21. List structures

Learning objectives:

- static vs dynamic data structures
- linear, circular and two-way lists
- fifo queue implemented as a linear list
- breadth-first and depth-first tree traversal
- traversing a binary tree without any auxiliary memory: triple tree traversal algorithm
- dictionary implemented as a binary search tree
- Balanced trees guarantee that dictionary operations can be performed in logarithmic time
- height-balanced trees
- multiway trees

Lists, memory management, pointer variables

The spectrum of data structures ranges from static objects, such as a table of constants, to dynamic structures, such as lists. A list is designed so that not only the data values stored in it, but its *size* and *shape* can change at run time, due to insertions, deletions, or rearrangement of data elements. Most of the data structures discussed so far can change their size and shape to a limited extent. A circular buffer, for example, supports insertion at one end and deletion at the other, and can grow to a predeclared maximal size. A heap supports deletion at one end and insertion anywhere into an array. In a list, any local change can be done with an effort that is independent of the size of the list - provided that we know the memory locations of the data elements involved. The key to meeting this requirement is the idea of abandoning memory allocation in large contiguous chunks, and instead allocating it dynamically in the smallest chunk that will hold a given object. Because data elements are stored randomly in memory, not contiguously, an insertion or deletion into a list does not propagate a ripple effect that shifts other elements around. An element inserted is allocated anywhere in memory where there is space and tied to other elements by **pointers** (i.e. addresses of the memory locations where these elements happen to be stored at the moment). An element deleted does not leave a gap that needs to be filled as it would in an array. Instead, it leaves some free space that can be reclaimed later by a memory management process. The element deleted is likely to break some chains that tie other elements together; if so, the broken chains are relinked according to rules specific to the type of list used.

Pointers are the language feature used in modern programming languages to capture the equivalent of a memory address. A pointer value is essentially an address, and a pointer variable ranges over addresses. A pointer, however, may contain more information than merely an address. In Pascal and other strongly typed languages, for example, a pointer also references the type definition of the objects it can point to - a feature that enhances the compiler's ability to check for consistent use of pointer variables.

Let us illustrate these concepts with a simple example: a **one-way linear list** is a sequence of cells each of which (except the last) points to its successor. The first cell is the head of the list, the last cell is the tail. Since the

tail has no successor, its pointer is assigned a predefined value 'nil', which differs from the address of any cell. Access to the list is provided by an external pointer 'head'. If the list is empty, 'head' has the value 'nil'. A cell stores an element x_i and a pointer to the successor cell (Exhibit 21.1):

```
type cptr = ^cell;
  cell = record  e: elt;  next: cptr  end;
```

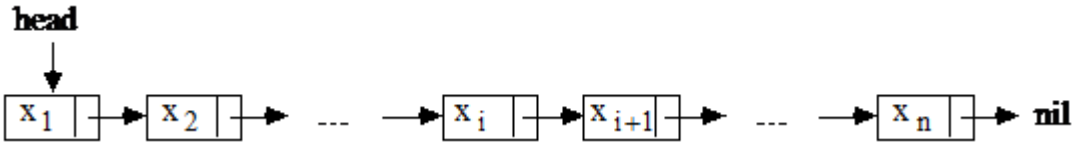


Exhibit 21.1: A one-way linear list.

Local operations, such as insertion or deletion at a position given by a pointer p, are efficient. For example, the following statements insert a new cell containing an element y as successor of a cell being pointed at by p (Exhibit 21.2):

```
new(q);  q^.e := y;  q^.next := p^.next;  p^.next := q;
```

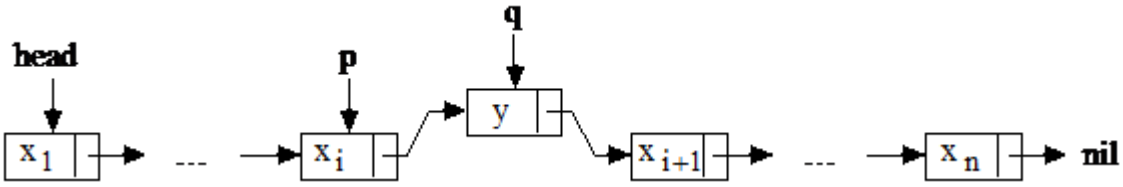


Exhibit 21.2: Insertion as a local operation.

The successor of the cell pointed at by p is deleted by a single assignment statement (Exhibit 21.3):

```
p^.next := p^.next^.next;
```

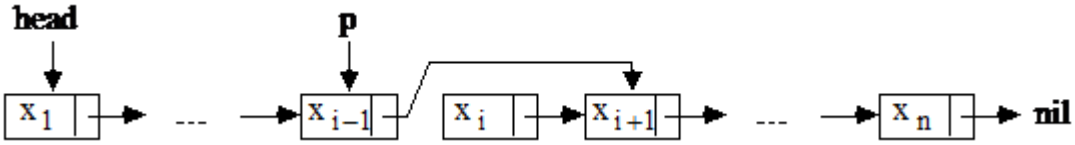


Exhibit 21.3: Deletion as a local operation.

An insertion or deletion at the head or tail of this list is a special case to be handled separately. To support insertion at the tail, an additional pointer variable 'tail' may be set to point to the tail element, if it exists.

A one-way linear list sometimes is handier if the tail points back to the head, making it a **circular list**. In a circular list, the head and tail cells are replaced by a single entry cell, and any cell can be reached from any other without having to start at the external pointer 'entry' (Exhibit 21.4).

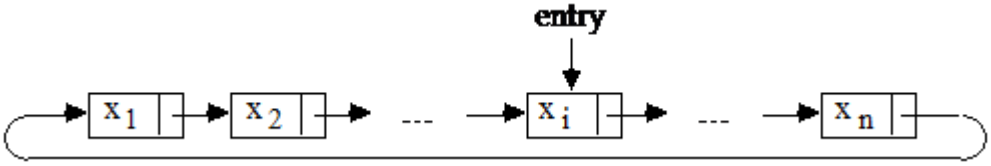


Exhibit 21.4: A circular list combines head and tail into a single entry point

In a *two-way* (or *doubly linked*) *list* each cell contains two pointers, one to its successor, the other to its predecessor. The list can be traversed in both directions. Exhibit 21.5 shows a circular two-way list.

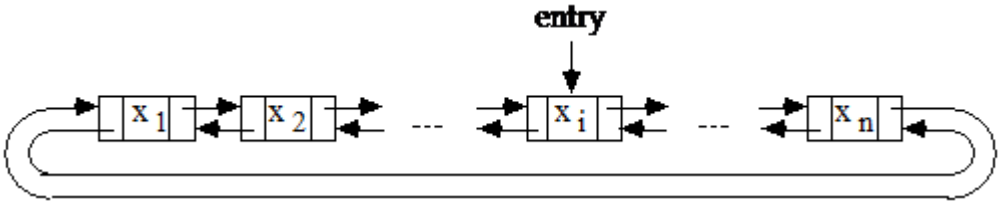


Exhibit 21.5: A circular two-way or doubly-linked list

Exercise: traversal of a singly linked list in both directions

Write a recursive

```
procedure traverse(p: cptr);
```

to traverse a singly linked list from the head to the tail and back again. At each visit of a node, call the

```
procedure visit(p: cptr);
```

Solve the same problem iteratively without using any additional storage beyond a few local pointers. Your traversal procedure may modify the structure of the list temporarily.

Solution

```
(a)    procedure traverse(p: cptr);
  begin  if  p ≠ nil  then  { visit(p);  traverse(p^.next);
visit(p) }  end;

  The initial call of this procedure is
  traverse(head);
```

```
(b)    procedure traverse(p: cptr);
  var  o, q: cptr;  i: integer;
  begin
    for  i := 1  to  2  do  { forward and back again }  begin
      o := nil;
      while  p ≠ nil  do  begin
        visit(p);  q := p^.next;  p^.next := o;
        o := p;  p := q  { the fork advances }
      end;
      p := o
    end
  end;

  Traversal becomes simpler if we let the 'next' pointer of the tail
cell point to this cell itself:

    procedure traverse(p: cptr);
  var o, q: cptr;
  begin
    o := nil;
    while  p ≠ nil  do  begin
      visit(p);  q := p^.next;  p^.next := o;
      o := p;  p := q  { the fork advances }
    end
  end;
```

The fifo queue implemented as a one-way list

It is natural to implement a fifo queue as a one-way linear list, where each element points to the next one "in line". The operation 'dequeue' occurs at the pointer 'head', and 'enqueue' is made fast by having an external pointer 'tail' point to the last element in the queue. A crafty implementation of this data structure involves an empty cell, called a *sentinel*, at the tail of the list. Its purpose is to make the list-handling procedures simpler and faster by making the empty queue look more like all other states of the queue. More precisely, when the queue is empty, the external pointers 'head' and 'tail' both point to the sentinel rather than having the value 'nil'. The sentinel allows insertion into the empty queue, and deletion that results in an empty queue, to be handled by the same code that handles the general case of 'enqueue' and 'dequeue'. The reader should verify our claim that a sentinel simplifies the code by programming the plausible, but less efficient, procedures which assume that an empty queue is represented by head = tail = nil.

The queue is empty if and only if 'head' and 'tail' both point to the sentinel (i.e. if head = tail). An 'enqueue' operation is performed by inserting the new element into the sentinel cell and then creating a new sentinel.

```
type cptr = ^cell;
  cell = record  e: elt;  next: cptr  end;
  fifoqueue = record  head, tail: cptr  end;

procedure create(var f: fifoqueue);
begin  new(f.head);  f.tail := f.head  end;

function empty(f: fifoqueue): boolean;
begin  return(f.head = f.tail)  end;

procedure enqueue(var f: fifoqueue; x: elt);
begin  f.tail^.e := x;  new(f.tail^.next);  f.tail := f.tail^.next
end;

function front(f: fifoqueue): elt;
{ not to be called if the queue is empty }
begin  return(f.head^.e)  end;

procedure dequeue(var f: fifoqueue);
{ not to be called if the queue is empty }
begin  f.head := f.head^.next  end;
```

Tree traversal

When we speak of trees in computer science, we usually mean **rooted, ordered trees**: they have a distinguished node called the root, and the subtrees of any node are ordered. Rooted, ordered trees are best defined recursively: a tree T is either empty, or it is a tuple (N, T_1, ... , T_k), where N is the root of the tree, and T_1, ... , T_k is a sequence of trees. Binary trees are the special case k = 2.

