VI

Data Structures in Languages and Libraries

40

Functional Data Structures

Chris Okasaki
United States Military Academy

40.1 Introduction

A *functional data structure* is a data structure that is suitable for implementation in a functional programming language, or for coding in an ordinary language like C or Java using a functional style. Functional data structures are closely related to *persistent data structures* and *immutable data structures*—in fact, the three terms are often used interchangeably. However, there are subtle differences.

- The term *persistent data structures* refers to the general class of data structures in which an update does not destroy the previous version of the data structure, but rather creates a new version that co-exists with the previous version. See Chapter 31 for more details about persistent data structures.
- The term *immutable data structures* emphasizes a particular implementation technique for achieving persistence, in which memory devoted to a particular version of the data structure, once initialized, is never altered.
- The term *functional data structures* emphasizes the language or coding style in which persistent data structures are implemented. Functional data structures are always immutable, except in a technical sense discussed in Section 40.4.

In this chapter, we will discuss the main issues surrounding the implementation of data structures in functional languages, and illustrate these issues with several extended examples. We will also show how to adapt functional data structures to a mainstream language such as Java, for use when a persistent data structure is required. Readers wishing more details about functional data structures should consult Okasaki [24].

40.1.1 Data Structures in Functional Languages

Functional programming languages differ in several important ways from ordinary programming languages like C or Java, and these differences can sometimes have a large effect on how data structures are implemented. The main differences (at least from the perspective of data structures) are *immutability*, *recursion*, *garbage collection*, and *pattern matching*.

Immutability

In functional languages, variables and records cannot be modified, or *mutated*, once they have been created.* Many textbook data structures depend critically on the ability to mutate variables and records via assignments. Such data structures can be difficult to adapt to a functional setting.

Recursion

Functional languages frequently do not support looping constructs, such as for-loops or while-loops, because such loops depend on being able to mutate the loop control variable. Functional programmers use recursion instead.†

Garbage Collection

Functional languages almost always depend on automatic garbage collection. Because objects are immutable in functional languages, they are shared much more widely than in ordinary languages, which makes the task of deciding when to deallocate an object very complicated. In functional languages, programmers ignore deallocation issues and allow the garbage collector to deallocate objects when it is safe to do so.

Pattern Matching

Pattern matching is a method of defining functions by cases that are essentially textual analogues of the kinds of pictures data-structure designers often draw. Pattern matching is not supported by all functional languages, but, when available, it allows many data structures to be coded very concisely and elegantly.

40.1.2 Functional Data Structures in Mainstream Languages

Even if you are programming in a mainstream language, such as C or Java, you may find it convenient to use a functional data structure. Functional data structures offer three main advantages in such a setting: *fewer bugs*, *increased sharing*, and *decreased synchronization*.

Fewer Bugs

A very common kind of bug arises when you observe a data structure in a certain state and shortly thereafter perform some action that assumes the data structure is still in that same state. Frequently, however, something has happened in the meantime to alter the state of the data structure, so that the action is no longer valid. Because functional data structures are immutable, such alterations simply cannot occur. If someone tries to change the data structure, they may produce a new version of it, but they will in no way effect the version that you are using.

*Actually, many functional languages do provide mechanisms for mutation, but their use is discouraged.

†Or higher-order functions, but we will not discuss higher-order functions further because they have relatively little effect on the implementation of data structures.

Increased Sharing

Precisely to avoid the kinds of bugs described above, programmers in mainstream languages are careful to limit access to their internal data structures. When sharing is unavoidable, programmers will often clone their data structures and share the clones rather than granting access to their own internal copies. In contrast, functional data structures can be shared safely without cloning. (Actually, functional data structures typically perform a substantial amount of cloning internally, but this cloning is of individual nodes rather than entire data structures.)

Decreased Synchronization

Again, precisely to avoid the kinds of bugs described above, programmers in concurrent settings are careful to synchronize access to their data structures, so that only a single thread can access the data structure at a time. On the other hand, because functional data structures are immutable, they can often be used with little or no synchronization. Even simultaneous writes are not a problem, because each writer thread will get a new version of the data structure that reflects only its own updates. (This assumes, of course, an application where participants do not necessarily want to see changes made by all other participants.)

40.2 Stacks: A Simple Example

Stacks (see Chapter 2) represented as singly-linked lists are perhaps the simplest of all data structures to make persistent. We begin by describing functional stacks supporting four main primitives:

- `empty`: a constant representing the empty stack.
- `push(`x`,`s`)`: push the element x onto the stack s and return the new stack.
- `top(`s`)`: return the top element of s.
- `pop(`s`)`: remove the top element of s and return the new stack.

We can see right away several differences between this interface and the interface for ordinary stacks. First, for ordinary stacks, `push` and `pop` would implicitly change the existing stack s rather than returning a new stack. However, the hallmark of functional data structures is that update operations return a new version of the data structure rather than modifying the old version. Second, ordinary stacks would support a function or constructor to *create* a fresh, new stack, rather than offering a single constant to represent all empty stacks. This highlights the increased sharing possible with functional data structures. Because pushing an element onto the empty stack will not change the empty stack, different parts of the program can use the same empty stack without interfering with each other.

Figure 40.1 shows an implementation of stacks in Haskell [28], a popular functional programming language. Figure 40.2 shows a similar implementation in Java. Like all code fragments in this chapter, these implementations are intended only to illustrate the relevant concepts and are not intended to be industrial strength. In particular, all error handling has been omitted. For example, the `top` and `pop` operations should check whether the stack is empty. Furthermore, programming conventions in Haskell and Java have been ignored where they would make the code harder for non-fluent readers to understand. For example, the Haskell code makes no use of currying, and the Java code makes no attempt to be object-oriented.

The Haskell code illustrates a simple use of pattern matching. The declaration

```
data Stack = Empty | Push(Element,Stack)

empty = Empty
push(x,s) = Push(x,s)
top(Push(x,s)) = x
pop(Push(x,s)) = s
```

FIGURE 40.1: Stacks in Haskell.

```
public class Stack {
  public static final Stack empty = null;
  public static Stack push(Element x,Stack s) { return new Stack(x,s); }
  public static Element top(Stack s) { return s.element; }
  public static Stack pop(Stack s) { return s.next; }

  private Element element;
  private Stack next;
  private Stack(Element element,Stack next) {
    this.element = element;
    this.next = next;
  }
}
```

FIGURE 40.2: Stacks in Java.

```
data Stack = Empty | Push(Element,Stack)
```

states that stacks have two possible shapes, `Empty` or `Push`, and that a stack with the `Push` shape has two fields, an element and another stack. The tags `Empty` and `Push` are called *constructors*. Later function declarations can match against these constructors. For example, the declaration

```
top(Push(x,s)) = x
```

says that when `top` is called on a stack with the `Push` shape, it returns the contents of the first field. If desired, more clauses can be added to deal with other shapes. For example, a second clause could be added to the definition of `top` to handle the error case:

```
top(Push(x,s)) = x
top(Empty) = ...signal an error...
```

How do these implementations achieve persistence? First, consider the `push` operation. Calling `push` creates a new node containing the new element and a pointer to the old top of stack, but it in no way alters the old stack. For example, if the old stack s contains the numbers 3, 2, 1 and we `push` 4, then the new stack s' contains the numbers 4, 3, 2, 1. Figure 40.3 illustrates the relationship between s and s'. Notice how the nodes containing 3, 2, and 1 are shared between both stacks. Because of this sharing, it is crucial that the nodes are immutable. Consider what would happen if nodes could be changed. If we were to change the 3 to a 5, for example, perhaps by calling an operation to update the top element of s, that change would affect not only s (which would now contain 5, 2, 1) but also s' (which would now contain 4, 5, 2, 1). Such unintended consequences would make sharing impossible.

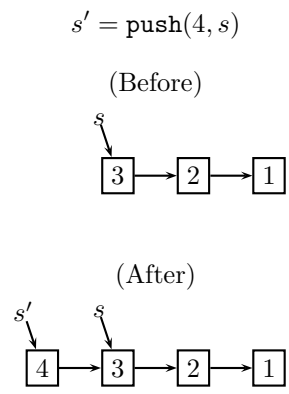


FIGURE 40.3: The `push` operation.

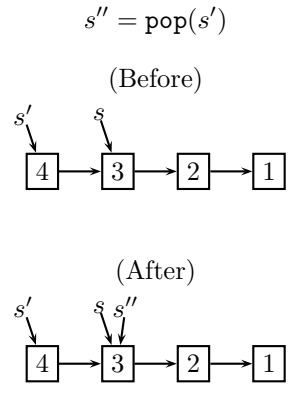


FIGURE 40.4: The `pop` operation.

Next, consider the `pop` operation, which simply returns the *next* pointer of the current node without changing the current node in any way. For example, Figure 40.4 illustrates the result of popping the stack s' to get the stack s'' (which shares its entire representation with the original stack s). Notice that, after popping s', the node containing 4 may or may not be garbage. It depends on whether any part of the program is still using the s' stack. If not, then automatic garbage collection will eventually deallocate that node.

```
data Tree = Empty | Node(Tree,Int,Tree)

empty = Empty
insert(x,Empty) = Node(Empty,x,Empty)
insert(x,Node(t1,y,t2)) = if x < y then Node(insert(x,t1),y,t2)
                          else if x > y then Node(t1,y,insert(x,t2))
                          else Node(t1,y,t2)
search(x,Empty) = False
search(x,Node(t1,y,t2)) = if x < y then search(x,t1)
                          else if x > y then search(x,t2)
                          else True
```

FIGURE 40.5: Binary search trees in Haskell.

40.3 Binary Search Trees: Path Copying

Stacks are unusual in that there is never a need to update an existing node. However, for most data structures, there is such a need. For example, consider inserting an element into a binary search tree (see Chapter 3). At the very least, we need to update the new node's parent to point to the new node. But how can we do this if nodes are immutable? The solution is a technique called *path copying*. To update an existing node, we copy the node and make the necessary changes in the copy. However, we then have to update the existing node's parent in a similar fashion to point to the copy. In this way, changes propagate all the way from the site of the update to the root, and we end up copying that entire path. That may seem like a lot of copying, but notice that all nodes not on that path are shared between the old and new versions.

To see how path copying works in practice, consider a simple implementation of integer sets as unbalanced binary search trees. Figure 40.5 shows an implementation in Haskell and Figure 40.6 shows the same implementation in Java.

The key to understanding path copying lies in the `insert` operation. Consider the case where the element being inserted is larger than the element at the current node. In the Java implementation, this case executes the code

```
return new Tree(t.left,t.element,insert(x,t.right));
```

First, `insert` calls itself recursively on the right subtree, returning a pointer to the new right subtree. It then allocates a new tree node, copying the `left` and `element` fields from the old node, and installing the new pointer in the `right` field. Finally, it returns a pointer to the new node. This process continues until it terminates at the root. Figure 40.7 illustrates a sample insertion. Notice how the parts of the tree not on the path from the root to the site of the update are shared between the old and new trees.

This functional implementation of binary search trees has exactly the same time complexity as an ordinary non-persistent implementation. The running time of `insert` is still proportional to the length of the search path. Of course, the functional implementation allocates more space, but even that issue is not clear cut. If the old tree is no longer needed, then the just-copied nodes can immediately be garbage collected, leaving a net space increase of one node—exactly the same space required by a non-persistent implementation. On the other hand, if the old tree is still needed, then the just-copied nodes cannot be garbage collected, but in that case we are actively taking advantage of functionality not supported by ordinary binary search trees.

```
public class Tree {
  public static final Tree empty = null;
  public static Tree insert(int x,Tree t) {
    if (t == null) return new Tree(null,x,null);
    else if (x < t.element)
      return new Tree(insert(x,t.left),t.element,t.right);
    else if (x > t.element)
      return new Tree(t.left,t.element,insert(x,t.right));
    else return t;
  }
  public static boolean search(int x,Tree t) {
    if (t == null) return false;
    else if (x < t.element) return search(x,t.left);
    else if (x > t.element) return search(x,t.right);
    else return true;
  }

  private int element;
  private Tree left,right;
  private Tree(Tree left,int element,Tree right) {
    this.left = left;
    this.element = element;
    this.right = right;
  }
}
```

FIGURE 40.6: Binary search trees in Java.

Of course, the binary search trees described above suffer from the same limitations as ordinary unbalanced binary search trees, namely a linear time complexity in the worst case. Whether the implementation is functional or not as no effect in this regard. However, we can easily apply the ideas of path copying to most kinds of *balanced* binary search trees (see Chapter 10), such as AVL trees [17, 29], red-black trees [25], 2-3 trees [30], and weight-balanced trees [2]. Such a functional implementation retains the logarithmic time complexity of the underlying design, but makes it persistent.

Path copying is sufficient for implementing many tree-based data structures besides binary search trees, including binomial queues [7, 15] (Chapter 7), leftist heaps [18, 24] (Chapter 5), Patricia tries [26] (Chapter 28), and many others.

40.4 Skew Heaps: Amortization and Lazy Evaluation

Next, we turn to priority queues, or *heaps*, supporting the following primitives:

- `empty`: a constant representing the empty heap.
- `insert(`x`,`h`)`: insert the element x into the heap h and return the new heap.
- `findMin(`h`)`: return the minimum element of h.
- `deleteMin(`h`)`: delete the minimum element of h and return the new heap.
- `merge(`h_1`,`h_2`)`: combine the heaps h_1 and h_2 into a single heap and return the new heap.

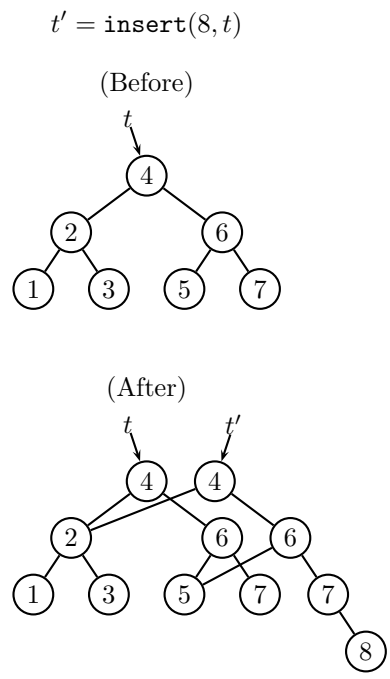


FIGURE 40.7: The `insert` operation.

Many of the standard heap data structures can easily be adapted to a functional setting, including binomial queues [7, 15] and leftist heaps [18, 24]. In this section, we describe a simple, yet interesting, design known as *skew heaps* [32]. (Non-persistent skew heaps are described in detail in Chapter 6.)

A skew heap is a heap-ordered binary tree. Each node contains a single element, and the nodes are ordered such that the element at each node is no larger than the elements at the node's children. Because of this ordering, the minimum element in a tree is always at the root. Therefore, the `findMin` operation simply returns the element at the root. The `insert` and `deleteMin` operations are defined in terms of `merge`: `insert` creates a new node and merges it with the existing heap, and `deleteMin` discards the root and merges its children.

The interesting operation is `merge`. Assuming both heaps are non-empty, `merge` compares their roots. The smaller root (that is, the root with the smaller element) becomes the new overall root and its children are swapped. Then the larger root is merged with the new left child of the smaller root (which used to be the right child). The net effect of a `merge` is to interleave the rightmost paths of the two trees in sorted order, swapping the children of nodes along the way. This process is illustrated in Figure 40.8. Notice how the nodes on the rightmost paths of the arguments end up on the leftmost path of the result. A Haskell implementation of skew heaps incorporating path copying is shown in Figure 40.9. A naive Java implementation is shown in Figure 40.10.

Skew heaps are not balanced, and individual operations can take linear time in the worst case. For example, Figure 40.11 shows an unbalanced shew heap generated by inserting the elements

5, 6, 4, 6, 3, 6, 2, 6, 1, 6

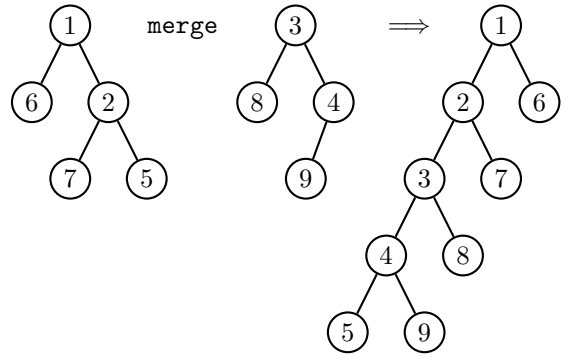


FIGURE 40.8: Merging two skew heaps.

```
data Skew = Empty | Node(Int,Skew,Skew)

empty = Empty
insert(x,s) = merge(Node(x,Empty,Empty),s)
findMin(Node(x,s1,s2)) = x
deleteMin(Node(x,s1,s2)) = merge(s1,s2)

merge(s1,Empty) = s1
merge(Empty,s2) = s2
merge(Node(x,s1,s2),Node(y,t1,t2)) =
  if x < y then Node(x,merge(Node(y,t1,t2),s2),s1)
           else Node(y,merge(Node(x,s1,s2),t2),t1)
```

FIGURE 40.9: Skew heaps in Haskell.

into an initially empty heap. Inserting a new element such as 7 into this unbalanced skew heap would take linear time. However, in spite of the fact that any one operation can be inefficient, the way that children are regularly swapped keeps the operations efficient in the amortized sense—`insert`, `deleteMin`, and `merge` run in logarithmic amortized time [32].

Or, at least, those would be the bounds for non-persistent skew heaps. When we analyze skew heaps in a persistent setting, we receive a nasty shock. Making an amortized data structure such as skew heaps persistent using path copying breaks its amortized bounds! In the case of skew heaps, naively incorporating path copying causes the logarithmic amortized bounds to degrade to the linear worst-case bounds.

Consider, for example, the unbalanced skew heap s in Figure 40.11, for which `insert` takes linear time for large elements. The result of the `insert` is a new skew heap s'. Performing another `insert` on s' would actually be quite efficient, but because these structures are persistent, we are free to ignore s' and perform the next `insert` on the old skew heap s instead. This `insert` again takes linear time. We can continue performing operations on the old skew heap as often as we want. The average cost per operation over a sequence of such operations is linear, which means that the amortized cost per operation is now linear, rather than logarithmic. Simple experiments on the Java implementation from Figure 40.10 confirm this analysis.

```
public class Skew {
  public static final Skew empty = null;
  public static Skew insert(int x,Skew s) { return merge(new Skew(x,null,null),s); }
  public static int findMin(Skew s) { return s.element; }
  public static Skew deleteMin(Skew s) { return merge(s.left,s.right); }

  public static Skew merge(Skew s,Skew t) {
    if (t == null) return s;
    else if (s == null) return t;
    else if (s.element < t.element)
      return new Skew(s.element,merge(t,s.right),s.left);
    else
      return new Skew(t.element,merge(s,t.right),t.left);
  }

  private int element;
  private Skew left,right;
  private Skew(int element, Skew left, Skew right) {
    this.element = element;
    this.left = left;
    this.right = right;
  }
}
```

FIGURE 40.10: First attempt at skew heaps in Java

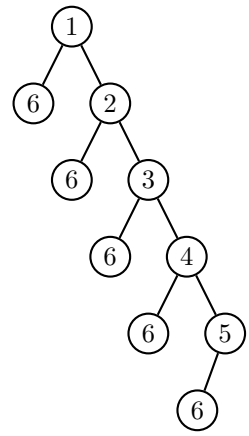


FIGURE 40.11: An unbalanced skew heap.

However, if we repeat those experiments on the Haskell implementation from Figure 40.9, we do not observe linear behavior. Instead, the operations appear to retain their logarithmic amortized bounds, even under persistent usage. This pleasant result is a consequence of a fortuitous interaction between path copying and a property of the Haskell language called *lazy evaluation*. (Many other functional programming languages also support lazy evaluation).

Under lazy evaluation, operations such as `merge` are not actually executed until their results are needed. Instead, a new kind of node that we might call a *pending merge* is automatically created. The pending merge lays dormant until some other operation such as `findMin` needs to know the result. Then and only then is the pending merge executed. The node representing the pending merge is overwritten with the result so that it cannot be executed twice.

Although Java does not directly support lazy evaluation, it is easy to simulate, as shown in Figure 40.12. A minor difference between the lazy Java implementation and the Haskell implementation is that the Java implementation avoids creating pending merge nodes when one of the arguments is `null`.

A crucial aspect of lazy evaluation is that a pending computation, once triggered, is executed only far enough to produce the part of the result that is needed. The remaining parts of the computation may be delayed further by creating new pending nodes. In the case of the `merge` operation, this means that when a pending merge is executed, the two roots are compared and the children of the smaller root are swapped as normal, but the recursive `merge` of the larger root with the former right child of the smaller root is *not* performed. Instead, a new pending merge is created and installed as the left child of the new root node. This process is illustrated in Figure 40.13, with the pending merges drawn as diamonds.

Figure 40.14 illustrates the propagation of pending merges through a sequence of operations. First, the initial tree is built via a series of `insert`s. Then `findMin` executes those pending merges to find the value of the root. Next, `deleteMin` deletes the root and creates a new pending merge of the two children. Finally, `findMin` again executes the pending merges to find the new value of the root.

Notice that pending nodes and lazy evaluation affect *when* the various steps of a `merge` are carried out, but that they do not affect the end results of those steps. After all the pending merges have been executed, the final tree is identical to the one produced by skew heaps without lazy evaluation.

Strictly speaking, the nodes of a lazy skew heap can no longer be called immutable. In particular, when a pending merge is executed, the node representing the pending merge is updated with the result so that it cannot be executed twice. Functional languages typically allow this kind of mutation, known as *memoization*, because it is invisible to the user, except in terms of efficiency. Suppose that memoization was not performed. Then pending merges might be executed multiple times. However, every time a given pending merge was executed, it would produce the same result. Therefore, memoization is an optimization that makes lazy evaluation run faster, but that has no effect on the output of any computation.

40.4.1 Analysis of Lazy Skew Heaps

Next, we prove that `merge` on lazy skew heaps runs in logarithmic amortized time. We use the banker's method, associating a certain number of credits with each pending merge.

We begin with a few definitions. The *logical view* of a tree is what the tree would look like if all of its pending merges were executed. The *logical left spine* of a tree is the leftmost path from the root in the logical view. Similarly, the *logical right spine* of a tree is the rightmost path from the root in the logical view. A pending node is called *left-heavy* if its left subtree is at least as large as its right subtree in the logical view, or *right-heavy* if its right subtree is larger than its left subtree in the logical view. The *successor* of a pending node is the new pending node that is created as the new left child of the existing node when the existing node is executed.

Now, we charge one credit to execute a single pending merge. This credit pays for the

```
public class Skew {
  public static final Skew empty = null;
  public static Skew insert(int x,Skew s) { return merge(new Skew(x,null,null),s); }
  public static int findMin(Skew s) {
    executePendingMerge(s);
    return s.element;
  }
  public static Skew deleteMin(Skew s) {
    executePendingMerge(s);
    return merge(s.left,s.right);
  }

  public static Skew merge(Skew s,Skew t) {
    if (t == null) return s;
    else if (s == null) return t;
    else return new Skew(s,t); // create a pending merge
  }

  private static void executePendingMerge(Skew s) {
    if (s != null && s.pendingMerge) {
      Skew s1 = s.left, s2 = s.right;
      executePendingMerge(s1);
      executePendingMerge(s2);
      if (s2.element < s1.element) { Skew tmp = s1; s1 = s2; s2 = tmp; }
      s.element = s1.element;
      s.left = merge(s2,s1.right);
      s.right = s1.left;
      s.pendingMerge = false;
    }
  }

  private boolean pendingMerge;
  private int element;
  private Skew left,right;
  private Skew(int element, Skew left, Skew right) {
    this.element = element;
    this.left = left;
    this.right = right;
    pendingMerge = false;
  }
  private Skew(Skew left,Skew right) { // create a pending merge
    this.left = left;
    this.right = right;
    pendingMerge = true;
  }
}
```

FIGURE 40.12: Skew heaps with lazy evaluation in Java.

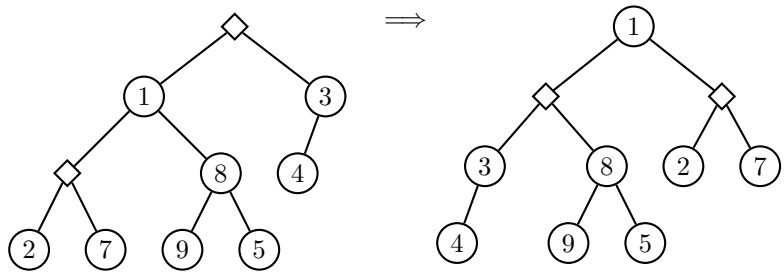


FIGURE 40.13: Executing a pending merge.

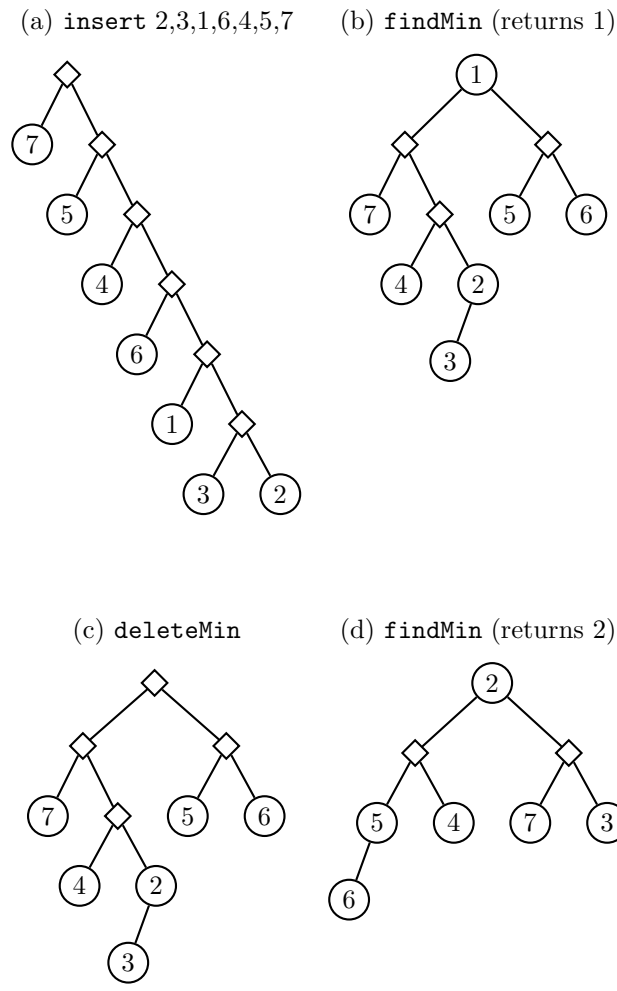


FIGURE 40.14: A sequence of operations on skew heaps.

comparison, the allocation of the successor node, and all necessary pointer manipulations, but it does not pay for executing the child nodes if they happen to be pending merges as well. When a right-heavy pending node is executed, it spends one of its own credits. When a left-heavy pending node is executed, the credit must be supplied either by its parent node, if it has one, or by the heap operation that originated the request, if the node is a root. This adds a single credit to the costs of the `insert` and `deleteMin` operations. After a pending node is executed, it passes any remaining credits to its successor node.

When we create a pending node, we must provide it with all the credits it will ever need. This includes

- one credit for itself, if it is right-heavy,
- one credit for each child that is a left-heavy pending node, and
- any credits needed by its successors.

Notice that a pending node and its successors all lay on the logical left spine of the resulting tree in the logical view. Similarly, the physical children of a pending node and its successors all lay on the logical right spines of the argument trees to the original `merge`. Therefore, the number of credits that we must create during a `merge` is bounded by the number of right-heavy nodes in the logical left spine of the resulting tree plus the numbers of left-heavy nodes in the logical right spines of the argument trees.

It is easy to see that the number of right-heavy nodes in a logical left spine is at most logarithmic in the size of the logical view. Similarly, the number of left-heavy nodes in a logical right spine is at most logarithmic in the size of the logical view. The total number of credits created by `merge` is therefore bounded by the sum of three logarithmic values, and thus is logarithmic itself.

40.5 Difficulties

As this chapter has shown, many data structures work quite nicely in a functional setting. However, some do not. We close with a description of several warning signs to watch for.

- *Random access*: All of the data structures described in this chapter have been pointer-based. Unfortunately, data structures that depend on arrays—such as hash tables—are more difficult to handle. No entirely satisfactory approach is known for making arrays persistent. The best known approach from a theoretical point of view is that of Dietz [6], in which array accesses run in $O(\log \log n)$ expected amortized time. However, his approach is quite complicated and difficult to implement. Competing approaches, such as [1, 4, 27], degrade to logarithmic (or worse!) time per access in common cases.
- *Cycles*: Not all pointer-based data structures are suitable for implementation in a functional setting. The most common problem is the presence of cycles, such as those found in doubly-linked lists or in binary search trees with parent pointers. Path copying requires copying *all* paths from the root to the site of the update. In the presence of cycles, this frequently means copying the entire data structure.
- *Multiple entrypoints*: Even without cycles, a pointer-based data structure can run into difficulties if it has multiple entrypoints. For example, consider a pointer-based implementation of the union-find data structure [33]. All the pointers go from children to parents, so there are no cycles (except sometimes trivial ones at the roots). However, it is common for every node of the union-find data structure to be a potential entrypoint, rather than just the root(s). Path copying requires

copying all paths from *any entrypoint* to the site of the update. With multiple entrypoints, this again frequently degrades into copying the entire data structure.

- *Unpredictable access patterns*: Section 40.4 described how to use lazy evaluation to make an amortized data structure persistent. Although this works for many amortized data structures, such as skew heaps [32], it does not work for all amortized data structures. In particular, data structures with highly unpredictable access patterns, such as splay trees [31] (see Chapter 12), are difficult to make persistent in this fashion.

40.6 Further Reading

The most complete general reference on functional data structures is Okasaki [24]. For more information on specific data structures, consult the following sources:

- queues and deques [5, 10, 11, 21]
- priority queues and priority search queues [3, 8]
- random-access lists and flexible arrays [9, 12, 13, 16, 20, 23]
- catenable lists and deques [14, 19, 22]

Acknowledgments

This work was supported, in part, by the National Science Foundation under grant CCR-0098288. The views expressed in this chapter are those of the author and do not reflect the official policy or position of the United States Military Academy, the Department of the Army, the Department of Defense, or the U.S. Government.

References

[1] A. Aasa, S. Holmström, and C. Nilsson. An efficiency comparison of some representations of purely functional arrays. *BIT*, 28(3):490–503, 1988.

[2] S. Adams. Efficient sets—a balancing act. *Journal of Functional Programming*, 3(4):553–561, Oct. 1993.

[3] G. S. Brodal and C. Okasaki. Optimal purely functional priority queues. *Journal of Functional Programming*, 6(6):839–857, Nov. 1996.

[4] T. Chuang. A randomized implementation of multiple functional arrays. In *ACM Conference on LISP and Functional Programming*, pages 173–184, June 1994.

[5] T. Chuang and B. Goldberg. Real-time deques, multihead Turing machines, and purely functional programming. In *Conference on Functional Programming Languages and Computer Architecture*, pages 289–298, June 1993.

[6] P. F. Dietz. Fully persistent arrays. In *Workshop on Algorithms and Data Structures*, volume 382 of *LNCS*, pages 67–74. Springer-Verlag, Aug. 1989.

[7] R. Hinze. Explaining binomial heaps. *Journal of Functional Programming*, 9(1):93–104, Jan. 1999.

[8] R. Hinze. A simple implementation technique for priority search queues. In *ACM SIGPLAN International Conference on Functional Programming*, pages 110–121, Sept. 2001.

[9] R. Hinze. Bootstrapping one-sided flexible arrays. In *ACM SIGPLAN International Conference on Functional Programming*, pages 2–13, Sept. 2002.

[10] R. Hood. *The Efficient Implementation of Very-High-Level Programming Language Constructs*. PhD thesis, Department of Computer Science, Cornell University, Aug. 1982. (Cornell TR 82-503).

[11] R. Hood and R. Melville. Real-time queue operations in pure Lisp. *Inf. Process. Lett.*, 13(2):50–53, Nov. 1981.

[12] R. R. Hoogerwoord. A logarithmic implementation of flexible arrays. In *Conference on Mathematics of Program Construction*, volume 669 of *LNCS*, pages 191–207. Springer-Verlag, July 1992.

[13] A. Kaldewaij and V. J. Dielissen. Leaf trees. *Sci. Comput. Programming*, 26(1–3):149–165, May 1996.

[14] H. Kaplan and R. E. Tarjan. Purely functional, real-time deques with catenation. *J. ACM*, 46(5):577–603, Sept. 1999.

[15] D. J. King. Functional binomial queues. In *Glasgow Workshop on Functional Programming*, pages 141–150, Sept. 1994.

[16] E. W. Myers. An applicative random-access stack. *Inf. Process. Lett.*, 17(5):241–248, Dec. 1983.

[17] E. W. Myers. Efficient applicative data types. In *ACM Symposium on Principles of Programming Languages*, pages 66–75, Jan. 1984.

[18] M. Núñez, P. Palao, and R. Peña. A second year course on data structures based on functional programming. In *Functional Programming Languages in Education*, volume 1022 of *LNCS*, pages 65–84. Springer-Verlag, Dec. 1995.

[19] C. Okasaki. Amortization, lazy evaluation, and persistence: Lists with catenation via lazy linking. In *IEEE Symposium on Foundations of Computer Science*, pages 646–654, Oct. 1995.

[20] C. Okasaki. Purely functional random-access lists. In *Conference on Functional Programming Languages and Computer Architecture*, pages 86–95, June 1995.

[21] C. Okasaki. Simple and efficient purely functional queues and deques. *Journal of Functional Programming*, 5(4):583–592, Oct. 1995.

[22] C. Okasaki. Catenable double-ended queues. In *ACM SIGPLAN International Conference on Functional Programming*, pages 66–74, June 1997.

[23] C. Okasaki. Three algorithms on Braun trees. *Journal of Functional Programming*, 7(6):661–666, Nov. 1997.

[24] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[25] C. Okasaki. Red-black trees in a functional setting. *Journal of Functional Programming*, 9(4):471–477, July 1999.

[26] C. Okasaki and A. Gill. Fast mergeable integer maps. In *Workshop on ML*, pages 77–86, Sept. 1998.

[27] M. E. O'Neill and F. W. Burton. A new method for functional arrays. *Journal of Functional Programming*, 7(5):487–513, Sept. 1997.

[28] S. Peyton Jones et al. Haskell 98: A non-strict, purely functional language. http://haskell.org/onlinereport/, Feb. 1999.

[29] F. Rabhi and G. Lapalme. *Algorithms: A Functional Programming Approach*. Addison-Wesley, 1999.

[30] C. M. P. Reade. Balanced trees with removals: an exercise in rewriting and proof. *Sci. Comput. Programming*, 18(2):181–204, Apr. 1992.

[31] D. D. K. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, July 1985.

[32] D. D. K. Sleator and R. E. Tarjan. Self-adjusting heaps. *SIAM J. Comput.*, 15(1):52–69, Feb. 1986.

[33] R. E. Tarjan. *Data Structures and Network Algorithms*, volume 44 of *CBMS Re-*

gional Conference Series in Applied Mathematics. Society for Industrial and Applied Mathematics, Philadelphia, 1983.

41

LEDA, a Platform for Combinatorial and Geometric Computing

Stefan Naeher
University of Trier

41.1 Introduction

LEDA, the **L**ibrary of **E**fficient **D**ata Types and **A**lgorithms, aims at being a comprehensive software platform for the area of combinatorial and geometric computing. It provides a sizable collection of data types and algorithms in C++. This collection includes most of the data types and algorithms described in the text books of the area ([1, 6, 9, 10, 15–17, 19–21]). LEDA supports a broad range of applications. It has already been used in such diverse areas as code optimization, VLSI design, graph drawing, graphics, robot motion planning, traffic scheduling, geographic information systems, machine learning and computational biology.

The LEDA project was started in 1988 by Kurt Mehlhorn and Stefan Näher. The first months they spent on the specification of different data types and on selecting the implementation language. At that time the item concept came up as an abstraction of the notion "pointer into a data structure". Items provide direct and efficient access to data and are similar to iterators in the standard template library. The item concept worked successfully for all test cases and is now used for most data types in LEDA. Concurrently with searching for the correct specifications several languages were investigated for their suitability as an implementation platform. Among the candidates were Smalltalk, Modula, Ada, Eiffel, and C++. The language had to support abstract data types and type parameters (genericity)

and should be widely available. Based on the experiences with different example programs C++ was selected because of its flexibility, expressive power, and availability.

We discuss some of the general aspects of the LEDA system.

41.1.1 Ease of Use

The library is easy to use. In fact, only a small fraction of the users are algorithms experts and many users are not even computer scientists. For these users the broad scope of the library, its ease of use, and the correctness and efficiency of the algorithms in the library are crucial. The LEDA manual [11] gives precise and readable specifications for the data types and algorithms mentioned above. The specifications are short (typically not more than a page), general (so as to allow several implementations) and abstract (so as to hide all details of the implementation).

41.1.2 Extensibility

Combinatorial and geometric computing is a diverse area and hence it is impossible for a library to provide ready-made solutions for all application problems. For this reason it is important that LEDA is easily extendible and can be used as a platform for further software development. In many cases LEDA programs are very close to the typical text book presentation of the underlying algorithms. The goal is the equation *Algorithm* + LEDA = *Program*.

LEDA *extension packages* (LEPs) extend LEDA into particular application domains and areas of algorithmics not covered by the core system. LEDA extension packages satisfy requirements, which guarantee compatibility with the LEDA philosophy. LEPs have a LEDA-style documentation, they are implemented as platform independent as possible and the installation process allows a close integration into the LEDA core library. Currently, the following LEPs are available: PQ-trees, dynamic graph algorithms, a homogeneous d-dimensional geometry kernel, and a library for graph drawing.

41.1.3 Correctness

Programming is a notoriously error-prone task; this is even true when programming is interpreted in a narrow sense: going from a (correct) algorithm to a program. The standard way to guard against coding errors is program testing. The program is exercised on inputs for which the output is known by other means, typically as the output of an alternative program for the same task. Program testing has severe limitations. It is usually only done during the testing phase of a program. Also, it is difficult to determine the "correct" suite of test inputs. Even if appropriate test inputs are known it is usually difficult to determine the correct outputs for these inputs: alternative programs may have different input and output conventions or may be too inefficient to solve the test cases.

Given that program verification, i.e., formal proof of correctness of an implementation, will not be available on a practical scale for some years to come, *program checking* has been proposed as an extension to testing [3, 4]. The cited papers explored program checking in the area of algebraic, numerical, and combinatorial computing. In [8, 14, 28] program checkers are presented for planarity testing and a variety of geometric tasks. LEDA uses program checkers for many of its implementations.

In computational geometry the correctness problem is even more difficult because geometric algorithms are frequently formulated under two unrealistic assumptions: computers are assumed to use exact real arithmetic (in the sense of mathematics) and inputs are

assumed to be in general position. The naive use of floating point arithmetic as an approximation to exact real arithmetic very rarely leads to correct implementations. In a sequence of papers [5, 12, 18, 22, 24] the degeneracy and precision issues have been investigated and LEDA was extended based on this theoretical work. It now provides exact geometric kernels for two-dimensional and higher dimensional computational geometry [26] and also correct implementations for basic geometric tasks, e.g., two-dimensional convex hulls, Delaunay diagrams, Voronoi diagrams, point location, line segment intersection, and higher-dimensional convex hulls and Delaunay triangulations.

An elegant (theoretical) approach to the degeneracy problem is *symbolic perturbation*. However, this method of forcing input data into general position can cause some serious problems in practice. In many cases, it increases the complexity of (intermediate) results considerably and furthermore, the final limit process turns out to be very difficult in particular in the presence of combinatorial structures. For this reason, LEDA follows a different approach. It copes with degeneracies directly by treating the degenerate case as the "normal" case. This approach proved to be very effective for many geometric problems.

41.1.4 Availability and Usage

LEDA is realized in C++ and can be used on many different platforms with many different compilers. LEDA is now used at many academic sites. A commercial version of LEDA is marketed by Algorithmic Solutions Software GmbH (<www.algorithmic-solutions.com>).

41.2 The Structure of LEDA

LEDA uses templates for the implementation of parameterized data types and for generic algorithms. However, it is not a pure template library and therefore is based on a number of object code libraries of precompiled code. Programs using LEDA data types or algorithms have to include the appropriate LEDA header files into their source code and have to be linked with one or more of these libraries. The four object code libraries are built on top of each other. Here, we only give a short overview. Consult the LEDA user manual ([11] or the LEDA book ([27]) for a detailed description.

- The Basic Library (*libL*)
  contains system dependent code, basic data structures, numbers and types for linear algebra, dictionaries, priority queues, partitions, and many more basic data structures and algorithms.
- The Graph Library (*libG*)
  contains different types of graphs and a large collection of graph and network algorithms
- The 2D Geometry Library (*libP*)
  contains the two-dimensional geometric kernels advanced geometric data structures, and a large number of algorithms for two-dimensional geometric problems.
- The 3D Geometry Library (*libP*)
  contains the three-dimensional kernels and some algorithms for three-dimensional problems.
- The Window Library(*libW*)
  supports graphical output and user interaction for both the X11 platform (Unix) and Microsoft Windows systems. It also contains animation support: GraphWin, a powerful graph editor, and GeoWin, a interactive tool for the visualization of geometric algorithms. See Section 41.5 for details.

41.3 Data Structures and Data Types

LEDA contains a large number of data structures from the different areas of combinatorial and geometric computing. However, as indicated in the name of the library, LEDA was not designed as a collection of *data structures* but as a library of (parameterized) *data types*. For each (abstract) data type in the library there is at least one data structure which implements this type. This separation of specification (by an abstract data type) and implementation (by an actual data structure) is crucial for a software component library. It allows to change the underlying implementation or to choose between different implementations without having to change the application program. In general, there is one default data structure for each of the advanced data types, e.g avl-trees for dictionaries, skiplists for sorted sequences, binary heaps for priority queues, and a union-find structure for partitions. For most of these data types a number of additional data structures are available and can be specified by an optional template argument. For instance `dictionary<string,int,skiplist>` specifies a dictionary type with key type *string* and information type *int* which is implemented by the *skiplist* data structure (see [27] for more details and Section 41.6.1 for an example).

41.3.1 Basic Data Types

Of course, LEDA contains a complete collection of all basic data types, such as strings, stacks, queues, lists, arrays, tuples ... which are ubiquitous in all areas of computing.

41.3.2 Numbers and Matrices

Numbers are at the origin of computing. We all learn about integers, rationals, and real numbers during our education. Unfortunately, the number types *int*, *float*, and *double* provided by C++ are only crude approximations of their mathematical counterparts: there are only finitely many numbers of each type and for floats and doubles the arithmetic incurs rounding errors. LEDA offers the additional number types *integer*, *rational*, *bigfloat*, and *real*. The first two are the exact realization of the corresponding mathematical types and the latter two are better approximations of the real numbers. Vectors and matrices are one- and two-dimensional arrays of numbers, respectively. They provide the basic operations of linear algebra.

41.3.3 Advanced Data Types

A collection of parameterized data types representing sets, partitions, dictionaries, priority queues, sorted sequences, partitions and sparse arrays (maps, dictionary and hashing arrays). Type parameters include the key, information, priority, or element type and (optional) the data structure used for the implementation of the type. The list of data structures includes skiplists (see Chapter 13), (a, b)-trees, avl-trees, red-black trees, $bb[\alpha]$-trees (see Chapter 10), randomized search trees (see Chapter 13), Fibonacci-heaps, binary heaps, pairing heaps (see Chapter 7), redistributive heaps, union find with path compression (Chapter 33), dynamic perfect hashing, cuckoo hashing (see Chapter 9), Emde-Boas trees, ... This list of data structures is continuously growing and adapted to new results from the area for data structures and algorithms.

41.3.4 Graph Data Structures

The graph data type (Chapter 4) is one of the central data types in LEDA. It offers the standard iterations such as "for all nodes v of a graph G do" (written $forall\_nodes(v, G)$) or "for all edges e incident to a node v do" (written $forall\_out\_edges(e, v)$), it allows to add and delete vertices and edges and it offers arrays and matrices indexed by nodes and edges, (see [27] chapter 6 for details and Section 41.6.2 for an example program). The data type *graph* allows to write programs for graph problems in a form very close to the typical text book presentation.

41.3.5 Geometry Kernels

LEDA offers kernels for two- and three-dimensional geometry, a kernel of arbitrary dimension is available as an extension package. In either case there exists a version of the kernel based on floating point Cartesian coordinates (called *float-kernel*) as well as a kernel based on rational homogeneous coordinates (called *rat-kernel*). All kernels provide a complete collection of geometric objects (points, segments, rays, lines, circles, simplices, polygons, planes, etc.) together with a large set of geometric primitives and predicates (orientation of points, side-of-circle tests, side-of-hyperplane, intersection tests and computation, etc.). For a detailed discussion and the precise specification see Chapter 9 of the LEDA book ([27]). Note that only for the rational kernel, which is based on exact arithmetic and floating-point filters, all operations and primitives are guaranteed to compute the correct result.

41.3.6 Advanced Geometric Data Structures

In addition to the basic kernel data structures LEDA provides many advanced data types for computational geometry. Examples are

- a general polygon type (`gen_polygon` or `rat_gen_polygon`) with a complete set of boolean operations. Its implementation is based on an efficient and robust plane sweep algorithms for the construction of the arrangement of a set of straight line segments (see [13] and [27] chapter 10.7 for a detailed description).
- two- and higher-dimensional geometric tree structures, such as range, segment, interval and priority search trees (see Chapter 18).
- partially and fully persistent search trees (Chapter 31).
- different kinds of geometric graphs (triangulations, Voronoi diagrams, and arrangements)
- a dynamic `point_set` data type supporting update, search, closest point, and different types of range query operations on one single representation which is based on a dynamic Delaunay triangulation (see [27] chapter 10.6).

41.4 Algorithms

The LEDA project had never the goal to provide a complete collection of all algorithms. LEDA was designed and implemented to establish a *platform* for combinatorial and geometric computing enabling programmers to implement these algorithms themselves more easily and customized to their particular needs. But of course the library already contains a considerable number of standard algorithms.

Here we give a brief overview and refer the reader to the user manual for precise specifications and to chapter 10 of the LEDA-book ([27]) for a detailed description and analysis of the corresponding implementations. In the current version LEDA offers different implementation of algorithms for the following problems:

- sorting and searching
- basic graph properties
- graph traversals
- different kinds of connected components
- planarity test and embeddings
- minimum spanning trees
- shortest paths
- network flows
- maximum weight and cardinality matchings
- graph drawing
- convex hulls (also three-dimensional)
- half-plane intersections
- (constraint) triangulations
- closest and farthest Delaunay and Voronoi diagrams
- Euclidean minimum spanning trees
- closest pairs
- boolean operations on generalized polygons
- segment intersection and construction of line arrangements
- Minkowski sums and differences
- nearest neighbors and closest points
- minimum enclosing circles and annuli
- curve reconstruction

41.5 Visualization

Visualization and animation of programs is very important for the understanding, presentation, and debugging of algorithms (see Chapter 44). LEDA provides two powerful tools for interactive visualization and animation of data structures and algorithms:

41.5.1 GraphWin

The *GraphWin* data type (see [27], chapter 12) combines the *graph* and the *window* data type. An object of type *GraphWin* (short: a GraphWin) is a window, a graph, and a drawing of the graph, all at once. The graph and its drawing can be modified either by mouse operations or by running a graph algorithm on the graph. The GraphWin data type can be used to:

- construct and display graphs,
- visualize graphs and the results of graph algorithms,
- write interactive demos for graph algorithms,

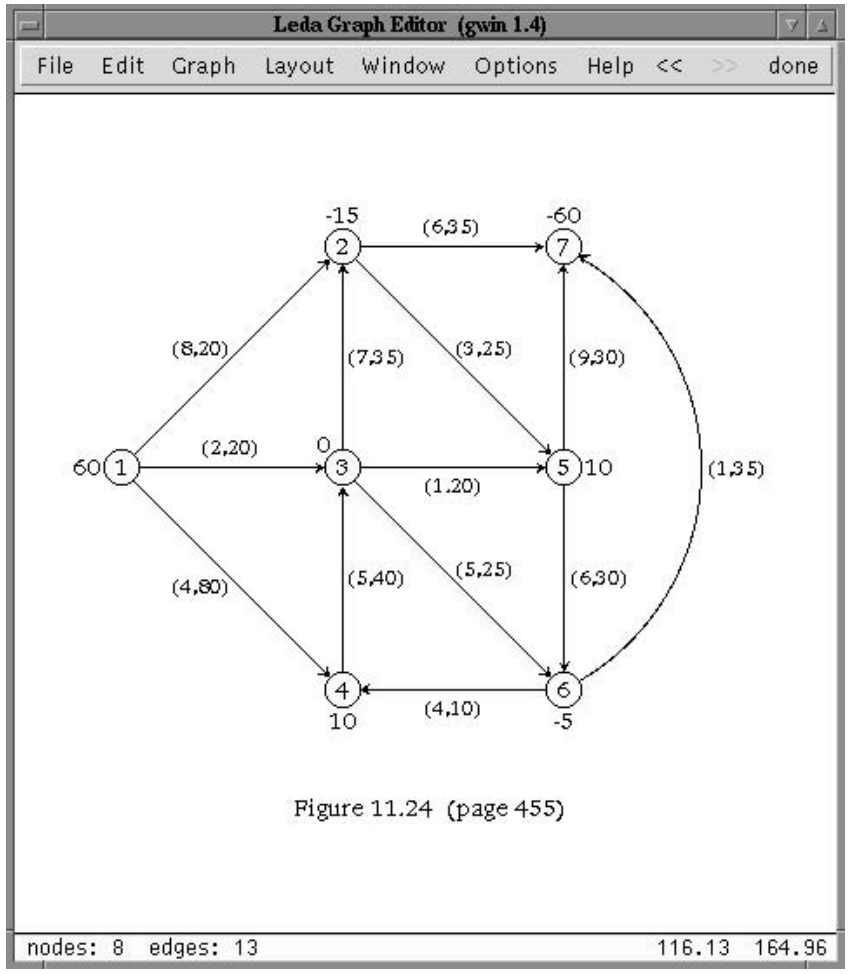


FIGURE 41.1: A GraphWin. The display part of the window shows a graph G and the panel part of the window features the default menu of a GraphWin. G can be edited interactively, e.g., nodes and edges can be added, deleted, and moved around. It is also possible to run graph algorithms on G and to display their result or to animate their execution.

