure 22(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

  $$k_{i-1} \le k < k_i, \tag{16}$$

  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).

¶41. 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 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 21) use this default choice. Another canonical choice is $a' = a, \quad b' = b$. As usual, we assume that the set of items in an (a, b)-search tree has unique keys. But, as seen in Figure 21, the keys in internal nodes may be the same as keys in the leaves. It is important to realize the different roles that keys play in the leaves and in internal nodes.

So (a', b') is implicit!

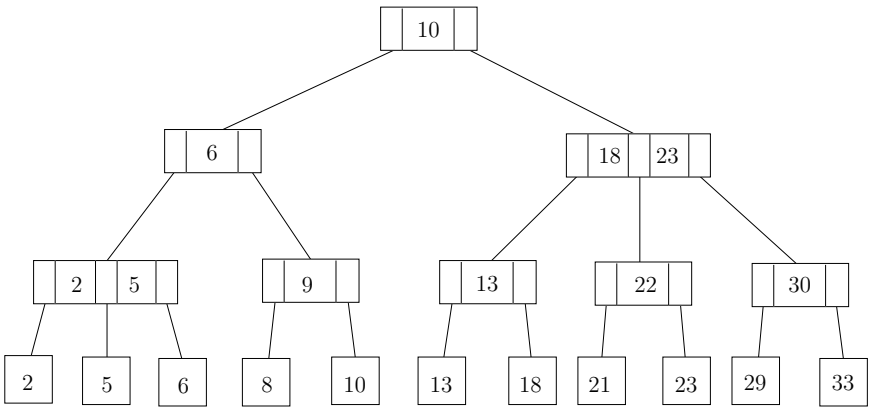


Figure 23: A $(2, 3)$-search tree.

Another (a, b)-search tree is shown in Figure 23, for the case $(a, b) = (2, 3)$. In contrast to Figure 21, here we use a slightly more standard convention of representing the pointers as tree edges.

¶42. 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

$$b = 2a - 1 \tag{17}$$

(for any $a \geq 2$), we obtain the generalization of $2 - 3$ trees called **B-trees**. These were introduced by McCreight and Bayer [2]. 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 [7].

The B-tree relationship (17) 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 (17) by $b = 2a$, the amortized complexity of such (a, b)-search trees algorithms can improve [3].

¶43. Searching. 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 (13). 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 (15). Then we find the p_i such that $k_{i-1} \le k < k_i$ (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. The following bounds the height of (a, b)-trees:

FIX: it is best to compute $M(h)$ and $\mu(h)$ as for AVL trees. This is quite easy: $M(h) := \sum_{i=0}^{h} b^i = \frac{b^{h+1}-1}{b-1}$. For $\mu(h)$, it is possible for the root to have two children and so

$$\mu(h) := 1 + 2\sum_{i=0}^{h-1} a^i = 1 + 2\frac{a^h - 1}{a - 1}.$$

The height h of an (a, b)-tree with n leaves can be bounded as follows:

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

If we take into account the parameters a', b' and let n be the number of items (not leaves), then we have:

LEMMA 6. *An (a, b)-tree with n items has height satisfying*

$$\log_b \lceil n/b' \rceil \le h \le 1 + \log_a \lfloor n/(2a') \rfloor . \tag{18}$$

Proof. The number ℓ of leaves clearly lies in the range $[\lfloor n/b' \rfloor, \lceil n/a' \rceil]$. However, with a little thought, we can improve it to:

$$\ell \in [\lceil n/b' \rceil, \lfloor n/a' \rfloor].$$

(Why?) With height h, we must have at least $2a^{h-1}$ leaves. Hence $\lfloor n/a' \rfloor \ge \ell \ge 2a^{h-1}$ or $\lfloor n/a' \rfloor /2 \ge a^{h-1}$. Since a^{h-1} is integer, we obtain $\lfloor \lfloor n/a' \rfloor /2 \rfloor = \lfloor n/(2a') \rfloor \ge a^{h-1}$ or $h \le 1 + \log_a(\lfloor n/(2a') \rfloor)$. For the lower bound on h, a similar (but simpler) argument holds. **Q.E.D.**