Trees are typically used to organize data or activities in a hierarchy: a top-level data set or activity is composed of a next level of data or activities, and so on. When one wishes to gather or survey all of the data or activities, it is necessary to traverse the tree, visiting (i.e. processing) the nodes in some systematic order. The visit at each node might be as simple as printing its contents or as complicated as computing a function that depends on all nodes in the tree. There are two major ways to traverse trees: breadth first and depth first.

Breadth-first traversal visits the nodes level by level. This is useful in heuristic search, where a node represents a partial solution to a problem, with deeper levels representing more complete solutions. Before pursuing any one solution to a great depth, it may be advantageous to assess all the partial solutions at the present

level, in order to pursue the most promising one. We do not discuss breadth-first traversal further, we merely suggest the following:

Exercise: breadth-first traversal

Decide on a representation for trees where each node may have a variable number of children. Write a procedure for breadth-first traversal of such a tree. *Hint:* use a fifo queue to organize the traversal. The node to be visited is removed from the head of the queue, and its children are enqueued, in order, at the tail end.

Depth-first traversal always moves to the first unvisited node at the next deeper level, if there is one. It turns out that depth-first better fits the recursive definition of trees than breadth-first does and orders nodes in ways that are more often useful. We discuss depth-first for binary trees and leave the generalization to other trees to the reader. Depth-first can generate three basic orders for traversing a binary tree: **preorder**, **inorder**, and **postorder**, defined recursively as:

```
preorder  Visit root, traverse left subtree, traverse right subtree.
Inorder Traverse left subtree, visit root, traverse right subtree.
postorder Traverse left subtree, traverse right subtree, visit root.
```

For the tree in Exhibit 21.6we obtain the orders shown.

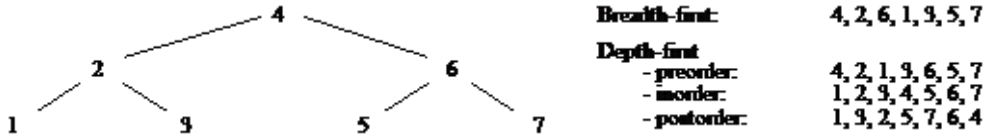


Exhibit 21.6: Standard orders defined on a binary tree

An arithmetic expression can be represented as a binary tree by assigning the operands to the leaves and the operators to the internal nodes. The basic traversal orders correspond to different notations for representing arithmetic expressions. By traversing the expression tree (Exhibit 21.7) in preorder, inorder, or postorder, we obtain the **prefix**, **infix**, or **suffix** notation, respectively.

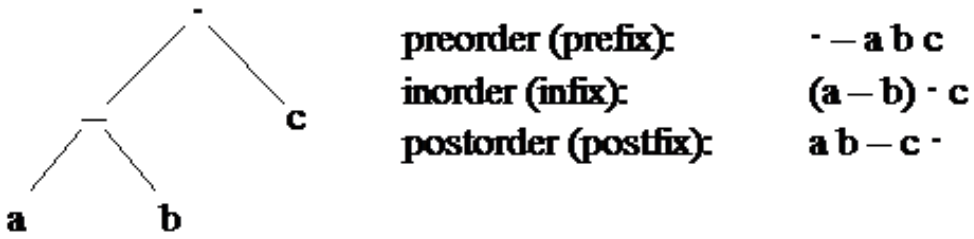


Exhibit 21.7: Standard traversal orders correspond to different notations for arithmetic expressions

A binary tree can be implemented as a list structure in many ways. The most common way uses an external pointer 'root' to access the root of the tree and represents each node by a cell that contains a field for an element to be stored, a pointer to the root of the left subtree, and a pointer to the root of the right subtree (Exhibit 21.8). An empty left or right subtree may be represented by the pointer value 'nil', or by pointing at a sentinel, or, as we shall see, by a pointer that points to the node itself.

```
type nptr =  ^node;
  node =  record  e: elt;  L, R: nptr  end;
var  root: nptr;
```

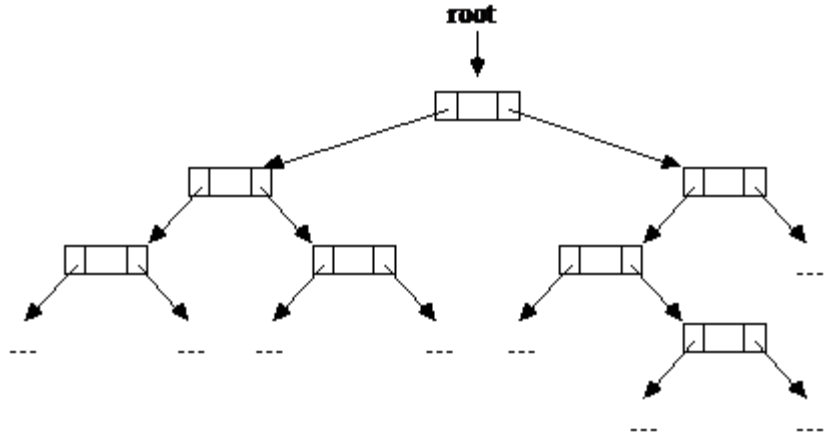


Exhibit 21.8: Straightforward implementation of a binary tree

The following procedure 'traverse' implements any or all of the three orders preorder, inorder, and postorder, depending on how the procedures 'visit$_1$', 'visit$_2$', and 'visit$_3$' process the data in the node referenced by the pointer p. The root of the subtree to be traversed is passed through the formal parameter p. In the simplest case, a visit does nothing or simply prints the contents of the node.

```
procedure traverse(p: nptr);
begin
  if  p ≠ nil  then  begin
    visit1(p);  { preorder }
    traverse(p^.L);
    visit2(p);  { inorder }
    traverse(p^.R);
    visit3(p)  { postorder }
  end
end;
```

Traversing a tree involves both advancing from the root toward the leaves, and backing up from the leaves toward the root. Recursive invocations of the procedure 'traverse' build up a stack whose entries contain references to the nodes for which 'traverse' has been called. These entries provide a means of returning to a node after the traversal of one of its subtrees has been finished. The bookkeeping done by a stack or equivalent auxiliary structure can be avoided if the tree to be traversed may be modified temporarily.

The following **triple-tree traversal** algorithm provides an elegant and efficient way of traversing a binary tree without using any auxiliary memory (i.e. no stack is used and it is not assumed that a node contains a pointer to its parent node). The data structure is modified temporarily to retain the information needed to find the way back up the tree and to restore each subtree to its initial condition after traversal. The triple-tree traversal algorithm assumes that an empty subtree is encoded not by a 'nil' pointer, but rather by an L (left) or R (right) pointer that points to the node itself, as shown in Exhibit 21.9.

Exhibit 21.9: Coding of a leaf used in procedure TTT

```
procedure TTT;
var  o, p, q: nptr;
begin
   o := nil;  p:= root;
   while  p ≠ nil  do  begin
      visit(p);
      q := p^.L;
      p^.L := p^.R;  { rotate left pointer }
      p^.R := o;  { rotate right pointer }
      o := p;
      p := q
   end
end;
```

In this procedure the pointers p ("present") and o ("old") serve as a two-pronged fork. The tree is being traversed by the pointer p and the companion pointer o, which always lags one step behind p. The two pointers form a two-pronged fork that runs around the tree, starting in the initial condition with p pointing to the root of the tree, and o = nil. An auxiliary pointer q is needed temporarily to advance the fork. The while loop in 'TTT' is executed as long as p points to a node in the tree and is terminated when p assumes the value 'nil'. The initial value of the o pointer gets saved as a temporary value. First it is assigned to the R pointer of the root, later to the L pointer. Finally, it gets assigned to p, the fork exits from the root of the tree, and the traversal of the tree is complete. The correctness of this algorithm is proved by induction on the number of nodes in the tree.

Induction hypothesis H: if at the beginning of an iteration of the while loop, the fork pointer p points to the root of a subtree with n > 0 nodes, and o has a value x that is different from any pointer value inside this subtree, then after $3 \cdot n$ iterations the subtree will have been traversed in triple order (visiting each node exactly three times), all tree pointers in the subtree will have been restored to their original value, and the fork pointers will have been reversed (i.e. p has the value x and o points to the root of the subtree).

Base of induction: H is true for n = 1.

Proof: The smallest tree we consider has exactly one node, the root alone. Before the while loop is executed for this subtree, the fork and the tree are in the initial state shown in Exhibit 21.10. Exhibit 21.11 shows the state of the fork and the tree after each iteration of the while loop. The node is visited in each iteration.

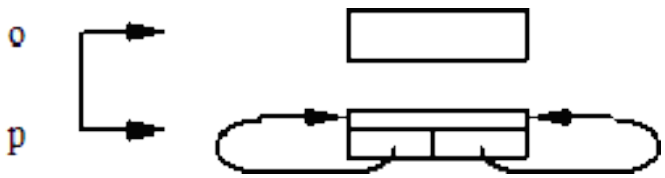


Exhibit 21.10 : Initial configuration for traversing a tree consisting of a single node

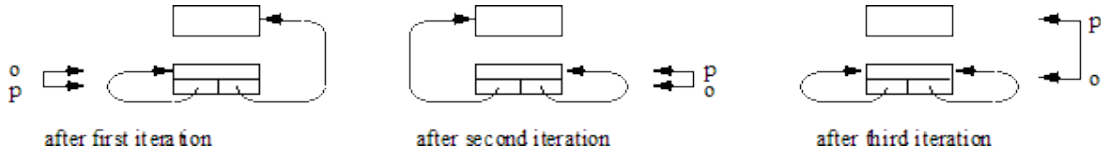


Exhibit 21.11: Tracing procedure TTT while traversing the smallest tree

Induction step: If H is true for all n, $0 < n \leq k$, H is also true for $k + 1$.

Proof: Consider a tree T with $k + 1$ nodes. T consists of a root and k nodes shared among the left and right subtrees of the root. Each of these subtrees has $\leq k$ nodes, so we apply the induction hypothesis to each of them. The following is a highly compressed account of the proof of the induction step, illustrated by Exhibit 21.12. Consider the tree with $k + 1$ nodes shown in state 1. The root is a node with three fields; the left and right subtrees are shown as triangles. The figure shows the typical case when both subtrees are nonempty. If one of the two subtrees is empty, the corresponding pointer points back to the root; these two cases can be handled similarly to the case $n = 1$. The fork starts out with p pointing at the root and o pointing at anything outside the subtree being traversed. We want to show that the initial state 1 is transformed in $3 \cdot (k + 1)$ iterations into the final state 6. In the final state the subtrees are shaded to indicate that they have been correctly traversed; the fork has exited from the root, with p and o having exchanged values. To show that the algorithm correctly transforms state 1 into state 6, we consider the intermediate states 2 to 5, and what happens in each transition.

