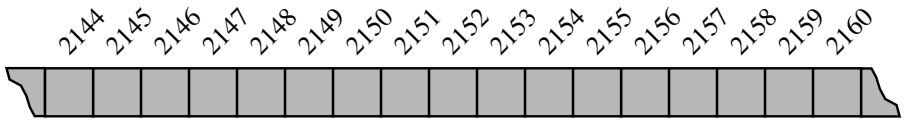
Our study of data structures thus far has focused primarily upon the efficiency of computations, as measured by the number of primitive operations that are executed on a central processing unit (CPU). In practice, the performance of a computer system is also greatly impacted by the management of the computer's memory systems. In our analysis of data structures, we have provided asymptotic bounds for the overall amount of memory used by a data structure. In this chapter, we consider more subtle issues involving the use of a computer's memory system.

We first discuss ways in which memory is allocated and deallocated during the execution of a computer program, and the impact that this has on the program's performance. Second, we discuss the complexity of multilevel memory hierarchies in today's computer systems. Although we often abstract a computer's memory as consisting of a single pool of interchangeable locations, in practice, the data used by an executing program is stored and transferred between a combination of physical memories (e.g., CPU registers, caches, internal memory, and external memory). We consider the use of classic data structures in the algorithms used to manage memory, and how the use of memory hierarchies impacts the choice of data structures and algorithms for classic problems such as searching and sorting.

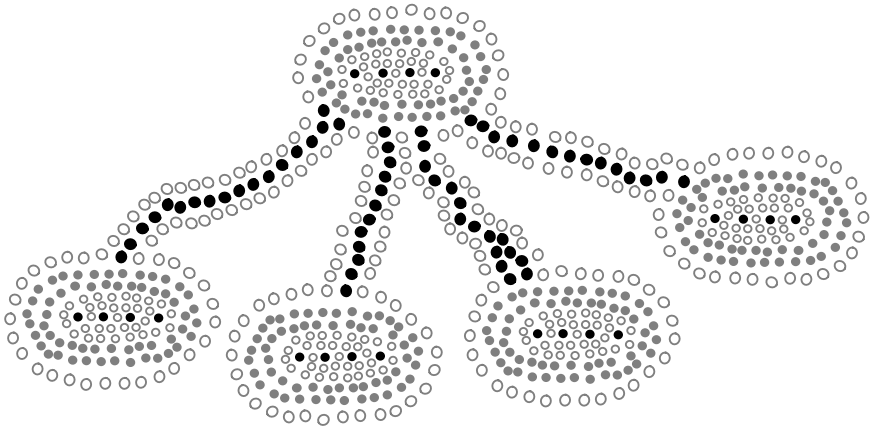
15.1 Memory Management

In order to implement any data structure on an actual computer, we need to use computer memory. Computer memory is organized into a sequence of ***words***, each of which typically consists of 4, 8, or 16 bytes (depending on the computer). These memory words are numbered from 0 to $N-1$, where N is the number of memory words available to the computer. The number associated with each memory word is known as its ***memory address***. Thus, the memory in a computer can be viewed as basically one giant array of memory words. For example, in Figure 5.1 of Section 5.2, we portrayed a section of the computer's memory as follows:



In order to run programs and store information, the computer's memory must be ***managed*** so as to determine what data is stored in what memory cells. In this section, we discuss the basics of memory management, most notably describing the way in which memory is allocated to store new objects, the way in which portions of memory are deallocated and reclaimed, when no longer needed, and the way in which the Python interpreter uses memory in completing its tasks.

Chapter 15 Memory Management and B-Trees



Contents

Contents and Organization

The chapters for this book are organized to provide a pedagogical path that starts with the basics of Python programming and object-oriented design. We then add foundational techniques like algorithm analysis and recursion. In the main portion of the book, we present fundamental data structures and algorithms, concluding with a discussion of memory management (that is, the architectural underpinnings of data structures). Specifically, the chapters for this book are organized as follows:

1. **Python Primer**
2. **Object-Oriented Programming**
3. **Algorithm Analysis**
4. **Recursion**
5. **Array-Based Sequences**
6. **Stacks, Queues, and Deques**
7. **Linked Lists**
8. **Trees**
9. **Priority Queues**
10. **Maps, Hash Tables, and Skip Lists**
11. **Search Trees**
12. **Sorting and Selection**
13. **Text Processing**
14. **Graph Algorithms**
15. **Memory Management and B-Trees**

A. **Character Strings in Python**

B. **Useful Mathematical Facts**

A more detailed table of contents follows this preface, beginning on page xi.