This lemma implies

$$\lceil \log_b \lceil n/b' \rceil \rceil \le 1 + \lfloor \log_a \lfloor n/(2a') \rfloor \rfloor . \tag{19}$$

For instance, with $n = 10^9$ (a billion), $(a, b) = (34, 51)$ and $a' = b' = 1$, this inequality is actually an equality (both sides are equal to 6). It become a strict inequality for n sufficiently large. For small n, the inequality may even fail. Hence it is clear that we need additional inequalities on our parameters.

This lemma shows that b, b' determine the lower bound 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.

¶44. 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 (15) and (13). 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. When we take into account the effects of secondary memory, 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!

¶45. 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 (11). 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").

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:

$$a \leq \left\lfloor \frac{b+1}{2} \right\rfloor. \tag{20}$$

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**. Again, we may consider borrowing a child from one of its **siblings** (there may be one or two siblings), 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:

$$a \leq \frac{b+1}{2}. \tag{21}$$

Clearly (20) implies (21). However, since a and b are integers, the reverse implication also holds! Thus (20) and (21) 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 (11) is $(a, b) = (2, 3)$; this case has been mentioned above. The case of equality in (20) and (21) 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 [7, Chap. III.5.3.1]).

¶46. 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 will 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 24. 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 $b - 1$ will be needed by the two new nodes. The extra key can be moved in the parent node as indicated.

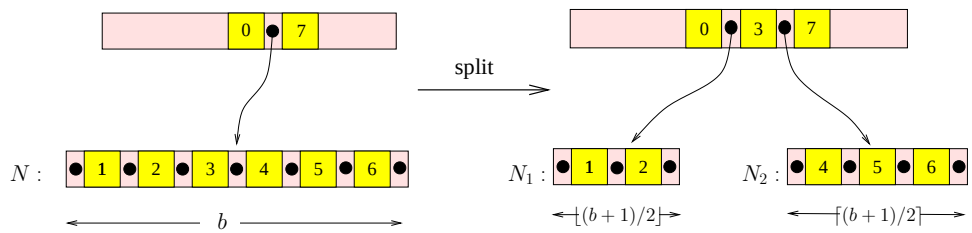


Figure 24: Splitting: N splits into N_1, N_2. Case of $(a, b) = (3, 6)$.

Next, suppose N is an underfull node with $a - 1$ children. First we try to **borrow** from a sibling if possible. This is because after borrowing, the rebalancing process can stop. To borrow, we look to a sibling (left or right), provided the sibling has more than a children. This is illustrated in Figure 25. 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.

Do not borrow from a cousin or distant cousins. Why?

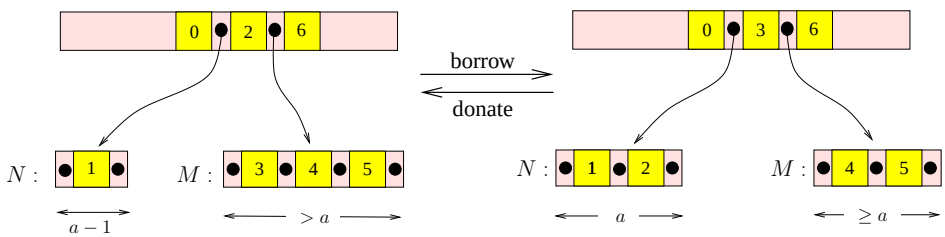


Figure 25: Borrowing: N borrows from M. Case of $(a, b) = (3, 6)$.

If N is unable to borrow, we resort to **merging**: let M be a 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 26.