- animate graph algorithms.

All demos and animations of graph algorithms in LEDA are based on GraphWin, many of the drawings in the LEDA book ([27]) have been made with GraphWin, and many of the geometry demos in LEDA have a GraphWin button that allows us to view the graph structure underlying a geometric object.

GraphWin can easily be used in programs for constructing, displaying and manipulating graphs and for animating and debugging graph algorithms. It offers both a programming and an interactive interface, most applications use both of them.

41.5.2 GeoWin

GeoWin ([23]) is a generic tool for the interactive visualization of geometric algorithms. *GeoWin* is implemented as a C++data type. It provides support for a number of programming techniques which have shown to be very useful for the visualization and animation of algorithms. The animations use *smooth transitions* to show the result of geometric algorithms on dynamic user-manipulated input objects, e.g., the Voronoi diagram of a set of moving points or the result of a sweep algorithm that is controlled by dragging the sweep line with the mouse (see Figure 41.2).

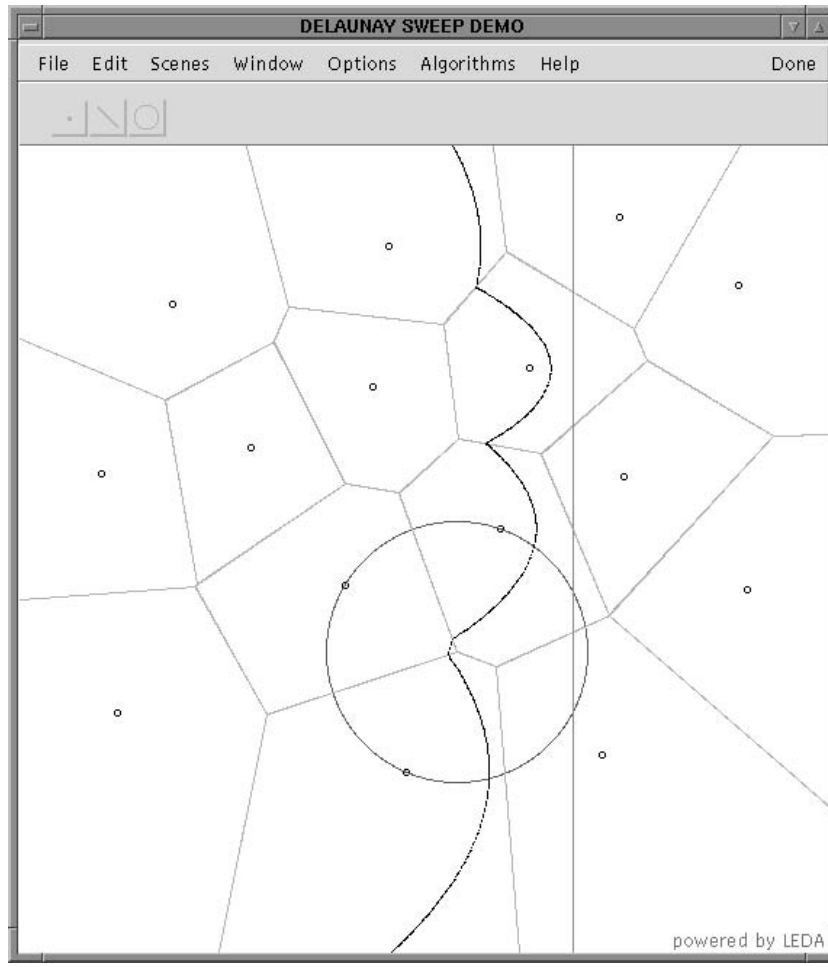


FIGURE 41.2: GeoWin animating Fortune's Sweep Algorithm.

A GeoWin maintains one or more geometric scenes. A geometric *scene* is a collection of geometric objects of the same type. A collection is simply either a standard C++-list (STL-list) or a LEDA-list of objects. GeoWin requires that the objects provide a certain functionality, such as stream input and output, basic geometric transformations, drawing and input in a LEDA window. A precise definition of the required operations can be found

in the manual pages [11]. GeoWin can be used for any collection of basic geometric objects (geometry kernel) fulfilling these requirements.

The visualization of a scene is controlled by a number of attributes, such as color, line width, line style, etc. A scene can be subject to user interaction and it may be defined from other scenes by means of an algorithm (a C++ function). In the latter case the scene (also called *result scene*) may be recomputed whenever one of the scenes on which it depends is modified. There are three main modes for re-computation: user-driven, continuous, and event-driven.

GeoWin has both an interactive and a programming interface. The interactive interface supports the interactive manipulation of input scenes, the change of geometric attributes, and the selection of scenes for visualization.

41.6 Example Programs

In this section we give several programming examples showing how LEDA can be used to implement combinatorial and geometric algorithms in a very elegant and readable way. In each case we will first state the algorithm and then show the program. It is not essential to understand the algorithms in full detail; our goal is to show:

- how easily the algorithms are transferred into programs and
- how natural and elegant the programs are.

In other words,

$$\text{Algorithm} + \text{LEDA} = \text{Program}.$$

41.6.1 Word Count

We start with a very simple program. The task is to read a sequence of strings from standard input, to count the number of occurrences of each string in the input, and to print a list of all occurring strings together with their frequencies on standard output.

The program uses the LEDA types *string* and *d_array* (dictionary arrays). The parameterized data type *d_array*<I, E> realizes arrays with index type I and element type E. We use it with index type *string* and element type *int*.

```
#include <LEDA/d_array.h>
#include <LEDA/impl/skiplist.h>
int main()
{ d_array<string,int,skiplist> N(0);
  string s;
  while (cin >> s) N[s]++;
  forall_defined(s,N) cout << s << " " << N[s] << endl;
}
```

We give some more explanations. The program starts with the include statement for dictionary arrays and skiplists. In the first line of the main program we define a dictionary array N with index type *string*, element type *int* and implementation *skiplist* and initialize all entries of the array to zero. Conceptually, this creates an infinite array with one entry for each conceivable string and sets all entries to zero; the implementation of d_arrays stores the non-zero entries in a balanced search tree with key type string. In the second line we define a string s. The while-loop does most of the work. We read strings from the standard input

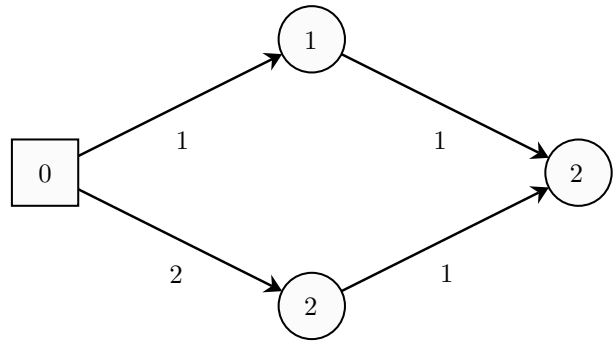


FIGURE 41.3: A shortest path in a graph. Each edge has a non-negative cost. The cost of a path is the sum of the cost of its edges. The source node s is indicated as a square. For each node the length of the shortest path from s is shown.

until the end of the input stream is reached. For every string s we increment the entry $N[s]$ of the array N by one. The iteration *forall_defined*(s, N) in the last line successively assigns all strings to s for which the corresponding entry of N was touched during execution. For each such string, the string and its frequency are printed on the standard output.

41.6.2 Shortest Paths

Dijkstra's shortest path algorithm [7] takes a directed graph $G = (V, E)$, a node $s \in V$, called the source, and a non-negative cost function on the edges $cost : E \to R_{\geq 0}$. It computes for each node $v \in V$ the distance from s, see Figure 41.3. A typical text book presentation of the algorithm is as follows:

forall nodes v **do**
  set $dist(v)$ to infinity.
  declare v unreached.
od
set $dist(s)$ to 0.

while there is an unreached node **do**
  let u be an unreached node with minimal dist-value. (*)
  declare u reached.
  forall edges $e = (u, v)$ out of u **do**
    set $dist(v) = \min(dist(v), dist(u) + cost(e))$.
  od
od

The text book presentation will then continue to discuss the implementation of line (*). It will state that the pairs $\{(v, dist(v)); v \text{ unreached}\}$ should be stored in a priority queue, e.g., a Fibonacci heap, because this will allow the selection of an unreached node with minimal distance value in logarithmic time. It will probably refer to some other chapter of the book for a discussion of priority queues.

We next give the corresponding LEDA program; it is very similar to the pseudo-code above. In fact, after some experience with LEDA you should be able to turn the pseudo-

code into code within a few minutes.

```
#include <LEDA/graph.h>
#include <LEDA/node_pq.h>
void DIJKSTRA(const graph& G, node s, const edge_array<double>& cost,
                                          node_array<double>& dist)
{ node_pq<double> PQ(G);
  node v;
  dist[s] = 0;
  forall_nodes(v,G)
  { if (v != s) dist[v] = MAXDOUBLE;
    PQ.insert(v,dist[v]);
   }

  while (!PQ.empty()) {
    node u = PQ.del_min();
    edge e;
    forall_out_edges(e,u) {
      v = target(e);
      double c = dist[u] + cost[e];
      if (c < dist[v]) {
        PQ.decrease_p(v,c);
        dist[v] = c;
      }
    }
  }
}
```

We start by including the graph and the node priority queue data types. The function *DIJKSTRA* takes a graph G, a node s, an *edge_array cost*, and a *node_array dist*. Edge arrays and node arrays are arrays indexed by edges and nodes, respectively. We declare a priority queue PQ for the nodes of graph G. It stores pairs $(v, dist[v])$ and is initially empty. The *forall_nodes*-loop initializes *dist* and PQ. In the main loop we repeatedly select a pair $(u, dist[u])$ with minimal distance value and then iterate over all out-going edges to update distance values of neighboring vertices.

We next incorporate the shortest path program into a small demo. We generate a random graph with n nodes and m edges and choose the edge costs as random number in the range $[0, 100]$. We call the function above and report the running time.

```
int main()
{ int n = read_int("number of nodes = ");
  int m = read_int("number of edges = ");
  graph G;
  random_graph(G,n,m);
  edge_array<double> cost(G);
  node_array<double> dist(G);
  edge e;
  forall_edges(e,G) cost[e] = rand_int(0,100);

  float T = used_time();
  DIJKSTRA(G,G.first_node(),cost,dist);
```

```
  cout << "The computation took " << used_time(T) << " seconds." << endl;
}
```

On a graph with 10000 nodes and 100000 edges the computation takes less than a second.

41.6.3 Curve Reconstruction

The reconstruction of a curve from a set of sample points is an important problem in computer vision. Amenta, Bern, and Eppstein [2] introduced a reconstruction algorithm which they called CRUST. Figure 41.4 shows a point set and the curves reconstructed by their algorithm. The algorithm *CRUST* takes a list S of points and returns a graph G. CRUST makes use of Delaunay diagrams and Voronoi diagrams and proceeds in three steps:

- It first constructs the Voronoi diagram VD of the points in S.
- It then constructs a set $L = S \cup V$, where V is the set of vertices of VD.
- Finally, it constructs the Delaunay triangulation DT of L and makes G the graph of all edges of DT that connect points in S.

The algorithm is very simple to implement

```
#include <LEDA/graph.h>
#include <LEDA/map.h>
#include <LEDA/float_kernel.h>
#include <LEDA/geo_alg.h>
void CRUST(const list<point>& S, GRAPH<point,int>& G)
{
  list<point> L = S;
  GRAPH<circle,point> VD;
  VORONOI(L,VD);
  // add Voronoi vertices and mark them
  map<point,bool> voronoi_vertex(false);
  node v;
  forall_nodes(v,VD)
  { if (VD.outdeg(v) < 2) continue;
    point p = VD[v].center();
    voronoi_vertex[p] = true;
    L.append(p);
  }
  DELAUNAY_TRIANG(L,G);
  forall_nodes(v,G)
    if (voronoi_vertex[G[v]]) G.del_node(v);
}
```

We give some explanations. We start by including graphs, maps, the floating point geometry kernel, and the geometry algorithms. In CRUST we first make a copy of S in L. Next we compute the Voronoi diagram VD of the points in L. In LEDA we represent Voronoi diagrams by graphs whose nodes are labeled with circles. A node v is labeled by a circle passing through the defining sites of the vertex. In particular, $VD[v].center(\,)$ is the position of the node v in the plane. Having computed VD we iterate over all nodes of VD and add all finite vertices (a Voronoi diagram also has nodes at infinity, they have degree one in our graph representation of Voronoi diagrams) to L. We also mark all added points

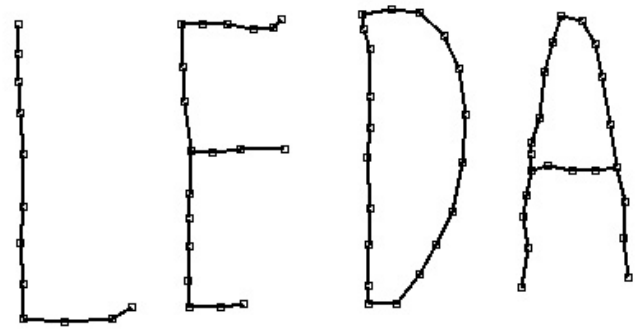


FIGURE 41.4: A set of points in the plane and the curve reconstructed by CRUST. The figure was generated by the program presented in Section 41.6.3.

as vertices of the Voronoi diagram. Next we compute the Delaunay triangulation of the extended point set in G. Having computed the Delaunay triangulation, we collect all nodes of G that correspond to vertices of the Voronoi diagram in a list *vlist* and delete all nodes in *vlist* from G. The resulting graph is the result of the reconstruction.

We next incorporate CRUST into a small demo which illustrates its speed. We generate n random points in the plane and construct their crust. We are aware that it does really make sense to apply CRUST to a random set of points, but the goal of the demo is to illustrate the running time.

```
int main()
{ int n = read_int("number of points = ");
  list<point> S;
  random_points_in_unit_square(n,S);
  GRAPH<point,int> G;

  float T = used_time();
  CRUST(S,G);
  cout << "The crust computation took " << used_time(T) << " seconds.";
  cout << endl;
}
```

For 3000 points the computation takes less than a second. We can now use the preceding program for a small interactive demo.

```
#include <LEDA/window.h>
int main()
{ window W;
  W.display();
  W.set_node_width(2);
  W.set_line_width(2);
  point p;
  list<point> S;
  GRAPH<point,int> G;
```

```
  while (W >> p)
  { S.append(p);
    CRUST(S,G);
    W.clear();
    node v;
    forall_nodes(v,G) W.draw_node(G[v]);
    edge e;
    forall_edges(e,G) W.draw_segment(G[source(e)], G[target(e)]);
   }
}
```

We give some more explanations. We start by including the window type. In the main program we define a window and open its display. A window will pop up. We state that we want nodes and edges to be drawn with width two. We define the list S and the graph G required for CRUST. In each iteration of the while-loop we read a point in W (each click of the left mouse button enters a point), append it to S and compute the crust of S in G. We then draw G by drawing its vertices and its edges. Each edge is drawn as a line segment connecting its endpoints. Figure 41.4 was generated with the program above.

41.6.4 Upper Convex Hull

In the next example we show how to use LEDA for computing the upper convex hull of a given set of points. The following function `UPPER_HULL` takes a list L of rational points (type `rat_point`) as input and returns the list of points of the upper convex hull of L in clockwise ordering. The algorithm is a variant of Graham's Scan [25]. First we sort L according to the lexicographic ordering of the Cartesian coordinates and remove multiple points. If the list contains not more than two points after this step we stop. Before starting the actual Graham Scan we first skip all initial points lying on or below the line connecting the two extreme points. Then we scan the remaining points from left to right and maintain the upper hull of all points seen so far in a list called *hull*. Note however that the last point of the hull is not stored in this list but in a separate variable p. This makes it easier to access the last two hull points as required by the algorithm. Note also that we use the rightmost point as a sentinel avoiding the special case that *hull* becomes empty.

```
using namespace leda;
list<rat_point> UPPER_HULL(list<rat_point> L)
{ L.sort();
  L.unique();
  if (L.length() <= 2) return L;

  rat_point p_min = L.front(); // leftmost point
  rat_point p_max = L.back();  // rightmost point

  list<rat_point> hull;
  hull.append(p_max); // use rightmost point as sentinel
  hull.append(p_min); // first hull point

  // goto first point p above (p_min,p_max)
  while (!L.empty() && !left_turn(p_min,p_max,L.front()) L.pop();
  if (L.empty()) return hull;
```

```
  rat_point p = L.pop();  // second (potential) hull point
  rat_point q;
  forall(q,L)
  { while (!right_turn(hull.back(),p,q)) p = hull.pop_back();
    hull.append(p);
    p = q;
  }

  hull.append(p); // add last hull point
  hull.pop();     // remove sentinel
  return hull;
}
```

41.6.5 Delaunay Flipping

LEDA represents triangulations by bidirected plane graphs (from the graph library) whose nodes are labeled with points and whose edges may carry additional information, e.g. integer flags indicating the type of edge (hull edge, triangulation edge, etc.). All edges incident to a node v are ordered in counter-clockwise ordering and every edge has a reversal edge. In this way the faces of the graph represent the triangles of the triangulation. The graph type offers methods for iterating over the nodes, edges, and adjacency lists of the graph. In the case of plane graphs there are also operations for retrieving the reverse edge and for iterating over the edges of a face. Furthermore, edges can be moved to new nodes. This graph operation is used in the following program to implement edge flips.

Function `DELAUNAY_FLIPPING` takes as input an arbitrary triangulation and turns into a Delaunay triangulation by the well-known flipping algorithm. This algorithm performs a sequence of local transformations as shown in Figure 41.5 to establish the Delaunay property: for every triangle the circumscribing sphere does not contain any vertex of the triangulation in its interior. The test whether an edge has to be flipped or not can be realized by a so-called *side_of_circle* test. This test takes four points a, b, c, d and decides on which side of the oriented circle through the first three points a,b, and c the last point d lies. The result is positive or negative if d lies on the left or on the right side of the circle, respectively, and the result is zero if all four points lie on one common circle. The algorithms uses a list of candidates which might have to be flipped (initially all edges). After a flip the four edges of the corresponding quadrilateral are pushed onto this candidate list. Note that `G[v]` returns the position of node v in the triangulation graph G. A detailed description of the algorithm and its implementation can be found in the LEDA book ([27]).

```
void DELAUNAY_FLIPPING(GRAPH<POINT,int>& G)
{
  list<edge> S = G.all_edges();

  while ( !S.empty() )
  { edge e = S.pop();
    edge r = G.rev_edge(e);

    // e1,e2,e3,e4: edges of quadrilateral with diagonal e
    edge e1 = G.face_cycle_succ(r);
    edge e2 = G.face_cycle_succ(e1);
```

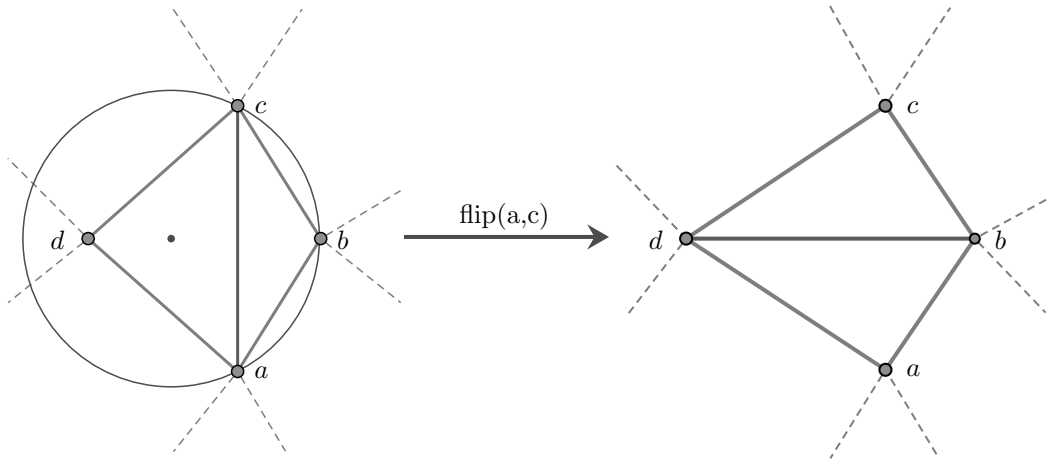


FIGURE 41.5: Delaunay Flipping

```
    edge e3 = G.face_cycle_succ(e);
    edge e4 = G.face_cycle_succ(e3);

    // a,b,c,d: corners of quadrilateral
    POINT a = G[G.source(e1)];
    POINT b = G[G.target(e1)];
    POINT c = G[G.source(e3)];
    POINT d = G[G.target(e3)];

    if (side_of_circle(a,b,c,d) > 0)
    { S.push(e1); S.push(e2); S.push(e3); S.push(e4);
      // flip diagonal
      G.move_edge(e,e2,source(e4));
      G.move_edge(r,e4,source(e2));
     }
   }
}
```

41.6.6 Discussion

In each of the above examples only a few lines of code were necessary to achieve complex functionality and, moreover, the code is elegant and readable. The data structures and algorithms in LEDA are efficient. For example, the computation of shortest paths in a graph with 10000 nodes and 100000 edges and the computation of the crust of 5000 points takes less than a second each. We conclude that LEDA is ideally suited for rapid prototyping as summarized in the equation

$$\text{Algorithm} + \text{LEDA} = \text{Efficient Program.}$$

41.7 Projects Enabled by LEDA

A large number academic and industrial projects from almost every area of combinatorial and geometric computing have been enabled by LEDA. Examples are graph drawing, algorithm visualization, geographic information systems, location problems, visibility algorithms, DNA sequencing, dynamic graph algorithms, map labeling, covering problems, railway optimization, route planning and many more. A detailed list of academic LEDA projects can be found on <http://www.mpi-sb.mpg.de/LEDA/friends> and a selection of industrial projects is shown on <http://www.algorithmic-solutions.com/enreferenzen.htm>.

References

[1] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley,1974.

[2] N. Amenta, M. Bern and D. Eppstein, *The Crust and the beta-Skeleton: Combinatorial Curve Reconstruction*, Graphical Models and Image Processing, Vol. 60, 125–135, 1998.

[3] M. Blum, M. Luby and R. Rubinfeld, *Self-testing/correcting with applications to numerical problems*, Proceedings of the 22nd Annual ACM Symposium on Theory of Computing (STOC90), 73–83, 1990.

[4] M. Blum and S. Kannan, *Designing Programs That Check Their Work*, Proceedings of the 21th Annual ACM Symposium on Theory of Computing (STOC89), 86–97, 1989.

[5] C. Burnikel, K. Mehlhorn and S. Schirra, *How to compute the Voronoi diagram of line Segments: Theoretical and experimental results*, Proceedings of the 2nd Annual European Symposium on Algorithms (ESA94), Lecture Notes in Computer Science 855, 227–239, 1994.

[6] T. H. Cormen and C. E. Leiserson and R. L. Rivest, *Introduction to Algorithms*, MIT Press/McGraw-Hill Book Company, 1990.

[7] E. W. Dijkstra, *A Note on Two Problems in Connection With Graphs*, Num. Math., Vol. 1, 269–271, 1959.

[8] C. Hundack, K. Mehlhorn and S. Näher, *A simple linear time algorithm for identifying Kuratowski subgraphs of non-planar graphs*, unpublished manuscript, 1996.

[9] J. H. Kingston, *Algorithms and Data Structures*, Addison-Wesley, 1990.

[10] K. Mehlhorn, *Data Structures and Algorithms 1,2, and 3*, Springer, 1984.

[11] K. Mehlhorn, S. Näher, M. Seel and C. Uhrig, *The LEDA User Manual*, Technical Report, Max-Planck-Institut für Informatik, 1999.

[12] K. Mehlhorn and S. Näher, *The implementation of geometric algorithms*, Proceedings of the 13th IFIP World Computer Congress, 223–231, Elsevier Science B.V. North-Holland, Amsterdam, 1994.

[13] K. Mehlhorn and S. Näher, *Implementation of a sweep line algorithm for the straight line segment intersection problem*, Technical Report, Max-Planck-Institut für Informatik, MPI-I-94-160, 1994.

[14] K. Mehlhorn and P. Mutzel, *On the Embedding Phase of the Hopcroft and Tarjan Planarity Testing Algorithm*, Algorithmica, Vol. 16, No. 2, 233–242, 1995.

[15] J. Nievergelt and K.H. Hinrichs, *Algorithms and Data Structures*, Prentice Hall, 1993.

[16] J. O'Rourke, *Computational Geometry in C*, Cambridge University Press, 1994.

[17] R. Sedgewick, *Algorithms*, Addison-Wesley, 1991.

[18] M. Seel, *Eine Implementierung abstrakter Voronoidiagramme*, Diplomarbeit, Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, 1994.

[19] R. E. Tarjan, *Data Structures and Network Algorithms*, CBMS-NSF Regional Conference Series in Applied Mathematics 44, 1983.

[20] C. J. van Wyk, *Data Structures and C Programs*, Addison-Wesley, 1988.

[21] D. Wood, *Data Structures, Algorithms, and Performance*, Addison-Wesley, 1993.

[22] C. Burnikel, K. Mehlhorn and S. Schirra, *On degeneracy in geometric computations*, Proc. 5th ACM-SIAM Sympos. Discrete Algorithms, 16–23, 1994.

[23] M. Bäsken and S. Näher, *GeoWin - A Generic Tool for Interactive Visualization of Geometric Algorithms*, Software Visualization, LNCS 2269, 88–100, 2002.

[24] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra and S. Schönherr, *On the Design of CGAL a Computational Geometry Algorithms Library*, Software – Practice and Experience, Vol. 30, No. 11, 1167–1202, 2000.

[25] R. L. Graham, *An efficient algorithm for determining the convex hulls of a finite point set*, Information Processing Letters, Vol. 1, 132–133, 1972.

[26] K. Mehlhorn, M. Müller, S. Näher, S. Schirra, M. Seel, C. Uhrig and J. Ziegler, *A Computational Basis for Higher-Dimensional Computational Geometry and Applications*, Comput. Geom. Theory Appl., Vol. 10, No. 4, 289–303, 1998.

[27] K. Mehlhorn and S. Näher, *LEDA: A Platform for Combinatorial and Geometric Computing*, Cambridge University Press, 2000.

[28] K. Mehlhorn, S. Näher, M. Seel, R. Seidel, T. Schilz, S. Schirra and C. Uhrig, *Checking Geometric Programs or Verification of Geometric Structures*, Comput. Geom. Theory Appl., Vol. 12, No. 1–2, 85–103, 1999.

42

Data Structures in C++

Mark Allen Weiss
Florida International University

42.1 Introduction

In C++, several classic data structures are implemented as a part of the standard library, commonly known as the *Standard Template Library*. The Standard Template Library (or simply, the STL) consists of a set of *container classes* (such as lists, sets, and maps), *iterator classes* that are used to traverse the container classes, and *generic algorithms*, such as sorting and searching. As its name implies, the library consists of (both function and class) templates. The STL is very powerful, and makes some use of advanced C++ features. Our discussion is focused on the most basic uses of the STL.

We partition our discussion of the Standard Template Library into the following sections.

1. Containers.
2. Iterators.
3. Generic Algorithms.

See [3–9] for additional material on the Standard Template Library.

42.2 Basic Containers

The STL defines several container templates that store collections of objects. Some collections are unordered, while others are ordered. Some collections allow duplicates, while others do not. All containers support the following operations:

- `int size( ) const`: returns the number of elements in the container.
- `void clear( )`: removes all elements from the container.

- `bool empty( ) const`: returns `true` if the container contains no elements and `false` otherwise.

There is no universal `add` or `insert` member function; different containers use different names.

The container classes can be split into three broad categories:

1. *Sequence containers*: maintains items with a notion of a position in the collection. Examples include `vector`, `list`, and `deque`.
2. *Sorted associative containers*: maintains items in sorted order. Examples include `set`, `multiset`, `map`, and `multimap`.
3. *Container adapters*: built on top of other containers to yield classic data structures. Examples include `stack`, `queue`, and `priority_queue`.

Associated with all containers is an *iterator*. An iterator represents the notion of a position in the container and is used to step through the container. All containers support the following operations:

- `iterator begin( )`: returns an appropriate iterator representing the first item in the container.
- `iterator end( )`: returns an appropriate iterator representing the end marker in the container (i.e. the position after the last item in the container).

We defer the discussion of iterators to Section 42.3.

42.2.1 Sequence Containers

The three basic sequence containers in the STL are the `vector`, `list`, and `deque`.

`vector` is a growable array. The `vector` wraps an internal array, maintaining a notion of its size, and internally its current capacity. If a sequence of additions would cause the size to exceed capacity, the capacity is automatically doubled. This makes additions at the end of the `vector` take constant amortized time. `list` is a doubly-linked list, in which pointers to both ends are maintained. `deque` is, in effect, a growable array that grows at both ends. There are several well-known ways of implement `deque` efficiently, but the standard is silent on which must be used.

For `vector`, an insertion or deletion takes *amortized time* that is proportional to the distance from the back, while for a `deque`, these operations take amortized time that is proportional to the smaller of the distances from the front and back. For a `list`, these operations take worst-case time that is proportional to the smaller of the distances from the front and back if an index is specified, but constant worst-case time if the position is specified by an existing iterator. `vector` and `deque` support indexing in constant worst-case time; `list` does not.

The basic operations that are supported by both containers are:

- `void push_back( const Object & x )`: adds `x` to the end of the container.
- `Object & back( )`: returns the object at the end of the container (an accessor that returns a constant reference is also provided).
- `void pop_back( )`: removes the object at the end of the container.
- `Object & front( )`: returns the object at the front of the container (an accessor that returns a constant reference is also provided).
- `iterator insert( iterator pos, const Object & x )`: adds `x` into the container, prior to the position given by the iterator `pos`. This is a constant-time

operation for `list`, but not for `vector` or `deque`. The return value is an iterator representing the position of the inserted item.

- `iterator erase( iterator pos )`: removes the object at the position given by the iterator. This is a constant-time operation for `list`, but not `vector` or `deque`. The return value is the position of the element that followed `pos` prior to the call. This operation invalidates `pos`, which is now stale.
- `iterator erase( iterator start, iterator end )`: removes all items beginning at position `start`, up to but not including `end`. Observe that an entire container can be erased by the call: `c.erase( c.begin( ), c.end( ) )`.

For `deque` and `list`, two additional operations are available with expected semantics:

- `void push_front( const Object & x )`: adds `x` to the front of the container.
- `void pop_front( )`: removes the object at the front of the container.

The `list` also provides a `splice` operation that allows the transfer of a sublist to somewhere else.

- `void splice( iterator pos, list & l, iterator start, iterator end )`: moves all items beginning at position `start`, up to but not including `end` to after `pos`. The moved items are assumed to have come from list `l`. The running time of this operation is proportional to the number of items moved, unless the items are moved within the same list (i.e. `&l==this`), in which case the running time is constant-time.

For `vector` and `deque`, additional operations include

- `Object & operator[] ( int idx )` returns the object at index `idx` in the container, with no bounds-checking (an accessor that returns a constant reference is also provided).
- `Object & at( int idx )`: returns the object at index `idx` in the container, with bounds-checking (an accessor that returns a constant reference is also provided).
- `int capacity( ) const`: returns the internal capacity.
- `void reserve( int newCapacity )`: for `vector` only, sets the new capacity. If a good estimate is available, it can be used to avoid array expansion.

42.2.2 Sorted Associative Containers

The STL provides two basic types of associative containers. Sets are used to store items. The `multiset` allows duplicates, while the `set` does not. Maps are used to store key/value pairs. The `multimap` allows duplicate keys, while the `map` does not. For sets, the items are logically maintained in sorted order, so an iterator to the "beginning item" represents the smallest item in the collection. For maps, the keys are logically maintained in sorted order. As a result, the most natural implementation of sets and maps make use of balanced binary search trees; typically a top-down red black tree (Chapter 10) is used.

The significant liability is that basic operations take logarithmic worst-case time. Some STL implementations (such as SGI's version) provide `hash_set` and `hash_map`, which make use of hash tables (Chapter 9) and provide constant average time per basic operation. A hash container will eventually make its way into the library, but at the time of this writing it is not Standard C++.

C++ sets and maps use either a default ordering (`operator<`) or one provided by a function object. In C++ a function object is implemented by providing a class that contains

an overloaded `operator()`, and then instantiating a template with the class name as a template parameter (an example of this is shown later in Figure 42.1).

Both sets and maps make use of the `pair` class template. The `pair` class template, stores two data members `first` and `second`, which can be directly accessed, without invoking member functions. Internally, maps store items of type `pair`. Additionally, several `set` member functions need to return two things; this is done by returning an appropriately instantiated `pair`.

Sets and Multisets

The `set` provides several member functions in addition to the usual suspects, including:

- `pair<iterator,bool> insert( const Object & x )`: adds `x` to the set. If `x` is not already present, the returned `pair` will contain the iterator representing the already contained `x`, and `false`. Otherwise, it will contain the iterator representing the newly inserted `x`, and `true`.
- `pair<iterator,bool> insert( iterator hint, const Object & x )`: same behavior as the one-parameter `insert`, but allows specification of a hint, representing the position where `x` should go. If the underlying implementation is a finger search tree (Chapter 11), or equivalent, the performance could be better than using the one-parameter `insert`.
- `iterator find( const Object & x ) const`: returns an iterator representing the position of `x` in the set. If `x` is not found, the endmarker is returned.
- `int erase( const Object & x )`: removes `x` (if present) and returns the number of items removed, which is either 0 or 1 in a `set`, and perhaps larger in a multiset.
- `iterator erase( iterator pos )`: same behavior as the sequence container version.
- `iterator erase( iterator start, iterator end )`: same behavior as the sequence container version.
- `iterator lower_bound( const Object & x )`: returns an iterator to the first element in the set with a key that is greater than or equal to `x`.
- `iterator upper_bound( const Object & x )`: returns an iterator to the first element in the set with a key that is greater than `x`.
- `pair<iterator,iterator> equal_range( const Object & x )`: returns a pair of iterators representing `lower_bound` and `upper_bound`.

A `multiset` is like a set except that duplicates are allowed. The return type of `insert` is modified to indicate that the insert always succeeds:

- `iterator insert( const Object & x )`: adds `x` to the multiset; returns an iterator representing the newly inserted `x`.
- `iterator insert( iterator hint, const Object & x )`: adds `x` to the multiset; returns an iterator representing the newly inserted `x`. Performance might be enhanced if `x` is inserted close to the position given by `hint`.

For the `multiset`, the `erase` member function that takes an `Object x` removes all occurrences of `x`. To simply remove one occurrence, first call `find` to obtain an iterator, and then use the `erase` member function that takes an iterator.

In a `multiset`, to find all occurrences of `x`, we cannot simply call `find`; that returns an iterator referencing one occurrence (if there is one), but which specific occurrence is

```
#include <set>
#include <string>
#include <iostream>
using namespace std;

class CaseInsensitive
{
  public:
    bool operator() ( const string & lhs, const string & rhs ) const
     { return stricmp( lhs.c_str( ), rhs.c_str( ) ) > 0; }
};

int main( )
{
   set<string>                 s1;
   set<string,CaseInsensitive> s2;

   s1.insert( "hello" );
   s1.insert( "world" );
   s1.insert( "HELLO" );
   s2.insert( "hello" );
   s2.insert( "world" );
   s2.insert( "HELLO" );

   cout << s1.size( ) << " " << s2.size( ) << endl;    // prints 3 2
   cout << (s1.find( "HeLlO" ) == s1.end( )) << endl;  // returns true
   cout << (s2.find( "HeLlO" ) == s2.end( )) << endl;  // returns false

   return 0;
}
```

FIGURE 42.1: Using the `set` class template.

returned is not guaranteed. Instead, the range returned by `lower_bound` and `upper_bound` (with `upper_bound` not included) contains all of occurrences of `x`; typically this is obtained by a call to `equal_range`.

Figure 42.1 illustrates two `set`s: `s1` which stores `string`s using the normal case-sensitive ordering, and `s2` which stores `string`s using case-insensitive comparisons. `s2` is instantiated by providing the class of a function object as a template parameter. As a result, `s1` contains three strings, but `s2` considers `"hello"` and `"HELLO"` to be identical, and thus only stores one of them.

Maps and Multimaps

A `map` behaves like a `set` instantiated with a `pair` representing a key and value, with a comparison function that refers only to the key. Thus it supports all of the `set` operations.

The `find` operation for maps requires only a key, but the iterator that it returns references a `pair`. Similarly, `erase` requires only a key, and otherwise behaves like the `set`'s `erase`. `insert` is supported, but to use `insert`, we must insert a properly instantiated `pair`, which is cumbersome, and thus rarely done. Instead, the `map` overloads the array indexing

`operator[]`:

- `ValueType & operator[] ( const KeyType & key )`: if the key is present in the map, a reference to the value is returned. If the key is not present in the map, it is inserted with a default value into the map, and then a reference to the inserted default value is returned. The default value is obtained by applying a zero-parameter constructor, or is zero for the primitive types.

The semantics of `operator[]` do not allow an accessor version of `operator[]`, and so `operator[]` cannot be used on an immutable `map`. For instance, if a `map` is passed by constant reference, inside the routine, `operator[]` is unusable. In such a case, we can use

```
#include <map>
#include <string>
#include <iostream>
using namespace std;

void addAlias( map<string,vector<string> > & aliasMap,
               const string & name,
               const string & alias )
{
   aliasMap[ name ].push_back( alias );
}

const vector<string> & getAliases(
               map<string,vector<string> > & aliasMap,
               const string & name )
{
   return aliasMap[ name ];
}

int main( )
{
   map<string,vector<string> > aliasMap;

   addAlias( aliasMap, "john", "john.doe@sillymail.com" );
   addAlias( aliasMap, "authors", "sahni@cise.ufl.edu" );
   addAlias( aliasMap, "authors", "dmehta@mines.edu" );
   addAlias( aliasMap, "authors", "weiss@fiu.edu" );

   cout << "authors is aliased to: ";
   const vector<string> & authors = getAliases( aliasMap, "authors" );
   for( int i = 0; i < authors.size( ); i++ )
       cout << authors[ i ] << " ";
   cout << endl;

   return 0;
}
```

FIGURE 42.2: Using the `map` class template.

`find` to obtain an iterator, test if the iterator represents the endmarker (in which case the `find` has failed), and if the iterator is valid, we can use it to access the `pair`'s `second` component, which is the value. (This alternate idiom is eventually seen in the larger example in Section 42.5).

A `multimap` is a `map` in which duplicate keys are allowed. `multimaps` behave like `maps` but do not support `operator[]`.

Figure 42.2 illustrates a combination of `map` and `vector` in which the keys are email aliases, and the values are the email addresses corresponding to each alias. Since there can be several addresses for each alias, the values are themselves `vectors` of `strings`. The trickiest part (other than the various parameter-passing modes) concerns the one-line implementation of `addAlias`. There, we see that `aliasMap[name]` refers to the `vector` value in the `map` for the `name` key. If `name` is not in the `map`, then the call to `operator[]` causes it to be placed into the `map`, with a default value being a zero-sized `vector`. Thus whether or not `name` was in the `map` prior to calling `addAlias`, after the access of the `map`, the call to `push_back` updates the `vector` value correctly.

42.2.3 Container Adapters

The STL provides adapter class templates to implement stacks, queues, and priority queues. These class templates are instantiated with the type of object to be stored and (optionally, if the default is unacceptable) the container class (such as `vector`, `list`, or `deque`) that is used to store the objects. Thus we can specify a linked list or array implementation for a stack, though in reality, this feature does little more than add nicer names than `push_back` to the (existing `list` or `vector`, respectively) interface.

`stack` and `queue`

For `stack`, the member functions are `push`, `pop`, and `top`. For `queue`, we get `push`, `front`, and `pop`. By default, a `vector` is used for `stack` and a `deque` for `queue`. Figure 42.3 shows how to use the `queue` adapter, implemented with a doubly-linked list.

```
#include <queue>
#include <iostream>
#include <list>
using namespace std;

int main( )
{
    queue<int,list<int> > q;
    q.push( 3 ); q.push( 7 );

    for( ; !q.empty( ); q.pop( ) )
        cout << q.front( ) << endl;

    return 0;
}
```

FIGURE 42.3: Using the `queue` adapter.

`priority_queue`

The `priority_queue` template contains member functions named `push`, `top`, and `pop`, that mirror `insert`, `findmax`, and `deletemax` in a classic max-heap.

Sometimes priority queues are set up to remove and access the smallest item instead of the largest item. In such a case, the priority queue can be instantiated with an appropriate `greater` function object to override the default ordering. The priority queue template is instantiated with an item type, the container type (as in `stack` and `queue`), and the comparator, with defaults allowed for the last two parameters. In Figure 42.4, the default instantiation of `priority_queue` allows access to the largest items, whereas an instantiation with a `greater` function object reverses the comparison order and allows access to the smallest item.

42.3 Iterators

The *iterator* in C++ abstracts the notion of a position in the container. As we have already seen, there are several ways to obtain such a position. All containers provide `begin` and `end` member functions that return iterators representing the first item and endmarker, respectively. Many containers provide a searching member function (e.g. `set::find`) that returns an iterator viewing the found item, or the endmarker if the item is not found.

42.3.1 Basics

Throughout our discussion we have used `iterator` as a type. In reality, in C++, each container defines several iterator types, nested in the scope of the container, and these specific iterator types are used by the programmer instead of simply using the word `iterator`. For instance, if we have a `vector<int>`, the basic iterator type is `vector<int>::iterator`. The basic iterator can be used to traverse and change the contents of the container.

Another iterator type, `vector<int>::const_iterator`, does not allow changes to the container on which the iterator is operating. All iterators are guaranteed to have at least the following set of operations:

- `++itr` and `itr++`: advance the iterator `itr` to the next location. Both the prefix and postfix forms are available. This does not cause any change to the container.
- `*itr`: returns a reference to the container object that `itr` is currently representing. The reference that is returned is modifiable for basic iterators, but is not modifiable (i.e. a constant reference) for `const_iterator`s.
- `itr1==itr2`: returns `true` if iterators `itr1` and `itr2` refer to the same position in the same container and `false` otherwise.
- `itr1!=itr2`: returns `true` if iterators `itr1` and `itr2` refer to different positions or different containers and `false` otherwise.

Some iterators efficiently support `--itr` and `itr--`. Those iterators are called *bidirectional iterators*. All of the iterators for the common containers `vector`, `list`, `deque`, `set`, and `map` are bidirectional.

Some iterators efficiently support both `itr+=k` and `itr+k`. Those iterators are called *random-access iterators*. `itr+=k` advances the iterator `k` positions. `itr+k` returns a new iterator that is `k` positions ahead of `itr`. Also supported by random-access iterators are `itr-=k` and `itr-k`, with obvious semantics, and `itr1-itr2` which yields a separation distance as an integer. The iterators for `vector` and `deque` support random access.

```
#include <vector>
#include <queue>
#include <functional>
#include <string>
#include <iostream>
using namespace std;

// Empty the priority queue and print its contents.
template <typename PQueue>
void dumpPQ( const string & msg, PQueue & pq )
{
    if( pq.empty( ) )
        cout << msg << " is empty" << endl;
    else
    {
        cout << msg << ": " << pq.top( );
        pq.pop( );
        while( !pq.empty( ) )
        {
            cout << " " <<  pq.top( );
            pq.pop( );
        }
        cout << endl;
    }
}

int main( )
{
    priority_queue<int>                              maxpq;
    priority_queue<int,vector<int>,greater<int> > minpq;

    minpq.push( 3 ); minpq.push( 7 ); minpq.push( 3 );
    maxpq.push( 3 ); maxpq.push( 7 ); maxpq.push( 3 );

    dumpPQ( "minpq", minpq );    // minpq: 3 3 7
    dumpPQ( "maxpq", maxpq );    // maxpq: 7 3 3

    return 0;
}
```

FIGURE 42.4: Using the `priority_queue` adapter.

All C++ containers have two member functions, `begin` and `end` that return iterators. Each collection defines four member functions:

- `iterator begin( )`
- `const_iterator begin( ) const`
- `iterator end( )`

- `const_iterator end( ) const`

`begin` returns an iterator that is positioned at the first item in the container. `end` returns an iterator that is positioned at the endmarker, which represents a position one past the last element in the container. On an empty container, `begin` and `end` return the same position. For random access iterators, the result of subtracting the return values of `end` and `begin` is always the size of the container.

Typically we initialize a local iterator to be a copy of the `begin` iterator, and have it step through the container, stopping as soon as it hits the endmarker.

As an example, Figure 42.5 shows a `print` function that prints the elements of any container, provided that the elements in the container have provided an `operator<<`. Note that we must use a `const_iterator` to traverse the container, because the container is itself immutable in the scope of `print`. The test program illustrates five different containers that invoke the `print` function, along with the expected output (in comments). Observe that both `set` and `multiset` output in sorted order, with `multiset` allowing the second insertion of `beta`. Additionally, for `map` to be compatible with the `print` routine, we must provide an `operator<<` that works for the elements of the `map`, namely the appropriate `pair`s.

42.3.2 Reverse Iterators

Sometimes it is important to be able to traverse a container in the reverse direction. Because of the asymmetric nature of `begin` and `end` (representing the first element and the endmarker, rather than the last element, respectively) this is cumbersome to do, even for bidirectional iterators. As a result, containers that support bidirectional iterators typically also provide a *reverse iterator*. The reverse iterator comes in two flavors: `reverse_iterator` and `const_reverse_iterator`. For reverse iterators, `++` retreats one position toward the beginning, while `--` advances one position toward the end. Container member functions `rbegin` and `rend` are used to obtain iterators representing the last element and the beginmarker, respectively. As a result, the code in Figure 42.6 prints any container in reverse order.

42.4 Additional Components of the STL

42.4.1 Sorting, Searching, and Selection

The Standard Library includes a rich set of functions that can be applied to the standard containers. Some of the algorithms include routines for sorting, searching, copying (possibly with substitutions), shuffling, reversing, rotating, merging, and so on. In all there are over sixty generic algorithms. In this section we highlight those related to efficient sorting, selection, and binary search.

Sorting

Sorting in C++ is accomplished by use of function template `sort`. The parameters to `sort` represent the start and endmarker of a (half-open range in a) container, and an optional comparator:

- `void sort( iterator begin, iterator end )`
- `void sort( iterator begin, iterator end, Comparator cmp )`

The iterators must support random access. As an example, in

```
#include <iostream>
#include <vector>
#include <list>
#include <set>
#include <string>
#include <map>
using namespace std;

template <typename Container>
void print( const Container & c, ostream & out = cout )
{
    typename Container::const_iterator itr;

    for( itr = c.begin( ); itr != c.end( ); ++itr )
        out << *itr << " ";
    out << endl;
}

template <typename Type1, typename Type2>
ostream & operator<<( ostream & out, const pair<Type1,Type2> & p )
{
    return out << "[" << p.first << "," << p.second << "]";
}

int main( )
{
    vector<int> vec;
    vec.push_back( 3 ); vec.push_back( 7 );

    list<double> lst;
    lst.push_back( 3.14 ); lst.push_front( 6.28 );

    set<string> s;
    s.insert( "beta" ); s.insert( "alpha" ); s.insert( "beta" );

    multiset<string> ms;
    ms.insert( "beta" ); ms.insert( "alpha" ); ms.insert( "beta" );

    map<string,string> zip;
    zip.insert( pair<string,string>( "Miami", "33199" ) );
    zip.insert( pair<string,string>( "Gainesville", "32611" ) );
    zip[ "Golden" ] = "80401";

    print( vec );     // 3 7
    print( lst );     // 6.28 3.14
    print( s );       // alpha beta
    print( ms );      // alpha beta beta
    print( zip );     // [Gainesville,32611] [Golden,80401] [Miami,33199]

    return 0;
}
```

FIGURE 42.5: Generic printing routine that works with five different containers.

```
#include <iostream>
using namespace std;

template <typename Container>
void printReverse( const Container & c, ostream & out = cout )
{
    typename Container::const_reverse_iterator itr;

    for( itr= c.rbegin( ); itr != c.rend( ); ++itr )
        out << *itr << " ";
    out << endl;
}
```

FIGURE 42.6: Printing a container in reverse.

```
sort( v.begin( ), v.end( ) );
sort( v.begin( ), v.end( ), greater<int>( ) );
sort( v.begin( ), v.begin( ) + ( v.size ( ) >= 5 ? 5 : 0 ) );
```

the first call sorts the entire container, `v`, in non-decreasing order. The second call sorts the entire container in non-increasing order. The third call sorts the first five elements of the container in non-decreasing order. The sorting algorithm is generally quicksort, which yields an $O(N \log N)$ algorithm on average. However, $O(N \log N)$ worst-case performance is not guaranteed.