Prerequisites

We assume that the reader is at least vaguely familiar with a high-level programming language, such as C, C++, Python, or Java, and that he or she understands the main constructs from such a high-level language, including:

- Variables and expressions.
- Decision structures (such as if-statements and switch-statements).
- Iteration structures (for loops and while loops).
- Functions (whether stand-alone or object-oriented methods).

For readers who are familiar with these concepts, but not with how they are expressed in Python, we provide a primer on the Python language in Chapter 1. Still, this book is primarily a data structures book, not a Python book; hence, it does not give a comprehensive treatment of Python.

C-15.20 Consider the page caching strategy based on the ***least frequently used*** (LFU) rule, where the page in the cache that has been accessed the least often is the one that is evicted when a new page is requested. If there are ties, LFU evicts the least frequently used page that has been in the cache the longest. Show that there is a sequence P of n requests that causes LFU to miss $\Omega(n)$ times for a cache of m pages, whereas the optimal algorithm will miss only $O(m)$ times.

C-15.21 Suppose that instead of having the node-search function $f(d) = 1$ in an order-d B-tree T, we have $f(d) = \log d$. What does the asymptotic running time of performing a search in T now become?

Projects

P-15.22 Write a Python class that simulates the best-fit, worst-fit, first-fit, and next-fit algorithms for memory management. Determine experimentally which method is the best under various sequences of memory requests.

P-15.23 Write a Python class that implements all the methods of the ordered map ADT by means of an (a,b) tree, where a and b are integer constants passed as parameters to a constructor.

P-15.24 Implement the B-tree data structure, assuming a block size of 1024 and integer keys. Test the number of "disk transfers" needed to process a sequence of map operations.

Chapter Notes

The reader interested in the study of the architecture of hierarchical memory systems is referred to the book chapter by Burger *et al.* [21] or the book by Hennessy and Patterson [50]. The mark-sweep garbage collection method we describe is one of many different algorithms for performing garbage collection. We encourage the reader interested in further study of garbage collection to examine the book by Jones and Lins [56]. Knuth [62] has very nice discussions about external-memory sorting and searching, and Ullman [97] discusses external memory structures for database systems. The handbook by Gonnet and Baeza-Yates [44] compares the performance of a number of different sorting algorithms, many of which are external-memory algorithms. B-trees were invented by Bayer and McCreight [11] and Comer [28] provides a very nice overview of this data structure. The books by Mehlhorn [76] and Samet [87] also have nice discussions about B-trees and their variants. Aggarwal and Vitter [3] study the I/O complexity of sorting and related problems, establishing upper and lower bounds. Goodrich *et al.* [46] study the I/O complexity of several computational geometry problems. The reader interested in further study of I/O-efficient algorithms is encouraged to examine the survey paper of Vitter [99].

C-15.11 Describe an external-memory data structure to implement the queue ADT so that the total number of disk transfers needed to process a sequence of k enqueue and dequeue operations is $O(k/B)$.

C-15.12 Describe an external-memory version of the PositionalList ADT (Section 7.4), with block size B, such that an iteration of a list of length n is completed using $O(n/B)$ transfers in the worst case, and all other methods of the ADT require only $O(1)$ transfers.

C-15.13 Change the rules that define red-black trees so that each red-black tree T has a corresponding $(4,8)$ tree, and vice versa.

C-15.14 Describe a modified version of the B-tree insertion algorithm so that each time we create an overflow because of a split of a node w, we redistribute keys among all of w's siblings, so that each sibling holds roughly the same number of keys (possibly cascading the split up to the parent of w). What is the minimum fraction of each block that will always be filled using this scheme?

C-15.15 Another possible external-memory map implementation is to use a skip list, but to collect consecutive groups of $O(B)$ nodes, in individual blocks, on any level in the skip list. In particular, we define an ***order-d B-skip list*** to be such a representation of a skip list structure, where each block contains at least $\lceil d/2 \rceil$ list nodes and at most d list nodes. Let us also choose d in this case to be the maximum number of list nodes from a level of a skip list that can fit into one block. Describe how we should modify the skip-list insertion and removal algorithms for a B-skip list so that the expected height of the structure is $O(\log n/\log B)$.