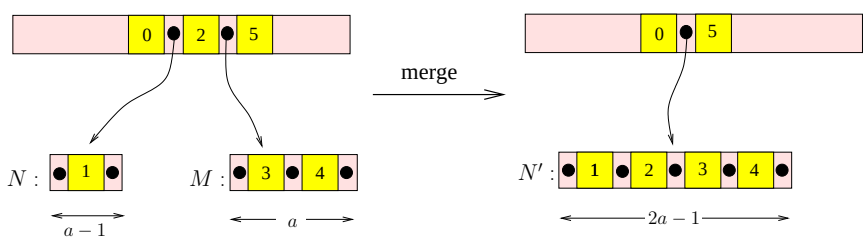


Figure 26: Merging: N and M merges into N'. Case of $(a, b) = (3, 6)$.

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 25, 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 algorithms in the literature. Below we will see the benefits of this.

¶47. Treatment of Leaves and Root. Now we must take care of these split, borrow, merge, 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 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.

Note that (i) and (ii) are the *only* means for increasing and decreasing the height of the (a, b)-tree.

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,

$$a' \leq \frac{b'+1}{2}. \tag{22}$$

But otherwise, they proceed as for the other internal nodes. 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

$$b'_0 \leq 2a' - 1. \tag{23}$$

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

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.

¶48. Mechanics of Insertion and Deletion. We are finally ready to describe insertion and deletion. It is very important that we describe these algorithms in an "I/O aware" manner, meaning that we acknowledge that the nodes are normally in secondary storage (e.g., a disk) and they need to be swapped in or out main memory. For that matter, the LookUp algorithm earlier should also be thought of in this 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 secondary storage.

We can unify the insertion/deletion algorithm by viewing both as a repeated application of the following while-loop:

INSERT/DELETE Algorithm

▷ *INITIALIZATION*

To insert an item (k, d) or delete a key k, first do a lookup on k.
Let u be the leaf node where k is inserted, or located for deletion.
Bring u into main memory and perform the insertion/deletion.
Call u the **current node**.

▷ *MAIN LOOP*

while u is overfull or underfull, do:

1. If u is root, handle as a special case and terminate.
2. Bring the parent v of u into main memory.
3. Bring sibling(s) u_j's ($j = 1, 2, \ldots$) of u into main memory.
4. Do the necessary transformations of u, u_j's and v.
   - ◁ *In main memory, nodes may temporarily have $> b$ or less than $< a$ children*
   - ◁ *We may have created a new node or deleted a node*
5. Write back into disk any modified node.
6. Make v the new current node (rename it as u) and repeat this loop.

Write the current node u to secondary memory and terminate.

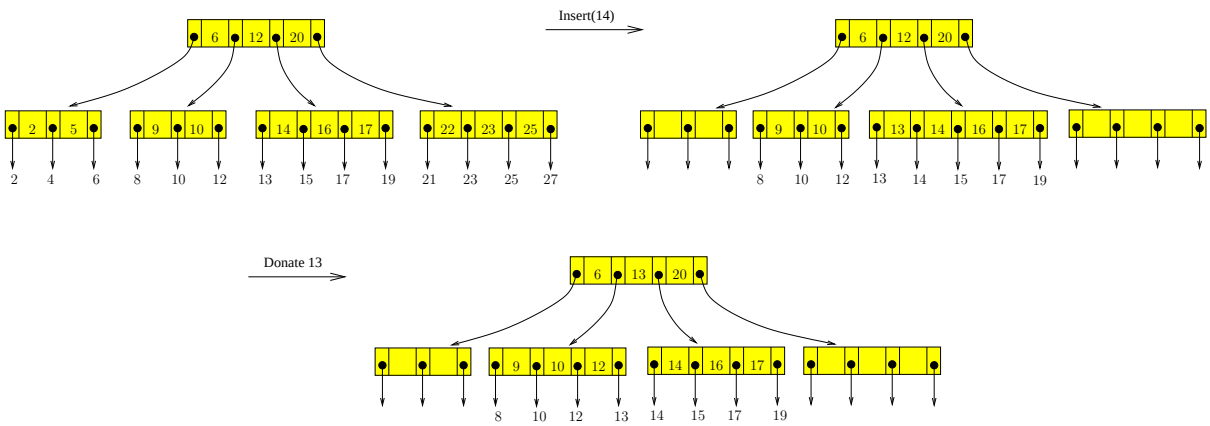


Figure 27: Inserting 14 into $(3, 4, 2)$-tree.