The C++ library inherits the sorting routine `qsort` from C. `qsort` uses pointers to functions to perform its comparison making it significantly slower on average than the `sort` routine. Furthermore, many implementations use a version of quicksort that has been shown to provide quadratic behavior on some commonly occurring inputs [1]. Avoid using `qsort`.

Selection

The function template `nth_element` is used for selection, and as expected has $O(N)$ average-case running time. The parameters include a pair of iterators, and k:

- `void nth_element( iterator begin, int k, iterator end )`
- `void nth_element( iterator begin, int k, iterator end, Comparator c)`

As a result of calling `nth_element`, the item in the kth position is the kth smallest element, where counting as usual in C++ begins at 0.

Thus, in

```
nth_element( v.begin( ), 0, v.end( ) );
nth_element( v.begin( ), 0, v.end( ), greater<int>( ) );
nth_element( v.begin( ), v.begin( ) + ( v.end( ) - vbegin( ) / 2 ), v.end( ) );
```

the first call places the smallest element in the position given by `v.begin( )`, the second call places the largest element in that position, and the third call places the median in the middle position.

Searching

Several generic searching algorithms are available for containers. The most basic is:

- `iterator find( iterator begin, iterator end, const Object & x )`: returns an iterator representing the first occurrence of `x` in the half-open range specified by `begin` and `end`, or `end` if `x` is not found.

`find` is simply a sequential search. If the range is sorted, `binary_search` can be used to find an object in logarithmic time. A comparator can be provided, or the default ordering can be used. The signatures for `binary_search` are:

- `iterator binary_search( iterator begin, iterator end, const Object & x )`
- `iterator binary_search( iterator begin, iterator end, const Object & x, Comparator cmp )`

`equal_range`, `lower_bound`, and `upper_bound` search sorted ranges and behave with the same semantics as the identically named member functions in `set`.

42.4.2 Non-Standard Extensions

Some implementations of the STL contain additional classes. Eventually, these might become part of C++.

A `rope` is a container that behaves like a `string` but is optimized for large strings. The implementation of a `rope` (the name is a play on *long string*) makes use of trees of reference-counted substrings, and allows operations such as assignment, concatenation, and substring to be performed more efficiently than with a regular `string`.

A `slist` is a singly-linked list. Because in a standard `list`, `insert` and `erase` add and remove prior to the position given in the iterator, those operations are linear-time in an `slist` in some implementations. If so, you'll find additional member functions `insert_after` and `erase_after` that provide constant-time performance.

The `hash_set`, `hash_map`, `hash_multiset`, and `hash_multimap` containers support sets and maps at $O(1)$ average time per operation, using hash tables. Generally speaking, these are instantiated with a type of object being stored, a comparison function (that defaults to `==`), and a hash function. Unfortunately, there are several competing designs which are not compatible with each other, and it remains to be seen which design will eventually be chosen to join the Standard Library.

42.5 Sample Code

To illustrate some of the STL routines in one place, we provide an implementation of a routine that finds *word ladders*. A word ladder transforms one word to another by changing one character at a time, with each change always resulting in a real word. For instance, we transform `those` to `where` using the word ladder: `those`, `these`, `there`, `where`. See [2] for a description of this problem. The word ladder problem is a standard unweighted shortest path problem, and in the typical application of five-letter words, a quadratic algorithm is acceptable. Figures 42.7 to 42.9 show some of the routines (omitting error checks, function prototypes, and a `main` that reads the word list and prompt for word pairs), which for the most part are straightforward.

`computeAdjacentWords`, shown in Figure 42.7, uses the same idiom seen in Figure 42.2 (dealing with email aliases).

In Figure 42.8 we see two versions of `findChain`. The return value for `findChain` is a `map` in which the key is a word, and the value is an adjacent word on the chain back towards

```
#include <vector>
#include <map>
#include <string>
#include <queue>
#include <algorithm>
using namespace std;

// Returns true if word1 and word2 are same length
// and differ in only one character.
bool oneCharOff( const string & word1, const string & word2 )
{
    if( word1.length( ) != word2.length( ) )
        return false;

    int diffs = 0;

    for( int i = 0; i < word1.length( ); i++ )
        if( word1[ i ] != word2[ i ] )
            if( ++diffs > 1 )
                return false;

    return diffs == 1;
}

// Computes a map in which the keys are words and values are vectors of words
// that differ in only one character from the corresponding key.
map<string,vector<string> > computeAdjacentWords( const vector<string> & words )
{
    map<string,vector<string> > adjacentWords;

    for( int i = 0; i < words.size( ); i++ )
        for( int j = i+1; j < words.size( ); j++ )
            if( oneCharOff( words[ i ], words[ j ] ) )
            {
                adjacentWords[ words[ i ] ].push_back( words[ j ] );
                adjacentWords[ words[ j ] ].push_back( words[ i ] );
            }

    return adjacentWords;
}
```

FIGURE 42.7: Word ladders ($O(N^2)$, and with `main` omitted (part 1 of 3)).

the first word (or `""` if there is no chain, or if the key is `first`). The shorter version of `findChain` takes a `vector` containing the words, computes the `map` of adjacent words, and then invokes the longer version of `findChain`. Because the `map` in `findChain` is passed by constant reference, `operator[]` is not available, and instead we access values in the `map` by using the `iterator` returned by a call to `find`.

```
// Runs the shortest path calculation from the original set of words, returning
// a vector that contains the sequence of word changes to get from first to
// second. Since this calls computeAdjacentWords, it is recommended that the
// user instead call computeAdjacents once and then the other findChain below
// for each word pair.
vector<string> findChain( const vector<string> & words,
                          const string & first,
                          const string & second )
{
    map<string,vector<string> > adjacentWords = computeAdjacentWords( words );
    return findChain( adjacentWords, first, second );
}

// Runs the shortest path calculation from the adjacency map, returning a vector
// that contains the sequence of word changes to get from first to second.
vector<string> findChain( const map<string,vector<string> > & adjacentWords,
                          const string & first,
                          const string & second )
{
    map<string,string> previousWord;
    queue<string> q;

    q.push( first );
    while( !q.empty( ) )
    {
        string current = q.front( ); q.pop( );

        map<string,vector<string> >::const_iterator itr;

        itr = adjacentWords.find( current );
        if( itr != adjacentWords.end( ) )
        {
            const vector<string> & adj = itr->second;
            for( int i = 0; i < adj.size( ); i++ )
            if( previousWord[ adj[ i ] ] == "" )
            {
                previousWord[ adj[ i ] ] = current;
                q.push( adj[ i ] );
            }
        }
    }

    return getChainFromPreviousMap( previousWord, first, second );
}
```

FIGURE 42.8: Word ladders ($O(N^2)$, and with `main` omitted (part 2 of 3)).

```
// After the shortest path calculation has run, computes the vector that
// contains the sequence of word changes to get from first to second.
vector<string> getChainFromPreviousMap( map<string,string> & previousWord,
                                        const string & first,
                                        const string & second )
{
    vector<string> result;
    if( previousWord[ second ] != "" )
    {
        string current = second;
        while( current != first )
        {
            result.push_back( current );
            current = previousWord[ current ];
        }
        result.push_back( first );
    }

    reverse( result.begin( ), result.end( ) );
    return result;
}
```

FIGURE 42.9: Word ladders ($O(N^2)$, and with `main` omitted (part 3 of 3)).

Finally, in Figure 42.9 we see the routine that returns, as a `vector<string>`, the chain of words given the `map` computed by the shortest path algorithm in `findChain`. The function `getChainFromPreviousMap` passes the first parameter by reference instead of the more natural constant reference to allow the use of `operator[]` to access the map. Although this is simpler to code, it is arguably inferior from a software design standpoint because it changes the parameter passing mode simply to allow expedient code.

Acknowledgment

This work was supported, in part, by the National Science Foundation under grant EIA-9906600. Parts of this chapter are based upon the author's work in [9].

References

[1] J. L. Bentley and M. D. McIlroy, Engineering a Sort Function, *Software - Practice and Experience* **23** (1993), 1249-1265.

[2] D. E. Knuth, *The Stanford Graphbase*, Addison-Wesley, Reading, Mass., 1993.

[3] J. Lajoie and S. Lippman, *C++ Primer*, Third Edition, Addison-Wesley, Reading, Mass., 1998.

[4] S. Meyers, *Effective STL*, Addison-Wesley, Reading, Mass., 2001.

[5] D. R. Musser and A. Saini, *C++ Programming with the Standard Template Library*, Addison-Wesley, Reading, Mass., 1996.

[6] B. Stroustrop, *The C++ Programming Language*, Third Edition, Addison-Wesley, Reading, Mass., 1997.

[7] B. Stroustrop, *The Design and Evolution of C++*, Addison-Wesley, Reading, Mass., 1994.

[8] M. A. Weiss, *Data Structures and Problem Solving Using C++*, Second Edition, Addison-Wesley, Reading, Mass., 2000.

[9] M. A. Weiss, *C++ for Java Programmers*, Prentice Hall, Englewood Cliffs, NJ., 2004. Chapter 10.

43

Data Structures in JDSL

Michael T. Goodrich
University of California, Irvine

Roberto Tamassia
Brown University

Luca Vismara
Brown University

43.1 Introduction

In the last four decades the role of computers has dramatically changed: once mainly used as *number processors* to perform fast numerical computations, they have gradually evolved into *information processors*, used to store, analyze, search, transfer, and update large collections of structured information. In order for computer programs to perform these tasks effectively, the data they manipulate must be well organized, and the methods for accessing and maintaining those data must be reliable and efficient. In other words, computer programs need advanced *data structures* and *algorithms*. Implementing advanced data structures and algorithms, however, is not an easy task and presents some risks:

Complexity Advanced data structures and algorithms are often difficult to understand thoroughly and to implement.

Unreliability Because of their complexity, the implementation of advanced data structures and algorithms is prone to subtle errors in boundary cases, which may require a considerable effort to identify and correct.

Long development time As a consequence, implementing and testing advanced data structures and algorithms is usually a time consuming process.

As a result, programmers tend to ignore advanced data structures and algorithms and to resort to simple ones, which are easier to implement and test but that are usually not as efficient. It is thus clear how the development of complex software applications, in particular their rapid prototyping, can greatly benefit from the availability of libraries of reliable and efficient data structures and algorithms.

Various libraries are available for the C++ programming language. They include the *Standard Template Library* (STL, see Chapter 42) [9], now part of the C++ standard, the

extensive *Library of Efficient Data Structures and Algorithms* (LEDA, see Chapter 41) [8], and the *Computational Geometry Algorithms Library* (CGAL) [3].

The situation for the Java programming language is the following: a small library of data structures, usually referred to as *Java Collections* (JC), is included in the java.util package of the Java 2 Platform.* An alternative to the Java Collections are the *Java Generic Libraries* (JGL) by Recursion Software, Inc.,† which are patterned after STL. Both the Java Collections and JGL provide implementations of basic data structures such as sequences, sets, maps, and dictionaries. JGL also provides a considerable number of generic programming algorithms for transforming, permuting, and filtering data.

None of the above libraries for Java, however, seems to provide a coherent framework, capable of accommodating both elementary and advanced data structures and algorithms, as required in the development of complex software applications. This circumstance motivated the development of the *Data Structures Library in Java* (JDSL) [10], a collection of Java interfaces and classes implementing fundamental data structures and algorithms, such as:

- sequences and trees;
- priority queues, binary search trees, and hash tables;
- graphs;
- sorting and traversal algorithms;
- topological numbering, shortest path, and minimum spanning tree.

JDSL is suitable for use by researchers, professional programmers, educators, and students. It comes with extensive documentation, including detailed Javadoc,‡ an overview, a tutorial with seven lessons, and several associated research papers. It is available free of charge for noncommercial use at http://www.jdsl.org/.

The development of JDSL began in September 1996 at the Center for Geometric Computing at Brown University and culminated with the release of version 1.0 in 1998. A major part of the project in the first year was the experimentation with different models for data structures and algorithms, and the construction of prototypes. A significant reimplementation, documentation, and testing [1] effort was carried out in 1999 leading to version 2.0, which was officially released in 2000. Starting with version 2.1, released in 2003 under a new license, the source code has been included in the distribution. During its life cycle JDSL 2.0 was downloaded by more than 5,700 users, while JDSL 2.1 has been downloaded by more than 3,500 users as of this writing. During these seven years a total of 25 people§ have been involved, at various levels, in the design and development of the library. JDSL has been used in data structures and algorithms courses worldwide as well as in two data structures and algorithms textbooks¶ written by the first two authors [6, 7].

In the development of JDSL we tried to learn from other approaches and to progress on them in terms of ease of use and modern design. The library was designed with the following goals in mind:

Functionality The library should provide a significant collection of existing data structures and algorithms.

*http://java.sun.com/j2se/
†http://www.recursionsw.com/jgl.htm
‡http://java.sun.com/j2se/javadoc/
§http://www.jdsl.org/team.html
¶http://java.datastructures.net/ and http://algorithmdesign.net/

Reliability Data structures and algorithms should be correctly implemented, with particular attention to boundary cases and degeneracies. All input data should be validated and, where necessary, rejected by means of exceptions.

Efficiency The implementations of the data structures and algorithms should match their theoretical asymptotic time and space complexity; constant factors, however, should also be considered when evaluating efficiency.

Flexibility Multiple implementations of data structures and algorithms should be provided, so that the user can experiment and choose the most appropriate implementation, in terms of time or space complexity, for the application at hand. It should also be possible for the user to easily extend the library with additional data structures and algorithms, potentially based on the existing ones.

Observe that there exist some trade-offs between these design goals, e.g., between efficiency and reliability, or between efficiency and flexibility.

In JDSL each data structure is specified by an interface and each algorithm uses data structures only via the interface methods. Actual classes need only be specified when objects are instantiated. Programming through interfaces, rather than through actual classes, creates more general code. It allows different implementations of the same interface to be used interchangeably, without having to modify the algorithm code.

A comparison of the key features of the Java Collections, JGL, and JDSL is shown in Table 43.1. The main advantages of JDSL are the definition of a large set of data structure APIs (including binary tree, general tree, priority queue and graph) in terms of Java interfaces, the availability of reliable and efficient implementations of those APIs, and the presence of some fundamental graph algorithms. Note, in particular, that the Java Collections do not include trees, priority queues and graphs, and provide only sorting algorithms.

	JC	JGL	JDSL
Sequences (lists, vectors)	✓	✓	✓
Trees			✓
Priority queues (heaps)		✓	✓
Dictionaries (hash tables, red-black trees)	✓		✓
Sets	✓		
Graphs			✓
Templated algorithms			✓
Sorting	✓	✓	✓
Data transforming, permuting, and filtering		✓	
Graph traversals			✓
Topological numbering			✓
Shortest path			✓
Minimum spanning tree			✓
Accessors (positions and locators)			✓
Iterators	✓	✓	✓
Range views	✓	✓	
Decorations (attributes)			✓
Thread-safety	✓		
Serializability	✓	✓	

TABLE 43.1 A comparison of the Java Collections (JC), the Generic Library for Java (JGL), and the Data Structures Library in Java (JDSL).

A good library of data structures and algorithms should be able to integrate smoothly with other existing libraries. In particular, we have pursued compatibility with the Java Collections. JDSL supplements the Java Collections and is not meant to replace them. No conflicts arise when using data structures from JDSL and from the Java Collections in the

same program. To facilitate the use of JDSL data structures in existing programs, adapter classes are provided to translate a Java Collections data structure into a JDSL one and vice versa, whenever such a translation is applicable.

43.2 Design Concepts in JDSL

In this section we examine some data organization concepts and algorithmic patterns that are particularly important in the design of JDSL.

43.2.1 Containers and Accessors

In JDSL each data structure is viewed as a *container*, i.e., an organized collection of objects, called the *elements* of the container. An element can be stored in many containers at the same time and multiple times in the same container. JDSL containers can store heterogeneous elements, i.e., instances of different classes.[‖]

JDSL provides two general and implementation-independent ways to access (but not modify) the elements stored in a container: individually, by means of accessors, and globally, by means of iterators (see Section 43.2.2). An *accessor* [5] abstracts the notion of membership of an element into a container, hiding the details of the implementation. It provides constant-time access to an element stored in a container, independently from its implementation. Every time an element is inserted in a container, an accessor associated with it is returned. Most operations on JDSL containers take one or more accessors as their operands.

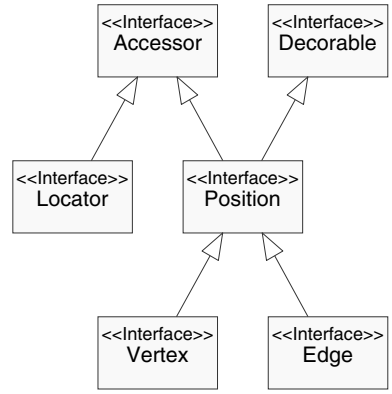


FIGURE 43.1: The accessors interface hierarchy.

We distinguish between two kinds of containers and, accordingly, of accessors (see Figure 43.1 for a diagram of the accessor interface hierarchy):

Positional containers Typical examples are sequences, trees, and graphs. In a positional container, some topological relation is established among the "place-

[‖]This is possible since in Java every class extends (directly or indirectly) java.lang.Object.

holders" that store the elements, such as the predecessor-successor relation in a sequence, the parent-child relation in a tree, and the incidence relation in a graph. It is the user who decides, when inserting an element in the container, what the relationship is between the new "placeholder" and the existing ones (in a sequence, for instance, the user may decide to insert an element before a given "placeholder"). A positional container does not change its topology, unless the user requests a change specifically. The implementation of these containers usually involves linked structures or arrays.

Positions The concept of position is an abstraction of the various types of "placeholders" in the implementation of a positional container (typically the nodes of a linked structure or the cells of an array). Each position stores an element. Position implementations may store the following additional information:

- the adjacent positions (e.g., the previous and next positions in a sequence, the right and left child and the parent in a binary tree, the list of incident edges in a graph);
- consistency information (e.g., what container the position is in, the number of children in a tree).

A position can be directly queried for its element through method element(), which hides the details of where the element is actually stored, be it an instance variable or an array cell. Through the positional container, instead, it is possible to replace the element of a position or to swap the elements between two positions. Note that, as an element moves about in its container, or even from container to container, its position changes. Positions are similar to the concept of items used in LEDA [8].

Key-based containers Typical examples are dictionaries and priority queues. Every element stored in a key-based container has a *key* associated with it. Keys are used as an indexing mechanism for their associated elements. Typically, a key-based container is internally implemented using a positional container; for example, a possible implementation of a priority queue uses a binary tree (a heap). The details of the internal representation, however, are completely hidden to the user. Thus, the user has no control over the organization of the positions that store the key/element pairs. It is the key-based container itself that modifies its internal representation based on the keys of the key/element pairs inserted or removed.

Locators The key/element pairs stored in a key-based container may change their positions in the underlying positional container, due to some internal restructuring, say, after the insertion of a new key/element pair. For example, in the binary tree implementation of a priority queue, the key/element pairs move around the tree to preserve the top-down ordering of the keys, and thus their positions change. Hence, a different, more abstract type of accessor, called locator, is provided to access a key/element pair stored in a key-based container. Locators hide the complications of dynamically maintaining the implementation-dependent binding between the key/element pairs and their positions in the underlying positional container.

A locator can be directly queried for its key and element, and through the key-based container it is possible to replace the key and the element of a locator. An example of using locators is given in Section 43.4.

43.2.2 Iterators

While accessors allow users to access single elements or key/element pairs in a container, *iterators* provide a simple mechanism for iteratively listing through a collection of objects. JDSL provides various iterators over the elements, the keys (where present), and the positions or the locators of a container (see Figure 43.2 for a diagram of the iterator interface hierarchy). They are similar to the iterators provided by the Java Collections.

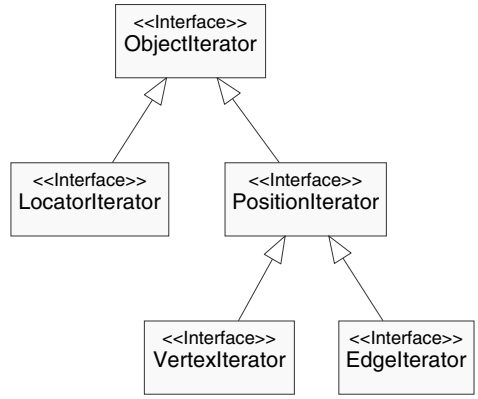


FIGURE 43.2: The iterators interface hierarchy.

All JDSL containers provide methods that return iterators over the entire container (e.g., all the positions of a tree or all the locators of a dictionary). In addition, some methods return iterators over portions of the container (e.g., the children of a position of a tree or the locators with a given key in a dictionary). JDSL iterators can be traversed only forward; however, they can be reset to start a new traversal.

For simplicity reasons iterators in JDSL have *snapshot* semantics: they refer to the state of the container at the time the iterator was created, regardless of the possible subsequent modifications of the container. For example, if an iterator is created over all the positions of a tree and then a subtree is cut off, the iterator will still include the positions of the removed subtree.

43.2.3 Decorations

Another feature of JDSL is the possibility to "decorate" individual positions of a positional container with attributes, i.e., with arbitrary objects. This mechanism is more convenient and flexible than either subclassing the position class to add new instance variables or creating global hash tables to store the attributes. Decorations are useful for storing temporary or permanent results of the execution of an algorithm. For example, in a depth-first search (DFS) traversal of a graph, we can use decorations to (temporarily) mark the vertices being visited and to (permanently) store the computed DFS number of each vertex. An example of using decorations is given in Section 43.4.

43.2.4 Comparators

When using a key-based container, the user should be able to specify the comparison relation to be used with the keys. In general, this relation depends on the type of the keys and on the specific application for which the key-based container is used: keys of the same type may be compared differently in different applications. One way to fulfill this requirement is to specify the comparison relation through a *comparator* object, which is passed to the key-based container constructor and is then used by the key-based container every time two keys need to be compared.

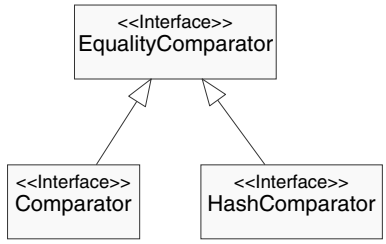


FIGURE 43.3: The comparators interface hierarchy.

Three comparator interfaces are defined in JDSL (see Figure 43.3 for a diagram of the comparators interface hierarchy). The concept of comparator is present also in the java.util package of the Java 2 Platform, where a Comparator interface is defined.

43.2.5 Algorithms

JDSL views algorithms as objects that receive the input data as arguments of their execute(.) method, and provide access to the output during or after the execution via additional methods. Most algorithms in JDSL are implemented following the *template method pattern* [4]. The invariant part of the algorithm is implemented once in an abstract class, deferring the implementation of the steps that can vary to subclasses. These varying steps are defined either as abstract methods (whose implementation must be provided by a subclass) or as "hook" methods (whose default implementation may be overridden in a subclass). In other words, algorithms perform "generic" computations that can be specialized for specific tasks by subclasses.

An example of applying the template method pattern is given in Section 43.4, where we use the JDSL implementation of Dijkstra's single-source shortest path algorithm [2]. The algorithm refers to the edge weights by means of an abstract method that can be specialized depending on how the weights are actually stored or computed in the application at hand.

43.3 The Architecture of JDSL

In this section we describe the interfaces of the data structures present in JDSL, the classes that implement those interfaces, and the algorithms that operate on them. Most containers are described by two interfaces, one (whose name is prefixed with Inspectable) that comprise all the methods to query the container, and the other, extending the first, that comprise all the methods to modify the container. Inspectable interfaces can be used as variable

or argument types in order to obtain an immutable view of a container (for instance, to prevent an algorithm from modifying the container it operates on).

As described in Section 43.2.1, we can partition the set of containers present in JDSL into two subsets: the positional containers and the key-based containers. Accordingly, the interfaces for the various containers are organized into two hierarchies (see Figures 43.4 and 43.5), with a common root given by interfaces InspectableContainer and Container. At the same time, container interfaces, their implementations, and algorithms that operate on them are grouped into various Java packages.

In the rest of this section, we denote with n the current number of elements stored in the container being considered.

43.3.1 Packages

JDSL currently consists of eight Java packages, each containing a set of interfaces and/or classes. Interfaces and exceptions for the data structures are defined in packages with the api suffix, while the reference implementations of these interfaces are defined in packages with the ref suffix. Interfaces, classes, and exceptions for the algorithms are instead grouped on a functional basis. As we will see later, the interfaces are arranged in hierarchies that may extend across different packages. The current packages are the following:

jdsl.core.api Interfaces and exceptions that compose the API for the core containers (sequences, trees, priority queues, and dictionaries), for their accessors and comparators, and for the iterators on their elements, positions and locators.

jdsl.core.ref Implementations of the interfaces in jdsl.core.api. Most implementations have names of the form *ImplementationStyleInterfaceName*. For instance, ArraySequence and NodeSequence implement the jdsl.core.api.Sequence interface with a growable array and with a linked structure, respectively. Classes with names of the form Abstract*InterfaceName* implement some methods of the interface for the convenience of developers building alternative implementations.

jdsl.core.algo.sorts Sorting algorithms that operate on the elements stored in a jdsl.core.api.Sequence object. They are parameterized with respect to the comparison rule used to sort the elements, provided as a jdsl.core.api.Comparator object.

jdsl.core.algo.traversals Traversal algorithms that operate on jdsl.core.api.InspectableTree objects. A traversal algorithm performs operations while visiting the nodes of the tree, and can be extended by applying the template method pattern.

jdsl.core.util This package contains a Converter class to convert some JDSL containers to the equivalent data structures of the Java Collections and vice versa.

jdsl.graph.api Interfaces and exceptions that compose the API for the graph container and for the iterators on its vertices and edges.

jdsl.graph.ref Implementations of the interfaces in jdsl.graph.api; in particular, class IncidenceListGraph is an implementation of interface jdsl.graph.api.Graph.

jdsl.graph.algo Basic graph algorithms, including depth-first search, topological numbering, shortest path, and minimum spanning tree, all of which can be extended by applying the template method pattern.

43.3.2 Positional Containers

All positional containers implement interfaces InspectablePositionalContainer and PositionalContainer, which extend InspectableContainer and Container, respectively (see Figure 43.4).

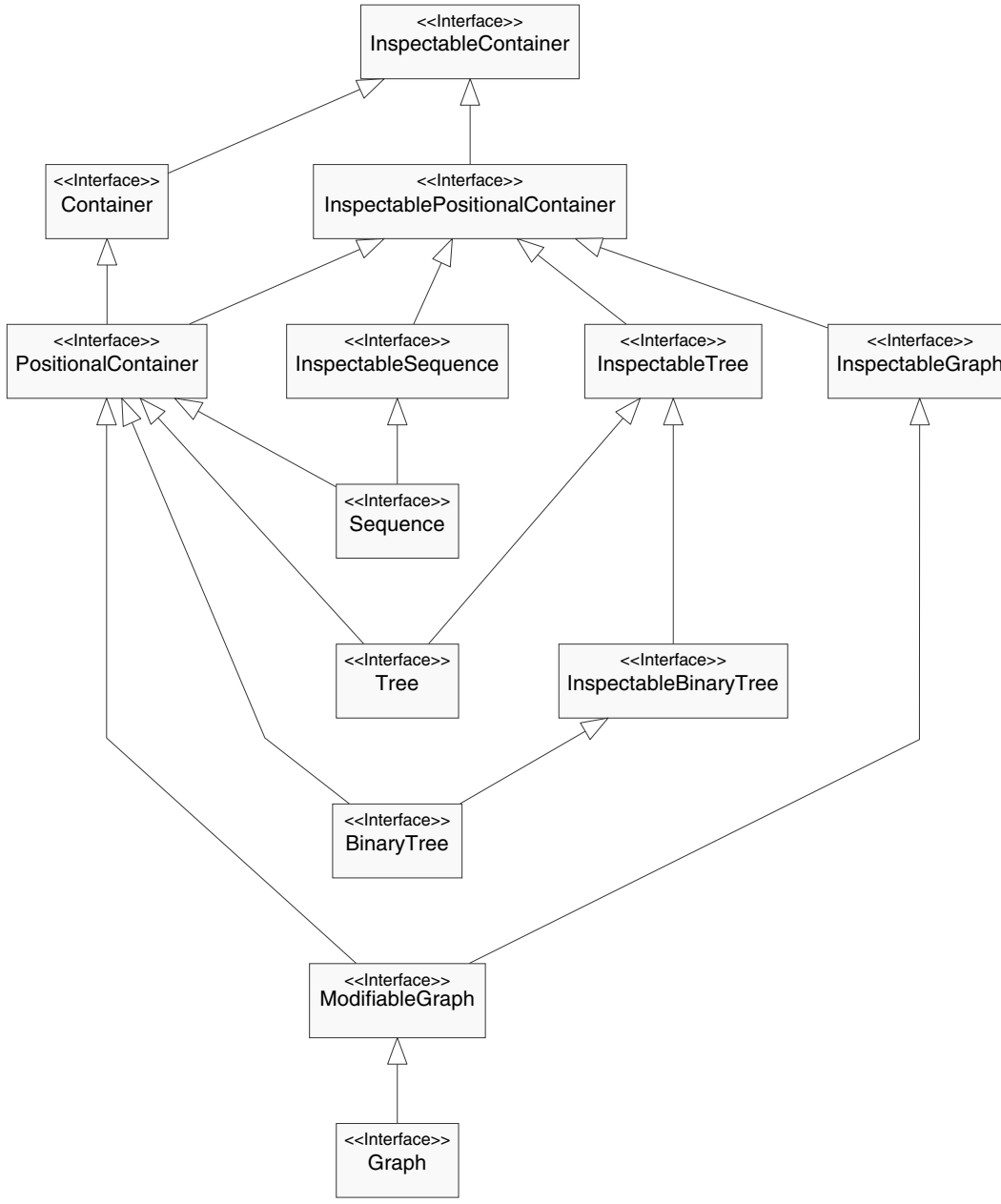


FIGURE 43.4: The positional containers interface hierarchy.

Every positional container implements a set of essential operations, including being able to determine its own size (size()), to determine whether it contains a specific position (contains(Accessor)), to replace the element associated with a position (replaceElement(Accessor,Object)), to swap the elements associated with two positions (swapElements(Position, Position)), and to get iterators over the positions (positions()) or the elements (elements()) of the container.

Sequences

A sequence is a basic data structure used for storing elements in a linear, ranked fashion (see Chapter 2). Sequences can be implemented in many ways, e.g., as a linked list of nodes or on top of an array. In JDSL, sequences are described by interfaces InspectableSequence and Sequence, which extend InspectablePositionalContainer and PositionalContainer, respectively. In addition to the basic methods common to all positional containers, the sequence interfaces provide methods to access and modify positions at the sequence ends (methods such as first(), insertLast() and removeFirst()) or specific positions along the sequence (methods such as after(Position), atRank(int), insertBefore(Position) and removeAtRank(int)).

NodeSequence is an implementation of Sequence based on a doubly linked list of nodes. The nodes are the positions of the sequence. It takes $O(1)$ time to insert, remove, or access both ends of the sequence or a position before or after a given one, while it takes $O(n)$ time to insert, remove, or access positions at a given rank in the sequence. Thus, NodeSequence instances can be suitably used as stacks, queues, or deques.

ArraySequence is an implementation of Sequence based on a growable array of positions. Instances can be created with an initial capacity, and can be told whether or not to reduce this capacity when their size drops below a certain value, depending on whether the user prefers space or time efficiency. It takes $O(1)$ time to access any position in the sequence, $O(1)$ amortized time over a series of operations to insert or remove elements at the end of the sequence, and $O(n)$ time to insert or remove elements at the beginning or middle of the sequence. Hence, ArraySequence instances can be suitably used for quick access to the elements after their initial insertion, when filled only at the end, or as stacks.

Trees

Trees allow more sophisticated relationships between elements than is possible with a sequence: they allow relationships between a child and its parent, or between siblings of a parent (see Chapter 3). InspectableTree and Tree are the interfaces describing a general tree; they extend InspectablePositionalContainer and PositionalContainer, respectively. InspectableBinaryTree, which extends InspectableTree, and BinaryTree, which extends PositionalContainer, are the interfaces describing a binary tree. In addition to the basic methods common to all positional containers, the tree interfaces provide methods to determine where in the tree a position lies (methods such as isRoot(Position) and isExternal(Position)), to return the parent (parent(Position)), siblings (siblings(Position)) or children (methods such as children(Position), childAtRank(Position,int) and leftChild(Position)) of a position, and to cut (cut(Position)) or link (link(Position,Tree)) a subtree.

NodeTree is an implementation of Tree based on a linked structure of nodes. The nodes are the positions of the tree. It is the implementation to use when a generic tree is needed or for building more specialized (nonbinary) trees. NodeTree instances always contain at least one node.

NodeBinaryTree is an implementation of BinaryTree based on a linked structure of nodes. The nodes are the positions of the tree. Similarly to NodeTree, NodeBinaryTree instances always contain at least one node; in addition, each node can have either zero or two children. If a more complex tree is not necessary, using NodeBinaryTree instances will be faster and easier than using NodeTree ones.

Graphs

A graph is a fundamental data structure describing a binary relationship on a set of elements (see Chapter 4) and it is used in a variety of application areas. Each vertex of the graph

may be linked to other vertices through edges. Edges can be either one-way, *directed* edges, or two-way, *undirected* edges. In JDSL, both vertices and edges are positions of the graph. JDSL handles all graph special cases such as self-loops, multiple edges between two vertices, and disconnected graphs.

The main graph interfaces are InspectableGraph, which extends InspectablePositionalContainer, ModifiableGraph, which extends PositionalContainer, and Graph, which extends both InspectableGraph and ModifiableGraph. These interfaces provide methods to determine whether two vertices are adjacent (areAdjacent(Vertex,Vertex)) or whether a vertex and an edge are incident (areIncident(Vertex,Edge)), to determine the degree of a vertex (degree(Vertex)), to determine the origin (origin(Edge)) or destination (destination(Edge)) of an edge, to insert (insertVertex(Object)) or remove (removeVertex(Vertex)) a vertex, to set the direction of an edge (setDirectionFrom(Edge,Vertex) and setDirectionTo(Edge,Vertex)), to insert (insertEdge(Vertex,Vertex,Object)), remove (removeEdge(Edge)), split (splitEdge(Edge, Object)), or unsplit (unsplitEdge(Vertex,Object)) an edge.

IncidenceListGraph is an implementation of Graph. As its name suggests, it is based on an incidence list representation of a graph.

43.3.3 Key-Based Containers

All key-based containers implement interfaces InspectableKeyBasedContainer and KeyBasedContainer, which extend InspectableContainer and Container, respectively (see Figure 43.5). Every key-based container implements a set of essential operations, including being able to determine its own size (size()), to determine whether it contains a specific locator

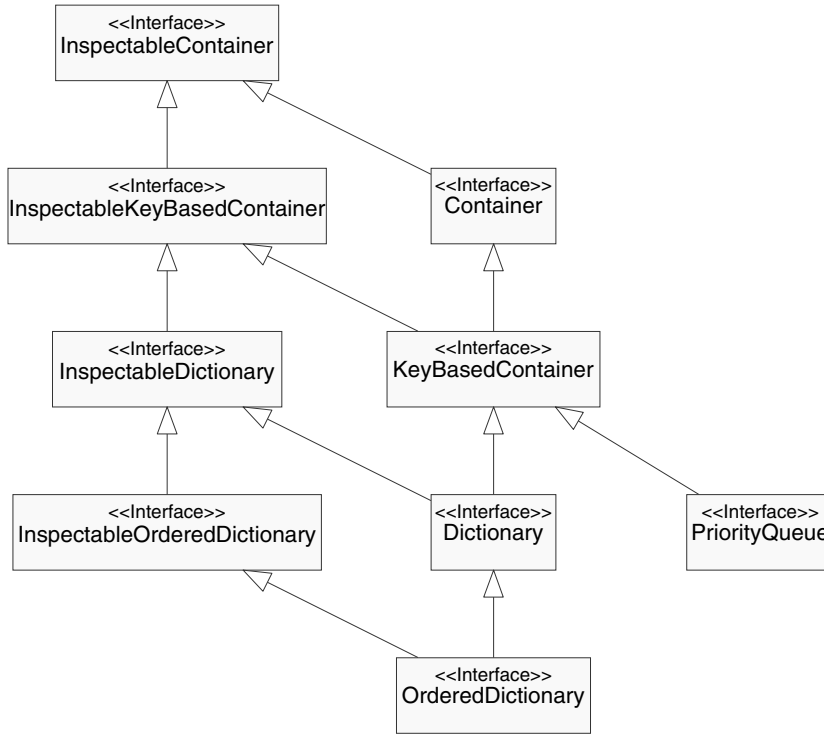


FIGURE 43.5: The key-based containers interface hierarchy.

(contains(Accessor)), to replace the key (replaceKey(Locator,Object)) or the element (replaceElement(Accessor,Object)) associated with a locator, to insert (insert(Object,Object)) or remove (remove(Locator)) a key/element pair, and to get iterators over the locators (locators()), the keys (keys()) or the elements (elements()) of the container.

Priority queues

A priority queue is a data structure used for storing a collection of elements prioritized by keys, where the smallest key value indicates the highest priority (see Part II). It supports arbitrary insertions and deletions of elements and keeps track of the highest-priority key. A priority queue is useful, for instance, in applications where the user wishes to store a queue of tasks of varying priority, and always process the most important task.

Interface PriorityQueue extends KeyBasedContainer. In addition to the basic methods common to all the key-based containers, it provides methods to access (min()) or remove (removeMin()) the key/element pair with highest priority, i.e., with minimum key. Note that the priority of an element can be changed using method replaceKey(Locator,Object), inherited from KeyBasedContainer.

ArrayHeap is an efficient implementation of PriorityQueue based on a heap. Inserting, removing, or changing the key of a key/element pair takes $O(\log n)$ time, while examining the key/element pair with the minimum key takes $O(1)$ time. The implementation is parameterized with respect to the comparison rule used to order the keys; to this purpose, a Comparator object is passed as an argument to the ArrayHeap constructors.

Dictionaries

A dictionary is a data structure used to store key/element pairs and then quickly search for them using their keys (see Part III). An ordered dictionary is a particular dictionary where a total order on the set of keys is defined. All JDSL dictionaries are *multi-maps*, i.e., they can store multiple key/element pairs with the same key.

The main dictionary interfaces are InspectableDictionary and Dictionary, which extend InspectableKeyBasedContainer and KeyBasedContainer, respectively. In addition to the basic methods common to all the key-based containers, these interfaces provide methods to find key/element pairs by their keys (find(Object) and findAll(Object)) and to remove all key/element pairs with a specific key (removeAll(Object)). Other dictionary interfaces are InspectableOrderedDictionary and OrderedDictionary, which extend InspectableDictionary and Dictionary, respectively. They provide additional methods to access the first (first()) or last (last()) key/element pair in the ordered dictionary, and to access the key/element pair before (before(Locator)) or after (after(Locator)) a given key/element pair.

HashtableDictionary is an implementation of Dictionary. As its name suggests, it is based on a hash table. Insertions and removals of key/element pairs usually take $O(1)$ time, although individual insertions and removals may require $O(n)$ time. The implementation is parameterized with respect to the hashing function used to store the key/element pairs; to this purpose, a HashComparator object is passed as an argument to the HashtableDictionary constructors. HashtableDictionary is a good choice when overall speed is necessary.

RedBlackTree is an implementation of OrderedDictionary. It is a particular type of binary search tree, where insertion, removal, and access to key/element pairs require each $O(\log n)$ time. The implementation is parameterized with respect to the comparison rule used to order the keys; to this purpose, a Comparator object is passed as an argument to the RedBlackTree constructors.

43.3.4 Algorithms

In addition to the data structures described above, JDSL includes various algorithms that operate on them.

Sequence sorting

JDSL provides a suite of sorting algorithms for different applications. They all implement the SortObject interface, whose only method is sort(Sequence,Comparator). Sorting algorithms with the prefix List are most efficient when used on instances of NodeSequence while those with the prefix Array are most efficient when used on instances of ArraySequence.

ArrayQuickSort is an implementation of the quicksort algorithm. This algorithm runs in $O(n \log n)$ expected time and performs very well in practice. Its performance, however, degrades greatly if the sequence is already very close to being sorted. Also, it is not *stable*, i.e., it does not guarantee that elements with the same value will remain in the same order they were in before sorting. In all cases whether neither of these caveats apply, it is the best choice.

ListMergeSort and ArrayMergeSort are two implementations of the mergesort algorithm. This algorithm is not as fast as quicksort in practice, even though its theoretical time complexity is $O(n \log n)$. There are no cases where its performance will degrade due to peculiarities in the input data, and it is a stable sort.

HeapSort is an implementation of the heapsort algorithm, and uses an instance of ArrayHeap (see Section 43.3.3) as a sorting device. Its performance, like that of mergesort, will not degrade due to peculiarities in the input data, but it is not a stable sort. Its theoretical time complexity is $O(n \log n)$.

Iterator-based tree traversals

JDSL provides two types of tree traversals. The first type is based on iterators: the tree is passed as an argument to the iterator constructor and is then iterated over using methods hasNext() and nextPosition(). Iterators give a quick traversal of the tree in a specific order, and are the proper traversals to use when this is all that is required. We recall that iterators in JDSL have snapshots semantics (see Section 43.2.2).

A preorder iterator visits the nodes of the tree in preorder, i.e., it returns a node before returning any of its children. Preorder iterators work for both binary and general trees; they are implemented in class PreOrderIterator.

A postorder iterator visits the nodes of the tree in postorder, i.e., it returns a node after returning all of its children. Postorder iterators work for both binary and general trees; they are implemented in class PostOrderIterator.

An inorder iterator visits the nodes of the tree in inorder, i.e., it returns a node in between its left and right children. Inorder iterators work only for binary trees; they are implemented in class InOrderIterator.

Euler tour tree traversal

The second type of tree traversals in JDSL is named Euler tour: it is implemented — in class EulerTour — as an algorithm object, which can be extended by applying the template method pattern.

The Euler tour visits each node of the tree several times, namely, a first time before traversing any of the subtrees of the node, then between the traversals of any two consecutive subtrees, and a last time after traversing all the subtrees. Each time a node is

visited, one of the methods visitFirstTime(Position), visitBetweenChildren(Position) and visitLastTime(Position), if the node is internal, or method visitExternal(Position), if the node is a leaf, is automatically called. A particular computation on the visited tree may be performed by suitably overriding those methods in a subclass of EulerTour.

The Euler tour is more powerful than the iterators described above as it can be used to implement more general kinds of algorithms. Note that, unlike the iterators, the Euler tour does not have snapshot semantics; this means that any modification of the tree during the execution of the Euler tour will cause undefined behavior.

Graph traversals

The depth-first search (DFS) traversal of a graph is available in JDSL. Depth-first search proceeds by visiting an unvisited vertex adjacent to the current one; if no such vertex exists, then the algorithm backtracks to the previous visited vertex.

Similarly to the Euler tour, depth-first search is implemented in JDSL as an algorithm object, which can be extended by applying the template method pattern. The basic implementation of depth-first search — DFS — is designed to work on undirected graphs. The user can specify actions to occur when a vertex is first or last visited or when different sorts of edges (such as "tree" edges of the DFS tree or "back" edges to previously visited vertices) are traversed by subclassing DFS and suitably overriding some methods.

DFS has two subclasses: FindCycleDFS, an algorithm for determining cycles in an undirected graph, and DirectedDFS, a depth-first search specialized for directed graphs. In turn, DirectedDFS has one subclass: DirectedFindCycleDFS, an algorithm for determining cycles in a directed graph. These subclasses are examples of how to apply the template method pattern to DFS in order to implement a more specific algorithm.

Topological numbering

A topological numbering is a numbering of the vertices of a directed acyclic graph such that, if there is an edge from vertex u to vertex v, then the number associated with v is higher than the number associated with u.

Two algorithms that compute a topological numbering are included in JDSL: TopologicalSort, which decorates each vertex with a unique number, and UnitWeightedTopologicalNumbering, which decorates each vertex with a nonnecessarily unique number based on how far the vertex is from the source of the graph. Both topological numbering algorithms extend abstract class AbstractTopologicalSort.

Dijkstra's algorithm

Dijkstra's algorithm computes the shortest path from a specific vertex to every other vertex of a weighted connected graph. The JDSL implementation of Dijkstra's algorithm — IntegerDijkstraTemplate — follows the template method pattern and can be easily extended to change its functionality. Extending it makes it possible, for instance, to set the function for calculating the weight of an edge, to change the way the results are stored, or to stop the execution of the algorithm after computing the shortest path to a specific vertex (as done in subclass IntegerDijkstraPathfinder). An example of using Dijkstra's algorithm is given in Section 43.4.

The Prim-Jarník algorithm

The Prim-Jarník algorithm computes a *minimum spanning tree* of a weighted connected graph, i.e., a tree that contains all the vertices of the graph and has the minimum total

weight over all such trees. The JDSL implementation of the Prim-Jarník algorithm — IntegerPrimTemplate — follows the template method pattern and can be easily extended to change its functionality. Extending it makes it possible, for instance, to set the function for calculating the weight of an edge, to change the way the results are stored, or to stop the execution of the algorithm after computing the minimum spanning tree for a limited set of vertices.

43.4 A Sample Application

In this section we explore the implementation of a sample application using JDSL. In particular, we show the use of some of the concepts described above, such as the graph and priority queue data structures, locators, decorations, and the template method pattern.

43.4.1 Minimum-Time Flight Itineraries

We consider the problem of calculating a minimum-time flight itinerary between two airports. The flight network can be modeled using a weighted directed graph: each vertex of the graph represents an airport, each directed edge represents a flight from the origin airport to the destination airport, and the weight of each directed edge is the duration of the flight. The problem we are considering can be solved by computing a shortest path between two vertices of the directed graph or determining that a path does not exists. To this purpose, we can suitably modify the classical algorithm by Dijkstra [2], which takes as input a graph G with nonnegative edge weights and a distinguished source vertex s, and computes a shortest path from s to any reachable vertex of G. Dijkstra's algorithm maintains a priority queue Q of vertices: at any time, the key of a vertex u in the priority queue is the length of the shortest path from s to u thus far. The priority queue is initialized by inserting vertex s with key 0 and all the other vertices with key $+\infty$ (some very large number). The algorithm repeatedly executes the following two steps:

1. Remove a minimum-key vertex u from the priority queue and mark it as *finished*, since a shortest path from s to u has been found.
2. For each edge e connecting u to an unfinished vertex v, if the path formed by extending a shortest path from s to u with edge e is shorter than the shortest known path from s to v, update the key of v (this operation is known as the *relaxation* of edge e).

43.4.2 Class IntegerDijkstraTemplate

As seen in Section 43.3.4, JDSL provides an implementation of Dijkstra's algorithm that follows the template method pattern. The abstract class implementing Dijkstra's algorithm is jdsl.graph.algo.IntegerDijkstraTemplate (see Figures 43.6–43.8; for brevity, the Javadoc comments present in the library code have been removed). The simplest way to run the algorithm is by calling execute(InspectableGraph,Vertex), which first initializes the various auxiliary data structures with init(g,source) and then repeatedly calls doOneIteration(). Note that the number of times doOneIteration() is called is controlled by shouldContinue(). Another possibility, instead of calling execute(InspectableGraph,Vertex), is to call init(InspectableGraph,Vertex) directly and then single-step the algorithm by explicitly calling doOneIteration().

