1 Introduction

This chapter surveys results in the design and analysis of self-organizing data structures for the search problem. The general search problem in pointer data structures can be phrased as follows. The elements of a set are stored in a collection of nodes. Each node also contains $O(1)$ pointers to other nodes and additional state data which can be used for navigation and self-organization. The elements have associated key values, which may or may not be totally ordered (almost always they are). Various operations may be performed on the set, including the standard dictionary operations of searching for an element, inserting a new element, and deleting an element. Additional operations such as set splitting or joining may be allowed. This chapter considers two simple but very popular data structures: the unsorted linear list, and the binary search tree.

A self-organizing data structure has a rule or algorithm for changing pointers and state data after each operation. The self-organizing rule is designed to respond to initially unknown properties of the input request sequence, and to get the data structure into a state that will take advantage of these properties and reduce the time per operation. As operations occur, a self-organizing data structure may change its state quite dramatically.

Self-organizing data structures can be compared to static or constrained data structures. The state of a static data structure is predetermined by some strong knowledge about the properties of the input. For example, if searches are generated according to some known probability distribution, then a linear list may
be sorted by decreasing probability of access. A constrained data structure must satisfy some structural invariant, such as a balance constraint in a binary search tree. As long as the structural invariant is satisfied, the data structure does not change.

Self-organizing data structures have several advantages over static and constrained data structures [64]. (a) The amortized asymptotic time of search and update operations is usually as good as the corresponding time of constrained structures. But when the sequence of operations has favorable properties, the performance can be much better. (b) Self-organizing rules need no knowledge of the properties of input sequence, but will adapt the data structure to best suit the input. (c) The self-organizing rule typically results in search and update algorithms that are simple and easy to implement. (d) Often the self-organizing rule can be implemented without using any extra space in the nodes. (Such a rule is called "memoryless" since it saves no information to help make its decisions.)

On the other hand, self-organizing data structures have several disadvantages. (a) Although the total time of a sequence of operations is low, an individual operation can be quite expensive. (b) Reorganization of the structure has to be done even during search operations. Hence self-organizing data structures may have higher overheads than their static or constraint-based cousins.

Nevertheless, self-organizing data structures represent an attractive alternative to constraint structures, and reorganization rules have been studied extensively for both linear lists and binary trees. Both data structures have also received considerable attention within the study of on-line algorithms. In Section 2 we review results for linear lists. Almost all previous work in this area has concentrated on designing on-line algorithms for this data structure. In Section 3 we discuss binary search trees and present results on on-line and off-line algorithms. Self-organizing data structures can be used to construct effective data compression schemes. We address this application in Section 4.

2 Unsorted linear lists

The problem of representing a dictionary as an unsorted linear list is also known as the list update problem. Consider a set $S$ of items that has to be maintained under a sequence of requests, where each request is one of the following operations.

- **Access(x).** Locate item $x$ in $S$.
- **Insert(x).** Insert item $x$ into $S$.
- **Delete(x).** Delete item $x$ from $S$.

Given that $S$ shall be represented as an unsorted list, these operations can be implemented as follows. To access an item, a list update algorithm starts at the front of the list and searches linearly through the items until the desired item is found. To insert a new item, the algorithm first scans the entire list to verify that the item is not already present and then inserts the item at the end of the list. To delete an item, the algorithm scans the list to search for the item and then deletes it.