Observe that the INSERT/DELETE Algorithm holds in main memory at most three nodes at any moment in time. Note that in Step 5, we do not write a node u back into disk unless it has been modified. In particular, if u is split into two nodes u_1, u_2, then both u_1 and u_2 must be written back to disk.

Insertion Example: Consider inserting the item (represented by its key) 14 into the tree in Figure 21. This is illustrated by Figure 27. Note that $a' = b' = 1$. After inserting 14, we get an overfull node with 5 children. Suppose we first try to donates to our left sibling. In this case, this is possible since the left sibling has less than 4 children.

But imagine that a slightly different algorithm which tries to first donate to the right sibling. In this case, the donation fails. Then our algorithm requires us to merge with the right sibling and then split into 3 nodes. Of course, it is also possible to imagine a variant where we try to donate to the left sibling if the right sibling is full. This variant may be slower since it involves bringing an additional disk I/O. The tradeoff is that it leads to better space utilization.

¶49. 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 (21) on (a, b)-trees implies that the space utilization in such trees can never[9] 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

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

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 28.

For any integer n, we have $\lfloor n/3 \rfloor + \lfloor n/3 \rceil + \lceil n/3 \rceil = n$

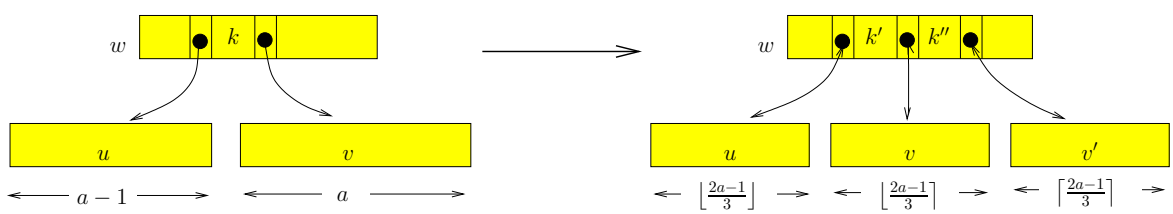


Figure 28: Generalized (2-to-3) split

We want these groups to have between a and b children. The largest of these groups has at most b children (assuming $b \geq 2$). However, for the smallest of these groups to have at least a children, we require

$$a \leq \left\lfloor \frac{2b+1}{3} \right\rfloor. \tag{24}$$

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[10] the nodes u, v, v' and then split the result into two groups, as evenly as possible. Again, this is a **generalized merge** that involves a split as well. The sizes of the two groups are $\lfloor (3a-1)/2 \rfloor$ and $\lceil (3a-1)/2 \rceil$ children, respectively. Assuming

$$a \geq 3, \tag{25}$$

v and v' exists (unless u is a child of the root). This means

$$\left\lceil \frac{3a-1}{2} \right\rceil \leq b \tag{26}$$

[9] 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.

[10] 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.

Because of integrality constraints, the floor and ceiling symbols could be removed in both (24) and (26), without changing the relationship. And thus both inequality are seen to be equivalent to

$$a \leq \frac{2b+1}{3} \tag{27}$$

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.