C-15.16 Describe how to use a B-tree to implement the partition (union-find) ADT (from Section 14.7.3) so that the union and find operations each use at most $O(\log n/\log B)$ disk transfers.

C-15.17 Suppose we are given a sequence S of n elements with integer keys such that some elements in S are colored "blue" and some elements in S are colored "red." In addition, say that a red element e ***pairs*** with a blue element f if they have the same key value. Describe an efficient external-memory algorithm for finding all the red-blue pairs in S. How many disk transfers does your algorithm perform?

C-15.18 Consider the page caching problem where the memory cache can hold m pages, and we are given a sequence P of n requests taken from a pool of $m+1$ possible pages. Describe the optimal strategy for the offline algorithm and show that it causes at most $m+n/m$ page misses in total, starting from an empty cache.

C-15.19 Describe an efficient external-memory algorithm that determines whether an array of n integers contains a value occurring more than $n/2$ times.

15.4.1 Multiway Merging

In a standard merge-sort (Section 12.2), the merge process combines two sorted sequences into one by repeatedly taking the smaller of the items at the front of the two respective lists. In a d-way merge, we repeatedly find the smallest among the items at the front of the d sequences and place it as the next element of the merged sequence. We continue until all elements are included.

In the context of an external-memory sorting algorithm, if main memory has size M and each block has size B, we can store up to M/B blocks within main memory at any given time. We specifically choose $d = (M/B) - 1$ so that we can afford to keep one block from each input sequence in main memory at any given time, and to have one additional block to use as a buffer for the merged sequence. (See Figure 15.5.)

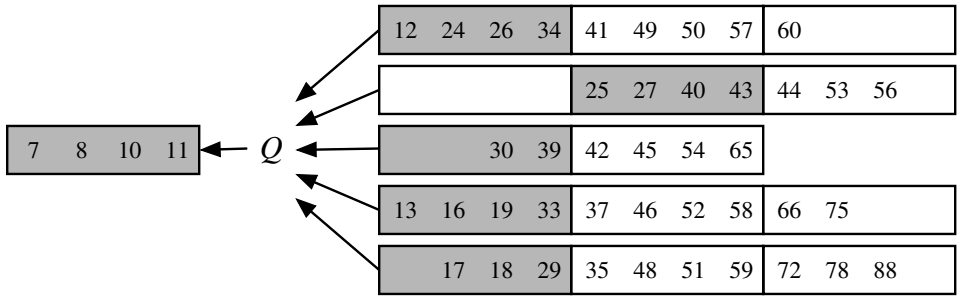


Figure 15.5: A d-way merge with $d = 5$ and $B = 4$. Blocks that currently reside in main memory are shaded.

We maintain the smallest unprocessed element from each input sequence in main memory, requesting the next block from a sequence when the preceding block has been exhausted. Similarly, we use one block of internal memory to buffer the merged sequence, flushing that block to external memory when full. In this way, the total number of transfers performed during a single d-way merge is $O(n/B)$, since we scan each block of list S_i once, and we write out each block of the merged list S' once. In terms of computation time, choosing the smallest of d values can trivially be performed using $O(d)$ operations. If we are willing to devote $O(d)$ internal memory, we can maintain a priority queue identifying the smallest element from each sequence, thereby performing each step of the merge in $O(\log d)$ time by removing the minimum element and replacing it with the next element from the same sequence. Hence, the internal time for the d-way merge is $O(n \log d)$.

Proposition 15.3: *Given an array-based sequence S of n elements stored compactly in external memory, we can sort S with $O((n/B)\log(n/B)/\log(M/B))$ block transfers and $O(n \log n)$ internal computations, where M is the size of the internal memory and B is the size of a block.*

15.3.2 B-Trees

A version of the (a,b) tree data structure, which is the best-known method for maintaining a map in external memory, is called the "B-tree." (See Figure 15.4.) A ***B-tree of order*** d is an (a,b) tree with $a = \lceil d/2 \rceil$ and $b = d$. Since we discussed the standard map query and update methods for (a,b) trees above, we restrict our discussion here to the I/O complexity of B-trees.