For an efficient implementation of the algorithm, it is important to access a vertex stored

```
package jdsl.graph.algo;

import jdsl.core.api.*;
import jdsl.core.ref.ArrayHeap;
import jdsl.core.ref.IntegerComparator;
import jdsl.graph.api.*;

public abstract class IntegerDijkstraTemplate {

  // instance variables

  protected PriorityQueue pq_;
  protected InspectableGraph g_;
  protected Vertex source_;
  private final Integer ZERO = new Integer(0);
  private final Integer INFINITY = new Integer(Integer.MAX_VALUE);
  private final Object LOCATOR = new Object();
  private final Object DISTANCE = new Object();
  private final Object EDGE_TO_PARENT = new Object();

  // abstract instance methods

  protected abstract int weight (Edge e);

  // instance methods that may be overridden for special applications

  protected void shortestPathFound (Vertex v, int vDist) {
    v.set(DISTANCE,new Integer(vDist));
  }

  protected void vertexNotReachable (Vertex v) {
    v.set(DISTANCE,INFINITY);
    setEdgeToParent(v,Edge.NONE);
  }

  protected void edgeRelaxed (Vertex u, int uDist, Edge uv, int uvWeight, Vertex v, int vDist) { }

  protected boolean shouldContinue () {
    return true;
  }

  protected boolean isFinished (Vertex v) {
    return v.has(DISTANCE);
  }

  protected void setLocator (Vertex v, Locator vLoc) {
    v.set(LOCATOR,vLoc);
  }

  protected Locator getLocator (Vertex v) {
    return (Locator)v.get(LOCATOR);
  }

  protected void setEdgeToParent (Vertex v, Edge vEdge) {
    v.set(EDGE_TO_PARENT,vEdge);
  }
```

FIGURE 43.6: Class IntegerDijkstraTemplate.

```
protected EdgeIterator incidentEdges (Vertex v) {
  return g_.incidentEdges(v,EdgeDirection.OUT | EdgeDirection.UNDIR);
}

protected Vertex destination (Vertex origin, Edge e) {
  return g_.opposite(origin,e);
}

protected VertexIterator vertices () {
  return g_.vertices();
}

protected PriorityQueue newPQ () {
  return new ArrayHeap(new IntegerComparator());
}

// output instance methods

public final boolean isReachable (Vertex v) {
  return v.has(EDGE_TO_PARENT) && v.get(EDGE_TO_PARENT) != Edge.NONE;
}

public final int distance (Vertex v) throws InvalidQueryException {
  try {
    return ((Integer)v.get(DISTANCE)).intValue();
  }
  catch (InvalidAttributeException iae) {
    throw new InvalidQueryException(v+" has not been reached yet");
  }
}

public Edge getEdgeToParent (Vertex v) throws InvalidQueryException {
  try {
    return (Edge)v.get(EDGE_TO_PARENT);
  }
  catch (InvalidAttributeException iae) {
    String s = (v == source_) ? " is the source vertex" : " has not been reached yet";
    throw new InvalidQueryException(v+s);
  }
}

// instance methods composing the core of the algorithm

public void init (InspectableGraph g, Vertex source) {
  g_ = g;
  source_ = source;
  pq_ = newPQ();
  VertexIterator vi = vertices();
  while (vi.hasNext()) {
    Vertex u = vi.nextVertex();
    Integer uKey = (u == source_) ? ZERO : INFINITY;
    Locator uLoc = pq_.insert(uKey,u);
    setLocator(u,uLoc);
  }
}
```

FIGURE 43.7: Class IntegerDijkstraTemplate (continued).

```
  protected final void runUntil () {
    while (!pq_.isEmpty() && shouldContinue())
      doOneIteration();
  }

  public final void doOneIteration () throws InvalidEdgeException {
    Integer minKey = (Integer)pq_.min().key();
    Vertex u = (Vertex)pq_.removeMin(); // remove a vertex with minimum distance from the source
    if (minKey == INFINITY)
      vertexNotReachable(u);
    else {   // the general case
      int uDist = minKey.intValue();
      shortestPathFound(u,uDist);
      int maxEdgeWeight = INFINITY.intValue()-uDist-1;
      EdgeIterator ei = incidentEdges(u);
      while (ei.hasNext()) {  // examine all the edges incident with u
        Edge uv = ei.nextEdge();
        int uvWeight = weight(uv);
        if (uvWeight < 0 || uvWeight > maxEdgeWeight)
          throw new InvalidEdgeException
            ("The weight of "+uv+" is either negative or causing overflow");
        Vertex v = destination(u,uv);
        Locator vLoc = getLocator(v);
        if (pq_.contains(vLoc)) { // v is not finished yet
          int vDist = ((Integer)vLoc.key()).intValue();
          int vDistViaUV = uDist+uvWeight;
          if (vDistViaUV < vDist) { // relax
            pq_.replaceKey(vLoc,new Integer(vDistViaUV));
            setEdgeToParent(v,uv);
          }
          edgeRelaxed(u,uDist,uv,uvWeight,v,vDist);
        }
      }
    }
  }

  public final void execute (InspectableGraph g, Vertex source) {
    init(g,source);
    runUntil();
  }

  public void cleanup () {
    VertexIterator vi = vertices();
    while (vi.hasNext()) {
      vi.nextVertex().destroy(LOCATOR);
      try {
        vi.vertex().destroy(EDGE_TO_PARENT);
        vi.vertex().destroy(DISTANCE);
      }
      catch (InvalidAttributeException iae) { }
    }
  }

}   // class IntegerDijkstraTemplate
```

FIGURE 43.8: Class IntegerDijkstraTemplate (continued).

in the priority queue in constant time, whenever its key has to be modified. This is possible through the locator accessors provided by class jdsl.core.ref.ArrayHeap (see Section 43.3.3). In init(InspectableGraph,Vertex), each vertex u of the graph is inserted in the priority queue and a locator uLoc for the key/element pair is returned. By calling setLocator(u,uLoc), each vertex u is decorated with its locator uLoc; variable LOCATOR is used as the attribute name. Later, in doOneIteration(), the locator of vertex v is retrieved by calling getLocator(v) in order to access and possibly modify the key of v; we recall that the key of v is the shortest known distance from the source vertex source_ to v. In addition to its locator in the priority queue, every unfinished vertex v is also decorated with its last relaxed incident edge uv by calling setEdgeToParent(v,uv); variable EDGE_TO_PARENT is used as the attribute name, in this case. When a vertex is finished, this decoration stores the edge to the parent in the shortest path tree, and can be retrieved with getEdgeToParent(Vertex).

Methods runUntil() and doOneIteration() are declared final and thus cannot be overridden. Following the template method pattern, they call some methods, namely, shouldContinue(), vertexNotReachable(Vertex), shortestPathFound(Vertex,int), and edgeRelaxed(Vertex,int,Edge, int,Vertex,int), that may be overridden in a subclass for special applications. For each vertex u of the graph, either vertexNotReachable(u) or shortestPathFound(u,uDist) is called exactly once, when u is removed from the priority queue and marked as finished. In particular, shortestPathFound(u,uDist) decorates u with uDist, the shortest distance from source_; variable DISTANCE is used as the attribute name. Method edgeRelaxed(u,uDist,uv,uvWeight,v,vDist) is called every time an edge uv from a finished vertex u to an unfinished vertex v is examined. The only method whose implementation must be provided by a subclass is abstract method weight(Edge), which returns the weight of an edge. Other important methods are isFinished(Vertex), which returns whether a given vertex is marked as finished, and distance(Vertex), which returns the shortest distance from source_ to a given finished vertex.

43.4.3 Class IntegerDijkstraPathfinder

JDSL also provides a specialization of Dijkstra's algorithm to the problem of finding a shortest path between two vertices of a graph. This algorithm is implemented in abstract class jdsl.graph.algo.IntegerDijkstraPathfinder (see Figure 43.9; for brevity, the Javadoc comments present in the library code have been removed), which extends IntegerDijkstraTemplate. The algorithm is run by calling execute(InspectableGraph,Vertex,Vertex). The execution of Dijkstra's algorithm is stopped as soon as the destination vertex is finished. To this purpose, shouldContinue() is overridden to return true only if the destination vertex has not been finished yet. Additional methods are provided in IntegerDijkstraPathfinder to test, after the execution of the algorithm, whether a path from the source vertex to the destination vertex exists (pathExists()), and, in this case, to return it (reportPath()).

43.4.4 Class FlightDijkstra

Our application for computing a minimum-time flight itinerary between two airports can be implemented as a specialization of IntegerDijkstraPathfinder. The distance of each vertex represents, in this case, the time elapsed from the beginning of the travel to the arrival at the airport represented by that vertex. In Figure 43.10 we show the code of class FlightDijkstra; this class is part of the tutorial** distributed with JDSL. All it takes to implement our

**http://www.jdsl.org/tutorial/tutorial.html

```
package jdsl.graph.algo;

import jdsl.core.api.*;
import jdsl.core.ref.NodeSequence;
import jdsl.graph.api.*;
import jdsl.graph.ref.EdgeIteratorAdapter;

public abstract class IntegerDijkstraPathfinder extends IntegerDijkstraTemplate {

  // instance variables

  private Vertex dest_;

  // overridden instance methods from IntegerDijkstraTemplate

  protected boolean shouldContinue () {
    return !isFinished(dest_);
  }

  // output instance methods

  public boolean pathExists () {
    return isFinished(dest_);
  }

  public EdgeIterator reportPath () throws InvalidQueryException {
    if (!pathExists())
      throw new InvalidQueryException("No path exists between "+source_+" and "+dest_);
    else {
      Sequence retval = new NodeSequence();
      Vertex currVertex = dest_;
      while (currVertex != source_) {
        Edge currEdge = getEdgeToParent(currVertex);
        retval.insertFirst(currEdge);
        currVertex = g_.opposite(currVertex,currEdge);
      }
      return new EdgeIteratorAdapter(retval.elements());
    }
  }

  // instance methods

  public final void execute (InspectableGraph g, Vertex source, Vertex dest) {
    dest_ = dest;
    init(g,source);
    if (source_ != dest_)
      runUntil();
  }

}  // class IntegerDijkstraPathfinder
```

FIGURE 43.9: Class IntegerDijkstraPathfinder.

application is to override method incidentEdges(), so that only the outgoing edges of a finished vertex are examined, and to define method weight(Edge). As noted before, the weighted graph representing the flight network is a directed graph. Each edge stores, as an element, an instance of auxiliary class FlightSpecs providing the departure time and the duration of the corresponding flight. Note that the weights of the edges are not determined before the execution of the algorithm, but rather depend on the computed shortest distance

```
import jdsl.graph.api.*;
import jdsl.graph.algo.IntegerDijkstraPathfinder;
import support.*;

public class FlightDijkstra extends IntegerDijkstraPathfinder {

  // instance variables

  private int startTime_;

  // overridden instance methods from IntegerDijkstraPathfinder

  protected int weight (Edge e) {
    FlightSpecs eFS = (FlightSpecs)e.element(); // the flightspecs for the flight along edge e
    int connectingTime = TimeTable.diff(eFS.departureTime(),startTime_+distance(g_.origin(e)));
    return connectingTime+eFS.flightDuration();
  }

  protected EdgeIterator incidentEdges (Vertex v) {
    return g_.incidentEdges(v,EdgeDirection.OUT);
  }

  // instance methods

  public void execute (InspectableGraph g, Vertex source, Vertex dest, int startTime) {
    startTime_ = startTime;
    super.execute(g,source,dest);
  }

}
```

FIGURE 43.10: Class FlightDijkstra.

between the source vertex and the origin of each edge. Namely, they are obtained by adding the duration of the flight corresponding to the edge and the connecting time at the origin airport for that flight.[††] Method TimeTable.diff(int,int) simply computes the difference between its two arguments modulo 24 hours. The algorithm is run by calling execute(InspectableGraph,Vertex,Vertex,int), where the fourth argument is the earliest time the passenger can begin traveling.

As we can see from this example, the availability in JDSL of a set of carefully designed and extensible data structures and algorithms makes it possible to implement moderately complex applications with a small amount of code, thus dramatically reducing the development time.

Acknowledgments

We would like to thank all the members of the JDSL Team for their fundamental role in the design, implementation, and testing of JDSL. It has been a great experience and a real pleasure working together.

[††]In this sample application we ignore the minimum connecting time requirement, which however could be accommodated with minor code modifications.

This work was supported in part by the National Science Foundation under grants CCR-0098068 and DUE-0231202.

References

[1] R. S. Baker, M. Boilen, M. T. Goodrich, R. Tamassia, and B. A. Stibel. Testers and visualizers for teaching data structures. In *Proc. 30th ACM SIGCSE Tech. Sympos.*, pages 261–265, 1999.

[2] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[3] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. On the design of CGAL a computational geometry algorithms library. *Softw. – Pract. Exp.*, 30(11):1167–1202, 2000.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, Reading, MA, 1995.

[5] M. T. Goodrich, M. Handy, B. Hudson, and R. Tamassia. Accessing the internal organization of data structures in the JDSL library. In M. T. Goodrich and C. C. McGeoch, editors, *Algorithm Engineering and Experimentation (Proc. ALENEX '99)*, volume 1619 of *Lecture Notes Comput. Sci.*, pages 124–139. Springer-Verlag, 1999.

[6] M. T. Goodrich and R. Tamassia. *Data Structures and Algorithms in Java*. John Wiley & Sons, New York, NY, 2nd edition, 2001.

[7] M. T. Goodrich and R. Tamassia. *Algorithm Design: Foundations, Analysis, and Internet Examples*. John Wiley & Sons, New York, NY, 2002.

[8] K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge, England, 1999.

[9] D. R. Musser, G. J. Derge, and A. Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison-Wesley, Reading, MA, 2nd edition, 2001.

[10] R. Tamassia, M. T. Goodrich, L. Vismara, M. Handy, G. Shubina, R. Cohen, B. Hudson, R. S. Baker, N. Gelfand, and U. Brandes. JDSL: The data structures library in Java. *Dr. Dobb's Journal*, (323):21–31, Apr. 2001.

44

Data Structure Visualization

John Stasko
Georgia Institute of Technology

44.1 Introduction

Important advances in programming languages have occurred since the early days of assembly languages and op codes, and modern programming languages have significantly simplified the task of writing computer programs. Unfortunately, tools for *understanding* programs have not achieved the accompanying levels of improvement. Software developers still face difficulties in understanding, analyzing, debugging, and improving code.

As an example of the difficulties still evident, consider a developer who has just implemented a new algorithm and is now ready to examine the program's behavior. After issuing a command to begin program execution, the developer examines output data and/or interactive behavior, attempting to ascertain if the program functioned correctly, and if not, the reasons for the program's failure. This task can be laborious and usually requires the use of a debugger or perhaps even the addition of explicit checking code, usually through output statements. Furthermore, the process can be quite challenging and the average developer may not be that skilled in fault diagnosis and correction. It would be extremely advantageous to have a tool that allows programmers to "look inside" programs as they execute, examining the changes to data and data structures that occur over time. Such a tool also could provide helpful context by identifying the current execution line, the active program unit, and the set of activations that have occurred to reach this configuration.

Tools, such as the one described above, are one form of *software visualization* [1]. Price, Baecker, and Small describe software visualization as, "the use of the crafts of typography, graphic design, animation, and cinematography with modern human-computer interaction and computer graphics technology to facilitate both the human understanding and effective use of computer software [2]." Fundamentally, software visualization seeks to take advantage of the visual perception and pattern matching skills of people to assist software development and software understanding. Human beings have very sophisticated visual perception systems that we constantly use to help us think [3]. Software visualization is a

subfield in the larger area of *information visualization* [4, 5] that studies how visualizations serve as external cognition aids.

Software visualization research and systems fall into two primary categories, *program visualization* and *algorithm visualization*.

Program visualization systems present data about and attributes of some existing software system. Typically, program visualizations are used in a software engineering context, that is, to help design, implement, optimize, test, or debug software. Thus, program visualization systems often illustrate source code, program data, execution statistics, program analysis information, or program objects such as data structures. The imagery in program visualization systems also is usually displayed with little or no input from the user/viewer. That is, a program (source code or executable) is analyzed by the system, which then automatically produces the graphics when the user requests them.

Algorithm visualization, conversely, strives to capture the abstractions and semantics of an algorithm or program primarily for pedagogical purposes. One often thinks of algorithm visualization as being more "high-level" as compared to program visualization being more "low level." The term *algorithm animation* is used interchangeably for this area simply because the displays are so dynamic and algorithm operations appear to animate over time. More specifically, Brown states, "Algorithm animation displays are, essentially, dynamic displays showing a program's fundamental *operations*—not merely its data or code [6]." Algorithm visualizations typically are hand-crafted, user-conceptualized views of what is "important" about a program. These types of views usually require some person, often the system developer or the program's creator, design and implement the graphics that accompany the program. The highly abstract, artificial nature of algorithm animations makes their automatic generation exceptionally difficult.

This chapter focuses on one specific subarea of software visualization, the visualization of data structures. By displaying pictures of data structures, one seeks to help people better understand the characteristics and the use of those structures.

How the data in computer programs are manipulated and organized by groups is a key aspect of successful programming. Understanding how elements of the data relate to other elements is a vital component in being able to comprehend and create sophisticated algorithms and programs. The word "structure" in "data structure" simply reinforces this point.

A person learning a new piece of software and seeking to modify it for some added functionality will likely ask a number of different questions. How are data elements organized? Is some kind of sequential structure used or is a more complicated organization present? Which items reference which other items? Understanding the answers to questions such as these helps a person to understand the utility and value of data structures. Providing pictures to express the relationships of data elements clearly then is one of the most useful ways to answer those questions.

44.2 Value of Data Structure Rendering

The graphical display of data structures can benefit a number of different activities, and one key role is as an aid for computer science education. It is not uncommon for students to have difficulty learning to understand and use non-trivial data structures such as arrays, lists, stacks, queues (Chapter 2), and trees (Chapter 3). One common problem occurs in the mapping back-and-forth between the pseudo code or programming language implementation of a structure and its conceptual model. In current educational methods, to foster understanding, the conceptual model (abstraction) is often presented in some graphical rep-

resentation or picture—an interesting challenge is to find a book about data structures that does not make liberal use of pictures.

Systems for visualizing data structures typically provide or operate in an environment containing both the structures' programming language implementation and their graphical display, which facilitates students making connections between the two. This allows an instructor to prepare a collection of "interesting" data structures for students to interact with and learn from. By examining the accompanying displays, students can make the connection from a structure's abstract properties to its concrete implementation.

Another activity in which data structure visualization systems can provide significant help is program debugging. When a data structure display tool is used in conjunction with a traditional textual debugger, it allows the active data structures to be displayed and examined in some graphical format as program execution occurs and units of code are examined. When the program's data change, the accompanying visualizations change too, reflecting the new program state.

The graphical display of data structures adds many advantages to strict textual debugging. Because a human's visual perception system is a highly-tuned information processing machine, the sheer amount of information that can be transmitted to the user is much greater using graphics. With a textual debugger, people usually need to issue queries variable-by-variable to discover values. One glance at a computer display showing simultaneous representations of multiple structures may convey as much information as several minutes of the "query-reply" loop in a conventional debugger.

In addition to discovering data values, programmers use debuggers to access the relationships and connections between different pieces of data. This is a particular area in which graphical display is much more beneficial than textual interpreters. For example, to check the structure of a linked list in a language such as C++ or Java, a programmer must print out a list item's values and references to other list items. By carefully examining the resulting values, the programmer can determine if the list is in the desired configuration. A data structure visualization tool that could display a linked list as a series of rectangular nodes with arrows acting as the references or pointers connecting the different nodes would tremendously simplify the programmer's task of determining relationships among the data.

Data structure visualization tools also can aid the acquisition of contextual information about the state of a debugging session. For instance, a display could show a simple persistent view of the call stack and current execution line number in addition to the data pictures. In most debuggers, a user must often make repeated explicit queries to determine this information.

For these reasons and many more, data structure visualization systems remain a relatively untapped source of value to programmers. It is quite surprising that, given these benefits, more attention and research have not been devoted to the area, and data structure visualization systems have yet to achieve widespread use and adoption.

44.3 Issues in Data Structure Visualization Systems

44.3.1 Purpose and Environment

The software environment in which a data structure visualization system resides will influence many of the system's capabilities. Data structure display systems usually target a specific programming language or set of languages for visualization. In particular, many systems display only strongly-typed languages, simply because one is able to infer more information concerning the structure and composition of data objects in a strongly typed language. It also is easier to "customize" a specific graphical look for the different data

structure types in a strongly typed language. A weakly typed language such as an assembly language generally forces a more generic looking display of data.

Clearly, the inherent characteristics of a particular programming language will also influence the language's resulting data visualizations. We would expect the node-pointer list data depiction to be an integral part of a LISP visualization system, while imperative languages such as C or Pascal would necessitate repeated use of a flexible structured, tiled image with easy to recognize and identify sub-fields. The display of graphical objects representing class instances and some form of arcs or arrows for messages (method invocations) would be important for illustrating an object-oriented language such as C++ or Java.

The intended audience of a data structure visualization system also affects the system's resulting displays. For example, a system designed for use by introductory programmers would probably stress consistent, simplified data structure views. Conversely, a system targeted to advanced programmers should support sophisticated, customized displays of complex, large data objects. Such a system should also allow a viewer to quickly suppress unwanted display objects and focus on items of special interest.

One common use of data structure visualization systems is in conjunction with other tools in a programming environment or an integrated development environment (IDE). For example, "attaching" a data structure visualization system to a program execution monitor allows the display of graphical data snap-shots at various points in the program's execution. By consistently displaying imagery, a form of program data animation results, showing how a program's data change over time.

Most often, a data structure visualization system functions in conjunction with a program debugger. In this arrangement, the display system provides a graphical window onto the program, permitting a persistent view of the data as opposed to the query-based approach of a textual debugger. While a programmer navigates through her program and its data by using a debugger, a data structure visualization system can map the particular data objects of interest onto a graphical format. As discussed in the previous section, the graphical format provides many benefits to typical debugging activities.

44.3.2 Data Structure Views

A primary distinguishing characteristic of a data structure visualization system is the types of data views it supports. Most systems provide a set of default views for the common data types such as integers, floats, and strings. Some systems also provide default views for more complex composite structures such as arrays, lists, and trees. Visualization systems frequently differ in their ability to handle a display request for a data object of a type other than those in the default view set. Reactions to such a query may range from taking no display action to utilizing a generic view applicable to all possible data types. Some systems also provide users with the ability to tailor a special graphical view for a particular, possibly uncommon data type. The process of defining these views is often tedious, however, discouraging such forms of improvisation.

Specific, relatively common data views that are often handled in varying ways include pointers/references, fields within composite structures, and arrays. Pointers/references, usually represented in a line-arrow manner, present a particularly tricky display problem because they involve issues similar issues to those evident in graph layout, a known difficult problem [7] (see also Chapter 46). Ideally, a view with pointers should minimize pointer overlap, edge crossings, and collisions with other display objects. Both polyline and spline display formats for pointers are common.

Representing fields within a composite data structure is a difficult problem because of spacing and layout concerns. Subfields can be complex structures themselves, complicating

efforts to make the data structure view clear and comprehensible.

Arrays present a challenging visualization problem because of the variety of displays that are possible. For instance, a simple one-dimensional array can be presented in a horizontal or vertical format. The ideal view for multi-dimensional arrays can also be a difficult rendering task, particularly for arrays of dimension three or higher.

The ability to display multiple views of a specific data instance is also a valuable capability of a data structure visualization system because certain display contexts may be more informative in particular situations. Allowing varying levels of view abstraction on data is another important feature. For example, consider an array of structures in the C programming language. At one level, we may wish to view the array formation globally, with each structure represented by a small rectangle. At a closer level, we may wish to view a particular array structure element in full detail, with little attention paid to the other array elements.

Some data structure display systems provide visualizations for components of program execution such as program flow of control and the call stack. While these components are not program data structures, their graphical visualizations can be helpful additions to a data structure display system, especially when used in conjunction with a debugger.

44.3.3 Interacting with a System

Data structure visualization systems provide many different ways for users to interact with the system. For instance, consider the manner that users employ to actually display a particular piece of data. Systems that work in conjunction with a debugger may provide a special *display* command in the debugger. Other systems may utilize a specialized user interface with a particular direct manipulation protocol for invoking data display, such as choosing a menu item and supplying a variable name or graphically selecting a variable in a source code view.

Visualization systems vary in the manner of interactions they provide to users as well. For instance, once data is selected for display, the corresponding image(s) must be rendered on the viewing area. One option for this rendering is to allow the viewer to position the images, usually with the mouse. This method has the advantage of giving the user explicit control, but often, such repeated positioning can become tedious. Another rendering option gives the system total placement control. This method has the advantage of requiring less viewer input, but it requires sophisticated layout algorithms to avoid poor layout decisions. Perhaps the most attractive rendering option is a combination of these two: automatic system display with subsequent user repositioning capabilities.

Limits in display window viewing area force data structure visualization systems to confront sizing issues also. One simple solution to the problem of limited viewing space is to provide an infinite, scrollable viewing plane. Another solution, one more closely integrated with the system, is to utilize varying display abstractions dependent upon the amount of space available to view a data object. For example, given no space limitations, an object could be rendered in its default view. With very limited space, the object could be rendered in a space-saving format such as a small rectangle.

Once data has been displayed in the viewing window, the viewer should be able to interact with and control the imagery. Allowing a user to interactively move and delete images is certainly desirable. Even more beneficial, however, is the capability to suppress aspects of the data that are not of interest. For instance, only a small section of an array may be "interesting," so a visualization system could deemphasize other portions as a user dictates. Similarly, only certain sections of linked lists and trees may require attention, and only certain fields with a particular structure type may be of interest. Allowing a viewer

to quickly dispose of uninteresting attributes and focus on the matter at hand is a very valuable feature.

If a data structure visualization system works in conjunction with a debugger, display interaction may take on a further role, that of interactive debugging itself. Graphical editing of data imagery by the viewer can be interpreted as corresponding changes to the underlying data. For example, a system may allow a viewer to enter a new data value for a variable by graphically selecting its accompanying image and overwriting the value displayed there. An even more sophisticated action is to allow a viewer to "grab" a pointer variable, usually represented by an arrow on the display, and alter its value by dragging and repositioning the arrow image in order to make the pointer reference some other object.

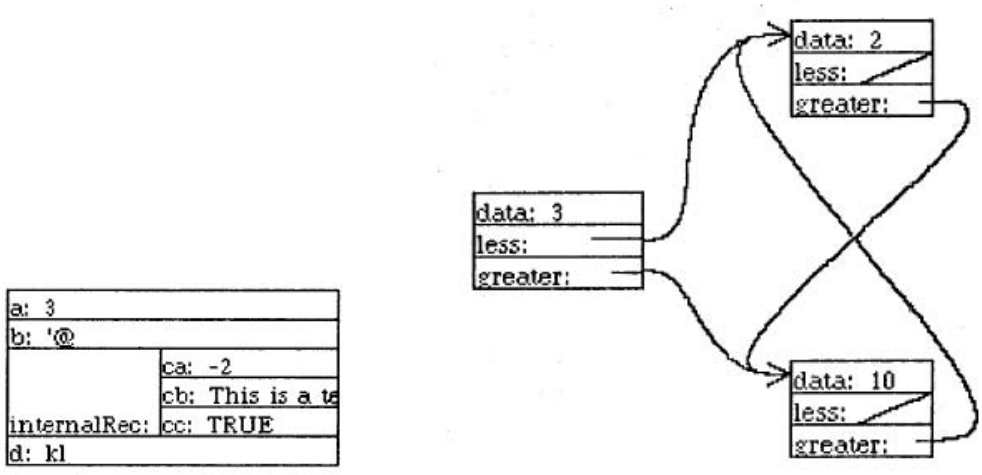
44.4 Existing Research and Systems

The idea of creating a system that would visualize data structures in computer programs is an old one, dating back even to the 1960's [8]. During this long history, a number of systems for data structure visualization have been developed. Most, however, have only been research prototypes and have never been used by anyone outside the system's creator(s). This relative lack of adoption is surprising in light of the potential benefits of data structure display identified above. A quick survey of widely-used current IDEs notes that none provide anything beyond the most limited abilities to persistently display data values, or more importantly, visualize data structures.

For a good overview of the aesthetic considerations and the general challenges in drawing data structures, see the article by Ding and Mateti [9]. The authors provide a rigorous survey of the different styles of drawings used in data structure diagrams. They enumerate a set of characteristics that influence people's perceptions of data structure illustrations, including visual complexity, regularity, symmetry, consistency, modularity, size, separation, shape, and traditional ways of drawing. Further, they develop a set of guidelines for drawing aesthetically appealing diagrams. The guidelines are based on rules, factors, and objectives, and include the ability to quantitatively assess the aesthetics of different diagrams. Ding and Mateti note that the automatic drawing of data structure representations is difficult because a diagram should be determined not only by a structure's declaration, but also by its intended usage.

Stasko and Patterson echo that thought by identifying the notion of *intention content* in program visualizations [10]. They note that a programmer's intent in creating and using a data structure can significantly influence how the structure should best be visualized. For instance, a simple array of integers may be shown as a row of rectangles with the values inside; a row of rectangles whose height is scaled to the value of each element; a round pie chart with a wedge whose percentage size of the circle corresponds to the value of that array element; or a stack with the array elements sitting on top of each other. To generate the latter two representations, some knowledge of the purpose and goal of the surrounding program code is necessary.

In the remainder of this section, we present brief summaries of a few of the most noteworthy data structure visualization systems that have been developed. Each of these grew from research projects, and the final one, DDD, has been used significantly by outside users. In reviewing each system, we highlight its unique aspects, and we evaluate it with respect to the issues raised in the previous section.



(a) Structure (b) Linked list

FIGURE 44.1: Example data structure views shown in the Incense system. Pictures provided by and reprinted with permission of Brad Myers.

44.4.1 Incense

The Incense system [11] developed by Myers was one of the earliest attempts to build a relatively full-featured data structure display system. Myers stated that Incense allowed a "programmer to interactively investigate data structures in actual programs."

The system operated on a Xerox Alto personal computer and displayed data structures for the Mesa strongly typed programming language. In the system, data structures could be displayed at debug time by supplying a variable's name; Incense examined the symbol table to determine the variable's type. Once a variable was chosen, the user specified via the mouse a rectangular viewing area inside the active window to display the data. Incense automatically chose the appropriate view abstraction given the space requirements. Data with insufficient screen space were displayed as grey boxes.

The system included a large number of sophisticated default views for the language's data types and structures, and it utilized a spline-based method for displaying pointers. Examples of views of different types of data structures are shown in Figure 44.1. Additionally, users could define their own data views using a predefined graphics library in Mesa. The difficulty of that process is unclear.

In order to implement Incense, Myers created the concept of an *artist*, a collection of procedures and data that handled display, erasure, and modification of the data being displayed. An artist had to be associated with a piece of data in the system in order to display it. To display pointers as arrows, Incense utilized *layouts*, special types of artists designed to locate and manage the various pointer and referent components of data.

The system was designed to allow users to also edit variables' values at run-time in the graphical presentation, but it does not appear that this feature was ever implemented. Unfortunately, the host hardware's space and processing limitations made Incense's performance relatively poor which likely restricted it to being a research prototype. The system

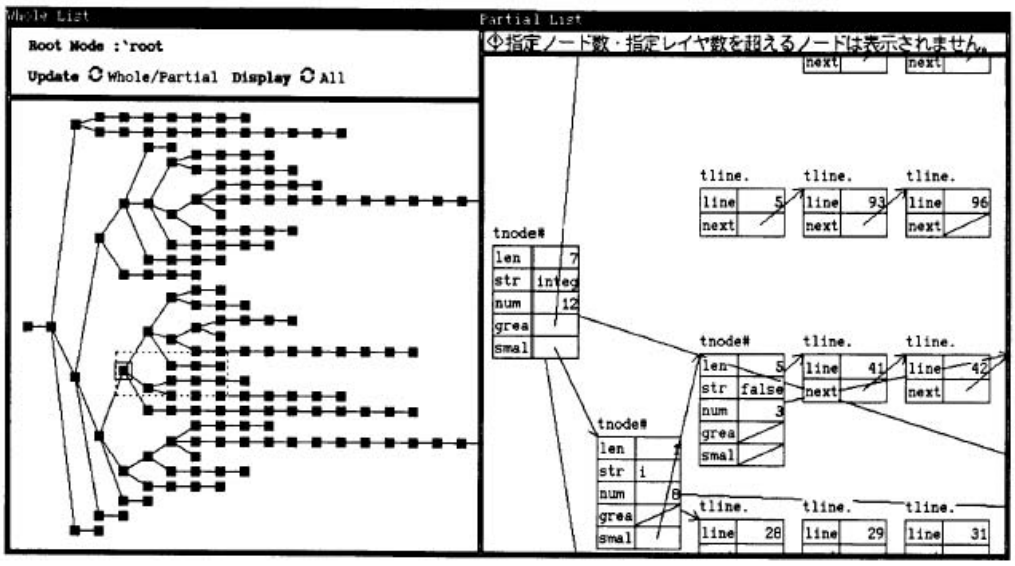


FIGURE 44.2: Data structure views in the VIPS system. Picture reprinted with permission from [13], page 47, ©1991 IEEE.

did, however, illustrate a multitude of valuable capabilities that a data structure visualization system could offer.

44.4.2 VIPS

The VIPS [13] visual debugger by Shimomura and Isoda focused on providing a flexible set of views of lists in programs. The system ran on top of the DBX debugger, with VIPS and DBX communicating through pipes. For instance, when a user would issue a request to display a list, the VIPS system would send necessary messages to DBX to extract information needed for the display. In addition to views of list data structures, VIPS provided views (windows) of program text, I/O, variable displays, as well as control windows for debugger interactions.

VIPS provided two main views of program lists, the whole list view and the partial list view. In the whole view, shown on the left side of Figure 44.2, list nodes were represented as small black rectangles thus allowing more than 100 nodes to be shown at once. The basic layout style used by the system was a classic binary tree node-link layout with the root at the left and the tree growing horizontally to the right. The whole list view assisted users in understanding structure. The partial list view, shown on the right-hand side of Figure 44.2, presented each node in a more magnified manner, allowing the user to see the values of individual fields within the node. By dragging a rectangle around a set of nodes in the whole list view, a user could issue a command to generate a new partial list view. VIPS also allowed users to select particular pointer fields in list structures and only have the nodes connected by those pointers to be displayed.

To assist debugging, VIPS also provided a *highlight* feature in which list structures with values that had recently changed were highlighted in order to draw the viewer's attention.

An earlier version of the system [12] provided multiple run-time views of executing Ada programs. Default data views in the system included scalars, linked lists, one-dimensional arrays and records. VIPS also allowed users to design their own data displays or "figures" by writing programs in the Figure Description Language FDL, a subset of Ada. FDL programs

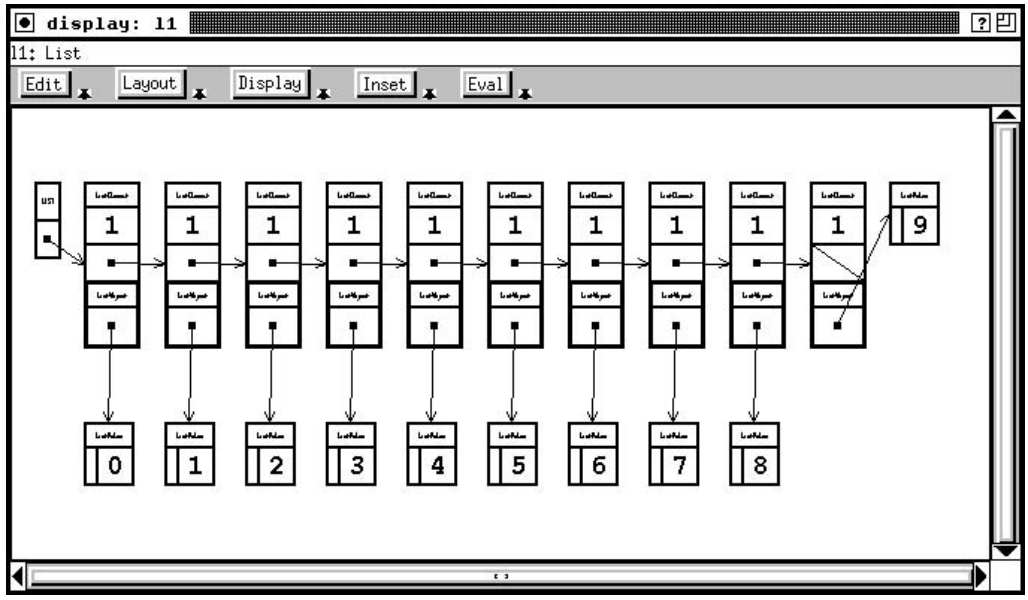


FIGURE 44.3: GELO view of a list. Picture provided courtesy of Steve Reiss.

contained parameters that could be associated with a variable or variables, promoting view control via data values. Data displays were rendered automatically by the system. When space was tight, smaller representations were utilized.

The VIPS debugger provided one of the most extensive graphically-aided debugging systems onto a particular language. By adding views of the call stack, data flow, and program structure to program data visualizations, the system explored the boundary of visual debugging efforts at that time.

44.4.3 GELO

The GELO system [14] was designed to ease the production and display of complex pictures of programs and data structures. In the system, strongly typed data structures were displayed as picture objects of types data, tile, layout, arc, and empty. GELO differed from many other data structure display systems in that diagrams were not described in a world coordinate space, but by giving a set of topological constraints for their views.

The system was organized as three components: GELO managed the specification and display of classical program and data structures; APPLE allowed a user to design and customize the way that a particular data structure would appear by providing mechanisms for defining the mapping between the data structure and the GELO structures (more than one mapping was allowed); PEAR used these mappings to allow the user to edit the structures' display, thereby modifying the underlying data structures.

GELO allowed users to specify both the data instances to be displayed and the drawing attributes of the display through fairly complex dialog box selections. It provided default displays for common structures such as lists and trees, as well as various heuristics for graph layout. A sample list view is shown in Figure 44.3 and a tree view is shown in Figure 44.4. GELO included view panning and zooming, abstractions on small objects, and scrollable windows.

Reiss noted that the system's topological layout scheme was sometimes overly restric-

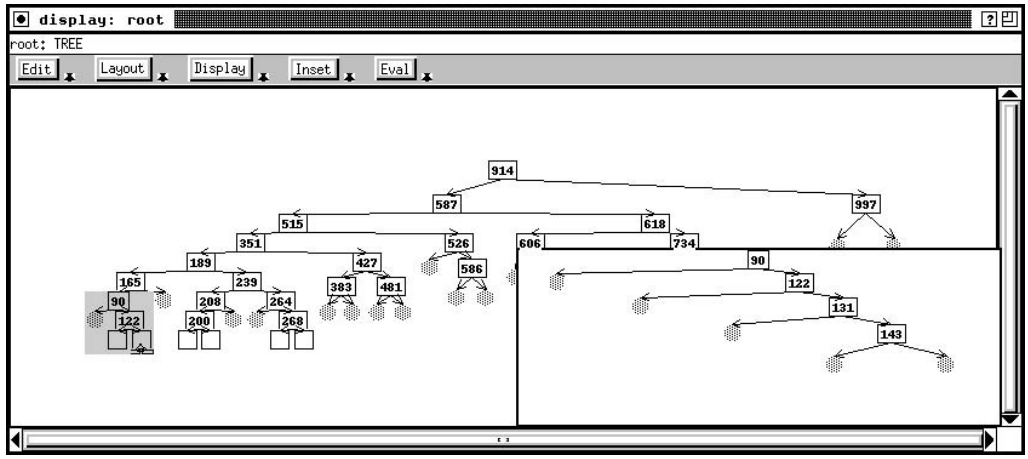


FIGURE 44.4: GELO view of a tree. Picture provided courtesy of Steve Reiss.

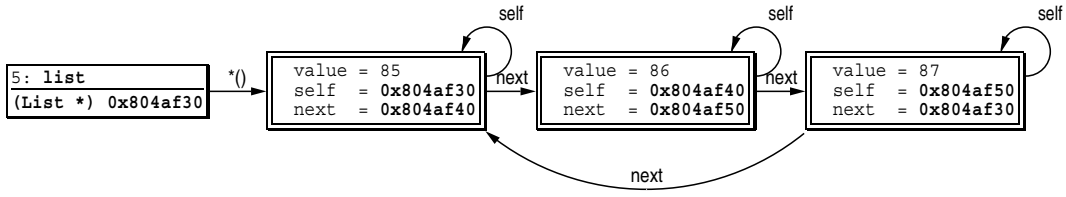


FIGURE 44.5: An example DDD representation of a linked list structure. Picture provided courtesy of Andreas Zeller.

tive; users may have wanted to have a picture "look just so" but GELO did not provide this capability. The sheer amount of dialog and menu choices for designing a display also appeared to be daunting, but the sophistication of automatic layout GELO provided was quite impressive.

44.4.4 DDD

The DDD System [15, 16], developed by Zeller, provides some of the most sophisticated graphical layout capabilities ever found in a data structure display system. DDD is technically a front-end to command-line debuggers such as GDB and DBX, and it provides all the capabilities of those debuggers. Additionally, program variables can be visualized in *displays*, rectangular windows that can be placed on a user's canvas.

DDD's sophistication comes in its techniques for visualizing pointer dereferencing. It draws a directed edge from one display to another to indicate the reference. DDD contains simple placement rules with edges pointing right and downward, or if desired, sophisticated graph layout algorithms can be invoked. Figure 44.5 shows an example data structure view from DDD.

In the default display mode, only one edge can point to any display. Thus, it is possible to have a visualization in which a display (variable or specific piece of memory) appears multiple times. DDD also provides alias detection, however, so that all program data residing at the same location are consolidated to the same display object. Thus, lists with cycles will be shown as the circular representation that one would expect.

DDD is particularly noteworthy in that the system is available as free software and it has

been widely downloaded and utilized. It is perhaps the data structure visualization that has been most used by people other than the system creators.

44.4.5 Other Systems

In addition to the systems mentioned above, a number of other data structure display tools have been developed over the years. Some of the more noteworthy ones include

- GDBX [17] - A graphical extension to the UNIX debugger DBX.
- PROVIDE [18] - An ambitious process visualization and debugging environment for programs written in a subset/variant of the C programming language.
- Amethyst [19] - A system intended to simplify data structure visualization for novice programmers.
- DS-Viewer [20] - A system that focused on the display of structures in a weakly typed language such as an assembler language.
- Lens [22] - A layer on top of the DBX debugger that provided, via programmer annotations, both data structure display capabilities as well as simple algorithm animation operations.
- SWAN [21] - A system that allowed instructors and students to easily annotate programs to produce data structure views.

44.5 Summary and Open Problems

Although the systems summarized in this paper have shown the promise that data structure visualization holds, data structure display systems still are not widely used in teaching and still are not commonly included in IDEs for programmers to use. Until data structure display systems achieve more general acceptance, the area will continue to exhibit unfulfilled potential. Anyone who has ever programmed will surely agree that a visualization of the complex interactions of data can only help comprehension and debugging of computer programs. Now that software and hardware improvements for graphical displays have made these types of visualizations routine, hopefully, this unfulfilled potential will be reached. Below we describe some specific problems that remain to be solved in order to help foster the growth of data structure visualization.

- **User-defined displays** – Invariably, advanced programmers or programmers in a specific application area will want to build customized views of particular data types and structures. Existing systems have only either provided generic default displays or have required people to do tedious graphical design in order to build custom views. A powerful data structure display system should allow users to *quickly* and *easily* demonstrate a new graphical form for a data structure. Thus, a data structure display system likely must include some form of sophisticated graphical editor or toolkit to facilitate view design.
- **Mapping data to their display** – In addition to designing new views, designers must be able to specify how data is to be interpreted to generate the views. Creating an easy-to-understand and easy-to-use mapping scheme is quite difficult, particularly if multiple pieces of data can "drive" a particular view or one piece of data can be presented in different view abstractions.
- **Complex, large data** – With the possible exception of DDD, prior data structure display systems have been better suited for programming-in-the-small with

relatively straightforward, moderate-sized data structures. Sophisticated display imagery becomes significantly more difficult as the complexity and sheer size of the data increases, thereby complicating screen layout issues. Further abstraction and mapping strategies are required to properly address these issues also.

References

[1] J. Stasko, J. Domingue, M. Brown, and B. Price, editors. *Software Visualization: Programming as a Multimedia Experience*. MIT Press, Cambridge, MA, 1998, 562 pages.

[2] B. Price, R. Baecker, and I. Small. An introduction to software visualization. In J. Stasko, J. Domingue, M. Brown, and B. Price, editors, *Software Visualization: Programming as a Multimedia Experience*, MIT Press, Cambridge, MA, 1998, 3–27.

[3] C. Ware. *Information Visualization: Perception for Design*. Morgan Kaufman, San Francisco, 2000, 438 pages.

[4] S. Card, J. Mackinlay, and B. Shneiderman, editors. *Readings in Information Visualization – Using Vision to Think*. Morgan Kaufmann, San Francisco, 1999, 686 pages.

[5] R. Spence. *Information Visualization*. ACM Press, Pearson Education, Essex, England, 2001, 206 pages.

[6] M. Brown. Perspectives on algorithm animation. *ACM CHI*, 1988, 33–38.

[7] G. Di Battista, P. Eades, R. Tamassia, and I. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, Upper Saddle River, NJ, 1998, 397 pages.

[8] R. Baecker. Experiments in on-line graphical debugging: The interrogation of complex data structures. *Hawaii International Conference on the System Sciences*, 1968, 128–129.

[9] C. Ding and P. Mateti. A framework for the automated drawing of data structure diagrams. *IEEE Transactions on Software Engineering*, 16, 5, 1990, 543–557.

[10] J. Stasko and C. Patterson. Understanding and characterizing software visualization systems. *IEEE Workshop on Visual Languages*, 1992, 3–10.

[11] B. Myers. A system for displaying data structures. *SIGGRAPH*, 1983, 115–125.

[12] S. Isoda, T. Shimomura, and Y. Ono. VIPS: A visual debugger. *IEEE Software*, 4, 3, 1987, 8–19.

[13] T. Shimomura and S. Isoda. Linked-list visualization for debugging. *IEEE Software*, 8, 3, 1991, 44–51.

[14] S. Reiss, S. Meyers, and C. Duby. Using GELO to visualize software systems. *ACM UIST*, 1989, 149–157.

[15] A. Zeller. Visual debugging with DDD. *Dr. Dobb's Journal*, 322, 2001, 21–28.

[16] T. Zimmermann and A. Zeller. Visualizing memory graphs. In S. Diehl, editor, *Software Visualization State-of-the-Art Survey*, LNCS 2269, Springer-Verlag, 2002, 191–204.

[17] D. Baskerville. Graphic presentation of data structures in the DBX debugger. Technical Report UCB/CSD 86/260, University of California at Berkeley, 1985.

[18] T. Moher. PROVIDE: A process visualization and debugging environment. *IEEE Transactions on Software Engineering*, 14, 6, 1988, 849–857.

[19] B. Myers, R. Chandhok, and A. Sareen. Automatic data visualization for novice Pascal programmers. *IEEE Workshop on Visual Languages*, 1988, 192–198.

[20] D. Pazel. DS-Viewer—an interactive graphical data structure presentation facility. *IBM Systems Journal*, 28, 2, 1989, 307–323.

[21] C. Shaffer, L. Heath, J. Nielsen, and J. Yang. SWAN: A student-controllable data structure visualization system. *ED-MEDIA*, 1996, 632–637.

[22] S. Mukherjea and J. Stasko. Toward visual debugging: Integrating algorithm animation capabilities within a source level debugger. *ACM Transactions on Computer-Human Interaction*, 1, 3, 1994, 215–244.

45

Drawing Trees

Sebastian Leipert
Center of Advanced European Studies and Research

45.1 Introduction

Constructing geometric representations of graphs in a readable and efficient way is crucial for understanding the inherent properties of the structures in many applications. The desire to generate a layout of such representations by algorithms and not by hand meeting certain aesthetics has motivated the research area *Graph Drawing*. Examples of these aesthetics include minimizing the number of edge crossings, minimizing the number of edge bends, minimizing the display area of the graph, visualizing a common direction (flow) in the graph, maximizing the angular resolution at the vertices, and maximizing the display of symmetries. Certainly, two aesthetic criteria cannot be simultaneously optimized in general and it depends on the data which criterion should be preferably optimized. Graph Drawing Software relies on a variety of mathematical results in graph theory, topology, geometry, as well as computer science techniques mainly in the areas algorithms and data structures, software engineering and user interfaces.

A typical graph drawing problem is to create for a graph $G = (V, E)$ a geometric representation where the nodes in V are drawn as geometric objects such as points or two dimensional shapes and edges $(u, v) \in E$ are drawn as simple Jordan curves connecting the geometric objects associated with u and v. Apart from the, in the context of this book obvious, visualization of Data Structures, other application areas are, e.g., software engineering (Unified Modeling Language (UML), data flow diagrams, subroutine-call graphs) databases (entity-relationship diagrams), decision support systems for project management (business process management, work flow).

A fundamental issue in Automatic Graph Drawing is to display trees, since trees are a common type of data structure (Chapter 3). Thus a good drawing of a tree is often a powerful intuitive guide for analyzing data structures and debugging their implementations. It is a trivial observation that a tree $T = (V, E)$ always admits a planar drawing, that is a drawing in the plane such that no two edges cross. Thus all algorithms that have been developed construct a planar drawing of a tree. Furthermore it is noticed that for trees the

condition $|E| = |V| - 1$ holds and therefore the time complexity of the layout algorithms is always given in dependency to the number of nodes $|V|$ of a tree.