$1 \rightarrow 2$ One iteration through the while loop advances the fork into the left subtree and rotates the pointers of the root.

$2 \rightarrow 3$ H applied to the left subtree of the root says that this subtree will be correctly traversed, and the fork will exit from the subtree with pointers reversed.

$3 \rightarrow 4$ This is the second iteration through the while loop that visits the root. The fork advances into the right subtree, and the pointers of the root rotate a second time.

$4 \rightarrow 5$ H applied to the right subtree of the root says that this subtree will be correctly traversed, and the fork will exit from the subtree with pointers reversed.

$5 \rightarrow 6$ This is the third iteration through the while loop that visits the root. The fork moves out of the tree being traversed; the pointers of the root rotate a third time and thereby assume their original values.

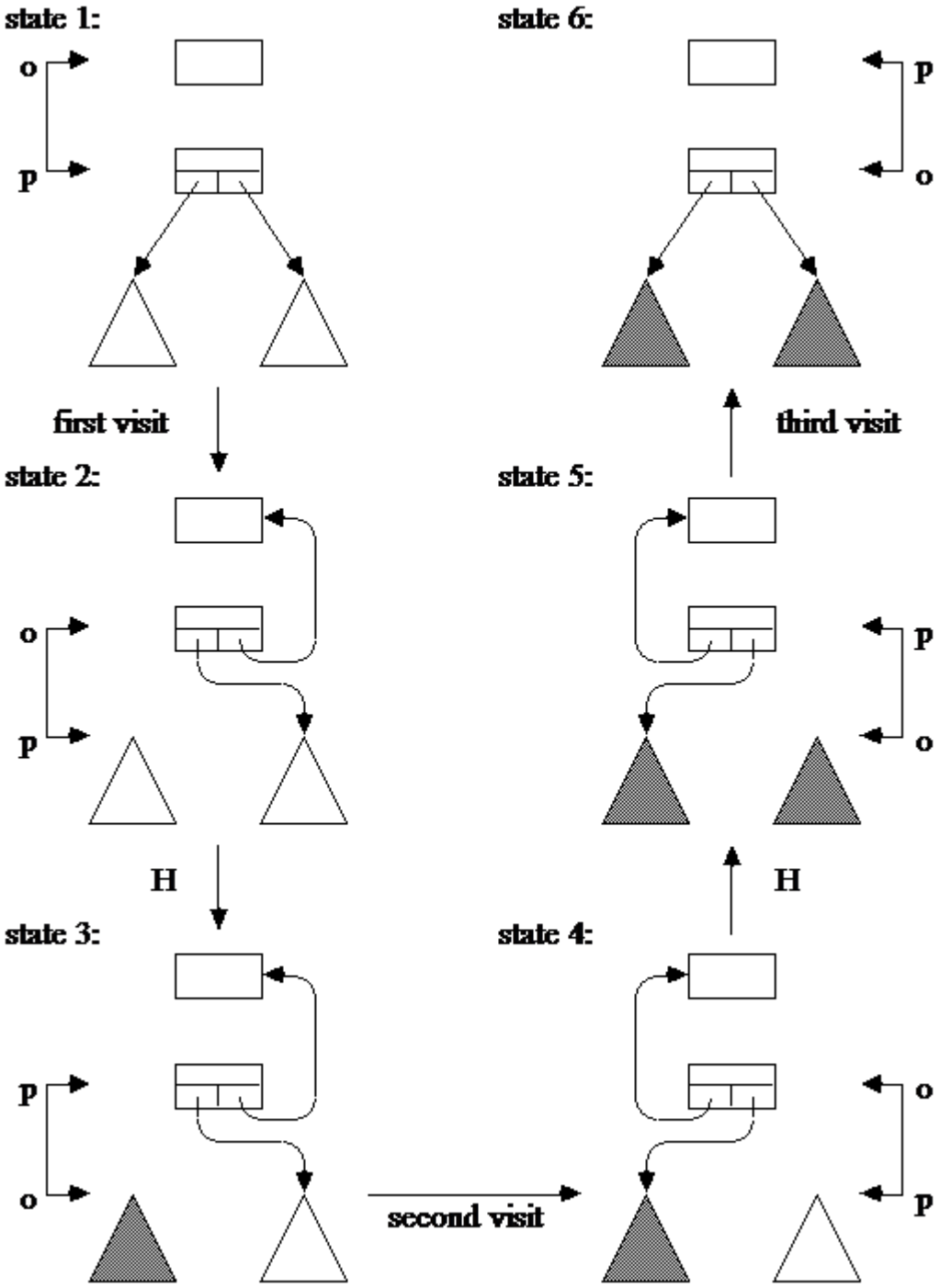


Exhibit 21.12: Trace of procedure TTT, invoking the induction hypothesis

Exercise: binary trees

Consider a binary tree declared as follows:

```
type nptr =  ^node;
  node =  record  L, R: nptr  end;
var  root: nptr;
```

(a) If a node has no left or right subtree, the corresponding pointer has the value 'nil'. Prove that a binary tree with n nodes, n > 0, has n + 1 'nil' pointers.

(b) Write a function nodes(...): integer; that returns the number of nodes, and a function depth(...): integer; that returns the depth of a binary tree. The depth of the root is defined to be 0; the depth of any other node is the depth of its parent increased by 1. The depth of the tree is the maximum depth of its nodes.

Solution

(a) Each node contains two pointers, for a total of $2 \cdot n$ pointers in the tree. There is exactly one pointer that points to each of $n - 1$ nodes, none points to the root. Thus $2 \cdot n - (n - 1) = n + 1$ pointers are 'nil'. This can also be proved by induction on the number of nodes in the tree.

(b)
```
function nodes(p: nptr): integer;
begin
  if  p = nil  then
    return(0)
  else
    return(nodes(p^.L) + nodes(p^.R) + 1)
end;

function depth(p: nptr): integer;
begin
  if  p = nil then return (-1)
      else return(1 + max(depth(p^.L), depth(p^.R)))
end;
```

where 'max' is

```
function max(a, b: integer): integer;
begin  if  a > b  then  return(a)  else  return(b)  end;
```

Exercise: list copying

Effective memory management sometimes makes it desirable or necessary to copy a list. For example, performance may improve drastically if a list spread over several pages can be compressed into a single page. List copying involves a traversal of the original concurrently with a traversal of the copy, as the latter is being built up.

(a) Consider binary trees built from nodes of type 'node' and pointers of type 'nptr'. A tree is accessed through a pointer to the root, which is 'nil' for an empty tree

```
type nptr =  ^ node;
     node =  record  e: elt;  L, R: nptr  end;
```

Write a recursive

```
function cptree(p: nptr): nptr;
```

to copy a tree given by a pointer p to its root, and return a pointer to the root of the copy.

(b) Consider arbitrary graphs built from nodes of a type similar to the nodes in (a), but they have an additional pointer field cn, intended to point to the copy of a node:

```
type  node = record  e: elt;  L, R: nptr;  cn: nptr  end;
```

A graph is accessed through a pointer to a node called the origin, and we are only concerned with nodes that can be reached from the origin; this access pointer is 'nil' for an empty graph. Write a recursive

```
function cpgraph(p: nptr): nptr;
```

to copy a graph given by a pointer p to its origin, and return a pointer to the origin of the copy. Use the field cn, assuming that its initial value is 'nil' in every node of the original graph; set it to 'nil' in every node of the copy.

Solution

```
(a)    function cptree(p: nptr): nptr;
  var  cp: nptr;
  begin
    if  p = nil  then
      return(nil)
    else  begin
      new(cp);
      cp^.e := p^.e;  cp^.L := cptree(p^.L);  cp^.R := cptree(p^.R);
      return(cp)
    end
  end;

(b)    function cpgraph(p: nptr): nptr;
  var  cp: nptr;
  begin
    if  p = nil  then
      return(nil)
    elsif  p^.cn ≠ nil  then  { node has already been copied }
      return(p^.cn)
    else  begin
      new(cp);  p^.cn := cp;  cp^.cn := nil;
      cp^.e := p^.e;  cp^.L := cpgraph(p^.L);  cp^.R := cpgraph(p^.R);
      return(cp)
    end
  end;
```

Exercise: list copying with constant auxiliary memory

Consider binary trees as in part (a) of the preceding exercise. Memory for the stack implied by the recursion can be saved by writing an iterative tree copying procedure that uses only a constant amount of auxiliary memory. This requires a trick, as any depth-first traversal must be able to back up from the leaves toward the root. In the triple-tree traversal procedure, the return path is temporarily encoded in the tree being traversed. This idea can again be used here, but there is a simpler solution: The return path is temporarily encoded in the R-fields of the copy; the L-fields of certain nodes of the copy point back to the corresponding node in the original. Work out the details of a tree-copying procedure that works with O(1) auxiliary memory.

Exercise: traversing a directed acyclic graph

A directed graph consists of nodes and directed arcs, where each arc leads from one node to another. A directed graph is *acyclic* if the arcs form no cycles. One way to ensure that a graph is acyclic is to label nodes with distinct integers and to draw each arc from a lower number to a higher number. Consider a binary directed acyclic graph, where each node has two pointer fields, L and R, to represent at most two arcs that lead out of that node. An example is shown in Exhibit 21.13.

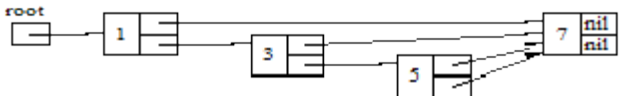


Exhibit 21.13: A rooted acyclic graph.

(a) Write a program to visit every node in a directed acyclic graph reachable from a pointer called 'root'. You are free to execute procedure 'visit' for each node as often as you like.

(b) Write a program similar to (a) where you are required to execute procedure 'visit' exactly once per node. *Hint:* Nodes may need to have additional fields.

Exercise: counting nodes on a square grid

Consider a network superimposed on a square grid: each node is connected to at most four neighbors in the directions east, north, west, south (Exhibit 21.14):

```
type nptr =  ^node;
  node =  record  E, N, W, S: nptr;  status: boolean  end;
var  origin: nptr;
```

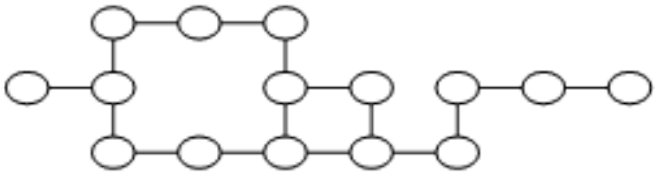


Exhibit 21.14: A graph embedded in a square grid.