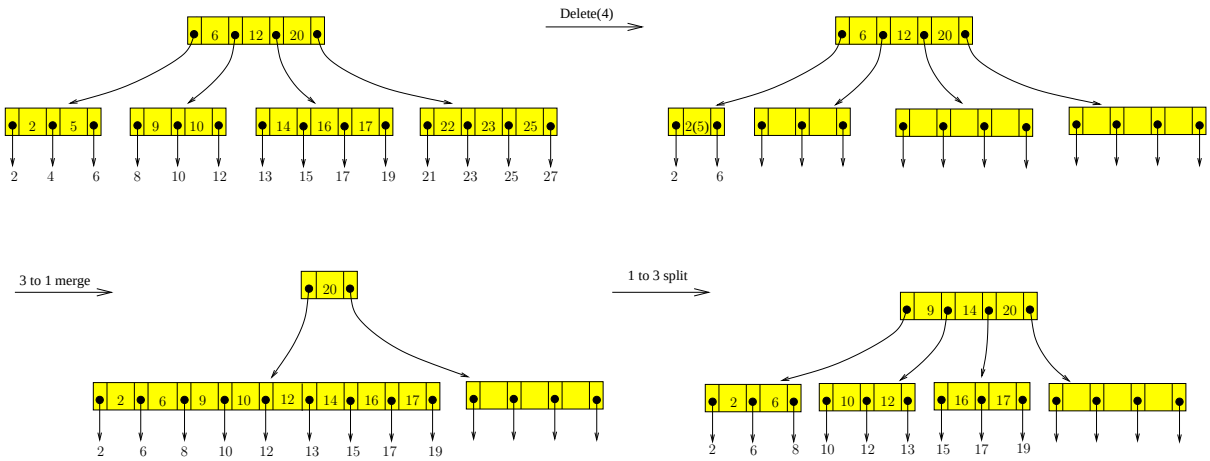


Figure 29: Deleting 4 from $(3, 4, 2)$-tree.

¶50. Example of Generalized Merge. Consider deleting the item (represented by its key) 4 from the tree in Figure 21. The is illustrated in Figure 29. 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.

> **Sharing with cousins?** In the above attempt to fix an overfull node u with $b+1$ children, we first try see to donate a child to a sibling v. Likewise, to fix an underfull node u with $a - 1$ children, we first try to borrow a child from a sibling v. By definition, two nodes u, v are siblings of each other if v and u share a common parent w. Now, the children of w are linearly ordered $u_1 < u_2 < \cdots < u_m$ in a natural way, based on their keys. We say u_i, u_{i+1} are **direct siblings** for $i = 1, \ldots, m - 1$. So each node u_i $(1 < i < m)$ has 2 direct siblings; but u_1 and u_m has only 1 direct sibling. It is important to realize that *a node can* **share** *(i.e., borrow or donate) with a direct sibling only.* In an Exercise, we consider a relaxed sharing condition, whereby sharing can be done between u and v if they are **direct cousins**.

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 (21); 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 memory at all times, unlike the other nodes.
(2) Allow 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 21. 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.

¶51. Exogenous and Endogenous Search Structures. Search trees store items. But where these items are stored constitute a major difference between (a, b)-search trees and the binary search trees which we have presented. Items in (a, b)-search trees are stored in the leaves only, while in binary search trees, items are stored in internal nodes as well. Tarjan [9, p. 9] calls a search structure **exogenous** if it stores items in leaves only; otherwise it is **endogenous**.

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 binary search trees (or their balanced versions such as AVL trees), we never explicitly discuss the data that are associated with keys. So how do we know that these data structures are endogenous? We deduce it from the observation that, in looking up a key k in a binary search tree, if k is found in an internal node u, we stop the search and return u. Implicitly, it means we have found the item with key k (effectively, the item is stored in u). 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. It is possible to modify binary search trees so that they become exogenous (Exercise).

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

There is another important consequence of this dual role of keys in (a, b)-search trees. The keys in the internal nodes *need not be the keys of items that are stored in the leaves*. This is seen in Figure 23 where the key 9 in an internal node does not correspond to any actual item in the tree. On the other hand, the key 13 appears in the leaves (as an item) as well as in an internal node.

¶52. Database Application. One reason for treating (a, b)-trees as exogenous 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 (13) 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.

¶53. Disk I/O Considerations: How to choose the parameter b. There is another reason for preferring exogenous structures. In databases, the number of items is very large and these are stored in disk memory. If there are n items, then we need at least n/b' internal nodes. This many internal nodes implies that the nodes of the (a, b)-trees is also stored in disk memory. Therefore, while searching through the (a, b)-tree, each node we visit must be brought into the main memory from disk. The I/O speed for transferring data between main memory and disk is relatively slow, compared to CPU speeds.