In 1979, Wetherell and Shannon [24] presented a linear time algorithm for drawing binary trees satisfying the following aesthetic requirements: the drawing is *strictly upward*, i.e. the y-coordinate of a node corresponds to its level, so that the hierarchical structure of the tree is displayed; the left child of a node is placed to the left of the right child, i.e., the order of the children is displayed; finally, each parent node is centered over its children. Moreover, edges are drawn *straight line*. Nevertheless, this algorithm showed some deficiencies. In 1981, Reingold and Tilford [17] improved the Wetherell-Shannon algorithm by adding the following feature: each pair of isomorphic subtrees is drawn identically up to translation, i.e., the drawing does not depend on the position of a subtree within the complete tree. They also made the algorithm symmetrical: if all orders of children in a tree are reversed, the computed drawing is the reflected original one. The width of the drawing is not always minimized subject to these conditions, but it is close to the minimum in general. The algorithm of Reingold and Tilford that runs in linear time, is given in Section 45.3. Figure 45.1 gives an example of a typical layout produced by Reingold and Tilford's algorithm.

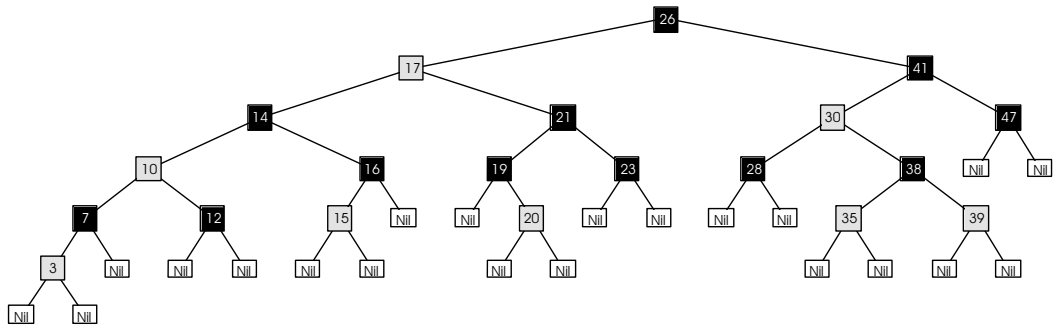


FIGURE 45.1: A typical layout of a binary tree. The tree is a red-black tree as given in [3].

Extending this algorithm to rooted ordered trees of unbounded degree in a straightforward way produces layouts where some subtrees of the tree may get clustered on a small space, even if they could be dispersed much better. This problem was solved in 1990 by the quadratic time algorithm of Walker [23], which spaces out subtrees whenever possible. Very recently Buchheim et al. [2] showed how to improve the algorithm of Walker to linear running time. The algorithm by Buchheim et al. is given in Section 45.4, including a pseudo code that allows the reader a straightforward application of the algorithm. This algorithm for n-ary trees gives similar results on binary trees as the algorithm of Reingold and Tilford.

In Section 45.5 algorithms for straight line circular drawings (see [1, 8, 9, 16]) are introduced. This type of layout proved to be useful for free trees that do not have a designated node as a root. Section 45.6 presents hv-drawings, an approach for drawing binary and n-ary trees on a grid. For binary trees, edges in an hv-drawing are drawn as rightward horizontal or downward vertical segments. This type of drawing is straight line *upward orthogonal*. It is, however, not strictly upward. Such drawings are e.g. investigated in [4, 5, 12–15, 20, 22].

Variations of the hv-layout algorithms have been used to obtain results on minimal area requirements of tree layouts. Shiloach and Crescenzi et al. [4, 20] showed that any rooted tree admits an upward straight-line drawing with area $O(|V| \log |V|)$. Crescenzi et al. [4]

moreover proved that there exists a class of rooted trees that require $\Omega(|V|\log|V|)$ area if drawn strictly upward straight-line. In [4, 7] algorithms are given that produce $O(|V|)$-area strictly upward straight line drawings for some classes of balanced trees such as complete binary trees, Fibonacci trees, and AVL trees. These results have been expanded in [6] to a class of balanced trees that include k-balanced trees, red-black trees, BB[α]-trees, and (a, b)-trees. Garg et al. [10] gave an $O(|V|\log|V|)$ area algorithm for ordered trees that produces upward layouts with polyline edges. Moreover they presented an upward orthogonal (not necessarily straight line) algorithm with an asymptotically optimal area of $O(|V|\log\log|V|)$.

Shin et al. [21] showed that bounded degree trees admit upward straight line layouts with an $O(|V|\log\log|V|)$ area. The result can be modified to derive an algorithm that gives an upward polyline grid drawing with an $O(|V|\log\log|V|)$ area, at most one bend per edge, and $O(|V|/\log|V|)$ bends in total. Moreover in [21] an $O(|V|\log\log|V|)$ area algorithm has been presented for non upward straight line grid layouts with arbitrary aspect ratio. Recently, Garg and Rusu [11] improved this result giving for binary trees an $O(|V|)$ area algorithm for non upward straight line orthogonal layouts with a pre specified aspect ratio in the range of $[1, |V|^{\alpha}]$, with $\alpha \in [0, 1)$ that can be constructed in $O(|V|)$.

45.2 Preliminaries

A (*rooted*) *tree* T is a directed acyclic graph with a single source, called the *root* of the tree, such that there is a unique directed path from the root to any other node. The *level* $l(v)$ of a node v is the length of this path. The largest level of any node in T is the height $h(T)$ of T. For each edge (v, w), v is the *parent* of w and w is a *child* of v. Children of the same node are called *siblings*. Each node w on the path from the root to a node v is called an *ancestor* of v, while v is called a *descendant* of w. A node that has no children is a *leaf*. If v_1 and v_2 are two nodes such that v_1 is not an ancestor of v_2 and vice versa, the *greatest distinct ancestors* of v_1 and v_2 are defined as the unique ancestors w_1 and w_2 of v_1 and v_2, respectively, such that w_1 and w_2 are siblings. Each node v of a rooted tree T induces a unique subtree $T(v)$ of T with root v.

Binary trees are trees with a maximum number of two children per node. In contrast to binary trees, trees that have an arbitrary number of children are called *n-ary trees*. A tree is said to be ordered if for every node the order of its children is fixed. The first (last) child according to this order is called the *leftmost* (*rightmost*) child. The *left* (*right*) *sibling* of a node v is its predecessor (successor) in the list of children of the parent of v. The *leftmost* (*rightmost*) *descendant* of v on a level l is the leftmost (rightmost) node on the level l belonging to the subtree $T(v)$ induced by v. Finally, if w_1 is the left sibling of w_2, v_1 is the rightmost descendant of w_1 on some level l, and v_2 is the leftmost descendant of w_2 on the same level l, the node v_1 is called the *left neighbor* of v_2 and v_2 is the *right neighbor* of v_1.

To *draw* a tree into the plane means to assign x- and y-coordinates to its nodes and to represent each edge (v, w) by a straight line connecting the points corresponding to v and w. Objects that represent the nodes are centered above the point corresponding to the node. The computation of the coordinates must respect the sizes of the objects. For simplicity however, we assume throughout this paper that all nodes have the same dimensions and that the minimal distance required between neighbors is the same for each pair of neighbors. Both restrictions can be relaxed easily, since we will always compare a single pair of neighbors.

Reingold and Tilford have defined the following aesthetic properties for drawings of trees:

(A1) The layout displays the hierarchical structure of the tree, i.e., the y-coordinate

of a node is given by its level.

(A2) A parent is centered above its children.

(A3) The drawing of a subtree does not depend on its position in the tree, i.e., isomorphic subtrees are drawn identically up to translation.

By their appearance, these drawings are called *level drawings*. If the tree that has to be drawn is ordered, we additionally require the following:

(A4) The order of the children of a node is displayed in the drawing, thus a left child is placed to the left with a smaller x-coordinate and a right child is placed to the right with a bigger x-coordinate.

(A5) A tree and its mirror image are drawn identically up to reflection.

Here, the *reflection* of an ordered tree is the tree with reversed order of children for each parent node. It is desirable to find a layout satisfying (A1) to (A5) with a small width. However, it has been shown by Supovit and Reingold [18] that achieving the minimum grid width is NP-hard. Moreover, there is no polynomial time algorithm for achieving a width that is smaller than $\frac{25}{24}$ time the minimum width, unless P = NP. If on the other hand continuous coordinates instead of integral coordinates are allowed, minimum width drawing can be found in linear time by applying linear programming techniques (see [18]).

45.3 Level Layout for Binary Trees

For ordered binary trees, the first linear time algorithm satisfying (A1) to (A5) was presented by Reingold and Tilford [17]. The algorithm follows the divide and conquer principle implemented in form of a postorder traversal of a tree $T = (V, E)$ and places the nodes on grid units. For each $v \in V$ with left and right child w_{left}, w_{right} the algorithm computes layouts for the trees $T(w_{left})$ and $T(w_{right})$ up to horizontal translation. When v is visited, the drawing of the right subtree $T(w_{right})$ is shifted to the right such that on every level l of the subtrees the rightmost node v_{left} of $T(w_{left})$ and its neighbor, the leftmost node v_{right} of $T(w_{right})$ are separated at least by two or three grid points. The separation value between v_{left} and v_{right} is chosen such that v can be centered above the roots of $T(w_{left})$ and $T(w_{right})$ at an integer grid coordinate.

Shifting $T(w_{right})$ is partitioned into two subtask: first, determining the amount of shift and second, performing the shift of the subtree. To determine the amount of shift, define the *left contour* of a tree T to be the vertices with minimum x-coordinate at each level in the tree. The *right contour* is defined analogously. For an illustration, see Figure 45.2, where nodes belonging to the contours are shaded. To place $T(w_{right})$ as close to $T(w_{left})$ as possible, the right contour of $T(w_{left})$ and the left contour of $T(w_{right})$ are traversed calculating for every level l the amount of shift to separate the two subtrees on that specific level l. The maximum over all shift then gives the displacement for the trees such that they do not overlap. Since each node belongs to the traversed part of the left contour of the right subtree at most for one subtree combination, the total number of such comparisons is linear for the complete tree.

In order to achieve linear running time it must be ensured that the contours are traversed without traversing (too many) nodes not belonging to the contours. This is achieved by introducing a *thread* for each leaf of the subtree that has a successor in the same contour. The thread is a pointer to this successor. See Figure 45.2 for an illustration where the threads are represented by dotted arrows. For every node of the contour, we now have a pointer to its successor in the left (right) contour given either by its leftmost (rightmost)

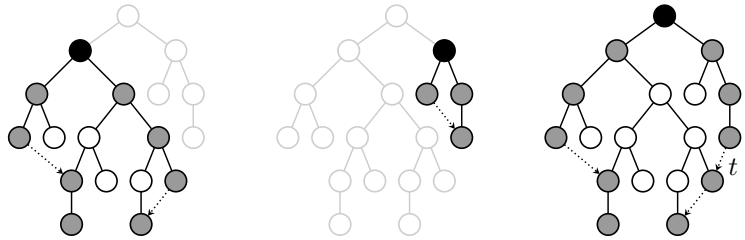


FIGURE 45.2: Combining two subtrees and adding a new thread t.

child or by the thread. Finally, to update the threads, a new thread has to be added whenever two subtrees of different height are combined.

Once the shift has been determined for $T(w_{right})$ we omit shifting the subtree by updating the coordinates of its nodes, since this would result in quadratic running time. Instead, the position of each node is only determined preliminary and the shift for $T(w_{right})$ is stored at w_{right} as the *modifier* $mod(w_{right})$ (see [24]). Only $mod(w_{right})$ and the preliminary x-coordinate $prelim(v)$ of the parent v are adjusted by the shift of $T(w_{right})$. The modifier of w_{right} is interpreted as a value to be added to all preliminary x-coordinates in the subtree rooted at w_{right}, except for w_{right} itself. Thus, the coordinates of a node v in T in the final layout is its preliminary position plus the aggregated modifier $modsum(v)$ given by the sum of all modifiers on the path from the parent of v to the root. Once all preliminary positions and modifiers have been determined the final coordinates can be easily computed by a top-down sweep.

THEOREM 45.1 *[Reingold and Tilford [17]] The layout algorithm for binary trees meets the aesthetic requirements (A1)–(A5) and can be implemented such that the running time is $O(|V|)$.*

Proof By construction of the algorithm it is obvious that the algorithm meets (A1)–(A5). So it is left to show that the running time is linear in the number of nodes. Every node of T is traversed once during the postorder and the preorder traversal. So it is left to show that the time needed to traverse the contour of the two subtrees $T(w_{left})$ and $T(w_{right})$ for every node v with children w_{left} and w_{right} is linear in the number of nodes of T over all such traversals.

It is obviously necessary to travel down the contours of $T(w_{left})$ and $T(w_{right})$ only as far as the tree with lesser height. Thus the time spent processing a vertex v in the postorder traversal is proportional to the minimum of the height of $h(T(w_{left}))$ and $h(T(w_{right}))$. The running time of the postorder traversal is then given by:

$$\sum_{v \in T} (1 + \min\{h(T(w_{left})), h(T(w_{right}))\}) = |V| + \sum_{v \in T} \min\{h(T(w_{left})), h(T(w_{right}))\}$$

The sum $\sum_{v \in T} \min\{h(T(w_{left})), h(T(w_{right}))\}$ can be estimated as follows. Consider a node w that is part of a contour of a subtree $T(w_{left})$. When comparing the right contour of $T(w_{left})$ and the left contour of $T(w_{right})$ two cases are possible. Either w is not traversed and therefore will be part of the contour of $T(v)$ or it is traversed when comparing $T(w_{left})$ and $T(w_{right})$. In the latter case, w is part of the right contour of $T(w_{left})$ and after merging $T(w_{left})$ and $T(w_{right})$ the node w is not part of the right contour of $T(v)$. Thus every node

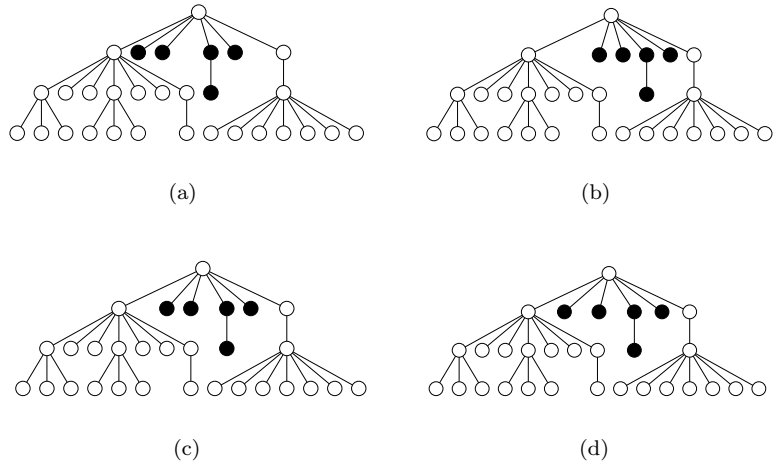


FIGURE 45.3: Extending the Reingold-Tilford algorithm to trees of unbounded degree.

of T is traversed at most twice and the total number of comparisons is bounded by $|V|$.

We do not give the full algorithm for drawing binary trees here. Instead, the layout algorithm for n-ary trees is given in full length in the next section. The methods presented there are an expansion of Reingold and Tilfords algorithm to the more general case and still proceed for binary trees as described in this section.

45.4 Level Layout for n-ary Trees

A straightforward manner to draw trees of unbounded degree is to adjust the Reingold-Tilford algorithm by traversing the children of each node v from left to right, successively computing the preliminary coordinates and the modifiers.

This however violates property (A5): the subtrees are placed as close to each other as possible and small subtrees between larger ones are piled to the left; see Figure 45.3(a). A simple trick to avoid this effect is to add an analogous second traversal from right to left; see Figure 45.3(b), and to take average positions after that. This algorithm satisfies (A1) to (A5), but smaller subtrees are usually clustered then; see Figure 45.3(c).

To obtain a layout where smaller subtrees are spaced out evenly between larger subtrees as for example shown in Figure 45.3(d), we process the subtrees for each node $v \in V$ from left to right, see Figure 45.4. In a first step, every subtree is then placed as close as possible to the right of its left subtrees. This is done similarly to the algorithm for binary trees as described in Section 45.3 by traversing the left contour of the right subtree $T(w_{right})$ and the right contour of the subforest induced by the left siblings of w_{right}.

Whenever two conflicting neighbors v_{left} and v_{right} are detected, forcing v_{right} to be shifted to the right by an amount of σ, we apply an appropriate shift to all smaller subtrees between the subtrees containing v_{left} and v_{right}.

More precisely, let w_{left} and w_{right} be the greatest distinct ancestors of v_{left} and v_{right}. Notice that both w_{left} and w_{right} are children of the node v that is currently processed. Let k be the number of children $w_1, w_2, \ldots, w_k$ of the current root between w_{left} and $w_{right}+1$. The subtrees between w_{left} and w_{right} are spaced out by shifting the subtree $T(w_i)$ to the

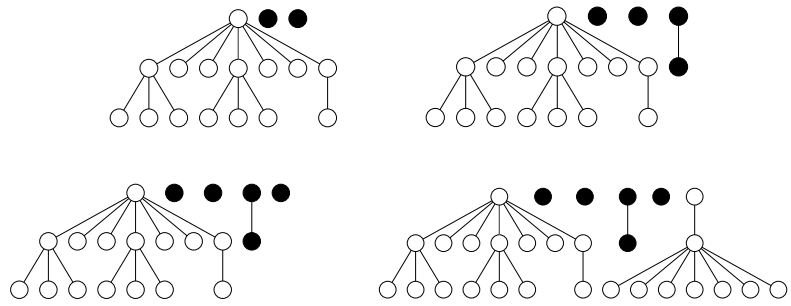


FIGURE 45.4: Spacing out the smaller subtrees.

right of w_{left} by an amount of $i \cdot \sigma / k$, for $i = 1, 2, \ldots, k$. We notice that spacing out the smaller subtrees may result in subtrees that are shifted more than once. Below, the effect on the running time for computing the shifts is determined and it is shown how to obtain a linear time implementation.

It is easy to see that this approach satisfies (A1)–(A5) and in addition spaces out the smaller subtrees between larger subtrees evenly. In contrast to the algorithm for binary trees (see Section 45.3) nodes are not placed on integer coordinates. Instead real values are used.

Figure 45.5 gives a layout example of a 5-ary tree. The layout algorithm considers different nodes sizes.

Figure 45.6 shows a layout of a PQ-tree produced by the algorithm for binary trees. A straightforward modification of the algorithm allows the application of different nodes sizes to the Q-nodes and the usage of strictly vertical edges for the children of the Q-nodes.

The algorithm TreeLayout given below works as a frame that initializes modifiers, threads, and ancestors (see Section 45.4.3), before evoking the methods PrePosition and Adjust. The method PrePosition computes the shifts and preliminary positions of the nodes.

forall *nodes* v *of* T **do**
    $mod(v) = thread(v) = 0$;
    $ancestor(v) = v$;
od
let r be the root of T;
PrePosition(r);
Adjust(r, $-prelim(r)$);

Algorithm 1: TreeLayout(T)

Based on the results of the function PrePosition the function Adjust given in Algorithm 2 computes the final coordinates by summing up the modifiers recursively.

45.4.1 PrePosition

Algorithm 3 presents the method PrePosition(v) that computes a preliminary x-coordinate for a node v. PrePosition is applied recursively to the children of v. After each call

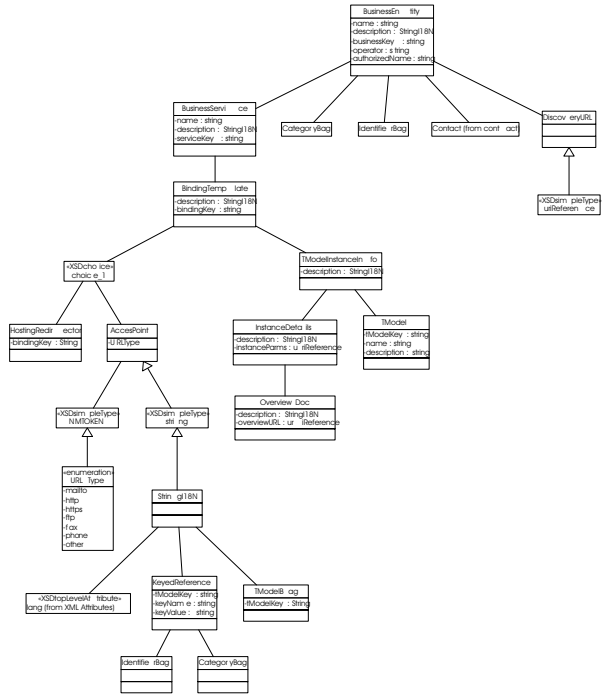


FIGURE 45.5: A level layout of an n-ary tree with different node sizes.

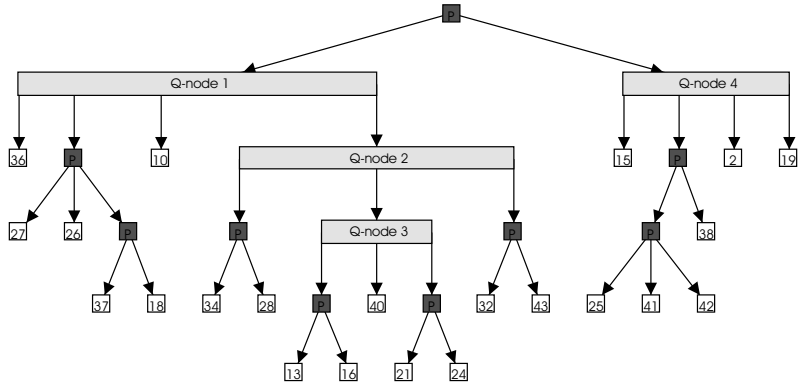


FIGURE 45.6: A level layout of a PQ-tree.

$x(v) = prelim(v) + m$;
let $y(v)$ be the level of v;
forall *children* w *of* v **do**
    ADJUST(w, $m + mod(v)$);
od

Algorithm 2: ADJUST(v, m)

of PREPOSITION on a child w a function APPORTION is executed on w. The procedure APPORTION is the core of the algorithm and shifts a subtree such that it does not conflict with its left subforest. After spacing out the smaller subtrees by calling EXECUTESHIFTS,

the node v is placed to the midpoint of its outermost children. The value *distance* prescribes the minimal distance between two nodes. If objects of different size are considered for the representation of the nodes, or if different minimal distances, i.e. between subtrees, are specified, the value *distance* has to be modified appropriately.

```
if v is a leaf then
    prelim(v) = 0;
    if v has a left sibling w prelim(v) = prelim(w) + distance;
else
    let defaultAncestor be the leftmost child of v;
    forall children w of v from left to right do
        PrePosition(w);
        Apportion(w, defaultAncestor);
    od
    ExecuteShifts(v);
    midpoint = 1/2(prelim(leftmost child of v) + prelim(rightmost child of v));
    if v has a left sibling w then
        prelim(v) = prelim(w) + distance;
        mod(v) = prelim(v) − midpoint;
    else
        prelim(v) = midpoint;
    end
end
```

Algorithm 3: PrePosition(v)

45.4.2 Combining a Subtree and Its Left Subforest

Before presenting Apportion in detail, we need to consider efficient strategies for the different tasks that are performed by this function.

Similar to the threads used for binary trees (see Sect. 45.3), Apportion uses threads to follow the contours of the right subtree and the left subforest. The fact that the left subforest is no tree in general does not create any additional difficulty.

One major task of Apportion is that it has to space out the smaller subtrees between the larger ones. More precisely, if Apportion shifts a subtree to the right to avoid a conflict with its left subforest, Apportion has to make sure that the shifts of the smaller subtrees of the left subforest are determined.

A straightforward implementation computes the shifts for the intermediate smaller subtrees after the right subtree has been moved. However, as has been shown in [2], this strategy has an aggregated runtime of $\Omega(|V|^{3/2})$. To prove this consider a tree T^k such that the root has k children $v_1, v_2, \ldots, v_k$ (see Figure 45.7 for $k = 3$). The children are numbered from left to right. Except for v_1 let the i-th child v_i be the root of a subtree $T^k(v_i)$ that consist of a chain of i nodes. Between each pair v_i, v_{i+1}, $i = 1, 2, \ldots, k-1$, of these children, add k children as leaves. Moreover, the subtree $T^k(v_1)$ is modified as follows. Its root v_1 has $2k+5$ children, and up to level $k-1$, every rightmost child of the $2k+5$ children again

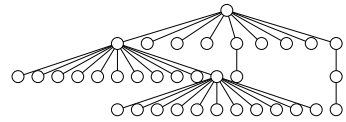


FIGURE 45.7: T^3.

has $2k + 5$ children. The number of nodes of T^k is

$$1 + \sum_{i=1}^{k} i + (k-1)k + (k-1)(2k+5) \in \Theta(k^2) \; .$$

When traversing the nodes $v_1, v_2, \ldots, v_k$ in order to determine their shifts, the subtree chain $T^k(v_i)$ conflicts with the first subtree $T^k(v_1)$ on level i. Hence, all $(i-1)(k+1)-1$ smaller subtrees between $T^k(v_1)$ and $T^k(v_i)$ are shifted. Thus, the total number of shifting steps is

$$\sum_{i=2}^{k}((i-1)(k+1)-1) = (k+1)k(k-1)/2 - k + 1 \in \Theta(k^3) \; .$$

Since the number of nodes in T_k is $|V| \in \Theta(k^2)$, this shows that a straightforward implementation needs $\Omega(|V|^{3/2})$ time in total.

Let $T(v_i)$, $i \in \{2, 3, \ldots, k\}$ be the subtree that has to be shifted. Let v_{left} and v_{right} be two neighboring nodes on some level l such that

- v_{left} is in the left subforest $\cup_{h=1}^{i-1} T(v_h)$ and v_{right} is in $T(v_i)$ respectively, and
- v_{left} and v_{right} determine the shift of $T(v_i)$.

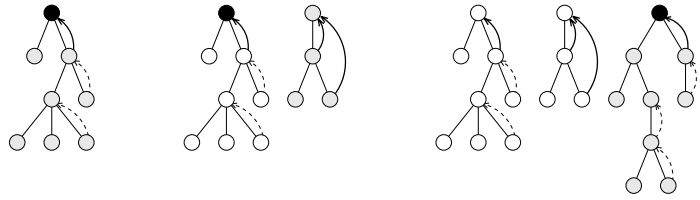
In order to develop an efficient method for spacing out the smaller subtrees two tasks have to be solved.

- The tree $T(v_j)$, $j \in \{1, 2, \ldots, i-1\}$, with $v_{left} \in T(v_j)$ has to be maintained in order to determine the smaller subtrees to the left of $T(v_i)$ that have to be spaced out.
- Shifting the smaller subtrees between $T(v_j)$ and $T(v_i)$ has to be done efficiently.

The next Section 45.4.3 shows how to obtain the tree $T(v_j)$ efficiently. Section 45.4.4 gives a detailed description on how to compute the shift of $T(v_i)$ and Section 45.4.5 presents a method that spaces out smaller subtrees.

45.4.3 Ancestor

We first describe how to obtain the subtree $T(v_j)$ that contains the node v_{left}. The problem is equivalent to finding the greatest distinct ancestors w_{left} and w_{right} of the nodes v_{left} and v_{right}, where in this case w_{left} is equal to the root v_j of the subtree that we need to determine and $w_{right} = v_i$. It is possible to apply an algorithm by Schieber and Vishkin [19] that determines for each pair of nodes its greatest distinct ancestors in constant time, after an $O(|V|)$ preprocessing step. Since their algorithm is somewhat tricky, and one of the greatest distinct ancestors, namely v_i, is known anyway, we apply a much simpler algorithm. Furthermore, as v_{right} is always the right neighbor of v_{left}, the left one of the greatest distinct ancestors only depends on v_{left}. Thus we may shortly call it *the ancestor* of v_{left} in the following.



(a) (b) (c)

FIGURE 45.8: Adjusting ancestor pointers when adding new subtrees: the pointer $ancestor(u)$ is represented by a solid arrow if it is up to date and by a dashed arrow if it has expired. In the latter case, the *defaultAncestor* is used and drawn black. When adding a small subtree, all ancestor pointers $ancestor(u)$ of its right contour are updated. When adding a large subtree, only *defaultAncestor* is updated.

To store the ancestor of a node u a pointer $ancestor(u)$ is introduced and initialized to u itself. The pointer $ancestor(u)$ for any node u is not updated throughout the algorithm. Instead, a *defaultAncestor* is used and ancestors are only determined for the nodes u on the right contour of the left subforest $\cup_{h=1}^{i-1} T(v_h)$ during the shift of the subtree $T(v_i)$. This strategy ensures that we obtain linear running time.

We make sure that for every $i = 2, 3, \ldots, k$ the following property for all nodes u on the right contour of $\cup_{h=1}^{i-1} T(v_h)$ holds:

(*) If $ancestor(u)$ is up to date, i.e., u is a child of v, then $ancestor(u) = w_{left}$; otherwise, *defaultAncestor* is the correct $ancestor(u)$.

For $i = 2$ we have that $\cup_{h=1}^{i-1} T(v_h) = T(v_1)$. The *defaultAncestor* for all nodes on the right contour is obviously v_1 and therefore *defaultAncestor* is set equal to v_1. It is easy to recognize if a node u on the right contour has a pointer $ancestor(u)$ that is up to date: either $ancestor(u) = v_1$, or the level $l(ancestor(u))$ is greater than $l(v_1)$. This obviously fulfills property (*), see Figure 45.8(a) for an illustration.

After adding a subtree $T(v_{i-1})$ to $\cup_{h=1}^{i-2} T(v_h)$ for each $i = 3, 4, \ldots, k$ two cases need to be considered. If the height $h(T(v_{i-1}))$ is lesser or equal to the height of the subforest $\cup_{h=1}^{i-2} T(v_h)$, the pointer $ancestor(u)$ is set to v_{i-1} for all nodes u on the right contour of $T(v_{i-1})$. This obviously fulfills property (*); see Figure 45.8(b) for an illustration. Moreover, the number of update operations is equal to the number of comparisons between the nodes on the left contour of $T(v_{i-1})$ and their neighbors in $\cup_{h=1}^{i-2} T(v_h)$. Hence the total number of all these update operations is in $O(|V|)$.

If the height $h(T(v_{i-1}))$ is greater than the height of $\cup_{h=1}^{i-2} T(v_h)$, we omit updating all nodes on the right contour of $T(v_{i-1})$ in order to obtain linear running time. Instead, it suffices to set *defaultAncestor* to v_{i-1}, since either $ancestor(u) = v_{i-1}$, or $l(ancestor(u)) > l(v_1)$ holds for any node in the right contour, and all smaller subtrees in the $\cup_{h=1}^{i-1} T(v_h)$ do not contribute to the right contour anymore. Thus property (*) is fulfilled; see Figure 45.8(c) for an illustration.

In algorithm 4 the function ANCESTOR is given. It returns w_{left} of u as described above.

if *ancestor(u) is a sibling of* w_{right} **then**
  return $ancestor(w_{right})$;
else
  return *defaultAncestor*;
end

Algorithm 4: ANCESTOR(u,w_{right},*defaultAncestor*)

45.4.4 Apportion

To give the function APPORTION in Algorithm 5 a more readable annotation, we use subscripts f and o to describe the different contours of the left subforest and the right subtree. The subscript f is used for neighboring nodes on the contours that are *facing* each other and thus is used for the nodes on the right contour of the left subforest $\cup_{h=1}^{i-1} T(v_h)$ and the left contour of the right subtree $T(v_i)$, $i = 2, 3, \ldots, k$. We use o for the left contour of $\cup_{h=1}^{i-1} T(v_h)$ and for the right contour of $T(v_i)$, describing the nodes that are on the *outside* of the combined subforest $\cup_{h=1}^{i} T(v_h)$.

The nodes traversing the contours are v^f_{right}, v^f_{left}, v^o_{left}, and v^o_{right}, where the subscript *left* describes nodes of the left subforest and *right* nodes of the right subtree. For summing up the modifiers along the contour (see also Sect. 45.3), respective variables s^f_{right}, s^f_{left}, s^o_{left}, and s^o_{right} are used.

Whenever two nodes of the facing contours conflict, we compute the left one of the greatest distinct ancestors using the function ANCESTOR and call MOVESUBTREE to shift the subtree and prepare the shifts of smaller subtrees.

Finally, a new thread is added (if necessary) as explained in Section 45.3. Observe that we have to adjust $ancestor(v^o_{right})$ or *defaultAncestor* to keep property (*). The functions NEXTLEFT and NEXTRIGHT return the next node on the left and right contour, respectively (see Algorithms 6 and 7).

45.4.5 Shifting the Smaller Subtrees

For spacing out the smaller subtrees evenly, the number of the smaller subtrees between the larger ones has to be maintained. Since simply counting the number of smaller children between two larger subtrees $T(v_j)$ and $T(v_i)$, $1 \leq j < i \leq k$ would result in $\Omega(n^{3/2})$ time in total, it is determined as follows. The children of v are numbered consecutively. Once the pair of nodes v_{left}, v_{right} that defines the maximum shift on $T(v_i)$ has been determined, the greatest distinct ancestors $w_{left} = v_j$ and $w_{right} = v_i$ are easily determined by the approach described in Section 45.4.3. The number $i - j - 1$ gives the number of in between subtrees in constant time.

In order to obtain a linear runtime, we make sure that each smaller subtree between a pair of larger ones is shifted at most once.

Thus whenever a subtree of $T(v_i)$ is considered to be placed next to the left subforest $\cup_{h=1}^{i-1} T(v_h)$, we do not modify the shift of the smaller subtrees. Such an approach would result in quadratic runtime by the fact that smaller subtrees are shifted every time they are in between a pair of greater subtrees. Instead, a system is installed, that allows to adopt the shift of the smaller subtrees by traversing the nodes $v_1, v_2, \ldots, v_k$ once after the last child v_k of v has been shifted.

For every node v_g, $g = 1, 2, \ldots, k$, real numbers $shift(v_g)$ and $change(v_g)$ are introduced and initialized by zero. Let $T(v_j)$, $1 \leq j < i \leq k$, be the subtree that defines the σ on the subtree $T(v_i)$. Let $\theta = i - j$ be the number of subtrees between v_i and v_j, plus 1. Then, for $t = 1, 2, \ldots, i - j - 1$, the t-th subtree $T(v_{j+t})$ has to be moved by $t \cdot \sigma/\theta$. In other words:

if v has a left sibling w **then**
  $v^f_{right} = v^o_{right} = v$;
  $v^f_{left} = w$;
  let v^o_{left} be the leftmost sibling of v^f_{right};
  $s^f_{right} = mod(v^f_{right})$;
  $s^o_{right} = mod(v^o_{right})$;
  $s^f_{left} = mod(v^f_{left})$;
  $s^o_{left} = mod(v^o_{left})$;
  while NEXTRIGHT$(v^f_{left}) \neq 0$ **and** NEXTLEFT$(v^f_{right}) \neq 0$ **do**
    $v^f_{left} =$ NEXTRIGHT(v^f_{left});
    $v^f_{right} =$ NEXTLEFT(v^f_{right});
    $v^o_{left} =$ NEXTLEFT(v^o_{left});
    $v^o_{right} =$ NEXTRIGHT(v^o_{right});
    $ancestor(v^o_{right}) = v$;
    $\sigma = (prelim(v^f_{left}) + s^f_{left}) - (prelim(v^f_{right}) + s^f_{right}) + distance$;
    if $\sigma > 0$ **then**
      MOVESUBTREE(ANCESTOR$(v^f_{left}, v, defaultAncestor), v, \sigma)$;
      $s^f_{right} = s^f_{right} + \sigma$;
      $s^o_{right} = s^o_{right} + \sigma$;
    end
    $s^f_{left} = s^f_{left} + mod(v^f_{left})$;
    $s^f_{right} = s^f_{right} + mod(v^f_{right})$;
    $s^o_{left} = s^o_{left} + mod(v^o_{left})$;
    $s^o_{right} = s^o_{right} + mod(v^o_{right})$;
  end
end
if NEXTRIGHT$(v^f_{left}) \neq 0$ **and** NEXTRIGHT$(v^o_{right}) = 0$ **then**
  $thread(v^o_{right}) =$ NEXTRIGHT(v^f_{left});
  $mod(v^o_{right}) = mod(v^o_{right}) + s^f_{left} - s^o_{right}$;
end
if NEXTLEFT$(v^f_{right}) \neq 0$ **and** NEXTLEFT$(v^o_{left}) = 0$ **then**
  $thread(v^o_{left}) =$ NEXTLEFT(v^f_{right});
  $mod(v^o_{left}) = mod(v^o_{left}) + s^f_{right} - s^o_{left}$;
  $defaultAncestor = v$;
end

Algorithm 5: APPORTION$(v, defaultAncestor)$

if *v has a child* **then**
  return the leftmost child of v;
else
  return $thread(v)$;
end

Algorithm 6: NEXTLEFT(v)

if *v has a child* **then**
    return the rightmost child of v;
else
    return $thread(v)$;
end

Algorithm 7: NEXTRIGHT(v)

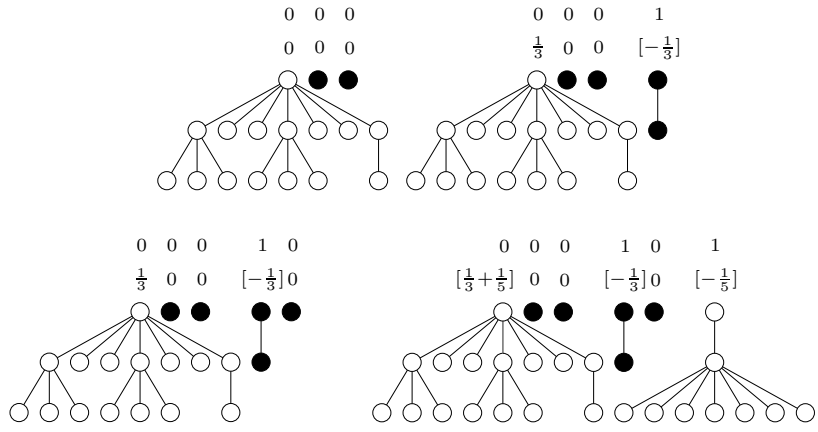


FIGURE 45.9: Aggregating the shifts: the top number at every node u indicates the value of $shift(u)$, and the bottom number indicates the value of $change(u)$. Example by [2].

the tree $T(v_{j+t})$ is shifted by $\sigma - (i-(j+t))\cdot\sigma/\theta$, e.g.

- for $t = i-j-1$ the subtree $T(v_{i-1})$ is shifted by $\sigma - \sigma/\theta$,
- for $t = i-j-2$ the subtree $T(v_{i-2})$ is shifted by $\sigma - 2\cdot\sigma/\theta$,
- and for $t = 1$ the subtree $T(v_{j+1})$ is shifted by $\sigma - (i-j-1)\cdot\sigma/\theta$.

The shift of the subtrees $T(v_{j+t})$, $t = 1, 2, \ldots, i-j-1$, depends on the shift of $T(v_i)$ and if traversed from right to left, is decreased linear by σ/θ.

Since the decrease σ/θ is linear for every pair of greater subtrees, we use a trick and aggregate the amount of shift for the intermediate smaller subtrees by storing σ in an array $shift()$ of size k and by storing the σ/θ in an array $change()$ of size k. The values for shifting the subtrees $T(v_{j+t})$, $t = 1, 2, \ldots, i-j-1$, are then stored at v_i and v_j:

1. the value $shift(v_i)$ is increased by σ
2. the value $change(v_i)$ is decreased by σ/θ
3. the value $change(v_j)$ is increased by σ/θ

Figure 45.9 shows an example on setting the values of $shift()$ and $change()$.

By construction of the arrays $shift()$ and $change()$ we then obtain the shift of $T(v_g)$, $g = 1, 2, \ldots, k$, (including the "original" shift of the subtree $T(v_i)$) as follows. The children v_g, $g = k, k-1, \ldots, 1$, are traversed from right to left. Two real values σ and $change$ are maintained to store the shifts and the decreases of shift per subtree, respectively. These values are initialized with zero. When visiting child v_g, $g \in \{k, k-1, \ldots, 1\}$ the subtree $T(v_g)$ is shifted to the right by σ (i.e., we increase $prelim(v_g)$ and $mod(v_g)$ by σ). Furthermore, $change$ is increased by $change(v_g)$, and σ is increased by $shift(v_g)$ and by $change(v_g)$ and we continue with v_{g-1}.

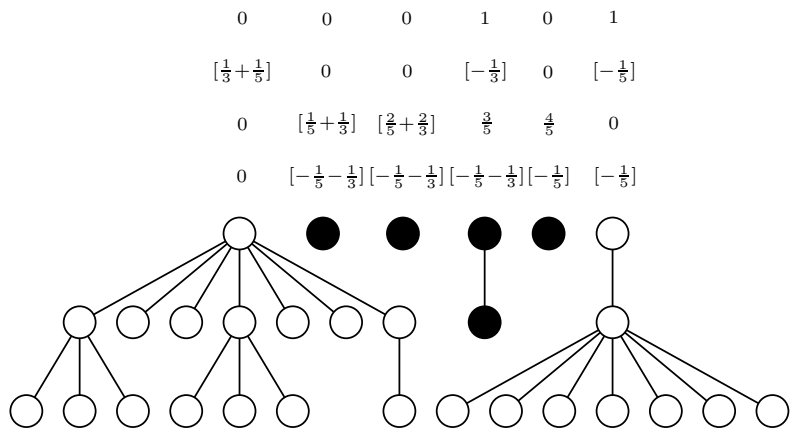


FIGURE 45.10: Executing the shifts: the third and the fourth line of numbers at a node u indicate the values of σ and *change* before shifting u, respectively.

It is easy to see that this algorithm shifts each subtree by the correct amount, see Figure 45.10 for an example.

The function MoveSubtree(w_{left},w_{right},σ) given in algorithm 8 performs an update of the arrays *shift*() and *change*(). We recall that w_{left} and w_{right} are the greatest distinct ancestors of v_{left} and v_{right} and correspond to children v_i and v_j, $1 \leq i < j \leq k$, respectively. MoveSubtree shifts the subtree $T(w_{right})$ by increasing $prelim(w_{right})$ and $mod(w_{right})$ by the amount of σ. The shifts of the intermediate smaller subtrees between $T(w_{left})$ and $T(w_{right})$ are prepared by adjust $change(w_{right})$, $shift(w_{right})$, and $change(w_{left})$.

$\theta = number(w_{right}) - number(w_{left})$;
$change(w_{right}) = change(w_{right}) - \sigma \ / \ \theta$;
$shift(w_{right}) = shift(w_{right}) + \sigma$;
$change(w_{left}) = change(w_{left}) + \sigma \ / \ \theta$;
$prelim(w_{right}) = prelim(w_{right}) + \sigma$;
$mod(w_{right}) = mod(w_{right}) + \sigma$;

Algorithm 8: MoveSubtree(w_{left},w_{right},σ)

The function ExecuteShifts(v) traverses its children from right to left and determines the total shift of the children based on the arrays *shift*() and *change*().

THEOREM 45.2 *[Buchheim et al. [2]] The layout algorithm for n-ary trees meets the aesthetic requirements (A1)–(A5), spaces out smaller subtrees evenly, and can be implemented to run in $O(|V|)$.*

Proof By construction of the algorithm it is obvious that the algorithm meets (A1)–(A5) and spaces out the smaller subtrees evenly. So it is left to show that the running time is linear in the number of nodes.

Every node of T is traversed once during the traversals PrePosition and Adjust.

```
σ= 0;
change = 0;
forall children w of v from right to left do
    prelim(w) = prelim(w) + σ;
    mod(w) = mod(w) + σ;
    change = change + change(w);
    σ = σ + shift(w) + change;
od
```

Algorithm 9: ExecuteShifts

Similar reasoning as in the proof of theorem 45.1 for binary trees shows that the time needed to traverse the left contour of the subtree $T(v_i)$ and the right contour of subforest $\cup_{h=1}^{i-1} T(v_h)$, $i = 2, 3, \ldots, k$ for every node v with children v_i, $i = 1, 2, \ldots, k$ is linear in the number of nodes of T over all such traversals. Moreover, we have that by construction the number of extra operations for spacing out the smaller subtrees is linear in $|V|$ plus the number of nodes traversed in the contours. This proves the theorem.

45.5 Radial Layout

A *radial layout* of a tree is a variation of a level drawing, where the levels are concentric circles $c_1, c_2, \ldots, c_{h(T)}$ around the root placed at the origin c_0. Figure 45.11 shows an example of a radial layout. The radius of a circle c_i, $i = 1, 2, \ldots, h(T)$, is given by an increasing function $r(i)$. This type of layout is frequently used for representing a free tree. A free tree is a tree without a specific root. To layout a free tree, a node is chosen as a fictitious root that minimizes the height of the resulting subtrees. A straightforward manner to obtain a radial layout of a tree is to modify the algorithms for level drawings as presented in Sections 45.3 and 45.4.

To guarantee that the resulting drawing is planar, the subtree $T(v)$ of each node v is drawn within an *annulus wedge* $w(v)$. In order to permit edges with endpoints in $w(v)$ to extend outside $w(v)$ and thus to conflict with other edges, the nodes in the subtree $T(v)$ are placed within a convex subset $s(v)$ of $w(v)$. Given the level $l(v)$, the node v is placed on $c_{l(v)}$. Suppose that the tangent to $c_{l(v)}$ through v meets the points a and b on $c_{l(v)+1}$. Then $s(v)$ is chosen to be the unbounded region that is given by the line segment ab and the rays from the root of T and the node a and b. The children v_j, $j = 1, 2, \ldots, k$ of v are then arranged on $c_{l(v)+1}$ within $s(v)$ according to the number of leaves in $T(v_j)$.

If the distance between consecutive circles c_i, c_{i+1} $i = 0, 1, \ldots, h(T) - 1$, is equal, it can be easily shown that the area occupied by the layout is in

$$O(h(T)^2 \max_{v \in T}\{k \mid k \text{ number of children of } v\}) \quad .$$

Different algorithms for the radial layout of trees have been presented by [1, 8, 9] depending on the choice of the root, the radii of the circles and the angle of the annulus wedge. Symmetry oriented algorithms have also been developed, see e.g. [16].

45.6 HV-Layout

An *hv*-layout of a binary tree is an upward (but not strictly upward) straight line orthogonal drawing with edges drawn as rightward horizontal or downward vertical segments. The "hv"

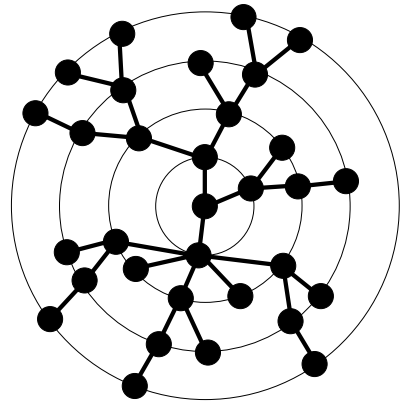


FIGURE 45.11: A radial layout of a binary tree.

stands for horizontal-vertical. For any vertex v in a binary tree T we either have:

1. A child of v is either
   - aligned horizontally with v and to the right of v or
   - vertically aligned below v.
2. The smallest rectangles that cover the area of the subtrees of the children of v in the layout do not intersect.

Figure 45.12 shows an example of a *hv*-layout

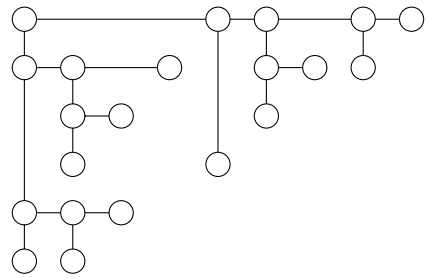


FIGURE 45.12: A hv-layout of a binary tree.

A *hv*-layout is generated by applying a dived and conquer approach. The divide step constructs the *hv*-layout of the left and the right subtrees, while the conquer step either performs a *horizontal* or a *vertical combination*. Figure 45.13 shows the two types of combination.

If the left subtree a node v is placed to the left in a horizontal combination and below in a vertical combination, the layout preserves the ordering of the children of v. The height and width of such a drawing is at most $|V| - 1$. A straightforward way to reduce the size of a *hv*-drawing to a height of at most $\log |V|$ is to use only horizontal combinations and to place the larger subtree (in terms of number of nodes) to the right of the smaller subtree. This *right heavy hv layout* is not order preserving and can be produced in $O(|V|)$ requiring

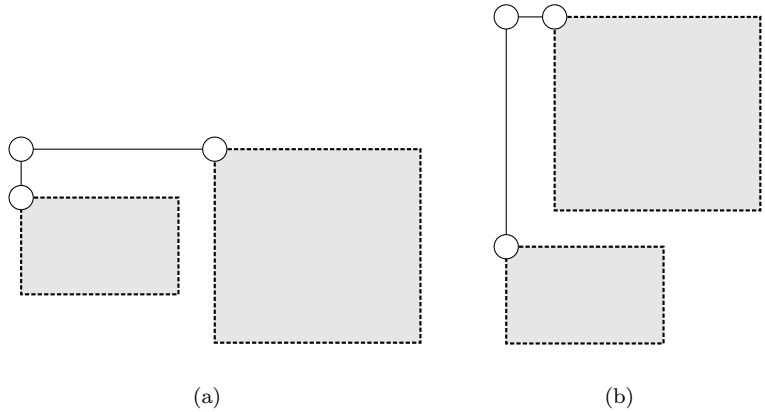


FIGURE 45.13: A horizontal combination (a) and a vertical combination (b).

only an area of $O(|V| \log |V|)$. The right heavy hv approach can be easily extended to draw n-ary trees as sketched in Figure 45.14.