A 'nil' pointer indicates the absence of a neighbor. Neighboring nodes are doubly linked: if a pointer in node p points to node q, the reverse pointer of q points to p; (e.g., p^.W = q and q^.E = p). The pointer 'origin' is 'nil' or points to a node. Consider the problem of counting the number of nodes that can be reached from 'origin'. Assume that the status field of all nodes is initially set to false. How do you use this field? Write a function nn(p: nptr): integer; to count the number of nodes.

Solution

```
function nn(p: nptr): integer;
begin
  if  p = nil cor p^.status  then
    return(0)
  else  begin
    p^.status:= true;
    return(1 + nn(p^.E) + nn(p^.N) + nn(p^.W) + nn(p^.S))
  end
end;
```

Exercise: counting nodes in an arbitrary network

We generalize the problem above to arbitrary directed graphs, such as that of Exhibit 21.15, where each node may have any number of neighbors. This graph is represented by a data structure defined by Exhibit 21.16 and the type definitions below. Each node is linked to an arbitrary number of other nodes.

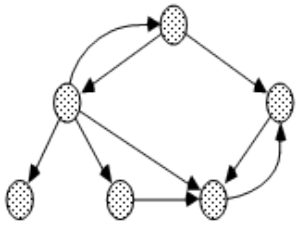


Exhibit 21.15: An arbitrary (cyclic) directed graph.

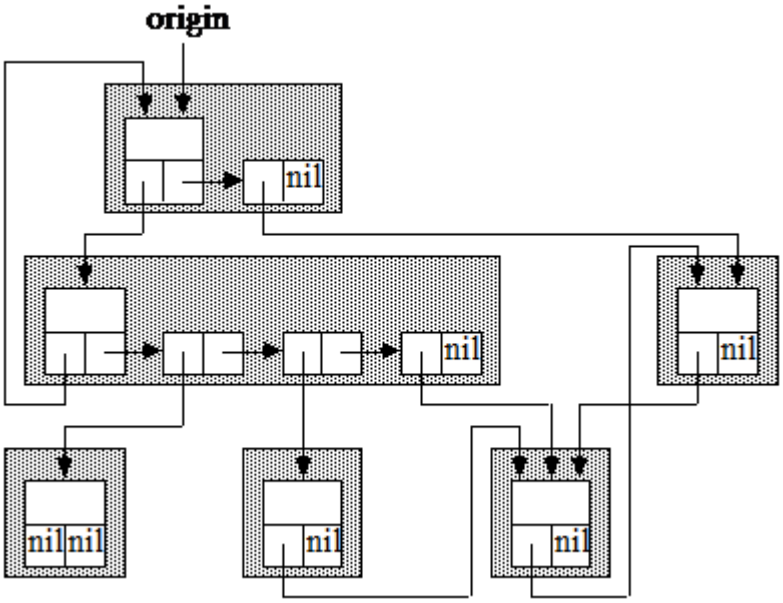


Exhibit 21.16: A possible implementation as a list structure.

```
type nptr = ^node;  cptr = ^cell;
  node = record  status: boolean;  np: nptr;  cp: cptr  end;
  cell = record  np: nptr;  cp: cptr  end;
var  origin: nptr;
```

The pointer 'origin' has the value 'nil' or points to a node. Consider the problem of counting the number n of nodes that can be reached from 'origin'. The status field of all nodes is initially set to false. How do you use it? Write a function nn(p: nptr): integer; that returns n.

Binary search trees

A **binary search tree** is a binary tree T where each node N stores a data element e(N) from a domain X on which a total order ≤ is defined, subject to the following order condition: For every node N in T, all elements in the left subtree L(N) of N are < e(N), and all elements in the right subtree R(N) of N are > e(N). Let $x_1, _2, \ldots, x_n$ be n elements drawn from the domain X.

Definition: A binary search tree for $x_1, x_2, \ldots, x_n$ is a binary tree T with n nodes and a one-to-one mapping between the n given elements and the n nodes, such that

$\forall$ N in T $\forall$ N' $\in$ L(N) $\forall$ N" $\in$ R(N): e(N') < e(N) < e(N")

Exercise

Show that the following statement is equivalent to this order condition: The inorder traversal of the nodes of T coincides with the natural order < of the elements assigned to the nodes.

Remark: The order condition can be relaxed to e(N') ≤ e(N) < e(N") to accommodate multiple occurrences of the same value, with only minor modifications to the statements and algorithms presented in this section. For simplicity's sake we assume that all values in a tree are distinct.

The order condition permits binary search and thus guarantees a worst-case search time O(h) for a tree of height h. Trees that are well balanced (in an intuitive sense; see the next section for a definition), that have not degenerated into linear lists, have a height h = O(log n) and thus support search in logarithmic time.

Basic operations on binary search trees are most easily implemented as recursive procedures. Consider a tree represented as in the preceding section, with empty subtrees denoted by 'nil'. The following function 'find' searches for an element x in a subtree pointed to by p. It returns a pointer to a node containing x if the search is successful, and 'nil' if it is not.

```
function find(x: elt; p: nptr): nptr;
begin
  if  p = nil  then  return(nil)
  elsif  x < p^.e  then  return(find(x, p^.L))
  elsif  x > p^.e  then  return(find(x, p^.R))
  else  { x = p^.e }  return(p)
end;
```

The following procedure 'insert' leaves the tree alone if the element x to be inserted is already stored in the tree. The parameter p initially points to the root of the subtree into which x is to be inserted.

```
procedure insert(x: elt; var p: nptr);
begin
  if  p = nil  then  { new(p);  p^.e := x;  p^.L := nil;  p^.R :=
nil }
  elsif  x < p^.e  then  insert(x, p^.L)
  elsif  x > p^.e  then  insert(x, p^.R)
end;

Initial call:
  insert(x, root);
```

To delete an element x, we first have to find the node N that stores x. If this node is a leaf or semileaf (a node with only one subtree), it is easily deleted; but if it has two subtrees, it is more efficient to leave this node in place and to replace its element x by an element found in a leaf or semileaf node, and delete the latter (Exhibit 21.17). Thus we distinguish three cases:

1. If N has no child, remove N.
2. If N has exactly one child, replace N by this child node.
3. If N has two children, replace x by the largest element y in the left subtree, or by the smallest element z in the right subtree of N. Either of these elements is stored in a node with at most one child, which is removed as in case (1) or (2).

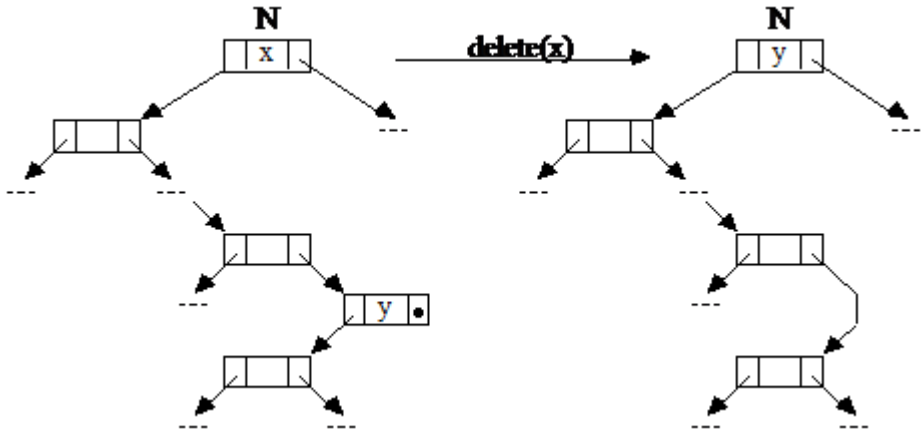


Exhibit 21.17: Element x is deleted while preserving its node N. Node N is filled with a new value y, whose old node is easier to delete.

A sentinel is again the key to an elegant iterative implementation of binary search trees. In a node with no left or right child, the corresponding pointer points to the sentinel. This sentinel is a node that contains no element; its left pointer points to the root and its right pointer points to itself. The root, if it exists, can only be accessed through the left pointer of the sentinel. The empty tree is represented by the sentinel alone (Exhibit 21.18). A typical tree is shown in Exhibit 21.19.

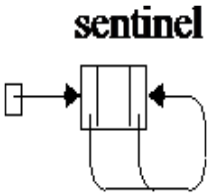


Exhibit 21.18: The empty binary tree is represented by the sentinel which points to itself.

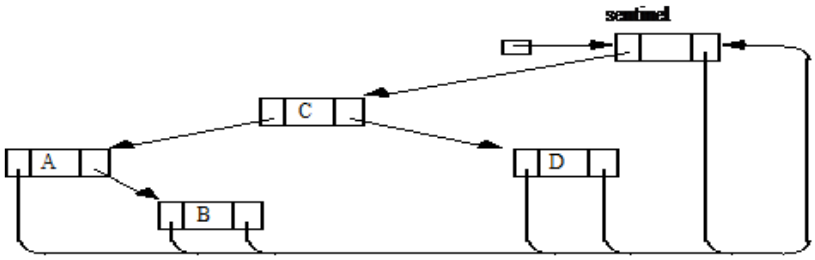


Exhibit 21.19: A binary tree implemented as a list structure with sentinel.