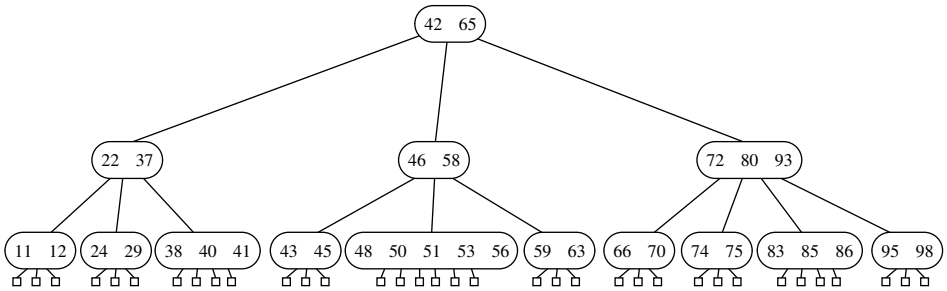


Figure 15.4: A B-tree of order 6.

An important property of B-trees is that we can choose d so that the d children references and the $d-1$ keys stored at a node can fit compactly into a single disk block, implying that d is proportional to B. This choice allows us to assume that a and b are also proportional to B in the analysis of the search and update operations on (a,b) trees. Thus, $f(b)$ and $g(b)$ are both $O(1)$, for each time we access a node to perform a search or an update operation, we need only perform a single disk transfer.

As we have already observed above, each search or update requires that we examine at most $O(1)$ nodes for each level of the tree. Therefore, any map search or update operation on a B-tree requires only $O(\log_{\lceil d/2 \rceil} n)$, that is, $O(\log n / \log B)$, disk transfers. For example, an insert operation proceeds down the B-tree to locate the node in which to insert the new entry. If the node would ***overflow*** (to have $d+1$ children) because of this addition, then this node is ***split*** into two nodes that have $\lfloor (d+1)/2 \rfloor$ and $\lceil (d+1)/2 \rceil$ children, respectively. This process is then repeated at the next level up, and will continue for at most $O(\log_B n)$ levels.

Likewise, if a remove operation results in a node ***underflow*** (to have $\lceil d/2 \rceil - 1$ children), then we move references from a sibling node with at least $\lceil d/2 \rceil + 1$ children or we perform a ***fusion*** operation of this node with its sibling (and repeat this computation at the parent). As with the insert operation, this will continue up the B-tree for at most $O(\log_B n)$ levels. The requirement that each internal node have at least $\lceil d/2 \rceil$ children implies that each disk block used to support a B-tree is at least half full. Thus, we have the following:

Proposition 15.2: *A B-tree with n entries has I/O complexity $O(\log_B n)$ for search or update operation, and uses $O(n/B)$ blocks, where B is the size of a block.*

15.3.1 (a,b) Trees

To reduce the number of external-memory accesses when searching, we can represent our map using a multiway search tree (Section 11.5.1). This approach gives rise to a generalization of the $(2,4)$ tree data structure known as the (a,b) tree.

An (a,b) tree is a multiway search tree such that each node has between a and b children and stores between $a-1$ and $b-1$ entries. The algorithms for searching, inserting, and removing entries in an (a,b) tree are straightforward generalizations of the corresponding ones for $(2,4)$ trees. The advantage of generalizing $(2,4)$ trees to (a,b) trees is that a generalized class of trees provides a flexible search structure, where the size of the nodes and the running time of the various map operations depends on the parameters a and b. By setting the parameters a and b appropriately with respect to the size of disk blocks, we can derive a data structure that achieves good external-memory performance.

Definition of an (a,b) Tree

An ***(a,b) tree***, where parameters a and b are integers such that $2 \leq a \leq (b+1)/2$, is a multiway search tree T with the following additional restrictions:

***Size Property*:** Each internal node has at least a children, unless it is the root, and has at most b children.

***Depth Property*:** All the external nodes have the same depth.

Proposition 15.1: *The height of an (a,b) tree storing n entries is $\Omega(\log n/\log b)$ and $O(\log n/\log a)$.*

Justification: Let T be an (a,b) tree storing n entries, and let h be the height of T. We justify the proposition by establishing the following bounds on h:

$$\frac{1}{\log b}\log(n+1) \leq h \leq \frac{1}{\log a}\log\frac{n+1}{2}+1.$$

By the size and depth properties, the number n'' of external nodes of T is at least $2a^{h-1}$ and at most b^h. By Proposition 11.7, $n'' = n+1$. Thus,

$$2a^{h-1} \leq n+1 \leq b^h.$$

Taking the logarithm in base 2 of each term, we get

$$(h-1)\log a + 1 \leq \log(n+1) \leq h\log b.$$

An algebraic manipulation of these inequalities completes the justification. ■