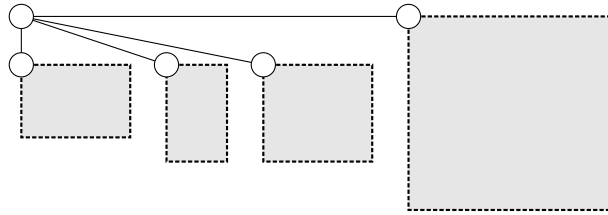


FIGURE 45.14: A hv-layout of a n-ary tree.

As already presented in the introduction, this type of layout has been extensively studied e.g. in [4–7, 10, 12–15, 20–22] to obtain results on minimal area requirements of tree layouts.

Recently Garg and Ruse [11] showed by flipping the subtrees rooted at a node v horizontally or vertically, it is possible to obtain tree layouts for binary trees with an $O(|V|)$ area and with a pre specified aspect ratio in the range of $[1, |V|^{\alpha}]$, with $\alpha \in [0, 1)$. These layouts are non upward straight line orthogonal layouts.

Acknowledgment

I gratefully acknowledge the contributions of Christoph Buchheim, who provided some of the figures and the implementation from our paper [2] and Merijam Percan for her careful proofreading.

References

[1] M. Bernard. On the automated drawing of graphs. In *3rd Caribbean Conference on Combinatorics and Computing*, pages 43–55, 1981.

[2] C. Buchheim, M. Jünger, and S. Leipert. Improving walker's algorithm to run in linear time. In M. Goodrich, editor, *Graph Drawing (Proc. GD 2002)*, volume 2528 of *LNCS*, pages 344–353. Springer-Verlag, 2002.

[3] Cormen, T., Leiserson, C., and Rivest, R. (1990). *Introduction to Algorithms*. MIT-Press.

[4] P. Crescenzi, G. Di Battista, and A. Piperno. A note on optimal area algorithms for upward drawings of binary trees. *Computational Geometry*, 2:187–200, 1992.

[5] P. Crescenzi and A. Piperno. Optimal-area upward drawings of AVL trees. In R. Tamassia and I.G. Tollis, editors, *Graph Drawing (Proc. GD 1994)*, volume 894 of *LNCS*, pages 307–317. Springer-Verlag, 1995.

[6] P. Crescenzi and P. Penna. Strictly-upward drawings of ordered search trees. *Theoretical Computer Science*, 203(1):51–67, 1998.

[7] P. Crescenzi, P. Penna, and A. Piperno. Linear-area upward drawings of AVL trees. *Comput. Geom. Theory Appl.*, 9:25–42, 1998. (special issue on Graph Drawing, edited by G. Di Battista and R. Tamassia).

[8] P. Eades. Drawing free trees. *Bulletin of the Inst. for Combinatorics and its Applications*, 5:10–36, 1992.

[9] C. Esposito. Graph graphics: Theory and practice. *Comput. Math. Appl.*, 15(4):247–253, 1988.

[10] A. Garg, M. T. Goodrich, and R. Tamassia. Planar upward tree drawings with optimal area. *Internat. J. Comput. Geom. Appl.*, 6:333–356, 1996.

[11] A. Garg and A. Rusu. Straight-line drawings of binary trees with linear area and arbitrary aspect ratio. In M. Goodrich, editor, *Graph Drawing (Proc. GD 2002)*, volume 2528 of *LNCS*, pages 320–331. Springer-Verlag, 2002.

[12] S. K. Kim. Simple algorithms for orthogonal upward drawing of binary and ternary trees. In *Proc. 7th Canad. Conf. Comput. Geometry*, pages 115–120, 1995.

[13] S. K. Kim. H-V drawings of binary trees. In *Software Visualization*, volume 7 of *Software Engineering and Knowledge Engineering*, pages 101–116, 1996.

[14] X. Lin P. Eades, T. Lin. Minimum size h-v drawings. In *Advanced Visual Interfaces*, volume 36 of *World Scientific Series in Computer Science*, pages 386–394, 1992.

[15] X. Lin P. Eades, T. Lin. Two tree drawing conventions. *International Journal of Computational Geometry and Applications*, 3(2):133–153, 1993.

[16] J. Manning and M. Atallah. Fast detection and display of symmetry in trees. *Congr. Numer.*, 64:159–169, 1988.

[17] E. Reingold and J. Tilford. Tidier drawing of trees. *IEEE Trans. Softw. Eng.*, SE-7(2):223–228, 1981.

[18] K.R. Supovit and E. Reingold. The complexity of drawing trees nicely. *Act Informatica*, 18:377–392, 1983.

[19] B. Schieber and U. Vishkin. On finding lowest common ancestors: Simplification and parallelization. In *Proceedings of the Third Aegean Workshop on Computing*, volume 319 of *Lecture Notes in Computer Science*, pages 111–123. Springer-Verlag, 1988.

[20] Y. Shiloach. *Arrangements of Planar Graphs on the Planar Lattice*. PhD thesis, Weizmann Institute of Science, 1976.

[21] C.-S. Shin, S. K. Kim, and K.-Y. Chwa. Area-efficient algorithms for straight-line tree drawings. *Comput. Geom. Theory Appl.*, 15:175–202, 2000.

[22] L. Trevisan. A note on minimum-area upward drawing of complete and Fibonacci trees. *Inform. Process. Lett.*, 57(5):231–236, 1996.

[23] J. Q. Walker II. A node-positioning algorithm for general trees. *Softw. – Pract. Exp.*, 20(7):685–705, 1990.

[24] C. Wetherell and A. Shannon. Tidy drawings of trees. *IEEE Transactions on Software Engineering*, 5(5):514–520, 1979.

46

Drawing Graphs

Peter Eades
University of Sydney and NICTA

Seok-Hee Hong
University of Sydney and NICTA

46.1 Introduction

Graph Drawing (see Chapter 4 for an introduction to graphs) is the art of making pictures of relationships. For example, consider the social network defined in Table 46.1. This table expresses the "has-written-a-joint-paper-with" relation for a small academic community.

Name	*has-a-joint-paper-with*
Jane	Harry, Paula, and Sally
Sally	Jane, Paula and Dennis
Dennis	Sally and Monty
Harry	Jane and Paula
Monty	Dennis and Kerry
Ying	Paula
Paula	Jane, Harry, Ying, Tan, Cedric, Chris, Kerry and Sally
Kerry	Paula and Monty
Tan	Paul and Cedric
Cedric	Tan, Paula, and Chris
Chris	Paula and Cedric

TABLE 46.1 Table representing the *has-a-joint-paper-with* relation.

It is easier to understand this social network if we draw it. A drawing is in Figure 46.1(a); a better drawing is in Figure 46.1(b). The challenge for Graph Drawing is to automatically create good drawings of graphs such as in Figure 46.1(b), starting with tables such as in Table 46.1.

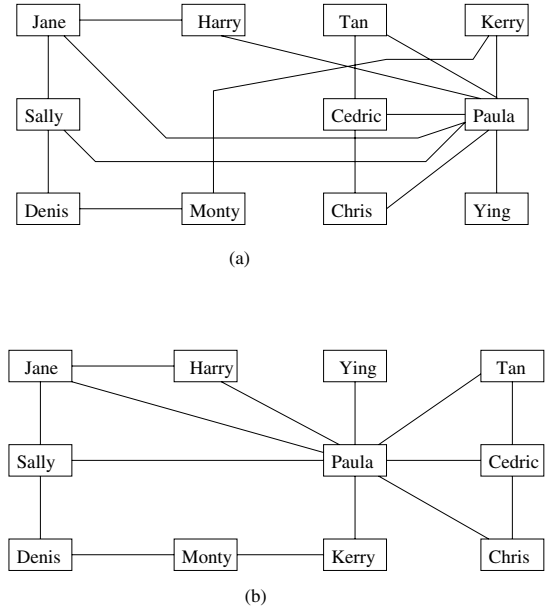


FIGURE 46.1: Two drawings of a social network.

The *criteria* of a good drawing of a graph have been the subject of a great deal of attention (see, for example, [3, 29, 30]). The four criteria below are among the most commonly used criteria.

- *Edge crossings* should be avoided. Human tests clearly indicate that edge crossings inhibit the readability of the graph [30]. Figure 46.1(a) has 6 edge crossings, and Figure 46.1(b) has none; this is the main reason that Figure 46.1(b) is more readable than Figure 46.1(a). The algorithms presented in this chapter deal with *planar drawings*, that is, drawings with no edge crossings.
- The *resolution* of a graph drawing should be as large as possible. There are several ways to define the intuitive concept of resolution; the simplest is the *vertex resolution*, that is, the ratio of the minimum distance between a pair of vertices to the maximum distance between a pair of vertices. High resolution helps readability because it allows a larger font size to be used in the textual labels on vertices. In practice it can be easier to measure the *area* of the drawing, given that the minimum distance between a pair of vertices is one. If the vertices in Figure 46.1 are on an integer grid, then the drawing is 4 units by 2 units, for both (a) and (b). Thus the vertex resolution is $1/2\sqrt{5} \simeq 0.2236$, and the area is 8. One can refine the concept of resolution to define *edge resolution* and *angular resolution*.
- The *symmetry* of the drawing should be maximized. Symmetry conveys the structure of a graph, and Graph Theory textbooks commonly use drawings that are as symmetric as possible. Intuitively, Figure 46.1(b) displays more symmetry than Figure 46.1(a). A refined concept of symmetry display is presented in Section 46.4.
- *Edge bends* should be minimized. There is some empirical evidence to suggest that bends inhibit readability. In Figure 46.1(a), three edges contain bends and

eleven are straight lines; in total, there are seven edge bends. Figure 46.1(b) is much better: all edges are straight lines.

Each of these criteria can be measured. Thus Graph Drawing problems are commonly stated as optimization problems: given a graph G, we need to find a drawing of G for which one or more of these measures are optimal. These optimization problems are usually NP-complete.

In most cases it is not possible to optimize one of the criteria above without compromise on another. For example, the graph in Figure 46.2(a) has 8 symmetries but one edge crossing. It is possible to draw this graph without edge crossings, but in this case we can only get 6 symmetries, as in Figure 46.2(b).

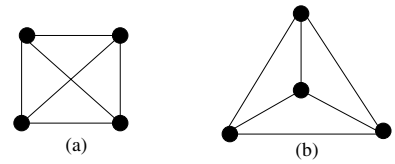


FIGURE 46.2: Two drawings of a graph.

The optimization problem may be constrained. For example, consider a graph representing prerequisite dependencies between the units in a course, as in Figure 46.3; in this case, the y coordinate of a unit depends on the semester in which the unit ought to be taken.

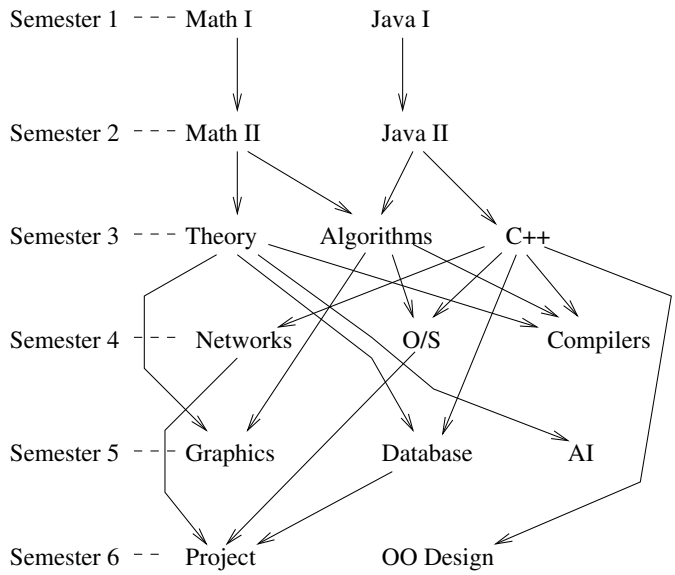


FIGURE 46.3: Prerequisite diagram.

Graph Drawing has a long history in Mathematics, perhaps beginning with the theorem of Wagner that every planar graph can be drawn with straight line edges [38]. The importance of graph drawing algorithms was noted early in the history of Computer Science (note the paper of Knuth on drawing flowcharts, in 1960 [23], and the software developed by Read in the 1970s [31]). In the 1980s, the availability of graphics workstations spawned a broad interest in visualization in general and in graph visualization in particular. The field became mature in the mid 1990s with a conference series (for example, see [15]), some books [8, 22, 33], and widely available software (see [20]).

In this Chapter we briefly describe just a few of the many graph available drawing algorithms. Section 46.3 presents some algorithms to draw planar graphs with straight line edges and convex faces. Section 46.4 describes two algorithms that can be used to construct planar symmetric drawings. Section 46.5 presents a method for constructing "visibility" representations of planar graphs, and shows how this method can be used to construct drawings with few bends and reasonable resolution.

46.2 Preliminaries

Basic concepts of mathematical Graph Theory are described in [4]. In this section we define mathematical notions that are used in many Graph Drawing algorithms.

A *drawing* D of a graph $G = (V, E)$ assigns a location $D(u) \in R^2$ for every vertex $u \in V$ and a curve $D(e)$ in R^2 for every edge $e \in E$ such that if $e = (u, v)$ then the curve $D(e)$ has endpoints $D(u)$ and $D(v)$. If $D(u)$ has integer coordinates for each vertex u, then D is a *grid* drawing. If the curve $D(e)$ is a straight line segment for each edge e, then D is a *straight line* drawing.

For convenience we often identify the vertex u with its location $D(u)$, and the edge e with its corresponding curve $D(e)$; for example, when we say "the edge e crosses the edge f", strictly speaking we should say "the curve $D(e)$ crosses the curve $D(f)$".

A graph drawing is *planar* if adjacent edges intersect only at their common endpoint, and no two nonadjacent edges intersect. A graph is *planar* if it has a planar drawing. Planarity is a central concern of Graph Theory, Graph Algorithms, and especially Graph Drawing; see [4].

A planar drawing of a graph divides the plane into regions called *faces*. One face is unbounded; this is the *outside face*. Two faces are *adjacent* if they share a common edge. The graph G together its faces and the adjacency relationship between the faces is a *plane graph*. The graph whose vertices are the faces of a plane graph G and whose edges are the adjacency relationships between faces is the *planar dual*, or just *dual*, of G. A graph and its planar dual are in Figure 46.4.

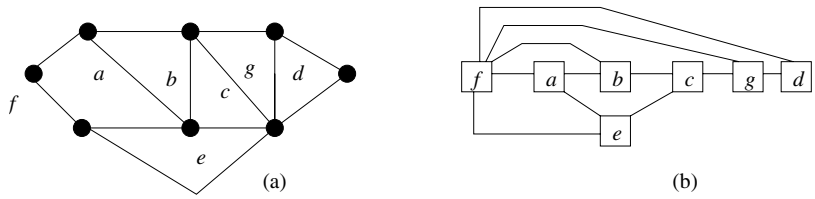


FIGURE 46.4: (a) A graph G and its faces. (b) The dual of G.

The *neighborhood* $N(u)$ of a vertex u is the list of vertices that are adjacent to u. If G is a plane graph, then $N(u)$ is given in the clockwise circular ordering about u.

46.3 Convex Drawing

A straight-line drawing of a planar graph is *convex* if all every face is drawn as convex polygon. This section reviews three different algorithms for constructing such a drawing for planar graphs.

46.3.1 Barycenter Algorithm

Tutte showed that a convex drawing exist for triconnected planar graphs and gave an algorithm for constructing such representations [36, 37].

The algorithm divides the vertex set V into two subsets; a set of *fixed* vertices and a set of *free* vertices. The fixed vertices are placed at the vertices of a strictly convex polygon. The positions of free vertices are decided by solving a system of $O(n)$ linear equations, where n is the number of vertices of the graph [37]. In fact, solving the equations is equivalent to placing each free vertex at the *barycenter* of its neighbors. That is, each position $p(v) = (x(v), y(v))$ of a vertex v is:

$$x(v) = \frac{1}{deg(v)} \sum_{(v,w)\in E} x(w), \quad y(v) = \frac{1}{deg(v)} \sum_{(v,w)\in E} y(w). \tag{46.1}$$

An example of a drawing computed by the barycenter algorithm is illustrated in Figure 46.5. Here the black vertices represent fixed vertices.

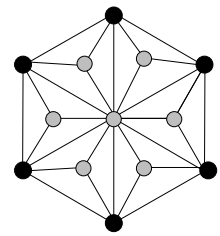


FIGURE 46.5: Example output from the algorithm of Tutte.

The main theorem can be described as follows.

THEOREM 46.1 *(Tutte [36, 37]) Suppose that f is a face in a planar embedding of a triconnected planar graph G, and P is a strictly convex planar drawing of f. Then the barycenter algorithm with the vertices of f fixed and positioned according to P gives a convex planar drawing of G.*

The matrix resulting from the equations can be solved in $O(n^{1.5})$ time at best, using a sophisticated sparse matrix elimination method [25]. However, in practice a Newton-Raphson iteration of the equation above converges rapidly.

46.3.2 Divide and Conquer Algorithm

Chiba, Yamanouchi and Nishizeki [6] present a linear time algorithm for constructing convex drawings of planar graphs, using divide and conquer. The algorithm constructs a convex drawing of a planar graph, if it is possible, with a given outside face. The drawing algorithm is based on a classical result by Thomassen [35].

The input to their algorithm is a biconnected plane graph with given outside face and a convex polygon. The output of their algorithm is a convex drawing of the biconnected plane graph with outside face drawn as the input convex polygon, if this is possible. The conditions under which such a drawing is possible come from the following theorem [35].

THEOREM 46.2 *(Thomassen [35]) Let G be a biconnected plane graph with outside facial cycle S, and let S^* be a drawing of S as a convex polygon. Let $P_1, P_2, \ldots, P_k$ be the paths in S, each corresponding to a side of S^*. (Thus S^* is a k-gon. It should be noted that not every vertex of the cycle S is necessarily an apex of the polygon S^*.) Then S^* can be extended to a convex drawing of G if and only if the following three conditions hold.*

1. *For each vertex v of $G - V(S)$ having degree at least three in G, there are three paths disjoint except at v, each joining v and a vertex of S;*
2. *The graph $G - V(S)$ has no connected component C such that all the vertices on S adjacent to vertices in C lie on a single path P_i; and no two vertices in each P_i are joined by an edge not in S; and*
3. *Any cycle of G which has no edge in common with S has at least three vertices of degree at least 3 in G.*

The basic idea of the algorithm of Chiba *et al.* [6] is to reduce the convex drawing of G to those of several subgraphs of G as follows:

Algorithm `ConvexDraw`(G, S, S^*)

1. Delete from G an arbitrary apex v of S^* together with edges incident to v.
2. Divide the resulting graph $G' = G - v$ into the biconnected components $B_1, B_2, \ldots, B_p$.
3. Determine a convex polygon S_i^* for the outside facial cycle S_i of each B_i so that B_i with S_i^* satisfies the conditions in Theorem 46.2.
4. Recursively apply the algorithm to each B_i with S_i^* to determine the positions of vertices not in S_i.

The main idea of the algorithm is illustrated in Figure 46.6; for more details see [6].

It is easy to show that the algorithm runs in linear time, and its correctness can be established using Theorem 46.2.

THEOREM 46.3 *(Chiba, Yamanouchi and Nishizeki [6]) Algorithm* `ConvexDraw` *constructs a convex planar drawing of a biconnected plane graph with given outside face in linear time, if such a drawing is possible.*

46.3.3 Algorithm Using Canonical Ordering

Both the Tutte algorithm and the algorithm of Chiba *et al.* give poor resolution. Kant [21] gives a linear time algorithm for constructing a planar convex grid drawing of a triconnected planar graph on a $(2n-4) \times (n-2)$ grid, where n is the number of vertices. This guarantees that the vertex resolution of the drawing is $\Omega(1/n)$.

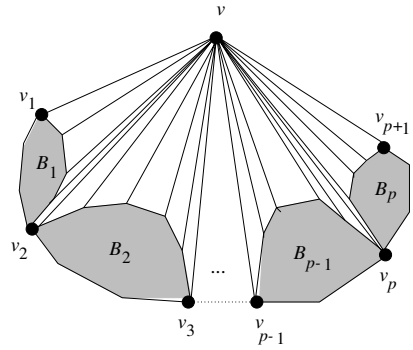


FIGURE 46.6: The algorithm of Chiba et al.

Kant's algorithm is based on a method of de Fraysseix, Pach and Pollack [10] for drawing planar *triangulated* graphs on a grid of size $(2n-4) \times (n-2)$.

The algorithm of de Fraysseix *et al.* [10] computes a special ordering of vertices called the *canonical ordering* based on fixed embedding. Then the vertices are added, one by one, in order to construct a straight-line drawing combined with *shifting* method. Note that the outside face of the drawing is always triangle, as shown in Figure 46.7.

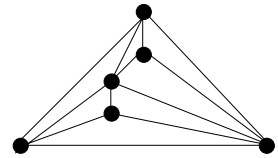


FIGURE 46.7: Planar straight-line grid drawing given by de Fraysseix, Pach and Pollack method.

Kant [21] uses a generalization of the canonical ordering called *leftmost canonical ordering*. The main difference from the algorithm of de Fraysseix *et al.* [10] is that it gives a convex drawing of *triconnected* planar graph. The following Theorem summarizes his main result.

THEOREM 46.4 *(Kant [21]) There is a linear time algorithm to construct a planar straight-line convex drawing of a triconnected planar graph with n vertices on a $(2n-4) \times (n-2)$ grid.*

Details of Kant's algorithm are in [21]. Chrobak and Kant improved the algorithm so that the size of the grid is $(n-2) \times (n-2)$ [7].

46.4 Symmetric Drawing

A graph drawing can have two kinds of symmetry: *axial* (or reflectional) symmetry and *rotational* symmetry. For example, see Figure 46.8. The drawing in Figure 46.8(a) displays four rotational symmetries and Figure 46.8(b) displays one axial symmetry. Figure 46.8(c)

displays eight symmetries, four rotational symmetries as well as four axial symmetries.

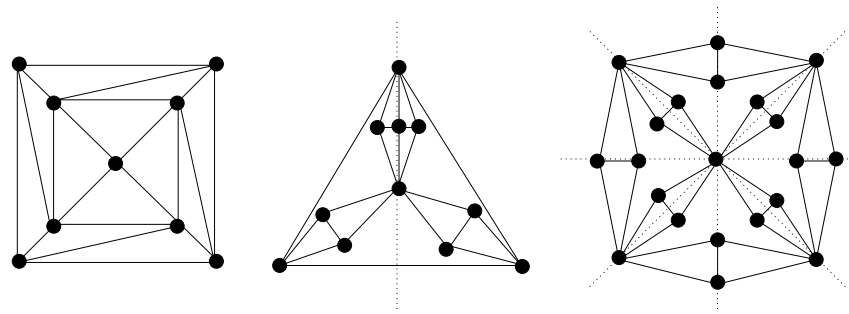


FIGURE 46.8: Three symmetric drawings of graphs.

Manning [28] showed that, in general, the problem of determining whether a given graph can be drawn symmetrically is NP-complete. De Fraysseix [9] presents heuristics for symmetric drawings of general graphs. Exact algorithms for finding symmetries in general graphs were presented by Buchheim and Junger [5], and Abelson, Hong and Taylor [1].

There are linear time algorithms available for restricted classes of graphs. Manning and Atallah present a liner time algorithm for detecting symmetries in trees [26], outerplanar graphs [27], and plane graphs [28]. Hong, Eades and Lee present a linear time algorithm for drawing series-parallel digraphs symmetrically [17].

In this section we briefly describe a linear time algorithm to draw triconnected planar graphs with maximum symmetry. Note that the algorithm of Tutte described in Section 46.3.1 can be used to construct symmetric drawings of triconnected planar graphs, but it does not work in linear time. Here we give the linear time algorithm of Hong, McKay and Eades [18]. It has two steps:

1. *Symmetry finding*: this step finds symmetries, or so-called *geometric automorphisms* [12], in graphs.
2. *Symmetric drawing*: this step constructs a drawing that displays these automorphisms.

More specifically, the symmetry finding step finds a plane embedding that displays the maximum number of symmetries. It is based on a classical theorem from group theory called the *orbit-stabilizer theorem* [2]. It also uses a linear time algorithm of an algorithm of Fontet [14] to compute orbits and generators for groups of geometric automorphisms. For details, see [18]. Here we present only the second step, that is, the drawing algorithm. This constructs a straight-line planar drawing of a given triconnected plane graph such that given symmetries are displayed.

The drawing algorithm has three routines, each corresponding to a type of group of symmetries in two dimensions:

1. the *cyclic* case: displaying k rotational symmetries;
2. one *axial* case: displaying one axial symmetry;

3. the *dihedral* case: displaying $2k$ symmetries (k rotational symmetries and k axial symmetries).

The dihedral case is important for displaying *maximum* number of symmetries, however it is the most difficult to achieve. Here, we concentrate on the first two cases; the cyclic case and the one axial case. In Section 46.4.1, we describe an algorithm for constructing a drawing displaying rotational symmetry and in Section 46.4.2 we describe an algorithm for constructing a drawing displaying axial symmetry. We briefly comments on the dihedral case in Section 46.4.3.

The input of the drawing algorithm is a triconnected plane graph; note that the outside face is given. Further, the algorithm needs to compute a triangulation as a preprocessing step. More specifically, given a plane embedding of triconnected planar graphs with a given outside face, we triangulate each internal face by inserting a new vertex in the face and joining it to each vertex of the face. This process is called *star-triangulation* and clearly it takes only linear time.

A major characteristic of symmetric drawings is the repetition of congruent drawings of isomorphic subgraphs. The algorithms use this property, that is, they compute a drawing for a subgraph and then use copies of it to draw the whole graph. More specifically, each symmetric drawing algorithm consists of three steps. First, it finds a subgraph. Then it draws the subgraph using the algorithm of Chiba *et al.* [6] as a subroutine. The last step is to replicate the drawing; that is, merge copies of the subgraphs to construct a drawing of the whole graph.

The application of Theorem 46.2 is to subgraphs of a triconnected plane graph defined by a cycle and its interior. In that case Thomassen's conditions can be simplified. Suppose that P is a path in a graph G; a *chord* for P in G is an edge (u, v) of G not in P, but whose endpoints are in P.

COROLLARY 46.1 Let G be a triconnected plane graph and let S be a cycle of G. Let W be the graph consisting of S and its interior. Let S^* be a drawing of S as a convex k-gon (where S might have more than k vertices). Let $P_1, P_2, \ldots, P_k$ be the paths in S corresponding to the sides of S^*. Then S^* is extendable to a straight-line planar drawing of W if and only if no path P_i has a chord in W.

Proof We can assume that the interior faces of W are triangles, since otherwise we can star triangulate them.

We need to show that Thomassen's three conditions are met. Condition 1 is a standard implication of the triconnectivity of G, and Condition 3 follows just from the observation that the internal vertices have degree at least 3.

To prove the first part of Condition 2, suppose on the contrary that C is a connected component of $W - S$ which is adjacent to P_i but not to any other of $P_1, \ldots, P_k$. Let u and v be the first and last vertices on P_i which are adjacent to C. If $u = v$, or u and v are adjacent on P_i, then $\{u, v\}$ is a cut (separation pair) in G, which is impossible as G is triconnected. Otherwise, there is an interior face containing u and v and so u and v are adjacent contrary to our hypothesis (since the internal faces are triangles).

The second part of Condition 2 is just the condition that we are imposing.

46.4.1 Displaying Rotational Symmetry

Firstly, we consider the cyclic case, that is, displaying k rotational symmetries. Note that after the star-triangulation, we may assume that there is either an edge or a vertex fixed by the symmetry. The fixed edge case can only occur if $k = 2$ and this case can be transformed into the fixed vertex case by inserting a dummy vertex into the fixed edge with two dummy edges. Thus we assume that there is a fixed vertex c.

The symmetric drawing algorithm for the cyclic case consists of three steps.

Algorithm `Cyclic`

1. `Find_Wedge_Cyclic`.
2. `Draw_Wedge_Cyclic`.
3. `Merge_Wedges_Cyclic`.

The first step is to find a subgraph W, called a *wedge*, as follows.

Algorithm `Find_Wedge_Cyclic`

1. Find the fixed vertex c.
2. Find a shortest path P_1, from c to a vertex v_1 on the outside face. This can be done in linear time by breadth first search.
3. Find the path P_2 which is a mapping of P_1 under the rotation.
4. Find the induced subgraph of G enclosed by the cycle formed from P_1, P_2 and a path P_0 along the outside face from v_1 to v_2 (including the cycle). This is the wedge W.

A wedge W is illustrated in Figure 46.9(a). It is clear that Algorithm `Find_Wedge_Cyclic` runs in linear time.

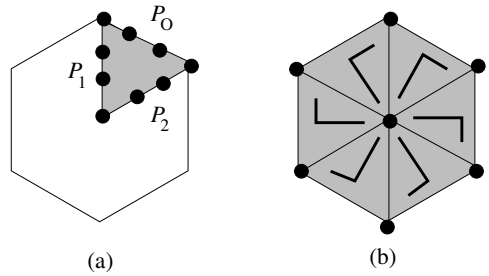


FIGURE 46.9: Example of (a) a wedge and (b) merging.

The second step, `Draw_Wedge_Cyclic`, constructs a drawing D of the wedge W using Algorithm `ConvexDraw`, such that P_1 and P_2 are drawn as straight-lines. This is possible by the next lemma.

LEMMA 46.1 Suppose that W is the wedge computed by Algorithm `Find_Wedge_Cyclic`, S is the outside face of W, and S^* is the drawing of the outside face S of W using three straight-lines as in Figure 46.9(a). Then S^* is extendable to a planar straight-line drawing of W.

Proof One of the three sides of S^* corresponds to part of the outside face of G, which cannot have a chord as G is triconnected. The other two sides cannot have chords either, as they are shortest paths in G. Thus W and S^* satisfy the conditions of Corollary 46.1.

The last step, `Merge_Wedges_Cyclic`, constructs a drawing of the whole graph G by replicating the drawing D of W, k times. Note that this merge step relies on the fact that P_1 and P_2 are drawn as straight-lines. This is illustrated in Figure 46.9 (b).

Clearly, each of these three steps takes linear time. Thus the main result of this section can be stated as follows.

THEOREM 46.5 *Algorithm* `Cyclic` *constructs a straight-line drawing of a triconnected plane graph which shows k rotational symmetry in linear time.*

46.4.2 Displaying Axial Symmetry

A critical element of the algorithm for displaying one axial symmetry is the subgraph that is fixed by the axial symmetry. Consider a drawing of a star-triangulated planar graph with one axial symmetry. There are fixed vertices, edges and/or fixed faces on the axis. The subgraph formed by these vertices, edges and faces, is called a *fixed string of diamonds*.

The first step of the algorithm is to identify the fixed string of diamonds. Then the second step is to use Algorithm `Symmetric_ConvexDraw`, a modified version of Algorithm `ConvexDraw`. Thus Algorithm `One_Axial` can be described as follows.

Algorithm `One_Axial`

1. Find a fixed string of diamonds. Suppose that $\omega_1, \omega_2, \ldots, \omega_k$ are the fixed edges and vertices in the fixed string of diamonds, in order from the outside face (ω_1 is on the outside face). For each ℓ, ω_ℓ may be a vertex or an edge. This is illustrated in Figure 46.10.
2. Choose an axially symmetric convex polygon S^* for the outside face of G.
3. `Symmetric_ConvexDraw`$(1, S^*, G)$.

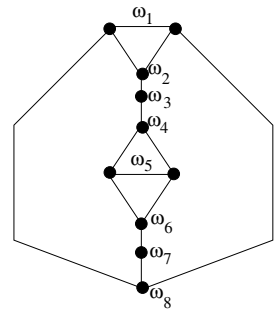


FIGURE 46.10: A string of diamonds.

The main subroutine in Algorithm `One_Axial` is Algorithm `Symmetric_ConvexDraw`. This algorithm, described below, modifies Algorithm `ConvexDraw` so that the following three

conditions are satisfied.

1. Choose v in Step 1 of Algorithm `ConvexDraw` to be ω_1. In Algorithm `ConvexDraw`, v must be a vertex; here we extend Algorithm `ConvexDraw` to deal with an edge or a vertex. The two cases are illustrated in Figure 46.11.
2. Let $D(B_i)$ be the drawing of B_i and α be the axial symmetry. Then, $D(B_i)$ should be a reflection of $D(B_j)$, where $B_j = \alpha(B_i)$, $i = 1, 2, \ldots, m$ and $m = \lfloor p/2 \rfloor$.
3. If p is odd, then $D(B_{m+1})$ should display one axial symmetry.

It is easy to see that satisfying these three conditions ensures the display a single axial symmetry.

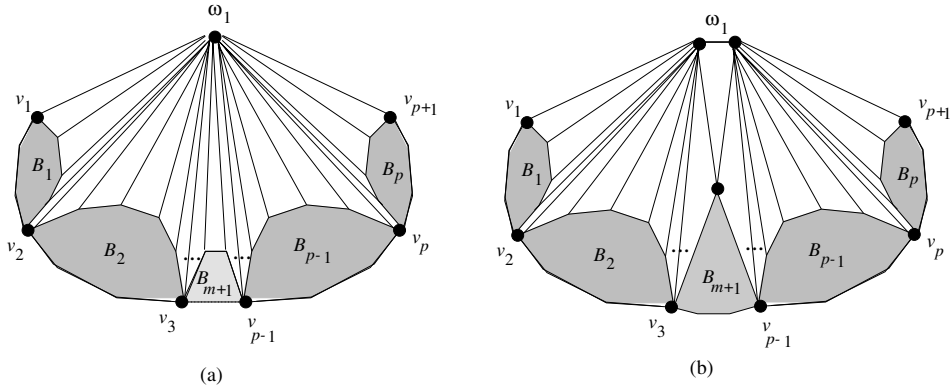


FIGURE 46.11: Symmetric version of ConvexDraw.

Let $B_j = \alpha(B_i)$. The second condition can be achieved as follows. First define S_j^* to be the reflection of S_i^*, $i = 1, 2, \ldots, m$. Then apply Algorithm `ConvexDraw` for $B_i, i = 1, 2, \ldots, m$ and construct $D(B_j)$ using a reflection of $D(B_i)$. If p is odd then we recursively apply Algorithm `Symmetric_ConvexDraw` to B_{m+1}. Thus Algorithm `Symmetric_ConvexDraw` can be described as follows.

Algorithm `Symmetric_ConvexDraw`(ℓ, S^*, G)

1. Delete ω_ℓ from G together with edges incident to ω_ℓ. Divide the resulting graph $G' = G - \omega_\ell$ into the blocks $B_1, B_2, \ldots, B_p$, ordered anticlockwise around the outside face. Let $m = \lfloor p/2 \rfloor$.
2. For each $i = 1$ to m, determine a convex polygon S_i^* of the outside facial cycle S_i of each B_i so that B_i with S_i^* satisfy the conditions in Theorem 46.2 and chose S_{p-i+1}^* to be a reflection of S_i^*.
3. For each $i = 1$ to m,
   (a) Construct a drawing $D(B_i)$ of B_i using Algorithm `ConvexDraw`.
   (b) Construct $D(B_{p-i+1})$ as a reflection of $D(B_i)$.
4. If p is odd, then construct a drawing $D(B_{m+1})$ using `Symmetric_ConvexDraw`$(\ell + 1, S_{m+1}^*, B_{m+1})$.
5. Merge the $D(B_i)$ to form a drawing of G.

Using the same argument as for the linearity of Algorithm `ConvexDraw` [6], one can show that Algorithm `One_Axial` takes linear time.

THEOREM 46.6 *Algorithm* `One_Axial` *constructs a straight-line drawing of a triconnected plane graph which shows one axial symmetry in linear time.*

46.4.3 Displaying Dihedral Symmetry

In this section, we briefly review an algorithm for constructing a drawing displaying k axial symmetries and k rotational symmetries.

As with the cyclic case, we assume that there is a vertex fixed by all the symmetries. The algorithm adopts the same general strategy as for the cyclic case: divide the graph into "wedges", draw each wedge, then merge the wedges together to make a symmetric drawing. However, the dihedral case is more difficult than the pure rotational case, because an axial symmetry in the dihedral case can have fixed edges and/or fixed faces. This requires a much more careful applications of the Algorithm `ConvexDraw` and Algorithm `Symmetric_ConvexDraw`; for details see [18].

Nevertheless, using three steps above, a straight-line drawing of a triconnected plane graph which shows dihedral symmetry can be constructed in linear time.

In conclusion, the symmetry finding algorithm together with the three drawing algorithms ensures the following theorem which summarizes their main result.

THEOREM 46.7 *(Hong, McKay and Eades [18]) There is a linear time algorithm that constructs maximally symmetric planar drawings of triconnected planar graphs, with straight-line edges.*

46.5 Visibility Drawing

In general, a *visibility representation* of a graph has a geometric shape for each vertex, and a "line of sight" for each edge. Different visibility representations involve different kinds of geometric shapes and different restrictions on the "line of sight". A three dimensional visibility representation of the complete graph on five vertices is in Figure 46.12. Here the geometric objects are rectangular prisms, and each "line of sight" is parallel to a coordinate axis.

Visibility representations have been extensively investigated in both two and three dimensions.

For two dimensional graph drawing, the simplest and most common visibility representation uses horizontal line segments (called "bars") for vertices and vertical line segments for "lines of sight". More precisely, a *bar visibility representation* of a graph $G = (V, E)$ consists of a horizontal line segment ω_u for each vertex $u \in V$, and a vertical line segment λ_e for each edge $e \in E$, such that the following properties hold:

- P1: If u and u' are distinct vertices of G, then ω_u has empty intersection with $\omega_{u'}$.
- P2: If e and e' are distinct edges of G, then λ_e has empty intersection with $\lambda_{e'}$.
- P3: If e is not incident with u then λ_e has empty intersection with ω_u.
- P4: If e is incident with u then the intersection of λ_e and ω_u consists of an endpoint of λ_e on ω_u.

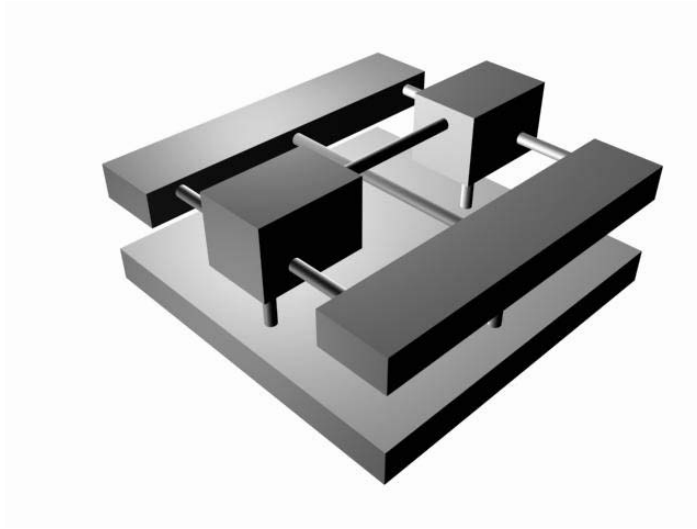


FIGURE 46.12: A visibility representation in three dimensions (courtesy of Nathalie Henry).

Figure 46.13 shows a bar visibility representation of the 3-cube.

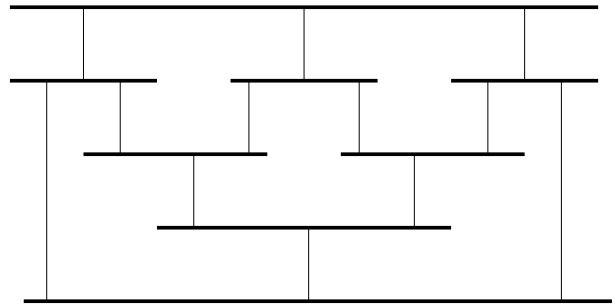


FIGURE 46.13: A bar visibility representation of the 3-cube.

It is clear that if G admits a bar visibility representation, then G is planar. Tamassia and Tollis [34] and Rosenstiehl and Tarjan [32] proved the converse: every planar graph has a bar visibility representation. Further, they showed that the resulting drawing has quadratic area, that is, the resolution is good. The algorithms from [34] and [32] construct such a representation in linear time.

In the Section 46.5.1 below we describe some concepts used to present the algorithm. Section 46.5.2 describes the basic algorithm for constructing bar visibility representations, and Sections 46.5.3 and 46.5.4 show how to apply these representations to obtain layered drawings and orthogonal drawings respectively.

The algorithm in Section 46.5.2 and some of the preceding results are stated in terms of biconnected planar graphs. Note that this restriction can be avoided by augmenting graphs of lower connectivity with dummy edges.

46.5.1 Planar st-Graphs

Bar visibility representation algorithms use the theory of *planar st-graphs*. This is a powerful theory which is useful in a variety of graph drawing applications. In this section we describe enough of this theory to allow presentation of a bar visibility algorithm; for proofs of the results here, and more details about planar st-graphs, see [8].

A directed plane graph is a *planar st-graph* if it has one source s and one sink t, and both s and t are on the outside face. We say that a vertex u on a face f of a planar st-graph is a *source for* f (respectively *sink for* f) if the edges incident with u on f are out-edges (respectively in-edges).

LEMMA 46.2 Every face f of a planar st-graph has one source s_f for f and one sink t_f for f.

We need to extend the concept of the "planar dual", defined in Section 46.2, to directed graphs. For a directed plane graph G, the *directed dual* G^* has a vertex v_f for each internal face f of G, as well as two vertices ℓ_{ext} and r_{ext} for the external face of G.

To define the edges of G^*, we must first define the $left$ and $right$ side face of each edge of G. Suppose that f is an internal face of G, and the cycle of edges traversing f in a clockwise direction is $(e_1, e_2, \ldots, e_k)$. The clockwise traversal may pass through some edges in the same direction as the edge, and some edges in the opposite direction. If we pass through e_i in the same direction as e_i then we define $right(e_i) = v_f$; if we pass through e_j in the opposite direction to e_j then we define $left(e_j) = v_f$.

Now consider a clockwise traversal $(e'_1, e'_2, \ldots, e'_k)$ of the external face of G. If we pass through e'_i in the same direction as e'_i then we define $left(e_i) = \ell_{ext}$; if we pass through e_j in the opposite direction to e_j then we define $right(e_j) = r_{ext}$.

One can easily show that the definitions of $left$ and $right$ are well founded, that is, that for each edge e there is precisely one face f of G such that $left(e) = v_f$ and precisely one one face f' of G such that $right(e) = v_{f'}$.

For each edge e of G, G^* has an edge from $left(e)$ to $right(e)$.

An example is in Figure 46.14. Here $\ell_{ext} = 1 = left(e)$, and $r_{ext} = 5 = right(j) = right(g)$, $left(f) = left(i) = 2$, $right(f) = right(h) = 4$, and $left(h) = right(i) = 3$.

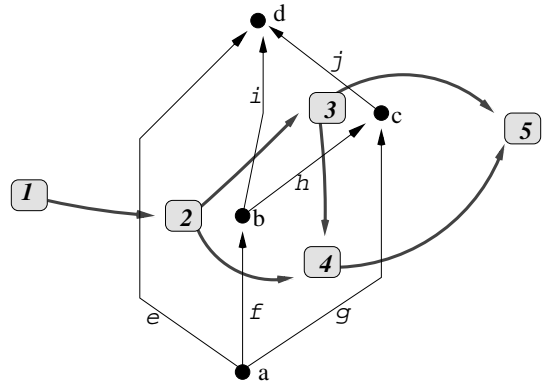


FIGURE 46.14: A directed planar graph and its directed dual.

The following Lemma (see [8] for a proof) is important for the bar visibility algorithm.

LEMMA 46.3
Suppose that f and f' are faces in a planar st-graph G. Then precisely one of the following statements is true.

(a) There is a directed path in G from the sink of f to the source of f'.
(b) There is a directed path in G from the sink of f' to the source of f.
(c) There is a directed path in G^* from v_f to $v_{f'}$.
(d) There is a directed path in G^* from $v_{f'}$ to v_f.

The next Lemma is, in a sense, dual to Lemma 46.2.

LEMMA 46.4 Suppose that u is a vertex in a planar st-graph. The outgoing edges from u are consecutive in the circular ordering of $N(u)$, and the incoming edges are consecutive in the circular ordering of $N(u)$.

Lemma 46.4 leads to a definition of the $left$ and $right$ of a vertex. Suppose that $(e_1, e_2, \ldots, e_k)$ is the circular list of edges around a non-source non-sink vertex u, in clockwise order. From Lemma 46.4, for some i, edge e_i comes into u and edge e_{i+1} goes out from u. we say that e_i is the *leftmost in-edge* of u and e_{i+1} is the *leftmost out-edge* of u. If f is the face shared by e_i and e_{i+1} then we define $left(u) = f$. Similarly, there is a j such that edge e_j goes out from u and edge e_{j+1} comes into u; we say that e_j is the *rightmost out-edge* of u and e_{j+1} is the *rightmost in-edge* of u. If f' is shared by e_j and e_{j+1} then we define $right(e_j) = f'$. If u is either the source or the sink, then $left(u) = \ell_{ext}$ and $right(u) = r_{ext}$.
In Figure 46.14, for example, $left(a) = left(d) = 1$, $left(b) = 2$, $left(c) = 3$, $right(b) = 4$, and $right(a) = right(c) = right(d) = 5$.
Finally, we need the following Lemma, which is a kind of dual to Lemma 46.3.

LEMMA 46.5
Suppose that u and u' are vertices in G. Then precisely one of the following statements is true.

(a) There is a directed path in G from u to u'.
(b) There is a directed path in G from u' to u.
(c) There is a directed path in G^* from $right(u)$ to $left(u')$.
(d) There is a directed path in G^* from $right(u')$ to $left(u)$.

46.5.2 The Bar Visibility Algorithm

The bar visibility algorithm takes a biconnected plane graph as input, and converts it to a planar st-graph. Then it computes the directed dual, and a "topological number" (defined below) for each vertex in both the original graph and the dual. Using these numbers, it assigns a y coordinate for each vertex and an x coordinate for each edge. The bar representing the vertex extends far enough to touch each incident edge, and the vertical line representing the edge extends far enough to touch each the bars representing its endpoints.
If $G = (V, E)$ is an acyclic directed graph with n vertices, a *topological numbering* Z of G assigns an integer $Z(u) \in \{0, 1, \ldots, n-1\}$ to every vertex $u \in V$ such that $Z(u) < Z(v)$ for

all directed edges $(u, v) \in E$. Note that we do not require that the function Z be one-one, that is, it is possible that two vertices are assigned the same number.

The algorithm is described below.

Algorithm `Bar_Visibility`
Input: a biconnected plane graph $G = (V, E)$
Output: a bar visibility representation of G

1. Choose two vertices s and t of G on the same face.
2. Direct the edges of G so that s is the only source, t is the only sink, and the resulting digraph is acyclic.
3. Compute a topological numbering Y for G.
4. Compute the directed planar dual $G^* = (V^*, E^*)$ of G.
5. Compute a topological numbering X for G^*.
6. For each vertex u, let ω_u be the line segment $[(X(left(u)), Y(u)), (X(right(u)) - 1, Y(u))]$.
7. For each edge $e = (u, v)$, let λ_e be the line segment $[(X(left(e)), Y(u)), (X(left(e)), Y(v))]$.

Step 2 can be implemented by a simple variation on the depth-first-search method, based on a biconnectivity algorithm.

There are many kinds of topological numberings for acyclic digraphs: for example, one can define $Z(u)$ to be the number of edges in the longest path from the source s to u. Any of these methods can be used in steps 3 and 5.

THEOREM 46.8 *(Tamassia and Tollis [34]; Rosenstiehl and Tarjan [32]) A visibility representation of a biconnected planar graph with area $O(n) \times O(n)$ can be computed in linear time.*

Proof We need to show that the drawing defined by ω and λ in Algorithm `Bar_Visibility` satisfies the four properties P1, P2, P3 and P4 above.

First consider P1, and suppose that u and u' are two vertices of G. If $Y(u) \neq Y(u')$ then ω_u has empty intersection with $\omega_{u'}$ and so P1 holds. Now suppose that $Y(u) = Y(u')$. Since Y is a topological numbering of G, it follows that there is no directed path in between u and u' (in either direction). Thus, from Lemma 46.5, in G^* either there is a directed path from $right(u)$ to $left(u')$ or a directed path from $right(u')$ to $left(u)$. The first case implies that $X(right(u)) < X(left(u')$, so that the whole of ω_u is to the left of $\omega_{u'}$; the second case implies that the whole of $\omega_{u'}$ is to the left of ω_u. This implies P1.