The following implementation of a dictionary as a binary search tree uses a sentinel accessed via the variable d:

```
type nptr =  ^node;
  node =  record  e: elt;  L, R: nptr  end;
  dictionary = nptr;

procedure create(var d: dictionary);
begin  {create sentinel }  new(d);  d^.L := d;  d^.R := d  end;
```

```
function member(d: dictionary; x: elt): boolean;
var  p: nptr;
begin
  d^.e := x;  { initialize element in sentinel }
  p := d^.L;  { point to root, if it exists }
  while  x ≠ p^.e  do
    if  x < p^.e  then  p := p^.L  else  { x > p^.e }  p := p^.R;
  return(p ≠ d)
end;
```

Procedure 'find' searches for x. If found, p points to the node containing x, and q to its parent. If not found, p points to the sentinel and q to the parent-to-be of a new node into which x will be inserted.

```
procedure find(d: dictionary; x: elt; var p, q: nptr);
begin
  d^.e := x;  p := d^.L;  q := d;
  while  x ≠ p^.e  do  begin
    q := p;
    if  x < p^.e  then  p := p^.L  else  { x > p^.e }  p := p^.R
  end
end;

procedure insert(var d: dictionary; x: elt);
var  p, q: nptr;
begin
  find(d, x, p, q);
  if  p = d  then  begin  { x is not yet in the tree }
    new(p);  p^.e := x;  p^.L := d;  p^.R := d;
    if  x ≤ q^.e  then  q^.L := p  else  { x > q^.e }  q^.R := p
  end
end;

procedure delete(var d: dictionary; x: elt);
var  p, q, t: nptr;
begin
  find(d, x, p, q);
  if  p ≠ d  then  { x has been found }
    if  (p^.L ≠ d) and (p^.R ≠ d)  then  begin
      { p has left and right children; find largest element in left
subtree }
      t := p^.L;  q:= p;
      while  t^.R ≠ d  do  { q := t;  t := t^.R };
      if  t^.e < q^.e  then  q^.L := t^.L  else  { t^.e > q^.e }
q^.R := t^.L
      p^.e := t^.e;
    end
    else  begin  { p has at most one child }
      if p^.L ≠ d  then { left child only }  p := p^.L
      elsif  p^.R ≠ d  then { right child only } p := p^.R
      else   { p has no children } p := d;
      if  x ≤ q^.e  then  q^.L := p  else  { x > q^.e }  q^.R := p
    end
end;
```

In the best case of a completely balanced binary search tree for n elements, all leaves are on levels $\lceil \log_2 n \rceil$ or $\lceil \log_2 n \rceil - 1$, and the search tree has the height $\lceil \log_2 n \rceil$. The cost for performing the 'member', 'insert', or 'delete' operation is bounded by the longest path from the root to a leaf (i.e. the height of the tree) and is therefore O(log n).

Without any further provisions, a binary search tree can degenerate into a linear list in the worst case. Then the cost for each of the operations would be O(n).

What is the expected average cost for the search operation in a *randomly generated* binary search tree? "Randomly generated" means that each permutation of the n elements to be stored in the binary search tree has the same probability of being chosen as the input sequence. Furthermore, we assume that the tree is generated by insertions only. Therefore, each of the n elements is equally likely to be chosen as the root of the tree. Let p_n be the expected path length of a randomly generated binary search tree storing n elements. Then

$$p_n = \frac{1}{n}\sum_{k=1}^{n}(p_{k-1} + p_{n-k}) + (n-1) = \frac{2}{n}\sum_{k=0}^{n-1} p_k + (n-1).$$

As shown in chapter 16 in the section “Recurrence relations”, this recurrence relation has the solution

$$p_n = (\ln 4)\cdot n \cdot \log_2 n + g(n) \quad \text{with} \quad g(n) \in O(n).$$

Since the average search time in randomly generated binary search trees, measured in terms of the number of nodes visited, is p_n / n and $\ln 4 \approx 1.386$, it follows that the cost is O(log n) and therefore only about 40 per cent higher than in the case of completely balanced binary search trees.

Balanced trees: general definition

If insertions and deletions occurred at random, and the assumption of the preceding section was realistic, we could let search trees grow and shrink as they please, incurring a modest increase of 40 per cent in search time over completely balanced trees. But real data are not random: they are typically clustered, and long runs of monotonically increasing or decreasing elements occur, often as the result of a previous processing step. Unfortunately, such deviation from randomness degrades the performance of search trees.

To prevent search trees from degenerating into linear lists, we can monitor their shape and restructure them into a more balanced shape whenever they have become too skewed. Several classes of balanced search trees guarantee that each operation 'member', 'insert', and 'delete' can be performed in time O(log n) in the worst case. Since the work to be done depends directly on the height of the tree, such a class B of search trees must satisfy the following two conditions (h_T is the height of a tree T, n_T is the number of nodes in T):

Balance condition: $\exists c > 0 \ \forall T \ \forall B: \ h_T \le c \cdot \log_2 n_T$

Rebalancing condition: If an 'insert' or 'delete' operation, performed on a tree $T \in B$, yields a tree $T' \notin B$, it must be possible to rebalance T' in time O(log n) to yield a tree $T'' \in B$.

Example: almost complete trees

The class of almost complete binary search trees satisfies the balance condition but not the restructuring condition. In the worst case it takes time O(n) to restructure such a binary search tree (Exhibit 21.20), and if 'insert' and 'delete' are defined to include any rebalancing that may be necessary, these operations cannot be guaranteed to run in time O(log n).

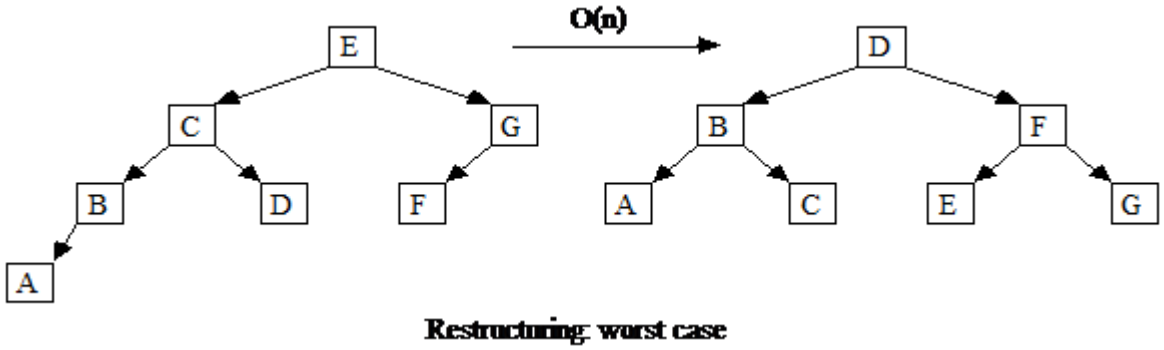


Exhibit 21.20: Restructuring: worst case

In the next two sections we present several classes of balanced trees that meet both conditions: the height-balanced or AVL-trees (G. Adel'son-Vel'skii and E. Landis, 1962) [AL 62] and various multiway trees, such as B-trees [BM 72, Com 79] and their generalization, (a,b)-trees [Meh 84a].

AVL-trees, with their small nodes that hold a single data element, are used primarily for storing data in main memory. Multiway trees, with potentially large nodes that hold many elements, are also useful for organizing data on secondary storage devices, such as disks, that allow direct access to sizable physical data blocks. In this case, a node is typically chosen to fill a physical data block, which is read or written in one access operation.

Height-balanced trees

Definition: A binary tree is *height-balanced* if, for each node, the heights of its two subtrees differ by at most one. Height-balanced search trees are also called *AVL-trees*. Exhibit 21.21 to Exhibit 21.23 show various AVL-trees, and one that is not.

Exhibit 21.21: Examples of height-balanced trees

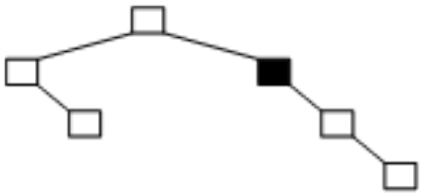


Exhibit 21.22: Example of a tree not height-balanced;the marked node violates the balance condition.

A "most-skewed" AVL-tree T_h is an AVL-tree of height h with a minimal number of nodes. Starting with T_0 and T_1 shown in Exhibit 21.23, T_h is obtained by attaching T_{h-1} and T_{h-2} as subtrees to a new root.

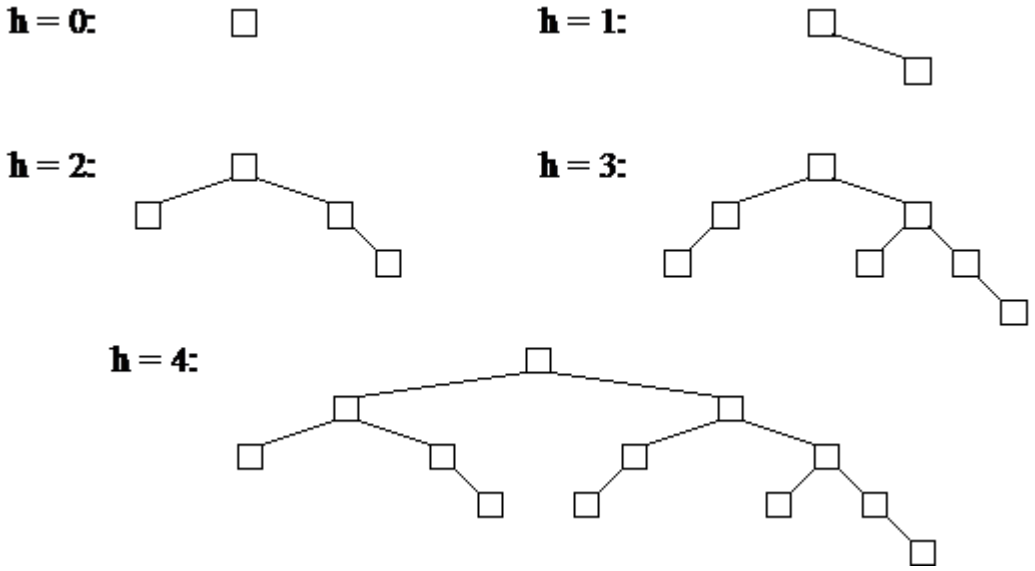


Exhibit 21.23: Most skewed AVL trees of heights h = 0 through h = 4

The number of nodes in a most-skewed AVL-tree of height h is given by the recurrence relation

$n_h = n_{h-1} + n_{h-2} + 1,\ n_0 = 1,\ n_1 = 2.$

In the section on recurrence relations in the chapter entitled “The mathematics of algorithm analysis”, it has been shown that the recurrence relation

$m_h = m_{h-1} + m_{h-2},\ m_0 = 0,\ m_1 = 1$

has the solution

$$m_h = \frac{1}{\sqrt{5}} \cdot \left(\frac{1+\sqrt{5}}{2}\right)^h - \frac{1}{\sqrt{5}} \cdot \left(\frac{1-\sqrt{5}}{2}\right)^h.$$

Since $n_h = m_{h+3} - 1$ we obtain

$$n_h = \left(1 + \frac{2}{\sqrt{5}}\right) \cdot \left(\frac{1+\sqrt{5}}{2}\right)^h + \left(1 - \frac{2}{\sqrt{5}}\right) \cdot \left(\frac{1-\sqrt{5}}{2}\right)^h - 1.$$

Since

$$\left|\frac{1-\sqrt{5}}{2}\right| < 1,$$

it follows that

$$\left(1 - \frac{2}{\sqrt{5}}\right) \cdot \left(\frac{1-\sqrt{5}}{2}\right)^h - 1 \in O(1),$$

and therefore n_h behaves asymptotically as

$$n_h \approx \left(1 + \frac{2}{\sqrt{5}}\right) \cdot \left(\frac{1+\sqrt{5}}{2}\right)^h.$$

Applying the logarithm results in

$$\log_2 n_h \approx \log_2\left(1+\frac{2}{\sqrt{5}}\right)+h\cdot\log_2\left(\frac{1+\sqrt{5}}{2}\right).$$

Therefore, the height of a worst-case AVL-tree with n nodes is about $1.44 \cdot \log_2 n$. Thus the class of AVL-trees satisfies the balance condition, and the 'member' operation can always be performed in time O(log n).