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.

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.

¶54. 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 $c \geq 1$. 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**,

$$c + 1 \leq a \leq \frac{cb+1}{c+1}. \tag{28}$$

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 (28) 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 (28) guarantees success.

We are interested in the maximum value of a in (28). Using the fact that a is integer, this amounts to

$$a = \left\lfloor \frac{cb+1}{c+1} \right\rfloor . \tag{29}$$

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.

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 $(a, b, c) = (4, 5, 3)$.

¶55. 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"). For any (a, b) parameter, we can compute the smallest c such that this could be a (a, b, c)-tree.

Therefore, we can freely modify the c as we wish. The c-parameter is only used during algorithm insertion/deletion, and this can be stored as a global variable. E.g., c can be a static member of the (a, b, c) class, if we implement this using `C++`). 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.

¶56. 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 $10b \leq 512$. 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. Assume 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.

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 (use Lemma 6).

¶57. 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 $c \geq 2$.

Hw5 from FunAlgo, Sp2011:

Exercise 7.1: Recall the "standard" insertion in (a, b)-search trees (see ¶43, p.43, in Lecture III). Each Lookup in a tree of height h takes $h+O(1)$ I/O operations where an I/O operation can be reading a node from disk, or writing a node into disk. Note that we do not need to write a node back to disk unless it has been modified. In the insertion/deletion algorithm in ¶46, p.45, we should have said "write back to secondary memory (i.e., disk) any node *that has been modified*". In other words, if a node has not been modified, we need not waste any I/O operation to write it back to disk.
(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 (this was discussed by Esther in recitation, and also in ¶54, p.50).
(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?

$\diamondsuit$

Exercise 7.2: Do the same analysis as the previous question, but for standard deletion algorithm. $\diamondsuit$

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.

¶58. 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 [7]). This was the beginning of a technique called "fringe analysis" which Yao [10] introduced in 1974. Nakamura and Mizoguchi [8] independently discovered the analysis, and Knuth used similar ideas in 1973 (see the survey of [1]).

Now consider the space utilization ratio of generalized B-trees. Under (29), 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 7.3: 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 7.4: 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 (this was discussed by Esther in recitation, and also in ¶54, p.50).
(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 7.5: 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 7.6: What is the the best ratio achievable under (21)? Under (27)? ♢

Exercise 7.7: Give a more 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 (21). ♢

Exercise 7.8: Describe the exogenous version of binary search trees. Give the insertion and deletion algorithms. NOTE: the keys in the leaves are now viewed as surrogates for the items. Moreover, we allow the keys in the internal nodes to duplicate keys in the leaves, and it is also possible that some keys in the internal nodes correspond to no stored item. ♢

Exercise 7.9: Consider the tree shown in Figure 21. 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 7.10: 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 7.11: Is it possible to design (a, b, c) trees so that the root is not treated as an exception? ♢

Exercise 7.12: 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 7.13: Our insertion and deletion algorithms tries to share (i.e., donate or borrow) children from siblings only. Suppose we now relax this condition to allow sharing among "cousins". Consider all the nodes in a given level: two nodes nodes u, v are **cousins** of each other if they belong to the same level but they are not siblings. All the nodes v_i ($i = 1, \ldots, M$) in a given level can be sorted based on their keys, $v_1 < v_2 < \cdots < v_M$. If v_i, v_{i+1} are not siblings, then we call them **direct cousins**. Modify our insert/delete algorithms so that we try to share with direct siblings or cousins before doing the generalized split/merge. ♢

Exercise 7.14: 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 7.15: How can we choose the a parameter (see (29)) in generalized B-trees in a more relaxed manner so that the repeated splits/merges during insertion and deletions are minimized? ♢

END EXERCISES

References

[1] R. A. Baeza-Yates. Fringe analysis revisited. *ACM Computing Surveys*, 27(1):109–119, 1995.

[2] R. Bayer and McCreight. Organization of large ordered indexes. *Acta Inform.*, 1:173–189, 1972.

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

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

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

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

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

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

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

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