Now consider P2, and suppose that $e = (u, v)$ and $e' = (u', v')$ are edges of G, and denote $left(e)$ by f and $left(e')$ by f'. If $X(f) \neq X(f')$ then λ_e cannot intersect $\lambda_{e'}$ and P2 holds. Now suppose that $X(f) = X(f')$; thus in G^* there is no directed path between f and f'. It follows from Lemma 46.3 that in G either there is a directed path from the sink t_f of f to the source $s_{f'}$ of f', or a directed path from the sink $t_{f'}$ of f' to the source s_f of f. In the first case we have $Y(t_f) < Y(s_{f'})$. Also, since e is on f and e' is on f', we have $Y(v) \leq Y(t_f)$ and $Y(s_{f'}) \leq Y(u')$; thus $Y(v) < Y(u')$ and λ_e cannot intersect $\lambda_{e'}$. The second case is similar and so P2 holds.

A similar argument shows that P3 holds.

Property P4 follows form the simple observation that for any edge $e = (u, v)$, $X(left(u)) \leq X(left(e)) \leq X(right(u))$.

The drawing has dimensions $\max_{v_f \in V^*} X(v_f) \times \max_{u \in V} Y(u)$, which is $O(n) \times O(n)$.

It is easy to see that each step can be implemented in linear time.

46.5.3 Bar Visibility Representations and Layered Drawings

A graph in which the nodes are constrained to specified y coordinates, as in Figure 46.3, is called a *layered graph*. More precisely, a layered graph consists of a directed graph $G = (V, E)$ as well as a topological numbering L of G. We say that $L(u)$ is the *layer* of u. A drawing of G that satisfies the layering constraint, that is, the y coordinate of u is $L(u)$ for each $u \in V$, is called a *layered drawing*. Drawing algorithms for layered graphs have been extensively explored [8].

A layered graph is *planar* (sometimes called *h-planar*) if it can be drawn with no edge crossings, subject to the layering constraint. Note that underlying the prerequisite Figure 46.3 is a planar graph; however, as a layered graph, it is not planar. The theory of planarity for layered graphs has received some attention; see [11, 13, 19, 24].

If a planar layered graph has one source and one sink, then it is a planar st-graph. Clearly the source s has $L(s) = \min_{u \in V} L_u$ and the sink t has $L(t) = \max_{v \in V} L_v$. Since the layers define a topological numbering of the graph, application of Algorithm `Bar_Visibility` yields a visibility representation that satisfies the layering constraints. Further, this can be used to construct a graph drawing by using simple local transformations, illustrated in Figure 46.15.

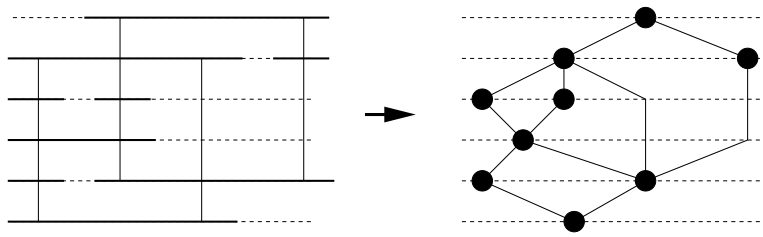


FIGURE 46.15: Transformation from a visibility representation of a layered graph to a layered drawing.

The following theorem is immediate.

THEOREM 46.9 *If G is a planar layered graph then we can construct a planar layered drawing of G in linear time, with the following properties:*

- *There are at most two bends in each edge.*
- *The area is $O(n) \times O(n)$.*

Although Algorithm `Bar_Visibility` produces the visibility representation with good resolution, it may not produce minimum area drawings. Lin and Eades [24] give a linear time variation on Algorithm *Bar Visibility* that, for a fixed embedding, maximizes resolution. The algorithm works by using a topological numbering for the dual computed from a dependency relation between the edges.

46.5.4 Bar Visibility Representations for Orthogonal Drawings

An *orthogonal drawing* of a graph G has each vertex represented as a point and each edge represented by a polyline whose line segments are parallel to a coordinate axis. An orthogonal drawing of the 3-cube is in Figure 46.16.

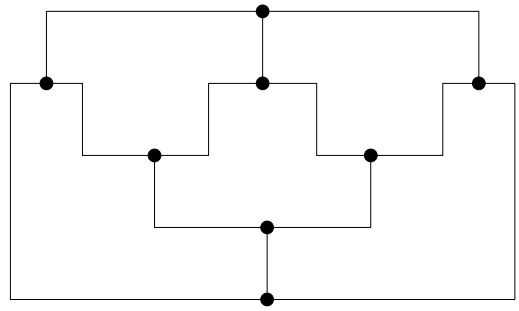


FIGURE 46.16: An orthogonal drawing of the 3-cube.

It is clear that a graph that has a planar orthogonal drawing, has maximum degree of G at most 4. Conversely, one can obtain a planar orthogonal drawing of a planar graph with maximum degree at most 4 from a visibility representation by using a simple set of local transformations, described in Figure 46.17.

Figure 46.16 can be obtained from Figure 46.13 by applying transformation (c) in Figure 46.17 to the source and sink, and applying transformation (d) to each other vertex.

Note that the transformations involve the introduction of a few bends in each edge. The worst case is for where a transformation introduces two bends near a vertex; this may occur at each end of an edge, and thus may introduce 4 bends into an edge.

THEOREM 46.10 *If G is a planar graph then we can construct a planar orthogonal drawing of G in linear time, with the following properties:*

- *There are at most four bends in each edge.*
- *The area is $O(n) \times O(n)$.*

In fact, a variation of Algorithm `Visibility` together with a careful application of some transformations along the lines of those in Figure 46.17 can ensure that the resulting drawing has a total of at most $2n + 4$ bends, where n is the number of vertices; see [8].

46.6 Conclusion

In this Chapter we have described just a small sample of graph drawing algorithms. Notable omissions include:

- *Force directed methods*: A graph can be used to define a system of forces. For example, we can define a Hooke's law spring force between two adjacent vertices, and magnetic repulsion between nonadjacent vertices. A minimum energy

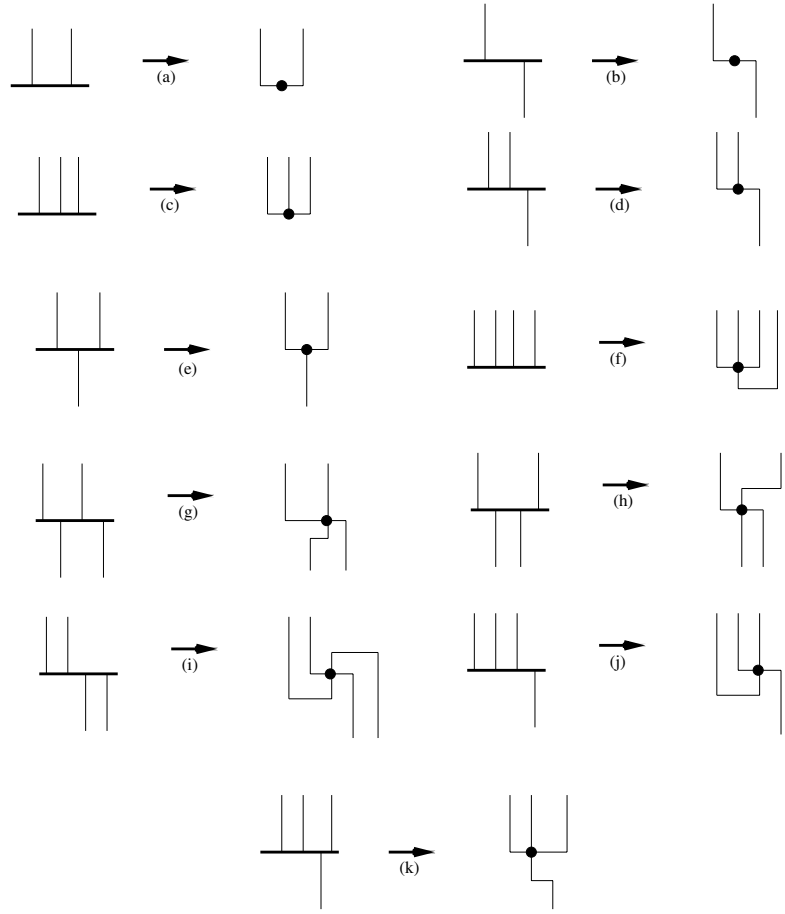


FIGURE 46.17: Local transformations to transform a visibility representation to an orthogonal drawing.

configuration of the graph can lead to a good drawing. For methods using the force-directed paradigm.

- *Clustered graph drawing methods*: in practice, to handle large graphs, one needs to form clusters of the vertices to form "super-vertices". Drawing graphs in which some vertices represent graphs is a challenging problem.
- *Three dimensional graph drawing methods*: the widespread availability of cheap three dimensional graphics systems has lead to the investigation of graph drawing in three dimensions.
- *Crossing minimization methods*: in this Chapter we have concentrated on planar graphs. In practice, we need to deal with non-planar graphs, by choosing a drawing with a small number of crossings.

References

[1] D. Abelson, S. Hong and D. E. Taylor, A Group-Theoretic Method for Drawing Graphs

Symmetrically, *Graph Drawing*, Proc. of GD'02, Lecture Notes in Computer Science 2528, Springer-Verlag, pp. 86-97, 2002.

[2] M. A. Armstrong, *Groups and Symmetry,* Springer-Verlag, 1988.

[3] C. Batini, E. Nardelli and R. Tamassia, A Layout Algorithm for Data-Flow Diagrams, *IEEE Trans. Softw. Eng.*, SE-12, no. 4, pp. 538-546, 1986.

[4] J. A. Bondy and U. S. R. Murty, *Graph Theory with Applications*, MacMillan Press, London, 1977.

[5] C. Buchheim and M. Jünger, Detecting Symmetries by Branch and Cut, *Graph Drawing*, Proc. of GD'01, Lecture Notes in Computer Science 2265, Springer-Verlag, pp. 178-188, 2002.

[6] N. Chiba, T. Yamanouchi and T. Nishizeki, Linear Algorithms for Convex Drawings of Planar Graphs, *Progress in Graph Theory*, Academic Press, pp. 153-173, 1984.

[7] M. Chrobak and G. Kant, Convex Grid Drawings of 3-connected Planar Graphs, *Int. Journal of Computational Geometry and Applications*, 7(3), pp. 211-224, 1997.

[8] G. Di Battista, P. Eades, R. Tamassia and I. G. Tollis, *Graph Drawing: Algorithms for the Visualization of Graphs*, Prentice-Hall 1998.

[9] H. de Fraysseix, An Heuristic for Graph Symmetry Detection, *Graph Drawing*, Proc. of GD'99, Lecture Notes in Computer Science 1731, Springer-Verlag, pp. 276-285, Springer-Verlag, 1999.

[10] H. de Fraysseix, J. Pach and R. Pollack, How to Draw a Planar Graph on a Grid, *Combinatorica*, 10, pp. 41-51, 1990.

[11] P. Eades, X. Lin and R. Tamassia, An Algorithm for Drawing a Hierarchical Graph, *International Journal of Computational Geometry and Applications*, 6(2), pp. 145-155, 1996.

[12] P. Eades and X. Lin, Spring Algorithms and Symmetry, *Theoretical Computer Science*, Vol. 240 No.2, pp. 379-405, 2000.

[13] P. Eades, Q. Feng, X. Lin and H. Nagamochi, Straight-Line Drawing Algorithms for Hierarchical Graphs and Clustered Graphs, *Algorithmica*, *to appear*.

[14] M. Fontet, Linear Algorithms for Testing Isomorphism of Planar Graphs, *Proceedings Third Colloquium on Automata, Languages, and Programming*, pp. 411-423, 1976.

[15] M. Goodrich and S. G. Kobourov (Eds.), *Graph Drawing 2002*, Lecture Notes in Computer Science 2528, Springer-Verlag, 2002.

[16] S. Hong, P. Eades and S. Lee, An Algorithm for Finding Geometric Automorphisms in Planar Graphs, *Algorithms and Computation,* Proc. of ISAAC'98, Lecture Notes in Computer Science 1533, Springer-Verlag, pp. 277-286, 1998.

[17] S. Hong, P. Eades and S. Lee, Drawing Series Parallel Digraphs Symmetrically, *Computational Geometry: Theory and Applicatons* 17(3-4), pp. 165-188, 2000.

[18] S. Hong, B. McKay and P. Eades, Symmetric Drawings of Triconnected Planar Graphs, *Proceeding of SODA 2002*, pp. 356-365, 2002.

[19] M. Junger, S. Leipert and P. Mutzel, Level Planarity Testing in Linear Time, *Graph Drawing 98*, Lecture Notes in Computer Science 1547, Springer-Verlag, pp. 167-182, 1999.

[20] M. Junger and P. Mutzel (Eds.), *Graph Drawing Software*, Series: Mathematics and Visualization, Springer-Verlag, 2003.

[21] G. Kant, Drawing Planar Graphs Using the Canonical Ordering, *Algorithmica*, 16, pp. 4-32, 1996.

[22] M. Kaufmann and D. Wagner (Eds.), *Drawing Graphs - Methods and Models*, Lecture Notes in Computer Science 2025, 2001.

[23] D. E. Knuth, Computer Drawn Flowcharts, *Commun, ACM*, 6, 1963.

[24] X. Lin and P. Eades, Towards Area Requirements for Drawing Hierarchically Planar

Graphs, *Theoretical Computer Science*, 292(3), pp. 679-695, 2003.

[25] R. J. Lipton, D. J. Rose and R. E. Tarjan, Generalized Nested Dissection, *SIAM J. Numer. Anal.*, 16, no. 2, pp. 346-358, 1979.

[26] J. Manning and M. J. Atallah, Fast Detection and Display of Symmetry in Trees, *Congressus Numerantium,* 64, pp. 159-169, 1988.

[27] J. Manning and M. J. Atallah, Fast Detection and Display of Symmetry in Outerplanar Graphs, *Discrete Applied Mathematics,* 39, pp. 13-35, 1992.

[28] J. Manning, Geometric Symmetry in Graphs, *Ph.D. Thesis, Purdue Univ.,* 1990.

[29] J. Martin and C. McClure, *Diagramming Techniques for Analyst and Programmers*, Prentice Hall, Inc., 1985.

[30] H. Purchase, Which Aesthetic Has the Greatest Effect on Human Understanding, *Graph Drawing*, Proc. of GD'97, Lecture Notes in Computer Science 1353, Springer-Verlag, pp. 248-261, 1998.

[31] R. Read, Some Applications of Computers in Graph Theory, In L.W. Beineke and R. Wilson, editors, *Selected Topics in Graph Theory*, chapter 15, pp. 415-444. Academic Press, 1978.

[32] P. Rosenstiehl and R. E. Tarjan, Rectilinear Planar Layouts and Bipolar Orientations of Planar Graphs, *Discrete Comput. Geom.*, 1, no. 4, pp. 342-351, 1986.

[33] K. Sugiyama, *Graph Drawing and Applications for Software and Knowledge Engineers*, Series on Software Engineering and Knowledge Engineering - Vol. 11, World Scientific, 2002.

[34] R. Tamassia and I. G. Tollis, A Unified Approach to Visibility Representations of Planar Graphs, *Discrete Comput. Geom.*, 1, no. 4, pp. 321-341, 1986.

[35] C. Thomassen, Planarity and Duality of Finite and Infinite Graphs, *J. of Combinatorial Theory,* Series B 29, pp. 244-271, 1980.

[36] W. T. Tutte, Convex Representations of Graphs, *Proc. London Math. Soc.*, 10, pp. 304-320, 1960.

[37] W. T. Tutte, How to Draw a Graph, *Proc. London Math. Soc.*, 13, pp. 743-768, 1963.

[38] K. Wagner, Bemerkungen zum Vierfarbenproblem, Jahres-bericht der Deutschen Mathematiker-Vereinigung, 46, pp. 26-32, 1936.

47

Concurrent Data Structures

Mark Moir
Sun Microsystems Laboratories

Nir Shavit
Sun Microsystems Laboratories

The proliferation of commercial shared-memory multiprocessor machines has brought about significant changes in the art of concurrent programming. Given current trends towards low-cost chip multithreading (CMT), such machines are bound to become ever more widespread.

Shared-memory multiprocessors are systems that concurrently execute multiple threads of computation which communicate and synchronize through data structures in shared memory. The efficiency of these data structures is crucial to performance, yet designing effective data structures for multiprocessor machines is an art currently mastered by few. By most accounts, concurrent data structures are far more difficult to design than sequential ones because threads executing concurrently may interleave their steps in many ways, each with a different and potentially unexpected outcome. This requires designers to modify the way they think about computation, to understand new design methodologies, and to adopt a new collection of programming tools. Furthermore, new challenges arise in designing *scalable* concurrent data structures that continue to perform well as machines that execute more and more concurrent threads become available. This chapter provides an overview of the challenges involved in designing concurrent data structures, and a summary of relevant work for some important data structure classes. Our summary is by no means comprehensive; instead, we have chosen popular data structures that illustrate key design issues, and hope that we have provided sufficient background and intuition to allow the interested reader to approach the literature we do not survey.

47.1 Designing Concurrent Data Structures

Several features of shared-memory multiprocessors make concurrent data structures significantly more difficult to design and to verify as correct than their sequential counterparts.

```
oldval = X;
X = oldval + 1;
return oldval;
```

```
acquire(Lock);
oldval = X;
X = oldval + 1;
release(Lock);
return oldval;
```

FIGURE 47.1: Code fragments for sequential and lock-based `fetch-and-inc` operations.

The primary source of this additional difficulty is concurrency: Because threads are executed concurrently on different processors, and are subject to operating system scheduling decisions, page faults, interrupts, etc., we must think of the computation as completely asynchronous, so that the steps of different threads can be interleaved arbitrarily. This significantly complicates the task of designing correct concurrent data structures.

Designing concurrent data structures for multiprocessor systems also provides numerous challenges with respect to performance and scalability. On today's machines, the layout of processors and memory, the layout of data in memory, the communication load on the various elements of the multiprocessor architecture all influence performance. Furthermore, the issues of correctness and performance are closely tied to each other: algorithmic enhancements that seek to improve performance often make it more difficult to design and verify a correct data structure implementation.

The following example illustrates various features of multiprocessors that affect concurrent data structure design. Suppose we wish to implement a *shared counter* data structure that supports a `fetch-and-inc` operation that adds one to the counter and returns the value of the counter immediately before the increment. A trivial sequential implementation of the `fetch-and-inc` operation contains code like that shown on the left in Figure 47.1:*

If we allow concurrent invocations of the `fetch-and-inc` operation by multiple threads, this implementation does not behave correctly. To see why, observe that most compilers will translate this source code into machine instructions that load `X` into a register, then add one to that register, then store that register back to `X`. Suppose that the counter is initially 0, and two `fetch-and-inc` operations execute on different processors concurrently. Then there is a risk that both operations read 0 from `X`, and therefore both store back 1 and return 0. This is clearly incorrect: one of the operations should return 1.

The incorrect behavior described above results from a "bad" interleaving of the steps of the two `fetch-and-inc` operations. A natural and common way to prevent such interleavings is to use a mutual exclusion lock (also known as a *mutex* or a *lock*). A lock is a construct that, at any point in time, is unowned or is owned by a single thread. If a thread t_1 wishes to acquire ownership of a lock that is already owned by another thread t_2, then t_1 must wait until t_2 releases ownership of the lock.

We can obtain a correct sequential implementation of the `fetch-and-inc` operation by using a lock as shown on the right in Figure 47.1. With this arrangement, we prevent the bad interleavings by preventing *all* interleavings. While it is easy to achieve a correct shared counter this way, this simplicity comes at a price: Locking introduces a host of problems related to both performance and software engineering.

*Throughout our examples, we ignore the fact that, in reality, integers are represented by a fixed number of bits, and will therefore eventually "wrap around" to zero.

47.1.1 Performance

The *speedup* of an application when run on P processors is the ratio of its execution time on a single processor to its execution time on P processors. It is a measure of how effectively the application is utilizing the machine it is running on. Ideally, we want *linear speedup*: we would like to achieve a speedup of P when using P processors. Data structures whose speedup grows with P are called *scalable*. In designing scalable data structures we must take care: naive approaches to synchronization can severely undermine scalability.

Returning to the lock-based counter, observe that the lock introduces a *sequential bottleneck*: at any point in time, at most one `fetch-and-inc` operation is doing useful work, i.e. incrementing the variable `X`. Such sequential bottlenecks can have a surprising effect on the speedup one can achieve. The effect of the sequentially executed parts of the code on performance is illustrated by a simple formula based on Amdahl's law [109]. Let b be the fraction of the program that is subject to a sequential bottleneck. If the program takes 1 time unit when executed on a single processor, then on a P-way multiprocessor the sequential part takes b time units, and the concurrent part takes $(1-b)/P$ time units in the best case, so the speedup S is at most $1/(b+(1-b)/P)$. This implies that if just 10% of our application is subject to a sequential bottleneck, the best possible speedup we can achieve on a 10-way machine is about 5.3: we are running the application at half of the machine's capacity. Reducing the number and length of sequentially executed code sections is thus crucial to performance. In the context of locking, this means reducing the number of locks acquired, and reducing *lock granularity*, a measure of the number of instructions executed while holding a lock.

A second problem with our simple counter implementation is that it suffers from *memory contention*: an overhead in traffic in the underlying hardware as a result of multiple threads concurrently attempting to access the same locations in memory. Contention can be appreciated only by understanding some aspects of common shared-memory multiprocessor architectures. If the lock protecting our counter is implemented in a single memory location, as many simple locks are, then in order to acquire the lock, a thread must repeatedly attempt to modify that location. On a *cache-coherent* multiprocessor[†] for example, exclusive ownership of the cache line containing the lock must be repeatedly transferred from one processor to another; this results in long waiting times for each attempt to modify the location, and is further exacerbated by the additional memory traffic associated with unsuccessful attempts to acquire the lock. In Section 47.1.7, we discuss lock implementations that are designed to avoid such problems for various types of shared memory architectures.

A third problem with our lock-based implementation is that, if the thread that currently holds the lock is delayed, then all other threads attempting to access the counter are also delayed. This phenomenon is called *blocking*, and is particularly problematic in *multiprogrammed* systems, in which there are multiple threads per processor and the operating system can preempt a thread while it holds the lock. For many data structures, this difficulty can be overcome by devising *nonblocking* algorithms in which the delay of a thread does not cause the delay of others. By definition, these algorithms cannot use locks.

Below we continue with our shared counter example, discussing blocking and nonblocking techniques separately; we introduce more issues related to performance as they arise.

[†]A *cache-coherent* multiprocessor is one in which processors have local caches that are updated by hardware in order to keep them consistent with the latest values stored.

47.1.2 Blocking Techniques

In many data structures, the undesirable effects of memory contention and sequential bottlenecks discussed above can be reduced by using a *fine-grained* locking scheme. In fine-grained locking, we use multiple locks of small granularity to protect different parts of the data structure. The goal is to allow concurrent operations to proceed in parallel when they do not access the same parts of the data structure. This approach can also help to avoid excessive contention for individual memory locations. For some data structures, this happens naturally; for example, in a hash table, operations concerning values that hash to different buckets naturally access different parts of the data structure.

For other structures, such as the lock-based shared counter, it is less clear how contention and sequential bottlenecks can be reduced because—abstractly—all operations modify the same part of the data structure. One approach to dealing with contention is to spread operations out in time, so that each operation accesses the counter in a separate time interval from the others. One widely used technique for doing so is called *backoff* [3]. However, even with reduced contention, our lock-based shared counter still lacks parallelism, and is therefore not scalable. Fortunately, more sophisticated techniques can improve scalability.

One approach, known as a *combining tree* [36, 37, 51, 137], can help implement a scalable counter. This approach employs a binary tree with one leaf per thread. The root of the tree contains the actual counter value, and the other internal nodes of the tree are used to coordinate access to the root. The key idea is that threads climb the tree from the leaves towards the root, attempting to "combine" with other concurrent operations. Every time the operations of two threads are combined in an internal node, one of those threads—call it the loser—simply waits at that node until a return value is delivered to it. The other thread—the winner—proceeds towards the root carrying the sum of all the operations that have combined in the subtree rooted at that node; a winner thread that reaches the root of the tree adds its sum to the counter in a single addition, thereby effecting the increments of all of the combined operations. It then descends the tree distributing a return value to each waiting loser thread with which it previously combined. These return values are distributed so that the effect is as if all of the increment operations were executed one after the other at the moment the root counter was modified.

The technique losers employ while waiting for winners in the combining tree is crucial to its performance. A loser operation waits by repeatedly reading a memory location in a tree node; this is called *spinning*. An important consequence in a cache-coherent multiprocessor is that this location will reside in the local cache of the processor executing the loser operation until the winner operation reports the result. This means that the waiting loser does not generate any unnecessary memory traffic that may slow down the winner's performance. This kind of waiting is called *local spinning*, and has been shown to be crucial for scalable performance [96].

In so-called *non-uniform memory access* (NUMA) architectures, processors can access their local portions of shared memory faster than they can access the shared memory portions of other processors. In such architectures, *data layout*—the way nodes of the combining tree are placed in memory—can have a significant impact on performance. Performance can be improved by locating the leaves of the tree near the processors running the threads that own them. (We assume here that threads are statically bound to processors.)

Data layout issues also affect the design of concurrent data structures for cache-coherent multiprocessors. Recall that one of the goals of the combining tree is to reduce contention for individual memory locations in order to improve performance. However, because cache-coherent multiprocessors manage memory in cache-line-sized chunks, if two threads are accessing different memory locations that happen to fall into the same cache line, performance

suffers just as if they had been accessing the same memory location. This phenomenon is known as *false sharing*, and is a common source of perplexing performance problems.

By reducing contention for individual memory locations, reducing memory traffic by using local spinning, and allowing operations to proceed in parallel, counters implemented using combining trees scale with the number of concurrent threads much better than the single lock version does [51]. If all threads manage to repeatedly combine, then a tree of width P will allow P threads to return P values after every $O(\log P)$ operations required to ascend and descend the tree, offering a speedup of $O(P/\log P)$ (but see Section 47.1.4).

Despite the advantages of the combining tree approach, it also has several disadvantages. It requires a known bound P on the number of threads that access the counter, and it requires $O(P)$ space. While it provides better throughout under *heavy loads*, that is, when accessed by many concurrent threads, its best-case performance under low loads is poor: It must still traverse $O(\log P)$ nodes in the tree, whereas a `fetch-and-inc` operation of the single-lock-based counter completes in constant time. Moreover, if a thread fails to combine because it arrived at a node immediately after a winner left it on its way up the tree, then it must wait until the winner returns before it can continue its own ascent up the tree. The coordination among ascending winners, losers, and ascending late threads, if handled incorrectly, may lead to *deadlocks*: threads may block each other in a cyclic fashion such that none ever makes progress. Avoiding deadlocks significantly complicates the task of designing correct and efficient implementations of blocking concurrent data structures.

In summary, blocking data structures can provide powerful and efficient implementations if a good balance can be struck between using enough blocking to maintain correctness, while minimizing blocking in order to allow concurrent operations to proceed in parallel.

47.1.3 Nonblocking Techniques

As discussed earlier, nonblocking implementations aim to overcome the various problems associated with the use of locks. To formalize this idea, various nonblocking progress conditions—such as wait-freedom [49, 82], lock-freedom [49], and obstruction-freedom [53]—have been considered in the literature. A *wait-free* operation is guaranteed to complete after a finite number of its own steps, regardless of the timing behavior of other operations. A *lock-free* operation guarantees that after a finite number of its own steps, *some* operation completes. An *obstruction-free* operation is guaranteed to complete within a finite number of its own steps after it stops encountering interference from other operations.

Clearly, wait-freedom is a stronger condition than lock-freedom, and lock-freedom in turn is stronger than obstruction-freedom. However, all of these conditions are strong enough to preclude the use of blocking constructs such as locks.[‡] While stronger progress conditions seem desirable, implementations that make weaker guarantees are generally simpler, more efficient in the common case, and easier to design and to verify as correct. In practice, we can compensate for the weaker progress conditions by employing backoff [3] or more sophisticated contention management techniques [54].

Let us return to our shared counter. It follows easily from results of Fischer et al. [32] (extended to shared memory by Herlihy [49] and Loui and Abu-Amara [92]) that such a shared counter cannot be implemented in a lock-free (or wait-free) manner using only `load` and

[‡]We use the term "nonblocking" broadly to include all progress conditions requiring that the failure or indefinite delay of a thread cannot prevent other threads from making progress, rather than as a synonym for "lock-free", as some authors prefer.

```
bool CAS(L, E, N) {
  atomically {
    if (*L == E) {
      *L = N;
      return true;
    } else
      return false;
  }
}
```

FIGURE 47.2: The semantics of the CAS operation.

`store` instructions to access memory. These results show that, in any proposed implementation, a bad interleaving of operations can cause incorrect behaviour. These bad interleavings are possible because the `load` and `store` are separate operations. This problem can be overcome by using a hardware operation that atomically combines a load and a store. Indeed, all modern multiprocessors provide such synchronization instructions, the most common of which are *compare-and-swap* (CAS) [61, 63, 136] and *load-linked/store-conditional* (LL/SC) [62, 69, 131]. The semantics of the CAS instruction is shown in Figure 47.2. For purposes of illustration, we assume an `atomically` keyword which requires the code block it labels to be executed *atomically*, that is, so that that no thread can observe a state in which the code block has been partially executed. The CAS operation atomically loads from a memory location, compares the value read to an expected value, and stores a new value to the location if the comparison succeeds. Herlihy [49] showed that instructions such as CAS and LL/SC are *universal*: there exists a wait-free implementation for *any* concurrent data structure in a system that supports such instructions.

A simple lock-free counter can be implemented using CAS. The idea is to perform the `fetch-and-inc` by loading the counter's value and to then use CAS to atomically change the counter value to a value greater by one than the value read. The CAS instruction fails to increment the counter only if it changes between the load and the CAS. In this case, the operation can retry, as the failing CAS had no effect. Because the CAS fails only as a result of another `fetch-and-inc` operation succeeding, the implementation is lock-free. However, it is not wait-free because a single `fetch-and-inc` operation can repeatedly fail its CAS.

The above example illustrates an *optimistic* approach to synchronization: the `fetch-and-inc` operation completes quickly in the hopefully common case in which it does not encounter interference from a concurrent operation, but must employ more expensive techniques under contention (e.g., backoff).

While the lock-free counter described above is simple, it has many of the same disadvantages that the original counter based on a single lock has: a sequential bottleneck and high contention for a single location. It is natural to attempt to apply similar techniques as those described above in order to improve the performance of this simple implementation. However, it is usually more difficult to incorporate such improvements into nonblocking implementations of concurrent data structures than blocking ones. Roughly, the reason for this is that a thread can use a lock to prevent other threads from "interfering" while it performs some sequence of actions. Without locks, we have to design our implementations to be correct despite the actions of concurrent operations; in current architectures, this often leads to the use of complicated and expensive techniques that undermine the improvements we are trying to incorporate. As discussed further in Section 47.1.7, transactional mechanisms make it much easier to design and modify efficient implementations of complex concurrent data structures. However, hardware support for such mechanisms does not yet exist.

47.1.4 Complexity Measures

A wide body of research is directed at analyzing the asymptotic complexity of concurrent data structures and algorithms in idealized models such as *parallel random access machines* [35, 121, 134]. However, there is less work on modeling such data structures in a realistic multiprocessor setting. There are many reasons for this, most of which have to do with the interplay of the architectural features of the hardware and the asynchronous execution of threads. Consider the combining tree example. Though we argued a speedup of $O(P/\log P)$ by counting instructions, this is not reflected in empirical studies [51, 128]. Real-world behavior is dominated by other features discussed above, such as cost of contention, cache behavior, cost of universal synchronization operations (e.g. CAS), arrival rates of requests, effects of backoff delays, layout of the data structure in memory, and so on. These factors are hard to quantify in a single precise model spanning all current architectures. Complexity measures that capture some of these aspects have been proposed by Dwork et al. [28] and by Anderson and Yang [7]. While these measures provide useful insight into algorithm design, they cannot accurately capture the effects of all of the factors listed above. As a result, concurrent data structure designers compare their designs empirically by running them using micro-benchmarks on real machines and simulated architectures [9, 51, 96, 102]. In the remainder of this chapter, we generally qualify data structures based on their empirically observed behavior and use simple complexity arguments only to aid intuition.

47.1.5 Correctness

It is easy to see that the behavior of the simple lock-based counter is "the same" as that of the sequential implementation. However, it is significantly more difficult to see this is also true for the combining tree. In general, concurrent data structure implementations are often subtle, and incorrect implementations are not uncommon. Therefore, it is important to be able to state and prove rigorously that a particular design correctly implements the required concurrent data structure. We cannot hope to achieve this without a precise way of specifying what "correct" means.

Data structure specifications are generally easier for sequential data structures. For example, we can specify the semantics of a sequential data structure by choosing a set of states, and providing a *transition function* that takes as arguments a state, an operation name and arguments to the operation, and returns a new state and a return value for the operation. Together with a designated initial state, the transition function specifies all acceptable sequences of operations on the data structure. The sequential semantics of the counter is specified as follows: The set of states for the counter is the set of integers, and the initial state is 0. The transition function for the `fetch-and-inc` operation adds one to the old state to obtain the new state, and the return value is the old state of the counter.

Operations on a sequential data structure are executed one-at-a-time in order, and we simply require that the resulting sequence of operations respects the sequential semantics specified as discussed above. However, with concurrent data structures, operations are not necessarily totally ordered. Correctness conditions for concurrent data structures generally require that *some* total order of the operations exists that respects the sequential semantics. Different conditions are distinguished by their different requirements on this total ordering.

A common condition is Lamport's *sequential consistency* [80], which requires that the total order preserves the order of operations executed by each thread. Sequential consistency has a drawback from the software engineering perspective: a data structure implemented using sequentially consistent components may not be sequentially consistent itself.

A natural and widely used correctness condition that overcomes this problem is Herlihy

and Wing's *linearizability* [57], a variation on the *serializability* [16] condition used for database transactions. Linearizability requires two properties: (1) that the data structure be sequentially consistent, and (2) that the total ordering which makes it sequentially consistent respect the *real-time ordering* among the operations in the execution. Respecting the real-time ordering means that if an operation O_1 finishes execution before another operation O_2 begins (so the operations are not concurrent with each other), then O_1 must be ordered before O_2. Another way of thinking of this condition is that it requires us to be able to identify a distinct point within each operation's execution interval, called its *linearization point*, such that if we order the operations according to the order of their linearization points, the resulting order obeys the desired sequential semantics.

It is easy to see that the counter implementation based on a single lock is linearizable. The state of the counter is always stored in the variable `X`. We define the linearization point of each `fetch-and-inc` operation as the point at which it stores its incremented value to `X`. The linearizability argument for the CAS-based lock-free implementation is similarly simple, except that we use the semantics of the CAS instruction, rather than reasoning about the lock, to conclude that the counter is incremented by one each time it is modified.

For the combining tree, it is significantly more difficult to see that the implementation is linearizable because the state of the counter is incremented by more than one at a time, and because the increment for one `fetch-and-inc` operation may in fact be performed by another operation with which it has combined. We define the linearization points of `fetch-and-inc` operations on the combining-tree-based counter as follows. When a winner thread reaches the root of the tree and adds its accumulated value to the counter, we linearize each of the operations with which it has combined in sequence immediately after that point. The operations are linearized in the order of the return values that are subsequently distributed to those operations. While a detailed linearizability proof is beyond the scope of our presentation, it should be clear from this discussion that rigorous correctness proofs for even simple concurrent data structures can be quite challenging.

The intuitive appeal and modularity of linearizability makes it a popular correctness condition, and most of the concurrent data structure implementations we discuss in the remainder of this chapter are linearizable. However, in some cases, performance and scalability can be improved by satisfying a weaker correctness condition. For example, the *quiescent consistency* condition [10] drops the requirement that the total ordering of operations respects the real-time order, but requires that every operation executed after a quiescent state—one in which no operations are in progress—must be ordered after every operation executed before that quiescent state. Whether an implementation satisfying such a weak condition is useful is application-dependent. In contrast, a linearizable implementation is always usable, because designers can view it as atomic.

47.1.6 Verification Techniques

In general, to achieve a rigorous correctness proof for a concurrent data structure implementation, we need mathematical machinery for specifying correctness requirements, accurately modeling a concurrent data structure implementation, and ensuring that a proof that the implementation is correct is complete and accurate. Most linearizability arguments in the literature treat at least some of this machinery informally, and as a result, it is easy to miss cases, make incorrect inferences, etc. Rigorous proofs inevitably contain an inordinate amount of mundane detail about trivial properties, making them tedious to write and to read. Therefore, computer-assisted methods for verifying implementations are required. One approach is to use a *theorem prover* to prove a series of assertions which together imply that an implementation is correct. Another approach is to use *model checking* tools,

```
void acquire(Lock *lock) {
  int delay = MIN_DELAY;
  while (true) {
    if (CAS(lock,UNOWNED,OWNED))
      return;
    sleep(random() % delay);
    if (delay < MAX_DELAY)
      delay = 2 * delay;
  }
}

void release(Lock *lock) {
  *lock = UNOWNED;
}
```

FIGURE 47.3: Exponential backoff lock.

which exhaustively check all possible executions of an implementation to ensure that each one meets specified correctness conditions. The theorem proving approach usually requires significant human insight, while model checking is limited by the number of states of an implementation that can be explored.

47.1.7 Tools of the Trade

Below we discuss three key types of tools one can use in designing concurrent data structures: locks, barriers, and transactional synchronization mechanisms. Locks and barriers are traditional low-level synchronization mechanisms that are used to restrict certain interleavings, making it easier to reason about implementations based on them. Transactional mechanisms seek to hide the tricky details of concurrency from programmers, allowing them to think in a more traditional sequential way.

Locks

As discussed earlier, locks are used to guarantee mutually exclusive access to (parts of) a data structure, in order to avoid "bad" interleavings. A key issue in designing a lock is what action to take when trying to acquire a lock already held by another thread. On uniprocessors, the only sensible option is to yield the processor to another thread. However, in multiprocessors, it may make sense to repeatedly attempt to acquire the lock, because the lock may soon be released by a thread executing on another processor. Locks based on this technique are called *spinlocks*. The choice between blocking and spinning is often a difficult one because it is hard to predict how long the lock will be held. When locks are supported directly by operating systems or threads packages, information such as whether the lock-holder is currently running can be used in making this decision.

A simple spinlock repeatedly uses a synchronization primitive such as compare-and-swap to atomically change the lock from unowned to owned. As mentioned earlier, if locks are not designed carefully, such spinning can cause heavy contention for the lock, which can have a severe impact on performance. A common way to reduce such contention is to use *exponential backoff* [3]. In this approach, which is illustrated in Figure 47.3, a thread that is unsuccessful in acquiring the lock waits some time before retrying; repeated failures cause longer waiting times, with the idea that threads will "spread themselves out" in time, resulting in lower contention and less memory traffic due to failed attempts.

Exponential backoff has the disadvantage that the lock can be unowned, but all threads attempting to acquire it have backed off too far, so none of them is making progress. One way to overcome this is to have threads form a queue and have each thread that releases the lock pass ownership of the lock to the next queued thread. Locks based on this approach

are called *queuelocks*. Anderson [8] and Graunke and Thakkar [38] introduce array-based queuelocks, and these implementations are improved upon by the list-based MCS queue locks of Mellor-Crummey and Scott [96] and the CLH queue locks of Craig and Landin and Hagersten [25, 93].

Threads using CLH locks form a virtual linked list of nodes, each containing a `done` flag; a thread enters the critical section only after the `done` flag of the node preceding its own node in the list is raised. The lock object has a pointer to the node at the tail of the list, the last one to join it. To acquire the lock, a thread creates a node, sets its `done` flag to false indicate that it has not yet released the critical section, and uses a synchronization primitive such as CAS to place its own node at the tail of the list while determining the node of its predecessor. It then spins on the `done` flag of the predecessor node. Note that each thread spins on a different memory location. Thus, in a cache-based architecture, when a thread sets its `done` flag to inform the next thread in the queue that it can enter the critical section, the `done` flags on which all other threads are spinning are not modified, so those threads continue to spin on a local cache line, and do not produce additional memory traffic. This significantly reduces contention and improves scalability in such systems. However, if this algorithm is used in a non-coherent NUMA machine, threads will likely have to spin on remote memory locations, again causing excessive memory traffic. The MCS queuelock [96] overcomes this problem by having each thread spin on a `done` flag in its *own* node. This way, nodes can be allocated in local memory, eliminating the problem.

There are several variations on standard locks that are of interest to the data structure designer in some circumstances. The queuelock algorithms discussed above have more advanced *abortable* versions that allow threads to give up on waiting to acquire the lock, for example, if they are delayed beyond some limit in a real-time application [122, 124], or if they need to recover from deadlock. The algorithms of [122] provide an abort that is nonblocking. Finally, [102] presents *preemption-safe* locks, which attempt to reduce the negative performance effects of preemption by ensuring that a thread that is in the queue but preempted does not prevent the lock from being granted to another running thread.

In many data structures it is desirable to have locks that allow concurrent readers. Such *reader-writer* locks allow threads that only read data in the critical section (but do not modify it) to access the critical section exclusively from the writers but concurrently with each other. Various algorithms have been suggested for this problem. The reader-writer queuelock algorithms of Mellor-Crummey and Scott [97] are based on MCS queuelocks and use read counters and a special pointer to writer nodes. In [75] a version of these algorithms is presented in which readers remove themselves from the lock's queue. This is done by keeping a doubly-linked list of queued nodes, each having its own simple "mini-lock." Readers remove themselves from the queuelock list by acquiring mini-locks of their neighboring nodes and redirecting the pointers of the doubly-linked list. The above-mentioned real-time queuelock algorithms of [122] provide a similar ability without locking nodes.

The reader-writer lock approach can be extended to arbitrarily many operation types through a construct called *group mutual exclusion* or *room synchronization*. The idea is that operations are partitioned into groups, such that operations in the same group can execute concurrently with each other, but operations in different groups must not. An interesting application of this approach separates push and pop operations on a stack into different groups, allowing significant simplifications to the implementations of those operations because they do not have to deal with concurrent operations of different types [18]. Group mutual exclusion was introduced by Joung [68]. Implementations based on mutual exclusion locks or `fetch-and-inc` counters appear in [18, 70].

More complete and detailed surveys of the literature on locks can be found in [6, 116].

Barriers

A barrier is a mechanism that collectively halts threads at a given point in their code, and allows them to proceed only when all threads have arrived at that point. The use of barriers arises whenever access to a data structure or application is layered into phases whose execution should not overlap in time. For example, repeated iterations of a numerical algorithm that converges by iterating a computation on the same data structure or the marking and sweeping phases of a parallel garbage collector.

One simple way to implement a barrier is to use a counter that is initialized to the total number of threads: Each thread decrements the counter upon reaching the barrier, and then spins, waiting for the counter to become zero before proceeding. Even if we use the techniques discussed earlier to implement a scalable counter, this approach still has the problem that waiting threads produce contention. For this reason, specialized barrier implementations have been developed that arrange for each thread to spin on a different location [21, 48, 123]. Alternatively, a barrier can be implemented using a *diffusing computation* tree in the style of Dijkstra and Scholten [27]. In this approach, each thread is the owner of one node in a binary tree. Each thread awaits the arrival of its children, then notifies its parent that it has arrived. Once all threads have arrived, the root of the tree releases all threads by disseminating the release information down the tree.

Transactional Synchronization Mechanisms

A key use of locks in designing concurrent data structures is to allow threads to modify multiple memory locations atomically, so that no thread can observe partial results of an update to the locations. Transactional synchronization mechanisms are tools that allow the programmer to treat sections of code that access multiple memory locations as a single atomic step. This substantially simplifies the design of correct concurrent data structures because it relieves the programmer of the burden of deciding which locks should be held for which memory accesses and of preventing deadlock.

As an example of the benefits of transactional synchronization, consider Figure 47.4, which shows a concurrent queue implementation achieved by requiring operations of a simple sequential implementation to be executed atomically. Such atomicity could be ensured either by using a global mutual exclusion lock, or via a transactional mechanism. However, the lock-based approach prevents concurrent `enqueue` and `dequeue` operations from executing in parallel. In contrast, a good transactional mechanism will allow them to do so in all but the empty state because when the queue is not empty, concurrent `enqueue` and `dequeue` operations do not access any memory locations in common.

The use of transactional mechanisms for implementing concurrent data structures is inspired by the widespread use of transactions in database systems. However, the problem of supporting transactions over shared memory locations is different from supporting transactions over databases elements stored on disk. Thus, more lightweight support for transactions is possible in this setting.

Kung and Robinson's *optimistic concurrency control* (OCC) [79] is one example of a transactional mechanism for concurrent data structures. OCC uses a global lock, which is held only for a short time at the end of a transaction. Nonetheless, the lock is a sequential bottleneck, which has a negative impact on scalability. Ideally, transactions should be supported without the use of locks, and transactions that access disjoint sets of memory locations should not synchronize with each other at all.

Transactional support for multiprocessor synchronization was originally suggested by Herlihy and Moss, who also proposed a hardware-based *transactional memory* mechanism for supporting it [55]. Recent extensions to this idea include *lock elision* [113, 114], in which the

```
typedef struct qnode_s { qnode_s *next; valuetype value; } qnode_t;

shared variables:
  // initially null
  qnode_t *head, *tail;

void enqueue(qnode_t *n) {
  atomically {
    if (tail == null)
      tail = head = n;
    else {
      tail->next = n;
      tail = n;
    }
  }
}

qnode_t * dequeue() {
  atomically {
    if (head == null)
      return null;
    else {
      n = head;
      head = head->next;
      if (head == null)
        tail = null;
      return n;
    }
  }
}
```

FIGURE 47.4: A concurrent shared FIFO queue.

hardware automatically translates critical sections into transactions, with the benefit that two critical sections that do not in fact conflict with each other can be executed in parallel. For example, lock elision could allow concurrent `enqueue` and `dequeue` operations of the above queue implementation to execute in parallel, even if the atomicity is implemented using locks. To date, hardware support for transactional memory has not been built.

Various forms of *software transactional memory* have been proposed by Shavit and Touitou [127], Harris et al. [44], Herlihy et al. [54], and Harris and Fraser [43].

Transactional mechanisms can easily be used to implement most concurrent data structures, and when efficient and robust transactional mechanisms become widespread, this will likely be the preferred method. In the following sections, we mention implementations based on transactional mechanisms only when no direct implementation is known.

47.2 Shared Counters and Fetch-and-ϕ Structures

Counters have been widely studied as part of a broader class of *fetch-and-ϕ* coordination structures, which support operations that fetch the current value of a location and apply some function from an allowable set ϕ to its contents. As discussed earlier, simple lock-based implementations of fetch-and-ϕ structures such as counters suffer from contention and sequential bottlenecks. Below we describe some approaches to overcoming these problems.

Combining

The combining tree technique was originally invented by Gottlieb et al. [37] to be used in the hardware switches of processor-to-memory networks. In Section 47.1.2 we discussed a software version of this technique, first described by Goodman et al. [36] and Yew et al. [137], for implementing a fetch-and-add counter. (The algorithm in [137] has a slight bug; see [51].) This technique can also be used to implement fetch-and-ϕ operations for a variety of sets of combinable operations, including arithmetic and boolean operations, and synchronization operations such as load, store, swap, test-and-set, etc. [76].

As explained earlier, scalability is achieved by sizing the tree such that the there is one leaf node per thread. Under maximal load, the throughput of such a tree is proportional to $O(P/\log P)$ operations per time unit, offering a significant speedup. Though it is pos-

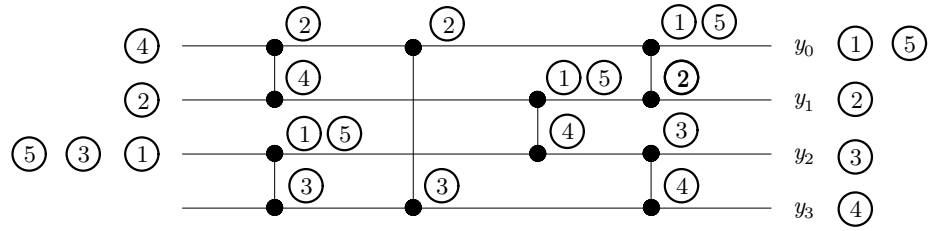


FIGURE 47.5: A bitonic counting network of width four.

sible to construct trees with fan-out greater than two in order to reduce tree depth, that would sacrifice the simplicity of the nodes and, as shown by Shavit and Zemach [130], will most likely result in reduced performance. Moreover, Herlihy et al. [51] have shown that combining trees are extremely sensitive to changes in the arrival rate of requests: as the load decreases, threads must still pay the price of traversing the tree while attempting to combine, but the likelihood of combining is reduced because of the reduced load.