We now show that the class of AVL-trees also satisfies the rebalancing condition. Thus AVL-trees support insertion and deletion in time O(log n). Each node N of an AVL-tree has one of the balance properties / (left-leaning), \ (right-leaning), or – (horizontal), depending on the relative height of its two subtrees.

Two local tree operations, **rotation** and **double rotation**, allow the restructuring of height-balanced trees that have been disturbed by an insertion or deletion. They split a tree into subtrees and rebuild it in a different way. Exhibit 21.24 shows a node, marked black, that got out of balance, and how a local transformation builds an equivalent tree (for the same elements, arranged in order) that is balanced. Each of these transformations has a mirror image that is not shown. The algorithms for insertion and deletion use these rebalancing operations as described below.

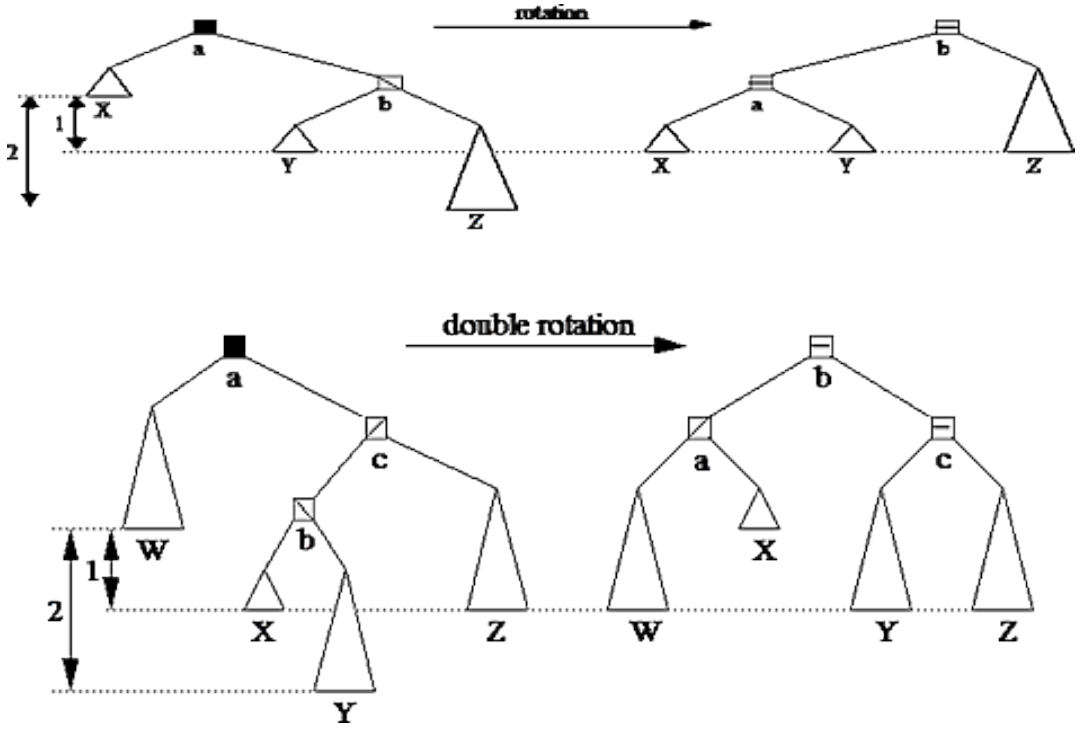


Exhibit 21.24: Two local rebalancing operations

Insertion

A new element is inserted as in the case of a binary search tree. The balance condition of the new node becomes – (horizontal). Starting at the new node, we walk toward the root of the tree, passing along the message that the height of the subtree rooted at the current node has increased by one. At each node encountered along this path, an operation determined by the following rules is performed. These rules depend on the balance condition of the node before the new element was inserted, and on the direction from which the node was entered (i.e. from its left or right child).

Rule I_1: If the current node has balance condition –, change it to / or \ depending on whether we entered from the node's left or from its right child. If the current node is the root, terminate; if not, continue to follow the path upward.

Rule I_2: If the current node has balance condition / or \ and is entered from the subtree that was previously shorter, change the balance condition to—and terminate (the height of the subtree rooted at the current node has not changed).

Rule I_3: If the current node has balance condition / or \ and is entered from the subtree that was previously taller, the balance condition of the current node is violated and gets restored as follows:

(a) If the last two steps were in the same direction (both from left children, or both from right children), an appropriate rotation restores all balances and the procedure terminates.

(b) If the last two steps were in opposite directions (one from a left child, the other from a right child), an appropriate double rotation restores all balances and the procedure terminates.

The initial insertion travels along a path from the root to a leaf, and the rebalancing process travels back up along the same path. Thus the cost of an insertion in an AVL-tree is O(h), or O(log n) in the worst case. Notice that an insertion calls for at most one rotation or double rotation, as shown in the example in Exhibit 21.25.

Example

Insert 1, 2, 5, 3, 4, 6, 7 into an initially empty AVL-tree (Exhibit 21.25). The balance condition of a node is shown below it. Boldfaced nodes violate the balance condition.

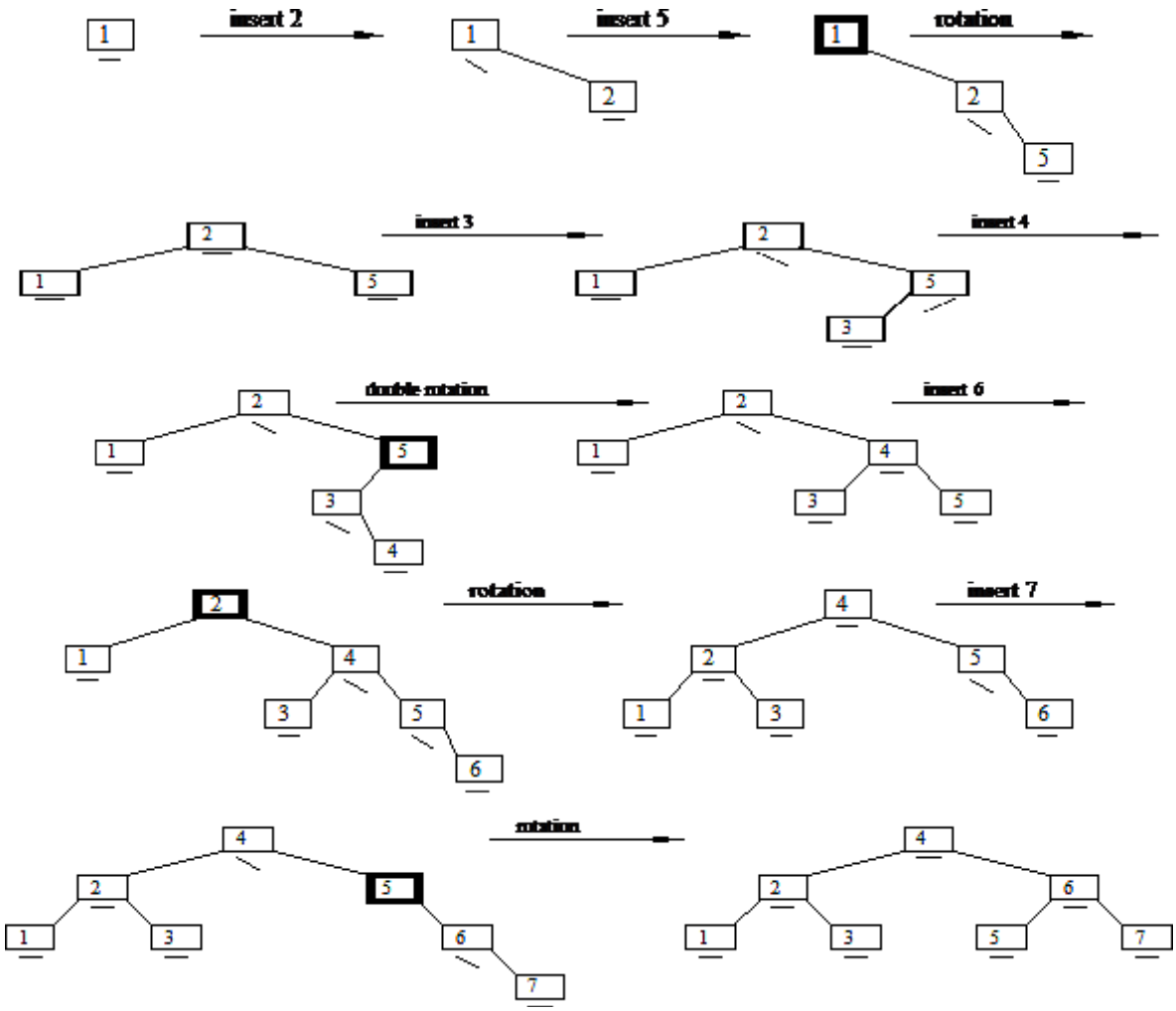


Exhibit 21.25: Trace of consecutive insertions and the rebalancings they trigger

Deletion

An element is deleted as in the case of a binary search tree. Starting at the parent of the deleted node, walk towards the root, passing along the message that the height of the subtree rooted at the current node has decreased by one. At each node encountered, perform an operation according to the following rules. These rules depend on the balance condition of the node before the deletion and on the direction from which the current node and its child were entered.

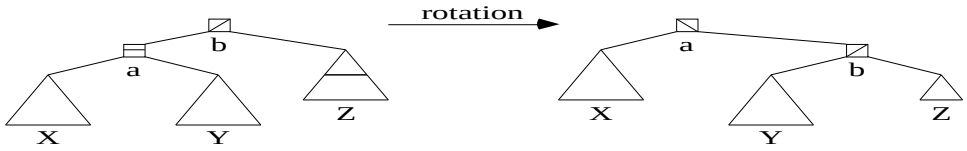
Rule D₁: If the current node has balance condition –, change it to \ or / depending on whether we entered from the node's left or from its right child, and terminate (the height of the subtree rooted at the current node has not changed).

Rule D₂: If the current node has balance condition / or \ and is entered from the subtree that was previously taller, change the balance condition to – and continue upward, passing along the message that the subtree rooted at the current node has been shortened.

Rule D₃: If the current node has balance condition / or \ and is entered from the subtree that was previously shorter, the balance condition is violated at the current node. We distinguish three subcases according to the

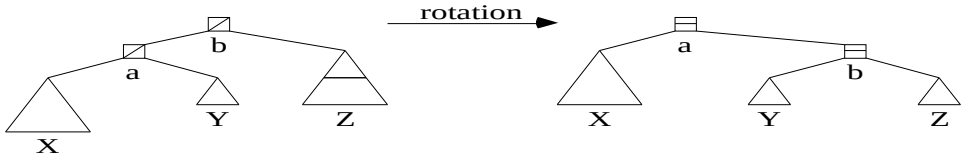
balance condition of the other child of the current node (consider also the mirror images of the following illustrations):

(a)



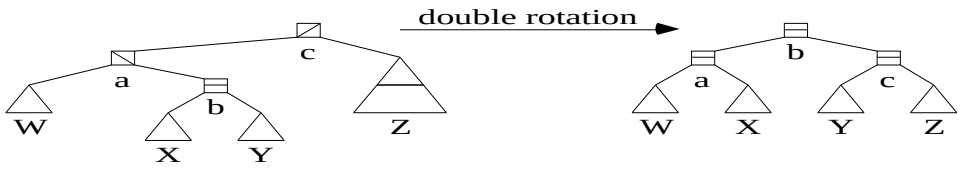
An appropriate rotation restores the balance of the current node without changing the height of the subtree rooted at this node. Terminate.

(b)



A rotation restores the balance of the current node. Continue upward, passing along the message that the subtree rooted at the current node has been shortened.

(c)



A double rotation restores the balance of the current node. Continue upward, passing along the message that the subtree rooted at the current node has been shortened. Similar transformations apply if either X or Y, but not both, are one level shorter than shown in this figure. If so, the balance conditions of some nodes differ from those shown, but this has no influence on the total height of the subtree. In contrast to insertion, deletion may require more than one rotation or double rotation to restore all balances. Since the cost of a rotation or double rotation is constant, the worst-case cost for rebalancing the tree depends only on the height of the tree, and thus the cost of a deletion in an AVL-tree is O(log n) in the worst case.

Multiway trees

Nodes in a multiway tree may have a variable number of children. As we are interested in balanced trees, we add two restrictions. First, we insist that all leaves (the nodes without children) occur at the same depth. Second, we constrain the number of children of all internal nodes by a lower bound a and an upper bound b. Many varieties of multiway trees are known; they differ in details, but all are based on similar ideas. For example, (2,3)-trees are defined by the requirement that all internal nodes have either two or three children. We generalize this concept and discuss (a,b)-trees.

Definition: Consider a domain X on which a total order ≤ is defined. Let a and b be integers with $2 \leq a$ and $2 \cdot a - 1 \leq b$. Let c(N) denote the number of children of node N. An *(a,b)-tree* is an ordered tree with the following properties:

- All leaves are at the same level
- $2 \le c(root) \le b$
- For all internal nodes N except the root, $a \le c(N) \le b$

A node with k children contains $k - 1$ elements $x_1 < x_2 < \ldots < x_{k-1}$ drawn from X; the subtrees corresponding to the k children are denoted by $T_1, T_2, \ldots, T_k$. An (a,b)-tree supports "c(N) search" in the same way that a binary tree supports binary search, thanks to the following order condition:

- $y \le x_i$ for all elements y stored in subtrees $T_1, \ldots, T_i$
- $x_i < z$ for all elements z stored in subtrees $T_{i+1}, \ldots, T_k$

Definition: (a,b)-trees with $b = 2 \cdot a - 1$ are known as *B-trees* [BM 72, Com 79].

The algorithms we discuss operate on internal nodes, shown in white in Exhibit 21.26, and ignore the leaves, shown in black. For the purpose of understanding search and update algorithms, leaves can be considered fictitious entities used only for counting. In practice, however, things are different. The internal nodes merely constitute a directory to a file that is stored in the leaves. A leaf is typically a physical storage unit, such as a disk block, that holds all the records whose key values lie between two (adjacent) elements stored in internal nodes.

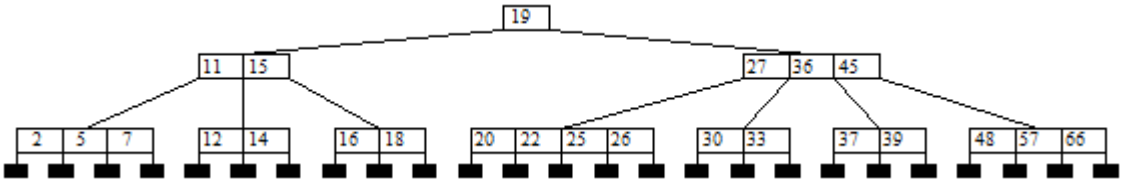


Exhibit 21.26: Example of a (3,5)-tree

The number n of elements stored in the internal nodes of an (a,b)-tree of height h is bounded by

$$2 \cdot a^{h-1} - 1 \le n \le b^h - 1,$$

and thus

$$\log_b (n+1) \le h \le 1 + \log_a \frac{n+1}{2}.$$

this shows that the class of (a,b)-trees satisfies the balance condition h = O(log n). We show that this class also meets the rebalancing condition, namely, that (a,b)-trees support insertion and deletion in time O(log n).

Insertion

Insertion of a new element x begins with a search for x that terminates unsuccessfully at a leaf. Let N be the parent node of this leaf. If N contained fewer than $b - 1$ elements before the insertion, insert x into N and terminate. If N was full, we imagine b elements temporarily squeezed into the overflowing node N. Let m be the median of these b elements, and use m to split N into two: a left node N_L populated by the $(b - 1) / 2$ elements smaller than m, and a right node N_R populated by the $(b - 1) / 2$ elements larger than m. The condition $2 \cdot a - 1 \le b$ ensures that $\lfloor (b - 1) / 2 \rfloor \ge a - 1$, in other words, that each of the two new nodes contains at least $a - 1$ elements.

The median element m is pushed upward into the parent node, where it serves as a separator between the two new nodes N_L and N_R that now take the place formerly inhabited by N. Thus the problem of insertion into a node at a given level is replaced by the same problem one level higher in the tree. The new separator element may be absorbed in a nonfull parent, but if the parent overflows, the splitting process described is repeated recursively. At

worst, the splitting process propagates to the root of the tree, where a new root that contains only the median element is created. (a,b)-trees grow at the root, and this is the reason for allowing the root to have as few as two children.

Deletion

Deletion of an element x begins by searching for it. As in the case of binary search trees, deletion is easiest at the bottom of the tree, at a node of maximal depth whose children are leaves. If x is found at a higher level of the tree, in a node that has internal nodes as children, x is the separator between two subtrees T_L and T_R. We replace x by another element z, either the largest element in T_L or the smallest element in T_R, both of which are stored in a node at the bottom of the tree. After this exchange, the problem is reduced to deleting an element z from a node N at the deepest level.

If deletion (of x or z) leaves N with at least $a - 1$ elements, we are done. If not, we try to restore N's occupancy condition by stealing an element from an adjacent sibling node M. If there is no sibling M that can spare an element, that is, if M is minimally occupied, M and N are merged into a single node L. L contains the $a - 2$ elements of N, the $a - 1$ elements of M, and the separator between M and N which was stored in their parent node, for a total of $2 \cdot (a - 1) \leq b - 1$ elements. Since the parent (of the old nodes M and N, and of the new node L) lost an element in this merger, the parent may underflow. As in the case of insertion, this underflow can propagate to the root and may cause its deletion. Thus (a,b)-trees grow and shrink at the root.

Both insertion and deletion work along a single path from the root down to a leaf and (possibly) back up. Thus their time is bounded by O(h), or equivalently, by O(log n): (a,b)-trees can be rebalanced in logarithmic time.

Amortized cost. The performance of (a,b)-trees is better than the worst-case analysis above suggests. It can be shown that the total cost of *any sequence of s insertions and deletions* into an initially empty (a,b)-tree is linear in the length s of the sequence: whereas the worst-case cost of a single operation is O(log n), the *amortized cost* per operation is O(1) [Meh 84a]. Amortized cost is a complexity measure that involves both an average and a worst-case consideration. The average is taken over all operations in a sequence; the worst case is taken over all sequences. Although any one operation may take time O(log n), we are guaranteed that the total of all s operations in any sequence of length s can be done in time O(s), as if each single operation were done in time O(1).

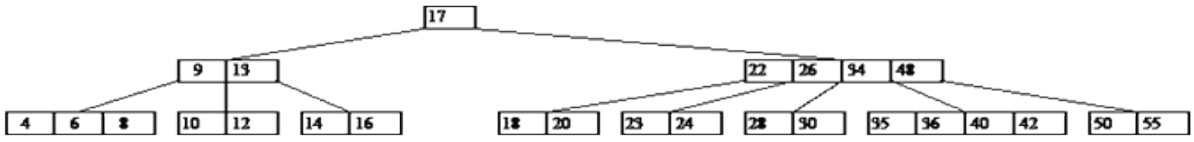


Exhibit 21.27: A slightly skewed (3,5)-tree.

Exercise: insertion and deletion in a (3,5)-tree

Starting with the (3,5)-tree shown in Exhibit 21.27, perform the sequence of operations: insert 38, delete 10, delete 12, delete 50. Draw the tree after each operation.

Solution

Inserting 38 causes a leaf and its parent to split (Exhibit 21.28). Deleting 10 causes underflow, remedied by borrowing an element from the left sibling (Exhibit 21.29). Deleting 12 causes underflow in both a leaf and its parent, remedied by merging (Exhibit 21.30). Deleting 50 causes merging at the leaf level and borrowing at the parent level (Exhibit 21.31).