Shavit and Zemach overcome the drawbacks of the static combining tree structures by introducing combining funnels [130]. A *combining funnel* is a linearizable fetch-and-ϕ structure that allows combining trees to form dynamically, adapting its overall size based on load patterns. It is composed of a (typically small) number of *combining layers*. Each such layer is implemented as a *collision array* in memory. Threads pass through the funnel layer by layer, from the first (widest) to the last (narrowest). These layers are used by threads to locate each other and combine their operations. As threads pass through a layer, they read a thread ID from a randomly chosen array element, and write their own in its place. They then attempt to combine with the thread whose ID they read. A successful combination allows threads to exchange information, allowing some to continue to the next layer, and others to await their return with the resulting value. Combining funnels can also support the *elimination* technique (described in Section 47.3) to allow two operations to complete without accessing the central data structure in some cases.

Counting Networks

Combining structures provide scalable and linearizable fetch-and-ϕ operations. However, they are blocking. An alternative approach to parallelizing a counter that overcomes this problem is to have multiple counters instead of a single one, and to use a counting network to coordinate access to the separate counters so as to avoid problems such as duplicated or omitted values. *Counting networks*, introduced by Aspnes et al. [10], are a class of data structures for implementing, in a highly concurrent and nonblocking fashion, a restricted class of fetch-and-ϕ operations, the most important of which is `fetch-and-inc`.

Counting networks, like sorting networks [24], are acyclic networks constructed from simple building blocks called balancers. In its simplest form, a *balancer* is a computing element with two input wires and two output wires. Tokens arrive on the balancer's input wires at arbitrary times, and are output on its output wires in a balanced way. Given a stream of input tokens, a balancer alternates sending one token to the top output wire, and one to the bottom, effectively balancing the number of tokens between the two wires.

We can wire balancers together to form a network. The *width* of a network is its number of output wires (wires that are not connected to an input of any balancer). Let $y_0, .., y_{w-1}$ respectively represent the number of tokens output on each of the output wires of a network of width w. A *counting network* is an acyclic network of balancers whose outputs satisfy the following *step property*:

In any quiescent state, $0 \leq y_i - y_j \leq 1$ for any $i < j$.

Figure 47.5 shows a sequence of tokens traversing a counting network of width four based on Batcher's Bitonic sorting network structure [13]. The horizontal lines are wires and the vertical lines are balancers, each connected to two input and output wires at the dotted points. Tokens (numbered 1 through 5) traverse the balancers starting on arbitrary input wires and accumulate on specific output wires meeting the desired step-property. Aspnes et al. [10] have shown that every counting network has a layout isomorphic to a sorting network, but not every sorting network layout is isomorphic to a counting network.

On a shared memory multiprocessor, balancers are records, and wires are pointers from one record to another. Threads performing increment operations traverse the data structure from some input wire (either preassigned or chosen at random) to some output wire, each time shepherding a new token through the network.

The counting network distributes input tokens to output wires while maintaining the step property stated above. Counting is done by adding a local counter to each output wire, so that tokens coming out of output wire i are assigned numbers $i, i + w, \ldots, i + (y_i - 1)w$. Because threads are distributed across the counting network, there is little contention on the balancers, and the even distribution on the output wires lowers the load on the shared counters. However, as shown by Shavit and Zemach [128], the dynamic patterns through the networks increase cache miss rates and make optimized layout almost impossible.

There is a significant body of literature on counting networks, much of which is surveyed by Herlihy and Busch [22]. An empirical comparison among various counting techniques can be found in [51]. Aharonson and Attiya [4] and Felten et al. [31] study counting networks with arbitrary fan-in and fan-out. Shavit and Touitou [126] show how to perform decrements on counting network counters by introducing the notion of "anti-tokens" and elimination. Busch and Mavronicolas [23] provide a combinatorial classification of the various properties of counting networks. Randomized counting networks are introduced by Aiello et al. [5] and fault-tolerant networks are presented by Riedel and Bruck [117].

The classical counting network structures in the literature are lock-free but not linearizable, they are only quiescently consistent. Herlihy et al. [56] show the tradeoffs involved in making counting networks linearizable.

Klugerman and Plaxton present an optimal $\log w$-depth counting network [72]. However, this construction is not practical, and all practical counting network implementations have $\log^2 w$ depth. Shavit and Zemach introduce *diffracting trees* [128], improved counting networks made of balancers with one input and two output wires laid out as a binary tree. The simple balancers of the counting network are replaced by more sophisticated *diffracting balancers* that can withstand high loads by using a randomized collision array approach, yielding lower depth counting networks with significantly improved throughput. An adaptive diffracting tree that adapts its size to load is presented in [26].

47.3 Stacks and Queues

Stacks and queues are among the simplest sequential data structures. Numerous issues arise in designing concurrent versions of these data structures, clearly illustrating the challenges involved in designing data structures for shared-memory multiprocessors.

Stacks

A concurrent *stack* is a data structure linearizable to a sequential stack that provides `push` and `pop` operations with the usual *LIFO* semantics. Various alternatives exist for the

behavior of these data structures in full or empty states, including returning a special value indicating the condition, raising an exception, or blocking.

Michael and Scott present several linearizable lock-based concurrent stack implementations: they are based on sequential linked lists with a top pointer and a global lock that controls access to the stack [102]. They typically scale poorly because even if one reduces contention on the lock, the top of the stack is a sequential bottleneck. Combining funnels [130] have been used to implement a linearizable stack that provides parallelism under high load. As with all combining structures, it is blocking, and it has a high overhead which makes it unsuitable for low loads.

Treiber [133] was the first to propose a lock-free concurrent stack implementation. He represented the stack as a singly-linked list with a top pointer and used CAS to modify the value of the top pointer atomically. Michael and Scott [102] compare the performance of Treiber's stack to an optimized nonblocking algorithm based on Herlihy's methodology [50], and several lock-based stacks (such as an MCS lock [96]) in low load situations. They concluded that Treiber's algorithm yields the best overall performance, and that this performance gap increases as the degree of multiprogramming grows. However, because the top pointer is a sequential bottleneck, even with an added backoff mechanism to reduce contention, the Treiber stack offers little scalability as concurrency increases [47].

Hendler et al. [47] observe that any stack implementation can be made more scalable using the *elimination* technique of Shavit and Touitou [126]. Elimination allows pairs of operations with reverse semantics—like pushes and pops on a stack—to complete without any central coordination, and therefore substantially aids scalability. The idea is that if a `pop` operation can find a concurrent `push` operation to "partner" with, then the `pop` operation can take the `push` operation's value, and both operations can return immediately. The net effect of each pair is the same as if the `push` operation was followed immediately by the `pop` operation, in other words, they eliminate each other's effect on the state of the stack. Elimination can be achieved by adding a collision array from which each operation chooses a location at random, and then attempts to coordinate with another operation that concurrently chose the same location [126]. The number of eliminations grows with concurrency, resulting in a high degree of parallelism. This approach, especially if the collision array is used as an adaptive backoff mechanism on the shared stack, introduces a high degree of parallelism with little contention [47], and delivers a scalable lock-free linearizable stack.

There is a subtle point in the Treiber stack used in the implementations above that is typical of many CAS-based algorithms. Suppose several concurrent threads all attempt a `pop` operation that removes the first element, located in some node "A," from the list by using a CAS to redirect the head pointer to point to a previously-second node "B." The problem is that it is possible for the list to change completely just before a particular `pop` operation attempts its CAS, so that by the time it does attempt it, the list has the node "A" as the first node as before, but the rest of the list including "B" is in a completely different order. This CAS of the head pointer from "A" to "B" may now succeed, but "B" might be anywhere in the list and the stack will behave incorrectly. This is an instance of the "ABA" problem [110], which plagues many CAS-based algorithms. To avoid this problem, Treiber augments the head pointer with a version number that is incremented every time the head pointer is changed. Thus, in the above scenario, the changes to the stack would cause the CAS to fail, thereby eliminating the ABA problem.[§]

[§]Note that the version number technique does not technically eliminate the ABA problem because the version number can wrap around; see [105] for a discussion of the consequences of this point in practice,

Queues

A concurrent *queue* is a data structure linearizable to a sequential queue that provides `enqueue` and `dequeue` operations with the usual *FIFO* semantics.

Michael and Scott [102] present a simple lock-based queue implementation that improves on the naive single-lock approach by having separate locks for the head and tail pointers of a linked-list-based queue. This allows an `enqueue` operation to execute in parallel with a `dequeue` operation (provided we avoid false sharing by placing the head and tail locks in separate cache lines). This algorithm is quite simple, with one simple trick: a "dummy" node is always in the queue, which allows the implementation to avoid acquiring both the head and tail locks in the case that the queue is empty, and therefore it avoids deadlock.

It is a matter of folklore that one can implement an array-based lock-free queue for a single enqueuer thread and a single dequeuer thread using only load and store operations [81]. A linked-list-based version of this algorithm appears in [46]. Herlihy and Wing [57] present a lock-free array-based queue that works if one assumes an unbounded size array. A survey in [102] describes numerous flawed attempts at devising general (multiple enqueuers, multiple dequeuers) nonblocking queue implementations. It also discusses some correct implementations that involve much more overhead than the ones discussed below.

Michael and Scott [102] present a linearizable CAS-based lock-free queue with parallel access to both ends. The structure of their algorithm is very simple and is similar to the two-lock algorithm mentioned above: it maintains head and tail pointers, and always keeps a dummy node in the list. To avoid using a lock, the `enqueue` operation adds a new node to the end of the list using CAS, and then uses CAS to update the tail pointer to reflect the addition. If the `enqueue` is delayed between these two steps, another `enqueue` operation can observe the tail pointer "lagging" behind the end of the list. A simple *helping technique* [50] is used to recover from this case, ensuring that the tail pointer is always behind the end of the list by at most one element.

While this implementation is simple and efficient enough to be used in practice, it does have a disadvantage. Operations can access nodes already removed from the list, and therefore the nodes cannot be freed. Instead, they are put into a *freelist*—a list of nodes stored for reuse by future `enqueue` operations—implemented using Treiber's stack. This use of a freelist has the disadvantage that the space consumed by the nodes in the freelist cannot be freed for arbitrary reuse. Herlihy et al. [52] and Michael [100] have presented nonblocking memory management techniques that overcome this disadvantage.

It is interesting to note that the elimination technique is not applicable to queues: we cannot simply pass a value from an `enqueue` operation to a concurrent `dequeue` operation, because this would not respect the FIFO order with respect to other values in the queue.

Deques

A concurrent double-ended queue (*deque*) is a linearizable concurrent data structure that generalizes concurrent stacks and queues by allowing pushes and pops at both ends [73]. (See Chapter 2 for an introduction to stacks, queues and deques.) As with queues, implementations that allow operations on both ends to proceed in parallel without interfering with each other are desirable.

Lock-based deques can be implemented easily using the same two-lock approach used

and also a "bounded tag" algorithm that eliminates the problem entirely, at some cost in space and time.

for queues. Given the relatively simple lock-free implementations for stacks and queues, it is somewhat surprising that there is no known lock-free deque implementation that allows concurrent operations on both ends. Martin et al. [95] provide a summary of concurrent deque implementations, showing that, even using nonconventional two-word synchronization primitives such as *double-compare-and-swap* (DCAS) [106], it is difficult to design a lock-free deque. The only known nonblocking deque implementation for current architectures that supports noninterfering operations at opposite ends of the deque is an obstruction-free CAS-based implementation due to Herlihy et al. [53].

47.4 Pools

Much of the difficulty in implementing efficient concurrent stacks and queues arises from the ordering requirements on when an element that has been inserted can be removed. A concurrent *pool* [94] is a data structure that supports `insert` and `delete` operations, and allows a `delete` operation to remove *any* element that has been inserted and not subsequently deleted. This weaker requirement offers opportunities for improving scalability.

A high-performance pool can be built using any quiescently consistent counter implementation [10, 128]. Elements are placed in an array, and a `fetch-and-inc` operation is used to determine in which location an `insert` operation stores its value, and similarly from which location a `delete` operation takes its value. Each array element contains a full/empty bit or equivalent mechanism to indicate if the element to be removed has already been placed in the location. Using such a scheme, any one of the combining tree, combining funnel, counting network, or diffracting tree approaches described above can be used to create a high throughput shared pool by parallelizing the main bottlenecks: the shared counters. Alternatively, a "stack like" pool can be implemented by using a counter that allows increments and decrements, and again using one of the above techniques to parallelize it.

Finally, the elimination technique discussed earlier is applicable to pools constructed using combining funnels, counting networks, or diffracting trees: if `insert` and `delete` operations meet in the tree, the `delete` can take the value being inserted by the `insert` operation, and both can leave without continuing to traverse the structure. This technique provides high performance under high load.

The drawback of all these implementations is that they perform rather poorly under low load. Moreover, when used for work-load distribution [9, 19, 118], they do not allow us to exploit locality information, as pools designed specifically for work-load distribution do.

Workload distribution (or *load balancing*) algorithms involve a collection of pools of units of work to be done; each pool is local to a given processor. Threads create work items and place them in local pools, employing a load balancing algorithm to ensure that the number of items in the pools is balanced. This avoids the possibility that some processors are idle while others still have work in their local pools. There are two general classes of algorithms of this type: *work sharing* [46, 118] and *work stealing* [9, 19]. In a work sharing scheme, each processor attempts to continuously offload work from its pool to other pools. In work stealing, a thread that has no work items in its local pool steals work from other pools. Both classes of algorithms typically use randomization to select the pool with which to balance or the target pool for stealing.

The classical work stealing algorithm is due to Arora et al. [9]. It is based on a lock-free construction of a *deque* that allows operations by only one thread (the thread to which the pool is local) at one end of the deque, allowing only pop operations at the other end, and allowing concurrent pop operations at that end to "abort" if they interfere. A deque with these restrictions is suitable for work stealing, and the restrictions allow a simple

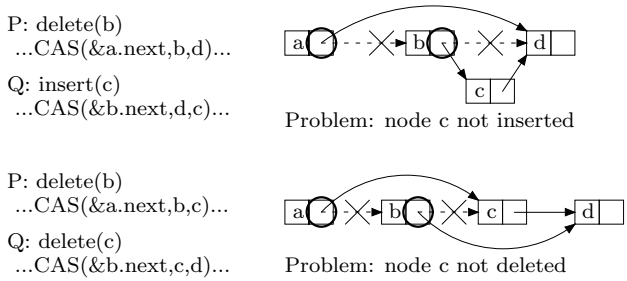


FIGURE 47.6: CAS-based list manipulation is hard. In both examples, P is deleting b from the list (the examples slightly abuse CAS notation). In the upper example, Q is trying to insert c into the list, and in the lower example, Q is trying to delete c from the list. Circled locations indicate the target addresses of the CAS operations; crossed out pointers are the values before the CAS succeeds.

implementation in which the local thread can insert and delete using simple low-cost load and store operations, resorting to a more expensive CAS operation only when it competes with the remote deleters for the last remaining item in the queue.

It has been shown that in some cases it is desirable to steal more than one item at a time [15, 103]. A nonblocking multiple-item work-stealing algorithm due to Hendler and Shavit appears in [45]. It has also been shown that in some cases it desirable to use affinity information of work items in deciding which items to steal. A locality-guided work stealing algorithm due to Acar et al. appears in [1].

47.5 Linked Lists

Consider implementations of concurrent search structures supporting `insert`, `delete`, and `search` operations. If these operations deal only with a key value, then the resulting data structure is a *set*; if a data value is associated with each key, we have a *dictionary* [24]. These are closely related data structures, and a concurrent set implementation can often be adapted to implement a dictionary. In the next three sections, we concentrate on implementing sets using different structures: linked lists, hash tables, and trees.

Suppose we use a linked list to implement a set. Apart from globally locking the linked list to prevent concurrent manipulation, the most popular approach to concurrent lock-based linked lists is *hand-over-hand locking* (sometimes called *lock coupling*) [14, 89]. In this approach, each node has an associated lock. A thread traversing the linked list releases a node's lock only after acquiring the lock of the next node in the list, thus preventing overtaking which may cause unnoticed removal of a node. This approach reduces lock granularity but significantly limits concurrency because insertions and deletions at disjoint list locations may delay each other.

One way to overcome this problem is to design lock-free linked lists. The difficulty in implementing a lock-free ordered linked list is ensuring that during an insertion or deletion, the adjacent nodes are still valid, i.e., they are still in the list and are still adjacent. As Figure 47.6 shows, designing such lock-free linked lists is not a straightforward matter.

The first CAS-based lock-free linked list is due to Valois [135], who uses a special auxiliary node in front of every regular node to prevent the undesired phenomena depicted in Figure 47.6. Valois's algorithm is correct when combined with a memory management solution due to Michael and Scott [101], but this solution is not practical. Harris [42] presents a

lock-free list that uses a special "deleted" bit that is accessed atomically with node pointers in order to signify that a node has been deleted; this scheme is applicable only in garbage collected environments. Michael [99] overcomes this disadvantage by modifying Harris's algorithm to make it compatible with memory reclamation methods [52, 100].

47.6 Hash Tables

A typical extensible hash table is a resizable array of buckets, each holding an expected constant number of elements, and thus requiring on average a constant time for `insert`, `delete` and `search` operations [24]. The principal cost of *resizing*—the redistribution of items between old and new buckets—is amortized over all table operations, thus keeping the cost of operations constant on average. Here resizing means extending the table, as it has been shown that as a practical matter, hash tables need only increase in size [58]. See Chapter 9 for more on hash tables.

Michael [99] shows that a concurrent non-extensible hash table can be achieved by placing a read-write lock on every bucket in the table. However, to guarantee good performance as the number of elements grows, hash tables must be extensible [30].

In the eighties, Ellis [29] and others [58, 77] extended the work of Fagin et al. [30] by designing an extensible concurrent hash table for distributed databases based on two-level locking schemes. A recent extensible hash algorithm by Lea [88] is known to be highly efficient in non-multiprogrammed environments [125]. It is based on a version of Litwin's sequential linear hashing algorithm [91]. It uses a locking scheme that involves a small number of high-level locks rather than a lock per bucket, and allows concurrent searches while resizing the table, but not concurrent inserts or deletes. Resizing is performed as a global restructuring of all buckets when the table size needs to be doubled.

Lock-based extensible hash-table algorithms suffer from all of the typical drawbacks of blocking synchronization, as discussed earlier. These problems become more acute because of the elaborate "global" process of redistributing the elements in all the hash table's buckets among newly added buckets. Lock-free extensible hash tables are thus a matter of both practical and theoretical interest.

As described in Section 47.5, Michael [99] builds on the work of Harris [42] to provide an effective CAS-based lock-free linked list implementation. He then uses this as the basis for a lock-free hash structure that performs well in multiprogrammed environments: a fixed-sized array of hash buckets, each implemented as a lock-free list. However, there is a difficulty in making a lock-free array of lists extensible since it is not obvious how to redistribute items in a lock-free manner when the bucket array grows. Moving an item between two bucket lists seemingly requires two CAS operations to be performed together atomically, which is not possible on current architectures.

Greenwald [39] shows how to implement an extensible hash table using his *two-handed emulation* technique. However, this technique employs a DCAS synchronization operation, which is not available on current architectures, and introduces excessive amounts of work during global resizing operations.

Shalev and Shavit [125] introduce a lock-free extensible hash table which works on current architectures. Their key idea is to keep the items in a single lock-free linked list instead of a list per bucket. To allow operations fast access to the appropriate part of the list, the Shalev-Shavit algorithm maintains a resizable array of "hints" (pointers into the list); operations use the hints to find a point in the list that is close to the relevant position, and then follow list pointers to find the position. To ensure a constant number of steps per operation on average, finer grained hints must be added as the number of elements in the list grows. To allow these

hints to be installed simply and efficiently, the list is maintained in a special *recursive split ordering*. This technique allows new hints to be installed incrementally, thereby eliminating the need for complicated mechanisms for atomically moving items between buckets, or reordering the list.

47.7 Search Trees

A concurrent implementation of any search tree (See Chapters 3 and 10 for more on sequential search trees) can be achieved by protecting it using a single exclusive lock. Concurrency can be improved somewhat by using a reader-writer lock to allow all read-only (`search`) operations to execute concurrently with each other while holding the lock in *shared mode*, while update (`insert` or `delete`) operations exclude all other operations by acquiring the lock in *exclusive mode*. If update operations are rare, this may be acceptable, but with even a moderate number of updates, the exclusive lock for update operations creates a sequential bottleneck that degrades performance substantially. By using fine-grained locking strategies—for example by using one lock per node, rather than a single lock for the entire tree—we can improve concurrency further.

Kung and Lehman [78] present a concurrent binary search tree implementation in which update operations hold only a constant number of node locks at a time, and these locks only exclude other update operations: search operations are never blocked. However, this implementation makes no attempt to keep the search tree balanced. In the remainder of this section, we focus on balanced search trees, which are considerably more challenging.

As a first step towards more fine-grained synchronization in balanced search tree implementations, we can observe that it is sufficient for an operation to hold an exclusive lock on the subtree in which it causes any modifications. This way, update operations that modify disjoint subtrees can execute in parallel. We briefly describe some techniques in this spirit in the context of B^+-trees. Recall that in B^+-trees, all keys and data are stored in leaf nodes; internal nodes only maintain routing information to direct operations towards the appropriate leaf nodes. Furthermore, an insertion into a leaf may require the leaf to be split, which may in turn require a new entry to be added to the leaf's parent, which itself may need to be split to accommodate the new entry. Thus, an insertion can potentially result in modifying all nodes along the path from the root to the leaf. However, such behavior is rare, so it does not make sense to exclusively lock the whole path just in case this occurs.

As a first step to avoiding such conservative locking strategies, we can observe that if an `insert` operation passes an internal B^+-tree node that is not full, then the modifications it makes to the tree cannot propagate past that node. In this case, we say that the node is *safe* with respect to the `insert` operation. If an update operation encounters a safe node while descending the tree acquiring exclusive locks on each node it traverses, it can safely release the locks on all ancestors of that node, thereby improving concurrency by allowing other operations to traverse those nodes [98, 120]. Because `search` operations do not modify the tree, they can descend the tree using lock coupling: as soon as a lock has been acquired on a child node, the lock on its parent can be released. Thus, `search` operations hold at most two locks (in shared mode) at any point in time, and are therefore less likely to prevent progress by other operations.

This approach still requires each update operation to acquire an exclusive lock on the root node, and to hold the lock while reading a child node, potentially from disk, so the root is still a bottleneck. We can improve on this approach by observing that most update operations will not need to split or merge the leaf node they access, and will therefore eventually release the exclusive locks on all of the nodes traversed on the way to the leaf.

This observation suggests an "optimistic" approach in which we descend the tree acquiring the locks in shared mode, acquiring only the leaf node exclusively [14]. If the leaf does not need to be split or merged, the update operation can complete immediately; in the rare cases in which changes do need to propagate up the tree, we can release all of the locks and then retry with the more pessimistic approach described above. Alternatively, we can use reader-writer locks that allow locks held in a shared mode to be "upgraded" to exclusive mode. This way, if an update operation discovers that it does need to modify nodes other than the leaf, it can upgrade locks it already holds in shared mode to exclusive mode, and avoid completely restarting the operation from the top of the tree [14]. Various combinations of the above techniques can be used because nodes near the top of the tree are more likely to conflict with other operations and less likely to be modified, while the opposite is true of nodes near the leaves [14].

As we employ some of the more sophisticated techniques described above, the algorithms become more complicated, and it becomes more difficult to avoid deadlock, resulting in even further complications. Nonetheless, all of these techniques maintain the invariant that operations exclusively lock the subtrees that they modify, so operations do not encounter states that they would not encounter in a sequential implementation. Significant improvements in concurrency and performance can be made by relaxing this requirement, at the cost of making it more difficult to reason that the resulting algorithms are correct.

A key difficulty we encounter when we attempt to relax the strict subtree locking schemes is that an operation descending the tree might follow a pointer to a child node that is no longer the correct node because of a modification by a concurrent operation. Various techniques have been developed that allow operations to recover from such "confusion", rather than strictly avoiding it.

An important example in the context of B^+-trees is due to Lehman and Yao [90], who define B^{link}-trees: B^+-trees with "links" from each node in the tree to its right neighbor at the same level of the tree. These links allow us to "separate" the splitting of a node from modifications to its parent to reflect the splitting. Specifically, in order to split a node n, we can create a new node n' to its right, and install a link from n to n'. If an operation that is descending the tree reaches node n while searching for a key position that is now covered by node n' due to the split, the operation can simply follow the link from n to n' to recover. This allows a node to be split without preventing access by concurrent operations to the node's parent. As a result, update operations do not need to simultaneously lock the entire subtree they (potentially) modify. In fact, in the Lehman-Yao algorithm, update operations as well as `search` operations use the lock coupling technique so that no operation ever holds more than two locks at a time, which significantly improves concurrency. This technique has been further refined, so that operations never hold more than one lock at a time [119].

Lehman and Yao do not address how nodes can be merged, instead allowing `delete` operations to leave nodes underfull. They argue that in many cases `delete` operations are rare, and that if space utilization becomes a problem, the tree can occasionally be reorganized in "batch" mode by exclusively locking the entire tree. Lanin and Shasha [83] incorporate merging into the `delete` operations, similarly to how `insert` operations split overflowed nodes in previous implementations. Similar to the Lehman-Yao link technique, these implementations use links to allow recovery by operations that have mistakenly reached a node that has been evacuated due to node merging.

In all of the algorithms discussed above, the maintenance operations such as node splitting and merging (where applicable) are performed as part of the regular update operations. Without such tight coupling between the maintenance operations and the regular operations that necessitate them, we cannot guarantee strict balancing properties. However, if we relax the balance requirements, we can separate the tree maintenance work from the update

operations, resulting in a number of advantages that outweigh the desire to keep search trees strictly balanced. As an example, the B^{link}-tree implementation in [119] supports a *compression process* that can run concurrently with regular operations to merge nodes that are underfull. By separating this work from the regular update operations, it can be performed concurrently by threads running on different processors, or in the background.

The idea of separating rebalancing work from regular tree operations was first suggested for red-black trees [40], and was first realized in [71] for AVL trees [2] supporting `insert` and `search` operations. An implementation that also supports `delete` operations is provided in [108]. These implementations improve concurrency by breaking balancing work down into small, local tree transformations that can be performed independently. Analysis in [84] shows that with some modifications, the scheme of [108] guarantees that each update operation causes at most $O(\log N)$ rebalancing operations for an N-node AVL tree. Similar results exist for B-trees [87, 108] and red-black trees [20, 107].

The only nonblocking implementations of balanced search trees have been achieved using Dynamic Software Transactional Memory mechanisms [33, 54]. These implementations use transactions translated from sequential code that performs rebalancing work as part of regular operations.

The above brief survey covers only basic issues and techniques involved with implementing concurrent search trees. To mention just a few of the numerous improvements and extensions in the literature, [104] addresses practical issues for the use of B^+-trees in commercial database products, such as recovery after failures; [74] presents concurrent implementations for *generalized search trees* (GiSTs) that facilitate the design of search trees without repeating the delicate work involved with concurrency control; and [85, 86] present several types of trees that support the efficient insertion and/or deletion of a group of values. Pugh [111] presents a concurrent version of his skiplist randomized search structure [112]. *Skiplists* are virtual tree structures consisting of multiple layers of linked lists. The expected search time in a skiplist is logarithmic in the number of elements in it. The main advantage of skiplists is that they do not require rebalancing: insertions are done in a randomized fashion that keeps the search tree balanced.

Empirical and analytical evaluations of concurrent search trees and other data structures can be found in [41, 66].

47.8 Priority Queues

A concurrent *priority queue* is a data structure linearizable to a sequential priority queue that provides `insert` and `delete-min` operations with the usual priority queue semantics. (See Part II of this handbook for more on sequential priority queues.)

Heap-Based Priority Queues

Many of the concurrent priority queue constructions in the literature are linearizable versions of the heap structures described earlier in this book. Again, the basic idea is to use fine-grained locking of the individual heap nodes to allow threads accessing different parts of the data structure to do so in parallel where possible. A key issue in designing such concurrent heaps is that traditionally `insert` operations proceed from the bottom up and `delete-min` operations from the top down, which creates potential for deadlock. Biswas and Brown [17] present such a lock-based heap algorithm assuming specialized "cleanup" threads to overcome deadlocks. Rao and Kumar [115] suggest to overcome the drawbacks of [17] using an algorithm that has both `insert` and `delete-min` operations proceed from the top down. Ayani [11] improved on their algorithm by suggesting a way to have consecutive

insertions be performed on opposite sides of the heap. Jones [67] suggests a scheme similar to [115] based on a skew heap.

Hunt et al. [60] present a heap-based algorithm that overcomes many of the limitations of the above schemes, especially the need to acquire multiple locks along the traversal path in the heap. It proceeds by locking for a short duration a variable holding the size of the heap and a lock on either the first or last element of the heap. In order to increase parallelism, insertions traverse the heap bottom-up while deletions proceed top-down, without introducing deadlocks. Insertions also employ a left-right technique as in [11] to allow them to access opposite sides on the heap and thus minimize interference.

On a different note, Huang and Weihl [59] show a concurrent priority queue based on a concurrent version of Fibonacci Heaps [34].

Nonblocking linearizable heap-based priority queue algorithms have been proposed by Herlihy [50], Barnes [12], and Israeli and Rappoport [64]. Sundell and Tsigas [132] present a lock-free priority queue based on a lock-free version of Pugh's concurrent skiplist [111].

Tree-Based Priority Pools

Huang and Weihl [59] and Johnson [65] describe concurrent *priority pools*: priority queues with relaxed semantics that do not guarantee linearizability of the `delete-min` operations. Their designs are both based on a modified concurrent B^+-tree implementation. Johnson introduces a "delete bin" that accumulates values to be deleted and thus reduces the load when performing concurrent `delete-min` operations. Shavit and Zemach [129] show a similar pool based on Pugh's concurrent skiplist [111] with an added "delete bin" mechanism based on [65]. Typically, the weaker pool semantics allows for increased concurrency. In [129] they further show that if the size of the set of allowable keys is bounded (as is often the case in operating systems) a priority pool based on a binary tree of combining funnel nodes can scale to hundreds (as opposed to tens) of processors.

47.9 Summary

We have given an overview of issues related to the design of concurrent data structures for shared-memory multiprocessors, and have surveyed some of the important contributions in this area. Our overview clearly illustrates that the design of such data structures provides significant challenges, and as of this writing, the maturity of concurrent data structures falls well behind that of sequential data structures. However, significant progress has been made towards understanding key issues and developing new techniques to facilitate the design of effective concurrent data structures; we are particularly encouraged by renewed academic and industry interest in stronger hardware support for synchronization. Given new understanding, new techniques, and stronger hardware support, we believe significant advances in concurrent data structure designs are likely in the coming years.

References

[1] U. Acar, G. Blelloch, and R. Blumofe. The data locality of work stealing. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 1–12, 2000.

[2] G. Adel'son-Vel'skii and E. Landis. An algorithm for the organization of information. *Doklady Akademii Nauk USSR*, 146(2):263–266, 1962.

[3] A. Agarwal and M. Cherian. Adaptive backoff synchronization techniques. In *Proceedings of the 16th International Symposium on Computer Architecture*, pages 396–406, May 1989.

[4] E. Aharonson and H. Attiya. Counting networks with arbitrary fan-out. *Distributed Computing*, 8(4):163–169, 1995.

[5] B. Aiello, R. Venkatesan, and M. Yung. Coins, weights and contention in balancing networks. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, pages 193–214, August 1994.

[6] J. Anderson, Y. Kim, and T. Herman. Shared-memory mutual exclusion: Major research trends since 1986. *Distributed Computing*, 16:75–110, 2003.

[7] J. Anderson and J. Yang. Time/contention trade-offs for multiprocessor synchronization. *Information and Computation*, 124(1):68–84, 1996.

[8] T. Anderson. The performance implications of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, 1990.

[9] N. Arora, R. Blumofe, and G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems*, 34(2):115–144, 2001.

[10] J. Aspnes, M. Herlihy, and N. Shavit. Counting networks. *Journal of the ACM*, 41(5):1020–1048, 1994.

[11] R. Ayani. LR-algorithm: concurrent operations on priority queues. In *In Proceedings of the 2nd IEEE Symposium on Parallel and Distributed Processing*, pages 22–25, 1991.

[12] G. Barnes. Wait free algorithms for heaps. Technical Report TR-94-12-07, University of Washington, 1994.

[13] K. Batcher. Sorting networks and their applications. In *Proceedings of AFIPS Joint Computer Conference*, pages 338–334, 1968.

[14] R. Bayer and M. Schkolnick. Concurrency of operations on B-trees. *Acta Informatica*, 9:1–21, 1979.

[15] P. Berenbrink, T. Friedetzky, and L. Goldberg. The natural work-stealing algorithm is stable. *SIAM Journal on Computing*, 32(5):1260–1279, 2003.

[16] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[17] J. Biswas and J. Browne. Simultaneous update of priority structures. In *In Proceedings of the 1987 International Conference on Parallel Processing*, pages 124–131, August 1987.

[18] G. Blelloch, P. Cheng, and P. Gibbons. Room synchronizations. In *Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures*, pages 122–133. ACM Press, 2001.

[19] R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, 1999.

[20] J. Boyar and K. Larsen. Efficient rebalancing of chromatic search trees. *Journal of Computer and System Sciences*, 49(3):667–682, December 1994.

[21] E.D. Brooks III. The butterfly barrier. *International Journal of Parallel Programming*, 15(4):295–307, October 1986.

[22] C. Busch and M. Herlihy. A survey on counting networks.

[23] C. Busch and M. Mavronicolas. A combinatorial treatment of balancing networks. *Journal of the ACM*, 43(5):794–839, September 1996.

[24] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, second edition, 2001.

[25] T. Craig. Building FIFO and priority-queueing spin locks from atomic swap. Technical Report TR 93-02-02, University of Washington, Department of Computer Science, February 1993.

[26] G. Della-Libera and N. Shavit. Reactive diffracting trees. *Journal of Parallel Dis-*

tributed Computing, 60(7):853–890, 2000.

[27] E. Dijkstra and C. Sholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, 1980.

[28] Cynthia Dwork, Maurice Herlihy, and Orli Waarts. Contention in shared memory algorithms. *Journal of the ACM*, 44(6):779–805, 1997.

[29] C. Ellis. Concurrency in linear hashing. *ACM Transactions on Database Systems (TODS)*, 12(2):195–217, 1987.

[30] R. Fagin, J. Nievergelt, N. Pippenger, and H. Strong. Extendible hashing: a fast access method for dynamic files. *ACM Transactions on Database Systems*, 4(3):315–355, September 1979.

[31] E. Felten, A. Lamarca, and R. Ladner. Building counting networks from larger balancers. Technical Report TR-93-04-09, University of Washington, 1993.

[32] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, pages 374–382, 1985.

[33] K. Fraser. *Practical Lock-Freedom.* Ph.D. dissertation, Kings College, University of Cambridge, Cambridge, England, September 2003.

[34] M. Fredman and R. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987.

[35] P. Gibbons, Y. Matias, and V. Ramachandran. The queue-read queue-write PRAM model: Accounting for contention in parallel algorithms. *SIAM Journal on Computing*, 28(2):733–769, 1998.

[36] J. Goodman, M. Vernon, and P. Woest. Efficient synchronization primitives for large-scale cache-coherent shared-memory multiprocessors. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, pages 64–75. ACM Press, April 1989.

[37] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, and M. Snir. The NYU ultracomputer - designing an MIMD parallel computer. *IEEE Transactions on Computers*, C-32(2):175–189, February 1984.

[38] G. Graunke and S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computer*, 23(6):60–70, June 1990.

[39] M. Greenwald. Two-handed emulation: How to build non-blocking implementations of complex data structures using DCAS. In *Proceedings of the 21st Annual Symposium on Principles of Distributed Computing*, 2002.

[40] L. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. *In Proceedings of the 19th IEEE Symposium on Foundations of Computer Science*, pages 8–21, October 1978.

[41] S. Hanke. The performance of concurrent red-black tree algorithms. *Lecture Notes in Computer Science*, 1668:286–300, 1999.

[42] T. Harris. A pragmatic implementation of non-blocking linked-lists. *15th International Conference on Distributed Computing, Lecture Notes in Computer Science*, 2180, pages 300–314, 2001.

[43] T. Harris and K. Fraser. Language support for lightweight transactions. In *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 388–402. ACM Press, 2003.

[44] T. Harris, K. Fraser, and I. Pratt. A practical multi-word compare-and-swap operation. In *Proceedings of the 16th International Symposium on DIStributed Computing*, pages 265–279, 2002.

[45] D. Hendler and N. Shavit. Non-blocking steal-half work queues. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing*, 2002.

[46] D. Hendler and N. Shavit. Work dealing. In *Proceedings of the 14th Annual ACM*

Symposium on Parallel Algorithms and Architectures (SPAA 2002), pages 164–172, 2002.

[47] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *Proceedings of the 16th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2004)*, pages 206–215, 2004.

[48] D. Hensgen, R. Finkel, and U. Manber. Two algorithms for barrier synchronization. *International Journal of Parallel Programming*, 17(1):1–17, 1988.

[49] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[50] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.

[51] M. Herlihy, B. Lim, and N. Shavit. Scalable concurrent counting. *ACM Transactions on Computer Systems*, 13(4):343–364, 1995.

[52] M. Herlihy, V. Luchangco, and M. Moir. The repeat offender problem: A mechanism for supporting lock-free dynamic-sized data structures. In *Proceedings of the 16th International Symposium on DIStributed Computing*, volume 2508, pages 339–353. Springer-Verlag Heidelberg, January 2002. A improved version of this paper is in preparation for journal submission; please contact authors.

[53] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, pages 522–529. IEEE, 2003.

[54] M. Herlihy, V. Luchangco, M. Moir, and W. Scherer. Software transactional memory for dynamic data structures. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing*, 2003.

[55] M. Herlihy and E. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the Twentieth Annual International Symposium on Computer Architecture*, 1993.

[56] M. Herlihy, N. Shavit, and O. Waarts. Linearizable counting networks. *Distributed Computing*, 9(4):193–203, 1996.

[57] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[58] M. Hsu and W. Yang. Concurrent operations in extendible hashing. In *Symposium on Very Large Data Bases*, pages 241–247, 1986.

[59] Q. Huang and W. Weihl. An evaluation of concurrent priority queue algorithms. In *IEEE Parallel and Distributed Computing Systems*, pages 518–525, 1991.

[60] G. Hunt, M. Michael, S. Parthasarathy, and M. Scott. An efficient algorithm for concurrent priority queue heaps. *Information Processing Letters*, 60(3):151–157, November 1996.

[61] IBM. System/370 principles of operation. Order Number GA22-7000.

[62] IBM. Powerpc microprocessor family: Programming environments manual for 64 and 32-bit microprocessors, version 2.0, 2003.

[63] Intel. *Pentium Processor Family User's Manual: Vol 3, Architecture and Programming Manual*, 1994.

[64] A. Israeli and L. Rappoport. Efficient wait-free implementation of a concurrent priority queue. In *The 7th International Workshop on Distributed Algorithms*, pages 1–17, 1993.

[65] T. Johnson. A highly concurrent priority queue based on the B-link tree. Technical Report 91-007, University of Florida, August 1991.

[66] T. Johnson and D. Sasha. The performance of current B-tree algorithms. *ACM Transactions on Database Systems (TODS)*, 18(1):51–101, 1993.

[67] D. Jones. Concurrent operations on priority queues. *Communications of the ACM*, 32(1):132–137, 1989.

[68] Y. Joung. The congenial talking philosophers problem in computer networks. *Distrib. Comput.*, 15(3):155–175, 2002.

[69] G. Kane. *MIPS RISC Architecture*. Prentice Hall, New York, 1989.

[70] P. Keane and M. Moir. A simple local-spin group mutual exclusion algorithm. In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 23–32. ACM Press, 1999.

[71] J. Kessels. On-the-fly optimization of data structures. *Communications of the ACM*, 26(11):895–901, November 1983.

[72] M. Klugerman and G. Plaxton. Small-depth counting networks. In *Proceedings of the Twenty-Fourth Annual ACM Symposium on the Theory of Computing*, pages 417–428, 1992.

[73] D. Knuth. *The Art of Computer Programming: Fundamental Algorithms*. Addison-Wesley, 2nd edition, 1968.

[74] M. Kornacker, C. Mohan, and J. Hellerstein. Concurrency and recovery in generalized search trees. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pages 62–72. ACM Press, 1997.

[75] O. Krieger, M. Stumm, R. Unrau, and J. Hanna. A fair fast scalable reader-writer lock. In *Proceedings of the 1993 International Conference on Parallel Processing*, volume II - Software, pages II–201–II–204, Boca Raton, FL, 1993. CRC Press.

[76] C. Kruskal, L. Rudolph, and M. Snir. Efficient synchronization on multiprocessors with shared memory. In *Fifth ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, August 1986.

[77] V. Kumar. Concurrent operations on extendible hashing and its performance. *Communications of the ACM*, 33(6):681–694, 1990.

[78] H. Kung and P. Lehman. Concurrent manipulation of binary search trees. *ACM Transactions on Programming Languages and Systems*, 5:354–382, September 1980.

[79] H. Kung and J. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, 1981.

[80] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):241–248, September 1979.

[81] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, 1983.

[82] Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974.

[83] V. Lanin and D. Shasha. A symmetric concurrent B-tree algorithm. In *Proceedings of the Fall Joint Computer Conference 1986*, pages 380–389. IEEE Computer Society Press, November 1986.

[84] K. Larsen. AVL trees with relaxed balance. In *Eighth International Parallel Processing Symposium*, pages 888–893. IEEE Computer Society Press, April 1994.

[85] K. Larsen. Relaxed multi-way trees with group updates. In *Proceedings of the Twentieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 93–101. ACM Press, 2001.

[86] K. Larsen. Relaxed red-black trees with group updates. *Acta Informatica*, 38(8):565–586, 2002.

[87] K. Larsen and R. Fagerberg. B-trees with relaxed balance. In *Ninth International Parallel Processing Symposium*, pages 196–202. IEEE Computer Society Press, April 1995.

[88] D. Lea. Concurrent hash map in JSR166 concurrency utilities. http://gee.cs.oswego.edu/dl/concurrency-interest/index.html.

[89] D. Lea. *Concurrent Programming in Java(TM): Design Principles and Pattern*. Addison-Wesley, second edition, 1999.

[90] P. Lehman and S. Bing Yao. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems (TODS)*, 6(4):650–670, December 1981.

[91] W. Litwin. Linear hashing: A new tool for file and table addressing. In *Sixth International Conference on Very Large Data Bases*, pages 212–223. IEEE Computer Society, October 1980.

[92] M. Loui and H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. In F. P. Preparata, editor, *Advances in Computing Research*, volume 4, pages 163–183. JAI Press, Greenwich, CT, 1987.

[93] P. Magnussen, A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. In *Proceedings of the 8th International Symposium on Parallel Processing (IPPS)*, pages 165–171. IEEE Computer Society, April 1994.

[94] Udi Manber. On maintaining dynamic information in a concurrent environment. In *Proceedings of the Sixteenth Annual ACM symposium on Theory of Computing*, pages 273–278. ACM Press, 1984.

[95] P. Martin, M. Moir, and G. Steele. Dcas-based concurrent deques supporting bulk allocation. Technical Report TR-2002-111, Sun Microsystems Laboratories, 2002.

[96] J. Mellor-Crummey and M. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

[97] J. Mellor-Crummey and M. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. *SIGPLAN Not.*, 26(7):106–113, 1991.

[98] J. Metzger. Managing simultaneous operations in large ordered indexes. Technical report, Technische Universität München, Institut für Informatik, TUM-Math, 1975.

[99] M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the Fourteenth Snnual ACM symposium on Parallel Algorithms and Architectures*, pages 73–82. ACM Press, 2002.

[100] M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *The 21st Annual ACM Symposium on Principles of Distributed Computing*, pages 21–30. ACM Press, 2002.

[101] M. Michael and M. Scott. Correction of a memory management method for lock-free data structures. Technical Report TR599, University of Rochester, 1995.

[102] M. Michael and M. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared - memory multiprocessors. *Journal of Parallel and Distributed Computing*, 51(1):1–26, 1998.

[103] M. Mitzenmacher. Analyses of load stealing models based on differential equations. In *Proceedings of the 10th ACM Symposium on Parallel Algorithms and Architectures*, pages 212–221, 1998.

[104] C. Mohan and F. Levine. Aries/im: an efficient and high concurrency index management method using write-ahead logging. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, pages 371–380. ACM Press, 1992.

[105] M. Moir. Practical implementations of non-blocking synchronization primitives. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed*

Computing, pages 219–228, 1997.

[106] Motorola. *MC68020 32-bit microprocessor user's manual*. Prentice-Hall, 1986.

[107] O. Nurmi and E. Soisalon-Soininen. Uncoupling updating and rebalancing in chromatic binary search trees. In *Proceedings of the Tenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 192–198. ACM Press, May 1991.

[108] O. Nurmi, E. Soisalon-Soininen, and D. Wood. Concurrency control in database structures with relaxed balance. In *Proceedings of the sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 170–176. ACM Press, 1987.

[109] D. Patterson and J. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, second edition, 1997.

[110] S. Prakash, Y. Lee, and T. Johnson. A non-blocking algorithm for shared queues using compare-and-swap. *IEEE Transactions on Computers*, 43(5):548–559, 1994.

[111] W. Pugh. Concurrent maintenance of skip lists. Technical Report CS-TR-2222.1, Institute for Advanced Computer Studies, Department of Computer Science, University of Maryland, 1989.

[112] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *ACM Transactions on Database Systems*, 33(6):668–676, 1990.

[113] R. Rajwar and J. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, pages 294–305, 2001.

[114] R. Rajwar and J. Goodman. Transactional lock-free execution of lock-based programs. In *10th Symposium on Architectural Support for Programming Languages and Operating Systems*, October 2003.

[115] V. Rao and V. Kumar. Concurrent access of priority queues. *IEEE Transactions on Computers*, 37:1657–1665, December 1988.

[116] M. Raynal. *Algorithms for Mutual Exclusion*. The MIT Press, Cambridge, MA, 1986.

[117] M. Riedel and J. Bruck. Tolerating faults in counting networks. Poster presented at *10th ACM Symposium on Parallel Algorithms and Architectures*, pages 27–36, 1998.

[118] L. Rudolph, M. Slivkin-Allalouf, and E. Upfal. A simple load balancing scheme for task allocation in parallel machines. In *In Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 237–245. ACM Press, July 1991.

[119] Y. Sagiv. Concurrent operations on B-trees with overtaking. *Journal of Computer and System Sciences*, 33(2):275–296, October 1986.

[120] B. Samadi. B-trees in a system with multiple users. *Information Processing Letter*, 5(4):107–112, October 1976.

[121] W. Savitch and M. Stimson. Time bounded random access machines with parallel processing. *Journal of the ACM*, 26:103–118, 1979.

[122] M. Scott. Non-blocking timeout in scalable queue-based spin locks. In *Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing*, pages 31–40. ACM Press, 2002.

[123] M. Scott and J. Mellor-Crummey. Fast, contention-free combining tree barriers. *International Journal of Parallel Programming*, 22(4), August 1994.

[124] M. Scott and W. Scherer. Scalable queue-based spin locks with timeout. In *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 44–52, 2001.

[125] O. Shalev and N. Shavit. Split-ordered lists: Lock-free extensible hash tables. In *The 22nd Annual ACM Symposium on Principles of Distributed Computing*, pages 102–111. ACM Press, 2003.

[126] N. Shavit and D. Touitou. Elimination trees and the construction of pools and stacks. *Theory of Computing Systems*, 30:645–670, 1997.

[127] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, February 1997.

[128] N. Shavit and A. Zemach. Diffracting trees. *ACM Transactions on Computer Systems*, 14(4):385–428, 1996.

[129] N. Shavit and A. Zemach. Scalable concurrent priority queue algorithms. In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 113–122. ACM Press, 1999.

[130] N. Shavit and A. Zemach. Combining funnels: a dynamic approach to software combining. *J. Parallel Distrib. Comput.*, 60(11):1355–1387, 2000.

[131] R. Sites. *Alpha Architecture Reference Manual*, 1992.

[132] H. Sundell and P. Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium*, pages 84–94. IEEE press, 2003.

[133] R. Treiber. Systems programming: Coping with parallelism. Technical Report RJ5118, IBM Almaden Research Center, 1986.

[134] L. Valiant. *Bulk-Synchronous Parallel Computers*. Wiley, New York, NY, 1989.

[135] J. Valois. Lock-free linked lists using compare-and-swap. In *ACM Symposium on Principles of Distributed Computing*, pages 214–222, 1995.

[136] D. Weaver and T. Germond (Editors). *The SPARC Architecture Manual (Version 9)*. PTR Prentice Hall, Englewood Cliffs, NJ), 1994.

[137] P. Yew, N. Tzeng, and D. Lawrie. Distributing hot-spot addressing in large-scale multiprocessors. *IEEE Transactions on Computers*, C-36(4):388–395, April 1987.