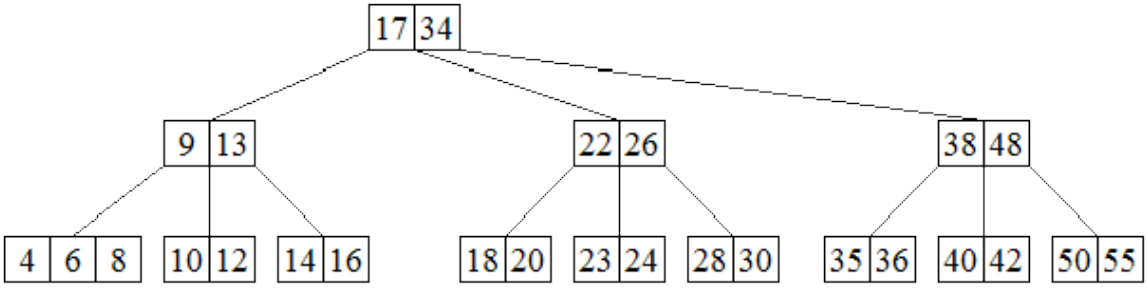


Exhibit 21.28: Node splits propagate towards the root

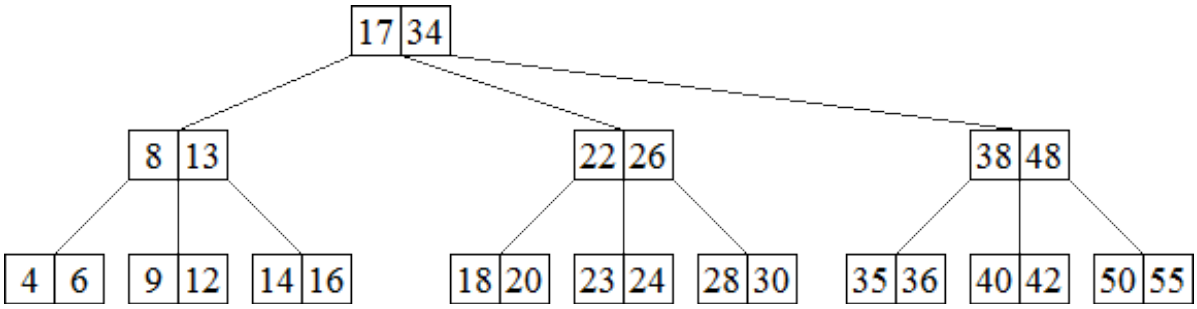


Exhibit 21.29: A deletion is absorbed by borrowing

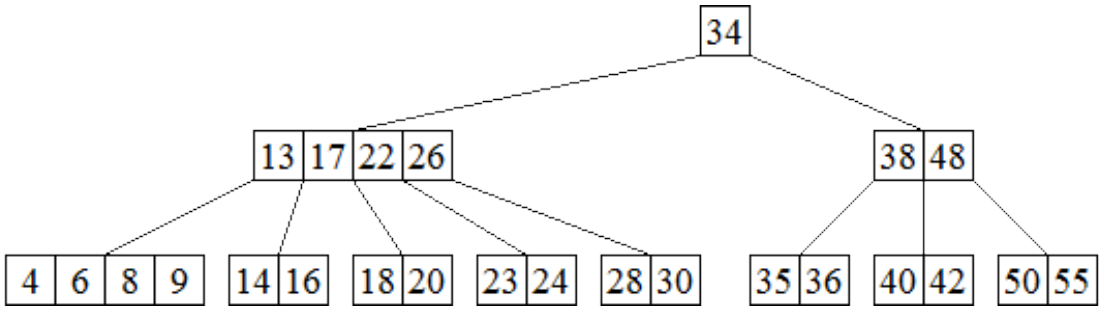


Exhibit 21.30: Another deletion propagates node merges towards the root

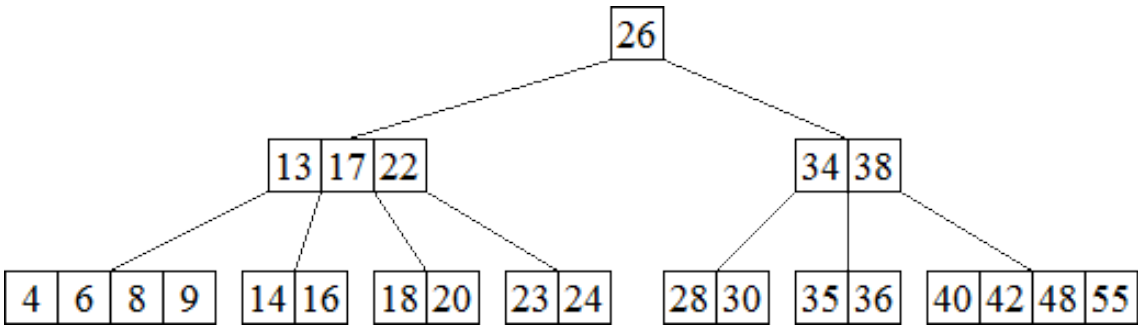


Exhibit 21.31: Node merges and borrowing combined

*(2,3)-tree*s are the special case a = 2, b = 3: each node has two or three children. Exhibit 21.32 omits the leaves. Starting with the tree in state 1 we insert the value 9: the rightmost node at the bottom level overflows and splits, the median 8 moves up into the parent. The parent also overflows, and the median 6 generates a new root (state 2). The deletion of 1 is absorbed without any rebalancing (state 3). The deletion of 2 causes a node to underflow, remedied by stealing an element from a sibling: 2 is replaced by 3 and 3 is replaced by 4 (state 4). The deletion of 3

triggers the merger of the nodes assigned to 3 and 5; this causes an underflow in their parent, which in turn propagates to the root and results in a tree of reduced height (state 5).

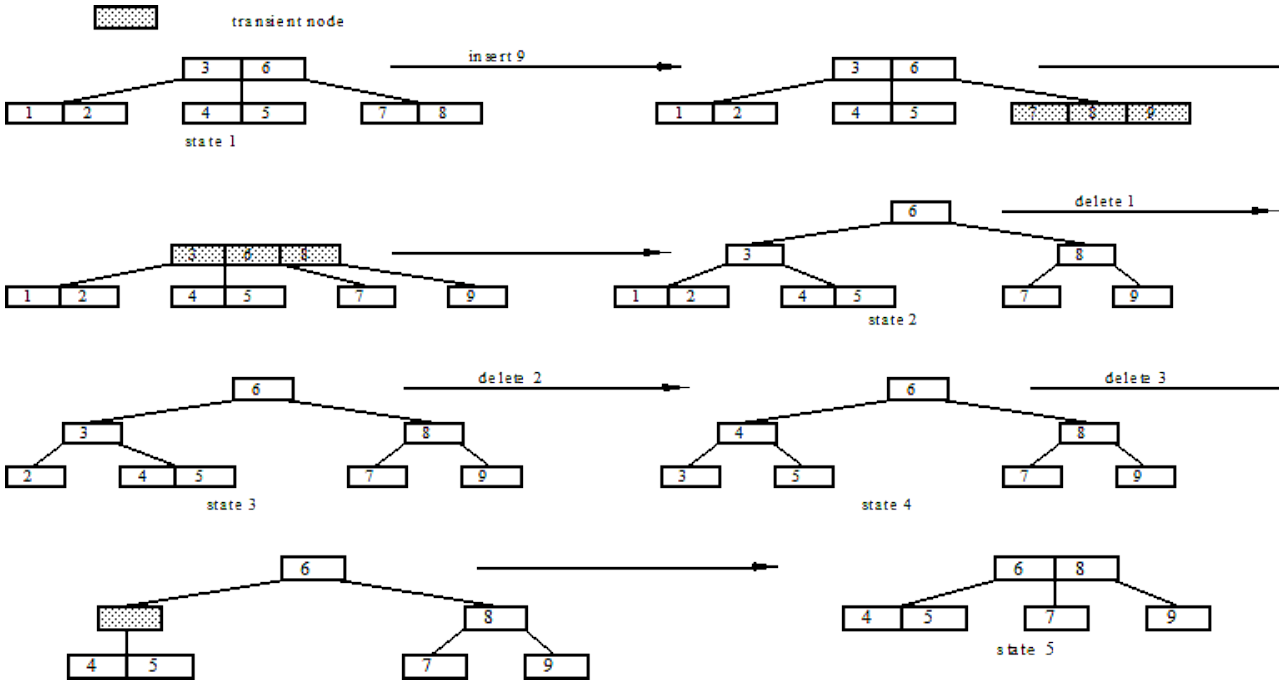


Exhibit 21.32: Tracing insertions and deletions in a (2,3)-tree

As mentioned earlier, multiway trees are particularly useful for managing data on a disk. If each node is allocated to its own disk block, searching for a record triggers as many disk accesses as there are levels in the tree. The depth of the tree is minimized if the maximal fan-out b is maximized. We can pack more elements into a node by shrinking their size. As the records to be stored are normally much larger than their identifying keys, we store keys only in the internal nodes and store entire records in the leaves (which we had considered to be empty until now). Thus the internal nodes serve as an index that assigns to a key value the path to the corresponding leaf.

Exercises and programming projects

1. Design and implement a list structure for storing a sparse matrix. Your implementation should provide procedures for inserting, deleting, changing, and reading matrix elements.
2. Implement a fifo queue by a circular list using only one external pointer f and a sentinel. f always points to the sentinel and provides access to the head and tail of the queue.
3. Implement a double-ended queue (deque) by a doubly linked list.
4. *Binary search trees and sorting* A binary search tree given by the following declarations is used to manage a set of integers:

```
type nptr = ^node
  node = record  L, R: nptr;  x: integer  end;
var  root: nptr;
```

The empty tree is represented as root = nil.

(a) Draw the result of inserting the sequence 6, 15, 4, 2, 7, 12, 5, 18 into the empty tree.

(b) Write a procedure smallest(var x: integer); which returns the smallest number stored in the tree, and a procedure remove smallest; which deletes it. If the tree is empty both procedures should call a procedure message('tree is empty');

(c) Write a procedure sort; that sorts the numbers stored in var a: array[1 .. n] of integer; by inserting the numbers into a binary search tree, then writing them back to the array in sorted order as it traverses the tree.

(d) Analyze the asymptotic time complexity of 'sort' in a typical and in the worst case.

(e) Does this approach lead to a sorting algorithm of time complexity $\Theta(n \cdot \log n)$

5. Extend the implementation of a dictionary as a binary search tree in the “Binary search trees” section to support the operations 'succ' and 'pred' as defined in chapter 19 in the section “Dictionary”.
6. *Insertion and deletion in AVL-trees:* Starting with an empty AVL-tree, insert 1, 2, 5, 6, 7, 8, 9, 3, 4, in this order. Draw the AVL-tree after each insertion. Now delete all elements in the opposite order of insertion (i.e. in last-in-first-out order). Does the AVL-tree go through the same states as during insertion but in reverse order?
7. Implement an AVL-tree supporting the dictionary operations 'insert', 'delete', 'member', 'pred', and 'succ'.
8. Explain how to find the smallest element in an (a,b)-tree and how to find the predecessor of a given element in an (a,b)-tree.
9. Implement a dictionary as a B-tree.